# Towards Self-Configuring Hardware for Distributed Computer Systems

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

## 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 the initial steps towards a system that can learn to alter its current configuration in reaction to the current workload. In particular, the advantages of shifting CPU and memory resources online are considered. Investigation on a publically available multi-machine, multi-process distributed system (the online transaction processing benchmark TPC-W) indicates that there is a real performance benefit to reconfiguration in reaction to workload changes. A learning framework is presented that does not require any instrumentation of the middleware, nor any special instrumentation of the operating system; rather, it learns to identify preferable configurations as well as their quantitative performance effects from system behavior as reported by standard monitoring tools. Initial results using the WEKA machine learning package suggest that automatic adaptive configuration can provide measurable performance benefits over any fixed configuration.*

## 1. Introduction

Recent advances in hardware devlopment have made adaptive hardware configuration possible. For example, processors and/or memory may be dynamically added to or removed from a running system. This paper establishes that such adaptive configuration can improve system performance when workloads vary.

As an experimental testbed for this research, we constructed an implementation of the TPC-W[1] benchmark using commonly available hardware and software. TPC-W provides a well-defined simulation of an online store with the capability to vary the workload and evaluate the system under varying demands.

This paper reports on three important steps towards the eventual goal of constructing a fully adaptive on-line system. First, we establish that dynamically reconfiguring hardware in response to workload changes has the *potential* to improve performance. That is, we show that there exists a set of hardware configurations and a set of workloads such that no configuration outperforms the others on all workloads and each configuration is best for at least one workload. We identify two configurations and five workloads for which the first configuration outperforms the second by at least 9% for two workloads and the second outperforms the first by over 7% for one workload.[2]

Second, we establish that, using uninstrumented middleware and given only raw, low-level system statistics, it is possible to *predict* which of the two configurations will outperform the other at any given time. In a scenario where the best constant guess gives about 60% accuracy, the system is able to identify the best configuration with over 90% accuracy.

Third, we extend this prediction capability to make precise numerical predictions. While knowing the best configuration for the current conditions may be useful, it is more beneficial to predict the quantitative change in performance when the system is switched to each possible alternative configuration. With such information, performance gains can be traded off against inevitable reconfiguration costs. In our experiments, we are able to predict performance change with an average error of approximately 15%

The remainder of this paper is organized as follows. The next section gives an overview of our experimental setup. Section 3 details our work in establishing the importance of autonomous reconfiguration. Sections 4 and 5 deal with our results in learning to predict the best configuration and the change in performance, respectively. Section 6 contains

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

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

[2]The remaining two workloads have statistically significant, but smaller, differences.

some discussion of our results and the impact of our work. Section 7 gives an overview of related work. Section 8 concludes.

## 2. Experimental setup

Large servers are now available that can be partitioned into one or more logical subsystems [12, 18, 24]. Each of these logical systems has 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.

However, this newfound flexibility brings with it new challenges. An allocation of resources to the subsystems that maximizes performance for one set of workloads may be suboptimal for other workloads. As a result, real-time reconfiguration may be needed to maximize performance under a variable workload.

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 can be found in section 2.1. The software products used are given in section 2.2. Finally, the hardware and simulation of subprocessor partitioning are explained in section 2.3.

### 2.1. TPC-W

The TPC-W Benchmark [23, 26] is a standardized benchmark put out by the Transaction Processing Performance Council. 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
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 [15].**

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. In order to give a single numeric result, results are normally measured in 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).

A commercially built, tested, and published system often has one main database server and many independant web servers. Additionally, they often have distinct web cache servers, image servers, and load balancers. For example, the current WIPS record holder [25] 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.

