Automated Cluster-Based Web Service Performance Tuning

I-Hsin Chung, and Jeffrey K. Hollingsworth
{ihchung, hollings}@cs.umd.edu

Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract - **In this paper, we apply the Active Harmony system to improve the performance of a cluster-based web service system. The performance improvement cannot easily be achieved by tuning individual components for such a system. The experimental results show that there is no single configuration for the system that performs well for all kinds of workloads. By tuning the parameters, the Active Harmony helps the system adapt to different workloads and improve the performance up to 16%. For scalability, we demonstrate how to reduce the time when tuning a large system with many tunable parameters. Finally an algorithm is proposed to automatically adjust the structure of cluster-based web systems, and the system throughput is improved up to 70% using this technology.**

I. INTRODUCTION

Online e-commerce sites are one of the main applications on the Internet today. They are used as a standard mechanism for online information distribution and exchange. In order to provide such service, e-commerce sites require large online web systems. Such systems must capable of running continuously and reliably 7 days a week, 24 hours a day. Besides, the systems must be able to accommodate widely varying service demands. They should be adaptive when the number or nature of requests changes. And finally, the systems should be cost-effective.

Clusters of commodity workstations interconnected by a high-speed network are frequently used to meet these challenges. The infrastructure can tolerate partial failures and allows scaling up by adding more components. The administration mechanism for such a large cluster does not have to be reinvented for each new service.

When these systems are designed and built, the developers usually can only imagine how they will be deployed and used based on their knowledge and experience. Besides, in order to make the system accommodate all kinds of possible environments, they tend to set the default configuration of the system (e.g., number of processes forked, memory size allocated) conservatively (i.e., appropriate values but not well tuned). Therefore, the customer environment may not be fully utilized and thus the performance for such a system may be improved if its configuration is “tuned” appropriately.

The Active Harmony system is designed to help systems become adaptive to their execution environment as well as to changes in workload. It changes the configuration of the system being tuned based on the performance monitored. By improving the performance iteratively, the Active Harmony system changes performance optimization from post-mortem style to real-time steering.

This paper differs from our previous work [10, 12, 20] in that we apply the Active Harmony to a coupled application. An e-commerce system contains multiple components (web server, application server, and database). Such a large-scale system cannot be tuned for each individual component. In this paper we show that Active Harmony is not only useful to improve the performance, but it is necessary to have such a tuning mechanism since there is no single best configuration for all kinds of workloads. One major difficulty when tuning such a system with numerous parameters is scalability. We propose *parameter replication* and *parameter partitioning* to solve this problem. This helps to speed up the tuning process.

The structure of this paper is organized as follows: Section two gives an overview of cluster-based web service systems, the Active Harmony system, and the TPC-W benchmark that we used as the performance metric in the experiments. Section three shows the tuning mechanisms and results. Section four demonstrates how to improve system performance with automatic reconfiguration. Section five discusses the challenges encountered in harmonizing an e-commerce system. Related work is given in the Section six and the Section seven concludes the paper.

II. SYSTEM

A cluster-based web service system consists of a collection of machines. The machines are separated into sets. Each set (or tier) of machines is focused on serving different parts of a request. The incoming requests are handled in a pipeline fashion by different tiers. In this project, we try to improve the overall system performance by automatic tuning across all tiers using the Active Harmony system. The performance metric we are focusing on is the TPC-W benchmark. It is a transactional web benchmark designed to emulate operations of an e-commerce site.

A. *Web Cluster*

In many web services today, there are (conceptually, at least) three tiers: presentation, middleware, and database. The presentation tier is the web server that provides the interface to the client. The middleware tier is what sits between the web server and the database. It receives requests for data from the web server, manipulates the data and queries the database. Then it generates results using existing data together with answers from database. Those results are presented to the client through the presentation tier. The third tier is the database, which holds the information accessible via the Web. It is the backend that provides reliable data storage and transaction semantics.