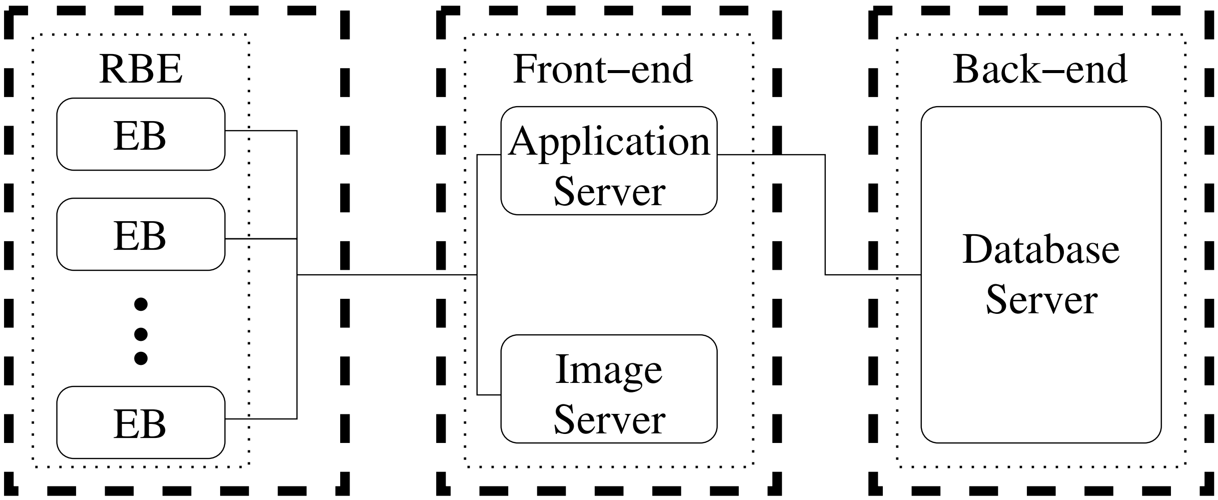


**Figure 1. The 3 machines used in the physical setup. The thick, dashed rectangles represent physical machines. The dotted rectangles are processes, and the innermost rounded rectangles are logical units. Network connections are shown as lines and come together at the point where they are managed; i.e., Tomcat 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 7.4.6 as the database server. The front-end uses Apache Jakarta Tomcat 5.5.4 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 [6]. This code implements both the servlets and a Java RBE, which is used to run the benchmark. Slight modifications were necessary to work with Tomcat and PostGreSQL [22]; additionally, the RBE was modified to retry any inabilities to connect to the front-end, rather than treating them as fatal errors.

### 2.3. Hardware

The physical setup of the system uses 3 identical Dell Precision 360n machines. Each machine has a 2.8 GHz procesor 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.

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 exmple, 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 configuratble 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.

By using both of these constraining processes simultaneously, simulation of any desired hardware configuration is possible. Additionally, the processes are designed to be reconfigurable on the fly, so we can simulate reconfiguring the system to give more memory to one machine by first constraining the other to the new requirement, and then unconstraining the newly available memory on the target machine. In this way, we never use more than a total of 2 GB memory. CPU reconfiguration is done similarly.

## 3. Adaptive configuration matters

At first appearance, it is not clear that adaptive configuration is necessary to maximize the throughput of a TPC-W system: it is possible that the best configuration is independent of workload. In order to verify that the best configuration *is* a function of workload, we identify two workloads and two configurations such that when running workload $x$, configuration **A** gives better results, but when running workload $y$, configuration **B** gives better results. This property is illustrated abstractly in Figure 2.

One way to achieve the dynamic resource allocation for workloads and configurations with the relationship shown in Figure 2 is to run both the web server and database in the same virtual machine and let the operating system manage the processor and memory. Such a system is unlikely to outperform our split system unless great care is taken with the CPU and memory allocation priorities of the database and web server processes. Commercial systems isolate their database on a single machine because database performance is sensitive to resource availability. When the OS takes resources from the database, it usually harms its performance, because it does not have sufficient information to choose just exactly the resources that will not harm performance. We plan to run an experiment to verify this effect as part of future work.

| | Configuration **A** | | Configuration **B** |
|---|---|---|---|
| workload $x$ | q | $>$ | r |
| workload $y$ | s | $<$ | t |

**Figure 2. q, r, s, and t are the WIPS results for the given configuration and workload. Notice that configuration A is better for workload $x$, while configuration B is better for workload $y$**

For all workloads in TPC-W, the database does more processing than the web server. Our initial experiments showed that the configurations which maximized throughput dedicated much of the CPU to the database back-end machine. Detailed experimentation on the shopping mix indicated that with about $\frac{7}{8}$ of the CPU (about 2.5 GHz) on the back-end, and the remaining $\frac{1}{8}$ on the front-end, the system is roughly in balance.

After high-level analysis of 15 workloads run on 15 configurations with roughly $\frac{7}{8}$ of the CPU on the back-end and various splits of memory, 5 likely workloads and 2 configurations were identified for further investigation.[3] The first configuration had $\frac{27}{32}$ of the CPU and $\frac{3}{8}$ of the memory on the back-end (referred to as "$\frac{27}{32}$ CPU, $\frac{3}{8}$ Mem") and maximized the throughput of the workload with 200 EBs running the ordering mix, as well as workloads of 250 and 300 EBs running the shopping mix. The second configuration had $\frac{30}{32}$ of the CPU and $\frac{5}{8}$ of the memory on the back-end (referred to as "$\frac{30}{32}$ CPU, $\frac{5}{8}$ Mem"), and led to higher performance for workloads of 350 and 400 EBs running the browsing mix.

In order to establish conclusively that configuration matters in this case, each configuration and workload was tested independently 53 times. To eliminate any interference between tests, the database and servlet engine were started before and shut down after each test, and the database files were copied over from originals. This eliminated any growth in the databases or servlet persistence issues from having an impact on the results.

Analysis of these data confirmed that there were significant differences between the configurations (see Table 2). A Student's t-test was used to analyze the results, showing that all differences between means are very statistically significant. The probability that the differences were due to random chance was less than $10^{-10}$ in all cases. Particularly of note are the large differences in the workloads with 250 and 300 EBs running the shopping mix (both over 1.25), as well as the difference in the workload with 400 EBs running the browsing mix (over 0.80). These large differences conclusively confirm that neither considered configuration is optimal in all situations. This result establishes the need for adaptive system configuration in order to take advantage of the optimal allocation of resources for the current workload.

## 4. Learning the best configuration

The results reported in Section 3 suggest that the system can improve its performance if it is able to adapt its configuration as the workload changes. However, in practice, the workload is not an observable quantity to the system. This section presents results indicating that the optimal configuration can be determined from *low-level operating system statistics* with no customized instrumentation. The lack of instrumentation allows this approach to work for any TPC-W implementation, regardless of the particular software used to implement the database, web server, etc.

In addition to collecting WIPS results during experiments, the individual front-end and back-end machines also collect low-level system statistics. These statistics are collected in parallel on both machines through the *vmstat* com-

[3] We were not able to run all 225 pairings sufficiently many times to establish statistically significant results, as each complete run took about 100 hours. The high-level analysis was done on data from 1 complete run and approximately 5 partial runs.

| configuration | Workload | | | | |
|---|---|---|---|---|---|
| | 200 ordering | 250 shopping | 300 shopping | 350 browsing | 400 browsing |
| $\frac{27}{32}$ CPU, $\frac{3}{8}$ Mem | **19.61(0.39)** | **16.30(0.22)** | **16.46(0.26)** | 12.73(0.27) | 12.22(0.51) |
| $\frac{30}{32}$ CPU, $\frac{5}{8}$ Mem | 18.79(0.35) | 14.96(0.12) | 14.86(0.21) | **13.10(0.22)** | **13.10(0.25)** |

**Table 2. Mean WIPS for chosen configurations over a variety of workloads. The better configuration is in bold. All tests involved 53 runs; standard deviations are in parentheses.**

| | | |
|---|---|---|
| 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 3. Statistics reported by** *vmstat*

mand, a commonly available system activity reporting utility (see Table 3). In order to determine the currently optimal configuration, we aim to create a model mapping current system state, as represented by *vmstat*, to the optimal configuration. Using the experiments reported in Table 2 as training data, standard machine learning methods can be used to learn such a model.

The WEKA [29] package implements many machine learning algorithms for exactly this purpose. In order to obtain human-understandable output, the JRip [11] rule learner was applied to the training data. As a baseline for analyzing the learned rules, accuracy can be compared to the model that always predicts the most likely outcome, in this case the $\frac{27}{32}$ CPU, $\frac{3}{8}$ Mem configuration (optimal for 3 of the 5 workloads in Table 2). The accuracy of this baseline learner is 61.9%.[4] By comparison, JRip learned the rules shown in Figure 3, yielding a prediction accuracy of 93.0%. The evaluation of JRip's rules was performed 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.

As desired, the rules learned by JRip are human-interpretable. The first 4 rules indicate cases in which the $\frac{30}{32}$ CPU, $\frac{5}{8}$ Mem case is the preferred configuration, while the last rule classifies all remaining cases as being in the $\frac{27}{32}$ CPU, $\frac{3}{8}$ Mem case. These four rules can be divided into two sets: the first two rules help identify situations where the front-end system has excess CPU available that the back-end could use, while the second two rules determine that the back end is over-utilized.