A scenario for such an architecture is that a user fills out a form on the web browser; the web server receives the request and passes the information to the middleware. The middleware translates the information into appropriate SQL and queries the database. It then takes the data from the database (does some manipulation or calculation if necessary) and turns the results into HTML pages. These pages are then sent back to the web server, which in turn serves them out to the web browser.

For small applications, it is possible to have all the three tiers on the same machine. However, this assignment is not feasible for configurations with high traffic. To increase performance, flexibility, and scalability, dedicated machines for different functionality are generally used. In addition, multiple machines can be used at each tier to increase throughput. Figure 1 illustrates a typical three-tier infrastructure.

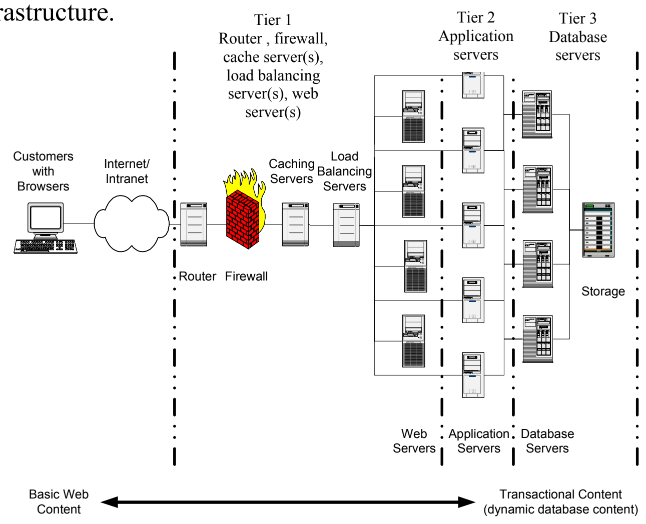


Figure 1: Multi-tier Web Architecture

Functionality by each Tier:

- Requests handled by the Tier 1:
  All the cacheable and static data are handled by the Tier 1. For example, the customer browses the company information or product specification sheets. This information is all handled by the Tier 1.
- Requests handled by the Tier 1 & 2:
  The server side applications are in the Tier 2. For example CGI or Java Servlet programs are in this tier. A customer may interact with the Web server to customize his or her merchandise. The data is received by the Tier 1 and then passed to the Tier 2. The interaction is then handled by the server side applications and then returned through the Tier 1.
- Requests handled by the Tier 1, 2 & 3:
  While the Tier 2 interacts with the customer, it may need to communicate with Tier 3, the database, for information about pricing, configuration parameters, transaction processing information, etc. After a customer placing an order, the Tier 2 first queries the price information from the Tier 3. Then it process the transaction based on the query results. Finally the receipt is presented back to the customer through the Tier 1.

An advantage of this structure is each machine can be optimized for its job. For example, the application server can be optimized for computation and the database server for I/O.

In order to optimize each machine with respect to its functionality, both the hardware and the software have to be tuned. In most systems today, software configuration tuning is done by either experienced system administrators or from the default configurations set by the system developers. The default configurations are set based on a general expectation of the environment on which the system to be executed. Those configurations will make the system work in most of environments but the performance may vary dramatically due to the difference in each customer's environment.

With automatic performance tuning, the system will be able to adapt itself to the execution environment. The adaptation includes changing the configuration on each machine and the number of servers in each tier. Section three and four describe how to do the tuning and the server reconfiguration.

B. *Active Harmony*

To provide automatic performance tuning, we developed the Active Harmony system [10, 12, 20]. Active Harmony is an infrastructure that allows applications to become tunable by applying very minimal changes to the application and library source code. This adaptability provides applications with a way to improve performance during a single execution based on the observed performance. The types of things that can be tuned at runtime range from parameters such as the size of a read-ahead parameter to what algorithm is being used (e.g., heap sort vs. quick-sort).