Of the first two rules, the first rule indicates a situation where the front system appears to be being underutilized. WEKA finds thresholds for the number of system interrupts taken by the front-end and how many blocks it is sending to assorted block devices (most likely the network sockets). If the front end falls under both thresholds, the back-end is the bottleneck. The second rule chooses a different method for determining if the front end is under-loaded; in this case it is the number of context switches per second. However, it also uses a threshold on the back-end machine to determine that there are more processes runnable than can be handled with the current configuration and that the additional CPU would be helpful.

The third rule indicates that the back end is receiving blocks (both from the network and the disk) at a fast enough rate that it would benefit from more CPU and memory. Finally, the fourth rule indicates that the CPU is spending very little time handling kernel-level work. This rule is a little odd. However, WEKA is likely determining that the back-end does not have enough CPU to handle the necessary system-space work, and is trying to handle too many things in user-space code simultaneously.

Notice that the above rules yield accurate prediction based entirely on low-level system data that is readily available independent of the system's software components. This feature of our approach allows it to generalize to a wide variety of scenarios, including to different implementations of TPC-W, and potentially to more varied distributed computer system scenarios.

Cross-validation as implemented by WEKA takes care to test prediction power on independent hold-out sets. However, it does not take care to keep data from the same workload out of the training and test sets. We plan future experiments to test the generalization power to completely new workloads.

## 5. Learning the benefit of switching configurations

While it is useful to predict the best configuration, even more useful is an ability to predict the actual benefit

[4] This baseline accuracy ought to be exactly 60% if all occurrences of each workload conformed exactly to the expected optimal configuration. Due to occasional outliers in the WIPS reported for a given configuration and workload, there were a few situations where the expected outcome was reversed.

1. **If** (Number of front-end system interrupts $\leq$ 1392.8)
   **and** (Number of blocks sent by the front-end to devices $\leq$ 201.2)
   **then** Best configuration=$\frac{30}{32}$ CPU, $\frac{5}{8}$ Mem
2. **If** (Number of front-end context switches $\leq$ 422.0)
   **and** (Number of runnable processes on the back-end $\geq$ 18.4)
   **then** Best configuration=$\frac{30}{32}$ CPU, $\frac{5}{8}$ Mem
3. **If** (Number of blocks received by the front-end from devices $\geq$ 499.5)
   **then** Best configuration=$\frac{30}{32}$ CPU, $\frac{5}{8}$ Mem
4. **If** (Percentage of CPU time spent by the back-end in the kernel $\leq$ 12.4%)
   **then** Best configuration=$\frac{30}{32}$ CPU, $\frac{5}{8}$ Mem
5. **else** Best configuration=$\frac{27}{32}$ CPU, $\frac{3}{8}$ Mem

**Figure 3. JRip rules learned by WEKA.**

of switching configurations in terms of increased (or decreased) WIPS. This ability is important as there will always be some cost involved in switching configurations. This cost needs to be taken into account when considering a configuration change.

For example, consider moving a CPU from one logical machine to another. Even when this move can be done without impacting the actual current work, the CPU will still be unavailable for a period of time. A similar problem applies to moving memory between machines. In a worst-case scenario, the machine may be temporarily unavailable while the system reallocates the hardware.

The WEKA package includes an algorithm for M5P model trees [28], which are able to learn function approximations. This algorithm was used to predict the changes in WIPS when changing configuration. As before, the learner only had access to the averaged raw system-level data from both systems and the current configuration. The change in WIPS was defined to be 0 for a change to the current configuration. Accuracy of the M5P trees was compared to a baseline learner that always predicted a constant change in WIPS equal to the overall average.

The complete M5P trees are too complex to display. However, the relative errors of the two methods can be seen in Table 4. The mean absolute error is the average of the absolute values of the prediction errors, and the root mean squared error is the square root of the average of the squares of the prediction errors. These results were determined using 10-fold cross validation, as before.

There is a sizeable improvement over the baseline learner; this indicates that the change in WIPS is a learnable function, without the need for internal instrumentation in the middleware. Although the differences that we are trying to predict are fairly small (around 1 WIPS), the accuracy given by the M5P trees is not only good enough that it can be used to predict if a configuration switch would help, but is even smaller than most of the standard deviations associated with the training data, implying that the error in the WIPS prediction could easily be masked by normal noise.

| Target | Learner | Mean abs. | Root mean sq. |
|---|---|---|---|
| $\frac{27}{32}$ CPU, $\frac{3}{8}$ Mem | Baseline | 0.61 | 0.78 |
| | M5P | 0.18 | 0.37 |
| $\frac{30}{32}$ CPU, $\frac{5}{8}$ Mem | Baseline | 0.61 | 0.78 |
| | M5P | 0.18 | 0.49 |

**Table 4. Error results of predicting the change in WIPS from one configuration to another.**

## 6. Discussion

This work is a step toward building a self-configuring system. As indicated earlier, it is possible to use the CPU and memory constraining processes to simulate moving resources from one machine to the other. While an implementation of this approach on real configurable hardware is still part of our future work, it is possible to analyze its potential impact. For example, consider a workload that alternated evenly between 300 EBs running the shopping mix and 400 EBs running the browsing mix. The data in table 2 indicates that the $\frac{27}{32}$ CPU, $\frac{3}{8}$ Mem system would have an approximate throughput of 14.34 WIPS, while the $\frac{30}{32}$ CPU, $\frac{5}{8}$ Mem system would have an approximate throughput of 13.98 WIPS. By comparison, a system running adaptively, using the JRip learner, would have a throughput of 14.69 WIPS, if switching time is negligible. This computation assumes that 93.0% of the time, the winning WIPS value is attained, and 7.0% of the time, the losing WIPS value is attained. These numbers represent a 2.4% performance gain over the $\frac{27}{32}$ CPU, $\frac{3}{8}$ Mem system and a 5.1% gain over the $\frac{30}{32}$ CPU, $\frac{5}{8}$ Mem system. As the disparities between the configurations grow (if the differences were 5 WIPS instead of 1), this gain will also grow.

This work makes two main contributions. First, it demonstrates that there is a need for self-configuring systems. While some systems may have a clear optimal division of resources to maximize the throughput of the system, there are certain situations where this division is workload-dependent. Because of this reliance on the workload, the learners we have outlined here appear to have a real use in distributed systems.

Second, having learners able to predict the change in WIPS is a critical step toward a fully functional self-configuring system able to maximize performance on the TPC-W benchmark. Because the cost involved in changing configurations may vary, this prediction allows a threshold to be set that would control when enough benefit would result from a configuration change to overcome the temporary cost of moving resources around.

A key feature of this work is that no instrumentation of any code is necessary. All learning was done based upon easily-available raw system statistics. Because no middleware instrumentation is necessary, it is possible to change any or all parts of the system, including the front-end and back-end software, the TPC-W implementation, or even the operating system or physical hardware. While retraining would be necessary, no additional code modifications would be needed. Finally, the entire system could be replaced with another benchmark, such as Sun's PetStore [21] web-based store simulation.

In contrast, most other adaptive systems appear to need instrumentation built into the applications and operating system; adaptive systems that work without any instrumentation seem to be rare. One other example of a system that works without instrumentation is Aguilera et al. [3], who investigated locating paths through a large distributed system by examining only the RPC headers of network packets. These paths could then be analyzed to identify high latency nodes.

As noted earlier, the differences between the results of the winning and losing configurations are fairly small (about 1 WIPS). Two design decisions made in the original PHARM code hamper its ability to get high performance out of a system and lead to these small disparities in the current implementation. First, the specification for the TPC-W benchmark defines a maximum length of time to keep certain pages cached. This bound is designed to force periodic updates of common pages, such as the "Best sellers" page. In order to avoid handling this scenario, the code always searches the database, rather than returning a previously generated copy. These searches place an undue amount of workload on the database, particularly for the browsing and shopping mixes.

The second important design decision involves the code that handles the processing of an order. The code, as written, does not allow for the database to assign a unique identifier to each new record and implements its own unique identifier algorithm. It therefore serializes all code that writes to the database using the Java *synchronized* keyword. This serialization prevents the database from being properly loaded (or overloaded) when running the ordering mix. These two properties of PHARM contributed to the difficulty of locating the breaking point for the CPU. They also make some results counterintuitive.

For example, the ordering mix is designed to be far more back-end intensive than the the browsing mix. However, because the browsing mix constantly accesses the database in parallel to retrieve records, while the front-end serializes many ordering accesses to the database in the ordering mix, the back-end is stressed much more in the browsing mixes. This property explains why the $\frac{30}{32}$ CPU, $\frac{5}{8}$ Mem configuration wins on the two browsing workloads, when intuition says that that is where it should lose. Modifying the PHARM code to eliminate these two problems is an area of future work.

## 7. Related Work

The concept of adaptive performance tuning has only recently become conceivable, so few papers address it directly. This section reviews the most related work to that reported here.

Diao et al. [13] 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, and make no claim that the parameters impact the performance of the web server in a known way.

Gomez et al. [16] 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.