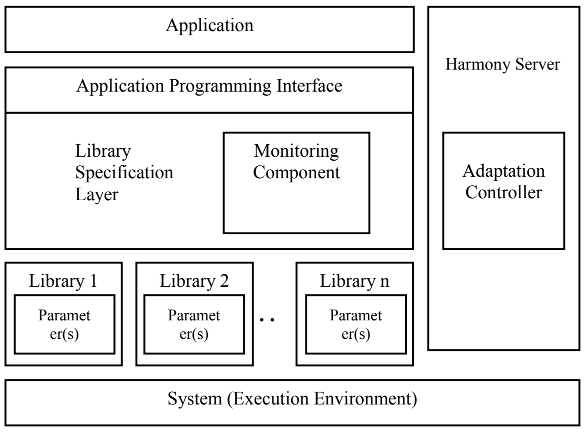


Figure 2: Active Harmony automated tuning system

Figure 2 shows the Active Harmony automated runtime tuning system. The Library Specification Layer provides a uniform API to library users by integrating different libraries with the same or similar functionality. This layer uses the Harmony Controller to select among different implementations of the library. The library specification layer also monitors the performance of the library to improve the decision for future usage of the program library.

The Adaptation Controller is the main part of the Harmony server. The Adaptability component manages the values of the different tunable parameters provided by the applications and changes them for better performance. The Adaptation Controller is written in Tcl since it can be easily adapted to the Active Harmony system requirements.

The kernel of the adaptation controller is a tuning algorithm. The algorithm is based on the simplex method for a finding a function's minimum value [14]. In the Active Harmony system, we treat each tunable parameter as a variable in an independent dimension. The algorithm makes use of a simplex, which is a geometrical figure defined by k+1 connected points in a k-dimensions space. In 2-dimensions, the simplex is a triangle, and for the 3-d space the simplex is a non-degenerated tetrahedron.

The Nelder-Mead simplex method approximates the extreme of a function by considering the worst point of the simplex and forming its symmetrical image through the center of the opposite (hyper) face. At each step a better point replaces the worst points and thus moves the simplex towards the extreme. In our case the algorithm slips down the valley towards the minimum. The concept of the simplex method is illustrated in Figure 3. The example is to search a minimum point in a three dimensional space. At the beginning of a step, there are four points: three points with low value are around the shadowed triangle and the point with high value is at left bottom corner of the pyramid as shown in Figure 3(a). Based on this given performance result, the possible points will be explored by the tuning algorithm will be i) a reflection point, ii) a contraction points, and iii) a multiple contraction point as shown in Figure 3(b).

The algorithm described above assumes a well-defined function and works in a continuous space. However, neither of these assumptions holds in our situation. Thus we have adapted the algorithm by simply using the resulting values from the nearest integer point in the space to approximate the performance at the selected point in the continuous space.

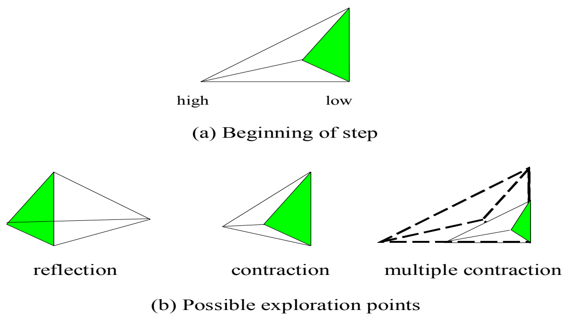


Figure 3: Possible outcomes for a simplex method step

C. *TPC-W Benchmark*

The major performance metric we use when tuning the cluster-based web service is the TPC-W benchmark. The TPC-W is a transactional web benchmark designed to mimic operations of an e-commerce site. The workload explores a breadth of system components together with the execution environment. Like all other TPC benchmarks, the TPC-W benchmark specification is a written document which defines how to setup, execute, and document a TPC-W benchmark run.