Other prior work addresses performance-tuning with the intention of maintaining a fixed level of service. 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. Additionally, Cohen et al. [10] use Tree-Augmented Naive Bayesian

Networks to correlate system statistics to a high-level performance metric (compliance or non-compliance with required service levels). Unlike our work, this work relies on a specialized instrumentation layer, a reliance that we actively and successfully avoid.

Additional work has also been done with offline analysis of a system. Aguilera et al. [3] worked on discovering the latency of individual nodes in a network. This work was done with no additional instrumentation to the system and was 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. [17] analyzed the performance of a system over large spans of time with statistical models, which could then determine online when unexpected changes occurred. Brown et al. [4] analyzed the dependency of each page in the TPC-W benchmark on each database table.

More broadly, this work falls within the emerging field of autonomic computing, which deals with self-managing, self-configuring, self-protecting, and/or self-healing systems. Brown et al. [5] have argued the need for a new benchmark to determine the performance of each of these categories, and put forward initial ideas as to how this could be accomplished. Other work within the field of autonomic computing focuses on failure diagnosis [8, 9], bug identification [19], file system organization [20], adaptive branch prediction [14], autonomous network creation [7], installation and configuration analysis [2] and utility function optimization [27].

## 8. Conclusion

The rapid development of reconfigurable servers indicates that they will become more commonly used. As this hardware is deployed, it may be used for distributed applications where isolating the front-end and back-ends is desirable, but where extra hardware is unneeded. In these cases, the server will need some form of adaptability to deal with changing workloads.

This paper has presented preliminary research into methods to handle variable workloads by dynamically reallocating hardware resources between the machines. In addition to showing that autonomous reconfiguration has the potential to improve performance, two learners were presented that predict preferable configuration changes with high accuracy. Our ongoing research agenda includes fully automating the adaptive process under dynamically changing workloads, first on our simulated reconfigurable machines and eventually on true reconfigurable hardware.

## Acknowledgments

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] G. Aggarwal, M. Datar, N. Mishra, and R. Motwani. On identifying stable ways to configure systems. In *Proceedings of the 1st International Conference on Autonomic Computing*, May 2004.

[3] 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.

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

[5] A. B. Brown, J. Hellerstein, M. Hogstrom, T. Lau, S. Lightstone, P. Shum, and M. P. Yost. Benchmarking autonomic capabilities: Promises and pitfalls. In *The 1st International Conference on Autonomic Computing*, May 2004.

[6] H. W. Cain, R. Rajwar, M. Marden, and M. H. Lipasti. An architectural evaluation of Java TPC-W. In *The 7th International Symposium on High-Performance Computer Architecture*, January 2001. Code available at `http://www.ece.wisc.edu/~pharm/tpcw.shtml`.

[7] Y.-H. Chang, T. Ho, and L. P. Kaelbling. Mobilized ad-hoc networks: A reinforcement learning approach. In *Proceedings of the 1st International Conference on Autonomic Computing*, May 2004.

[8] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer. Failure diagnosis using decision trees. In *Proceedings of the 1st International Conference on Autonomic Computing*, May 2004.

[9] M. Y. Chen, E. Kıcıman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of 2002 International Performance and Dependability Symposium*, Washington, DC, June 2002.

[10] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *6th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

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

[12] K. DeLira, A. Garcia, R. Hendrickson, L. Macedo, and R. Patel. LPAR heterogeneous workloads on the IBM®eServer pSeries™ 690 system. International Business Machines Corporation, February 2002.

http://www.redbooks.ibm.com/redpapers/pdfs/redp0425.pdf.

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

[14] A. Fern, R. Givan, B. Falsafi, and T. N. Vijaykumar. Dynamic feature selection for hardware prediction, 2004. http://web.engr.oregonstate.edu/~afern/papers/jsa-submission.pdf.

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

[16] 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.

[17] 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.

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

[19] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Programming Languages Design and Implementation (PLDI)*, June 2003.

[20] M. Mesnier, E. Thereska, G. R. Ganger, D. Ellard, and M. Seltzer. File classification in self-* storage systems. In *Proceedings of the 1st International Conference on Autonomic Computing*, May 2004.

[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] W. D. Smith. TPC-W: Benchmarking an ecommerce solution. Technical report, Intel Corporation, 2000. http://www.tpc.org/tpcw/TPC-W_Wh.pdf.

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

[25] 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.

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

[27] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das. Utility functions in autonomic systems. In *Proceedings of the 1st International Conference on Autonomic Computing*, May 2004.

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

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