TABLE 1: TPC-W BENCHMARK WORKLOADS

Web Interaction	Browsing (WIPSb)	Shopping (WIPS)	Ordering (WIPSo)
Browse	**95 %**	**80 %**	**50 %**
Home	29.00 %	16.00 %	9.12 %
New Products	11.00 %	5.00 %	0.46 %
Best Sellers	11.00 %	5.00 %	0.46 %
Product Detail	21.00 %	17.00 %	12.35 %
Search Request	12.00 %	20.00 %	14.53 %
Search Results	11.00 %	17.00 %	13.08 %
Order	**5 %**	**20 %**	**50 %**
Shopping Cart	2.00 %	11.60 %	13.53 %
Customer Registration	0.82 %	3.00 %	12.86 %
Buy Request	0.75 %	2.60 %	12.73 %
Buy Confirm	0.69 %	1.20 %	10.18 %
Order Inquiry	0.30 %	0.75 %	0.25 %
Order Display	0.25 %	0.66 %	0.22 %
Admin Request	0.10 %	0.10 %	0.12 %
Admin Confirm	0.09 %	0.09 %	0.11 %

The two primary performance metrics of the TPC-W benchmark are the number of Web Interaction Per Second (WIPS), and a price performance metric defined as Dollars/WIPS. However, some shopping applications attract users primarily interested in browsing, while others attract those planning to purchase. Two secondary metrics are defined to provide insight as to how a particular system will perform under these conditions. WIPSb is used to refer to the average number of Web Interaction Per Second completed during the Browsing Interval. WIPSo is used to refer to the average number of Web Interaction Per Second completed during the Ordering Interval.

The TPC-W workload is made up of a set of web interactions. Different workloads assign different relative weights to each of the web interactions based on the scenario. In general, these web interactions can be classified as either "Browse" or "Order" depending on whether they involve browsing and searching on the site or whether they play an explicit role in the ordering process. The details for each workload breakdown are shown in the Table 1.

D. *Environment*

The summary of the environment used for our experiment is shown in Table 2. The 10 machines used include the ones running emulated browsers and the servers for proxy, application and database services. Each machine is equipped with dual processors, 1 Gbyte memory and runs Linux as the operating system. For each tier, we select Squid as the proxy server, Tomcat as the application server and MySQL as the database server. All computer software components are open-source which allows us to look at source code to understand system performance. The TPC-W benchmark scale factor is 10,000 items. In other words, the number of the items that the store sells in the experiment is approximately 10,000.

TABLE 2: EXPERIMENT ENVIRONMENT

Hardware	
Processor	Dual AMD Athlon 1.67 GHz
Memory	1Gbyte
Network	100Mbps Ethernet
No. of machines	10
Software	
Operating System	Linux 2.4.18smp
TPC-W benchmark	Modified from the PHARM [7]
Proxy Server	Squid 2.5 [4]
Application Server	Tomcat 4.0.4 [1]
Database Server	MySQL 3.23.51 [3]

III. TUNING

Our goal is to improve the overall system performance using Active Harmony. We first show that there is no single configuration suitable for all the workloads. Active Harmony makes the system perform better by using different configurations when facing different workloads. Then we investigate Active Harmony's scalability as the number of machines grows. One way to solve this problem is to partition the parameters into sets. We show how to use an independent Active Harmony tuning server for each set to speed up the tuning process. Another method is to tune a representative set of parameters and use duplicated values on the rest of nodes. In addition to tuning parameters in each node, we also show how to adjust the number of nodes in each tier dynamically to reduce the hot spot inside the large-scale system in the Section four.

A. Impact of Varying Workload

In this experiment we show that the Active Harmony server can tune the system to adjust each tier's server to provide good performance. We use four machines in this experiment: one machine for the emulated browsers, one for the proxy server, one for the application server, and one for the database server.

In the experiment, we examine the tuning processes for two different workloads: browsing and ordering. Both tuning processes are started using the default configuration. We then let the system warm up for 100 seconds and measure the performance (WIPS) for 1000 seconds followed by 100 seconds for cooling down. We define such a cycle as one "iteration". The Active Harmony server will adjust the configuration (parameters values) between two iterations.

The tuning process with a browsing workload shows that the default configuration is not suitable for the system. The main reason is due to the characteristics of the browsing workload – the system components utilized by the requests in this workload are changing dramatically. Some emulated browsers browse web pages that consist solely of static data that can come directly from the proxy server or the application server without generating jobs to the database server. While some other browsers visit web pages that contain dynamic data such as product price. Dynamic data is gathered from the database server, processed by the application server and then sent back to the browser. However, even with the dynamically changing requests in the browsing workload, the Active Harmony tuning server is still able to improve the overall system performance. For the second 100 iterations, the average improvement is 3% and the performance of 78% of the iterations is better than it is in the default configuration. The overall performance improvement is 15%.

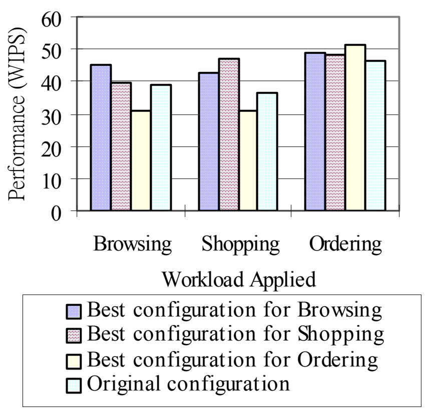
We observed there is higher variation in system throughput in a browsing workload. We believe this is because the tuning server sometimes uses a configuration that consists of parameters with extreme values. The extreme values are at the limit boundaries that can be assigned to parameters. From experience we know that the system often performs poorly when using a configuration with extreme values. In the future, we plan to modify the kernel of the Active Harmony tuning algorithm so it will avoid jumping to extreme values, but instead slowly approach them only when performance gains warrant it.

When the system is processing a predominately ordering workload, the results show that the performance for the default configuration is pretty good and thus the improvement after tuning is relatively limited. Unlike the browsing workload, most of the requests for ordering workload utilize all components in the system, including the database server. The characteristics of the requests do not change dramatically during execution. Therefore, it is easier to tune the system with the ordering workload. For the second 100 iterations, the performance of 85% iterations is better than it is of the default configuration. The performance improvement is only up to 5%.

Figure 4 shows that for different workloads, the system should apply different configurations. Each different bar represents the best configurations we determined after 200 tuning iterations for each of the workloads. We then apply those best configurations to the other two workloads for comparison. The results show that when using a configuration that is tuned for another workload, the system does not perform as well as using a configuration that is tuned for the current workload. The results show that there is no universal configuration good for all kinds of workloads. The table in Figure 4 shows the improvements for those best-tuned configurations compared to the default configuration. The improvements range from 5% to 16%.

Table 3 shows the details of all Harmony tunable parameters before, and after tuning for each of the workloads.

The results show for the proxy server, it first increases the main memory size for the cache to improve the performance. For the shopping and working workloads, the proxy server tries to cache larger objects in the memory compared to the browsing workload. For the HTTP server (which is part of the application server), the tuning results show that it spawns more threads to handle the requests during the ordering workload. We believe the main reason is that most of the requests in the ordering workload require high latency operations in the database server (i.e., performing update transactions on the database). Thus the average response time is longer compared to other workloads. As long as it is not over the system capacity, the HTTP server should use more threads (minProcessors/maxProcessors) and buffer space (bufferSize) to handle the incoming requests. The waiting queue capacity should also increase accordingly (acceptCount) as the results show. The same situation happens in the worker part (AJP connector) of the application server. For the database server, the tuning results show it increases the cache and buffer size when the utilization for the database is high (i.e., shopping and ordering workloads). However, it shows that reducing the join buffer size does not impact performance.



	Best configuration after 200 iterations		
	Browsing	Shopping	Ordering
Improvement compared to the default configuration	15%	16%	5%

Figure 4: Applying best configuration after 200 iterations to different workloads

From the results we can see that some parameters significantly affect the overall system performance such as the number of threads or the buffer size. However, there are some parameters that we thought to be performance related but they turn out not to be important. For example, the thresholds (cache_swap_low, cache_swap_high) which control whether the proxy server should swap out objects do not impact the overall system performance. Since it is automated, the Active Harmony tuning process is also helpful for system administrators and developers to identify those parameters that actually affect system performance.

Figure 5 shows the tuning system's responsiveness to the changing workloads. The system is started with the default configurations for all the servers. We change the workload every 100 iterations. As shown in the Figure, the response time it takes for the system to adjust itself when the workload changes, is fairly short. Only a few iterations are needed to adapt to the new workload. The Active Harmony tuning server not only helps the system react to the changing workload, it also makes the adjustments fairly quickly. This is helpful when the system is facing real-world traffic that can change at a rate faster than a person could hand tune the system.

TABLE 3: TUNING RESULTS FOR DIFFERENT WORKLOADS

Tunable parameters	Default config.	Best configuration after 200 iterations		
		Browsing	Shopping	Ordering
Proxy Server				
cache_mem	8	13	17	21
cache_swap_low	90	91	86	91
cache_swap_high	95	96	96	96
maximum_object_size	4,096	4,096	4,096	5,888
minimum_object_size	0	0	50	306
maximum_object _size_in_memory	8	6	256	2,560
store_objects_per _bucket	20	15	25	105
Web Server				
minProcessors	5	1	16	102
maxProcessors	20	11	16	131
acceptCount	10	6	21	136
bufferSize	2,048	2,049	3,585	6,657
AJPminProcessors	5	6	26	136
AJPmaxProcessors	20	86	296	161
AJPacceptCount	10	76	306	671
Database Server				
binlog_cache_size	32,768	63,488	153,600	284,672
Delayed_insert_limit	100	200	400	700
max_connections	100	201	451	701
delayed_queue_size	1000	2,600	9,100	7,100
join_buffer_size	8,388,600	407,552	407,552	407,552
Net_buffer_length	16,384	31,744	38,912	34,816
table_cache	64	873	905	761
thread_con	10	81	91	76
thread_stack	65,535	102,400	1,018,880	773,120

B. *Cluster Tuning*

When the number of servers increases, the number of the tunable parameters also increases accordingly. This makes the tuning process lengthy and the tuning results may not be useful since the environment could change during the tuning process. Therefore, an important question is how to make the tuning process scalable.

In the current Active Harmony system to tune *n* parameters at once requires exploring *n+1* configurations before improvements to the system will take effect. If there are numerous servers in the cluster and each server contains tens of parameters, the tuning process will be fairly long. In order to reduce the initial exploration period, we try to partition the components inside the cluster into groups based on the execution environment and use separate Active Harmony tuning servers for each of the groups. We compared different tuning configurations in different situations with the default infrastructure which use only single Active Harmony tuning server in this section.

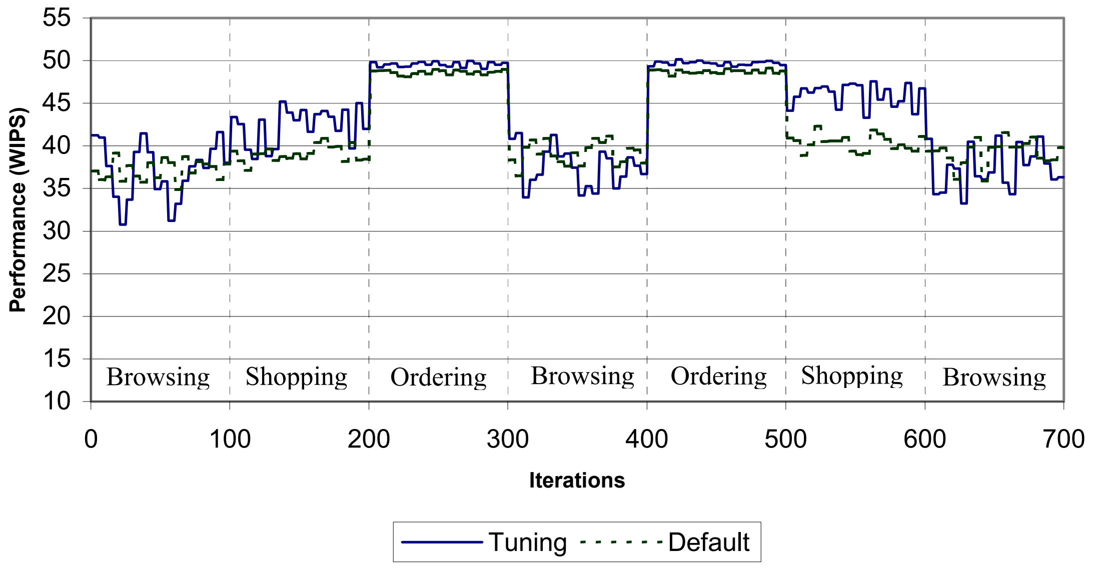


Figure 5: Tuning responsiveness to the changing workloads

When all the machines in the same tier are homogeneous, we try to partition all the servers into tuning groups using two methods. The first one is parameter duplication: we only tune one server for each tier, and the values for those parameters are duplicated to other servers in the same tier. This tuning mechanism is based on the assumptions that (a) servers in the same tier are running on the same execution environment, and thus will have the same or similar behavior for the same configuration; (b) the workload are evenly distributed among all the servers in the same tier.

The second method, parameter partitioning, is based on a work line. Each work line group consists of at least one server from each tier. A request to the web cluster system is only handled exactly by one work line group. In other words, any server in work line group A will not generate (serve) requests to (from) a server in work line group B. We use a different Active Harmony tuning server to tune the parameters for each work line. The assumption for this tuning mechanism is that (a) all the work lines are running in parallel; and (b) there is no interaction between any two of the work lines.

To compare these two approaches, we tuned the system using three different tuning methods: default, parameter duplication and parameter partitioning. Compared to the default tuning method (using a single Active Harmony server for all parameters in all servers), tuning using parameter duplication speedups the tuning process. Table 4 shows the best configuration after 200 iterations and the standard deviation for the second 100 iterations.

We then tuned the system using the parameter partitioning method. This method is more stable compared to the other methods. The second 100 iterations come with an average of 116 WIPS with a standard deviation 9.7 (where the default tuning method comes with an average of 110 WIPS with standard deviation 30). This is because this method has more detailed performance results for each of the parameters. The detailed performance results for each group help the dedicated tuning server to do a better job. In addition, the impact when changing one parameter in a tuning group is only limited to the work line performance result of the group and will not affect the performance result of other groups. This helps to make the whole tuning process more stable.

Table 4 shows the tuning results. The tuning results for all three methods are very close. The default method takes the longest time since there are many parameters and only one performance result per iteration. The parameter duplication method provides both a larger performance improvement and faster convergence to the tuned configuration. It speeds up the tuning process since the tunable parameters are distributed to multiple tuning servers and there are fewer parameters for each tuning server to tune. The time (iterations) spent for the grouping by parameter partitioning method is about 2/3 of the default method.

TABLE 4: PERFORMANCE FOR DIFFERENT METHODS FOR CLUSTER TUNING

Tuning method	WIPS[1]	Standard deviation[2]	Performance improvement	Iterations
None (No Tuning)	110.4	2.1	-	-
Default method	130.6	30.0	18.3%	159
Parameter duplication	133.7	29.5	21.2%	33
Parameter partitioning	131.3	9.7	19.0%	107

[1] Performance for the best configuration after 200 iterations
[2] For the second 100 iterations

Based on the time for the tuning process, parameter duplication tuning seems to be the best. It takes a much shorter time for tuning. However, if stable performance during tuning process is critical, parameter partitioning by work lines is a reasonable choice.

In the future, we plan to investigate the possibility to have the hybrid tuning. That is, using the parameter duplication method first, and then using separate tuning server for each group for fine-granularity tuning.

IV. Automatic Cluster Reconfiguration

One of the advantages for a cluster-based web service is the ability to reconfigure hardware easily. By dynamically changing the roles of servers for different workloads, it is possible to make the best of available resources.

The parameter tuning part of the Active Harmony system helps to tune the cluster-based web service at a fine time granularity. However, when the load is not balanced among tiers in the web service system, changing the parameters for all the servers will not provide much help to solve the problem. Instead, it is necessary to adjust the infrastructure by changing the number of servers in each tier dynamically to reduce the load imbalance.

The Active Harmony system applies a simple mechanism to achieve load balance among tiers. While the tuning is in progress, the Active Harmony system monitors the resource utilization for all nodes of all tiers. The resources that are monitored include CPU load, memory usage, network bandwidth used and disk I/O activity. Periodically, the Active Harmony detects whether (1) there is a resource on node A that is over utilized, (2) all the resources on node B are under utilized and node B is suitable reconfiguration. If both situation (1) and (2) exist, the Active Harmony tries to reconfigure node B to run the same server process as node A.

Unlike parameter tuning which is done for each iteration, the reconfiguration algorithm is run at a lower frequency (e.g., every 50 iterations) since it is designed to react to longer term trends, and incurs a greater overhead to make changes. Table 5 shows the definition for variables in the algorithm and Figure 6 shows the concept of the reconfiguration algorithm.

TABLE 5: Variable Description

Variable	Description
R_{ij}	Utilization of resource *j* on node *i*
LT_{ij}	Low threshold for resource *j* on node *i*
HT_{ij}	High threshold for resource *j* on node *i*
M_{pq}	Cost to move a job for node *p* to node *q*
A_i	Average process time on node *i*
F	Configuration cost in terms of time
L	List of nodes
N_i	Number of jobs on node *i*
Head(L)	First node in the List *L*
Tier(i)	The tier that node *i* belongs to
M(t)	Number of nodes in tier *t*

```
1. For all node $i$, resource $j$ do
      If $R_{ij} > HT_{ij}$ then add $i$ to the list $L_1$
      //find out what nodes are highly or over loaded
2. For all node $i$ do
      If $R_{ij} < LT_{ij}$ for all $j$ then add $i$ to the list $L_2$
      //find out what nodes are lightly loaded
3. Sort $L_1$ based on the "degree of urgency[3]"
      //decide the priority for the nodes to be relieved
4. Let $i = Head(L_1)$, find the node $k$ in $L_2$ such that satisfies (a)(b)(c)
      //find out the appropriate node to be reconfigured
      (a) $Tier(i) \neq Tier(k)$
      (b) $M(Tier(k)) > 1$
      (c) $F + N_k \times M_{km} - N_k \times A_k$ is minimal, where $k \neq m$
      and $Tier(k) = Tier(m)$
5. Reconfigure $k$ such that $Tier(i) = Tier(k)$
```

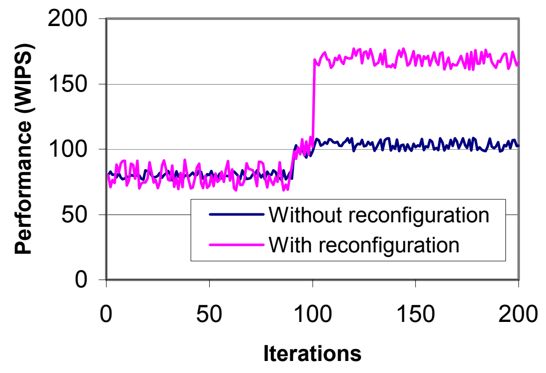
Figure 6: Reconfiguration algorithm for external tuning

Step 1 finds out what nodes are over loaded. It checks the resource utilization against the predefined high threshold. Step 2 tries to find nodes that are lightly loaded. If all the resources on the node are idling most of the time (i.e., utilization is smaller than the lower threshold), the node is considered under utilized. Step 3 finds out what is the most "urgent" node that should be relived first. Step 4 checks to ensure correct operation, there must be at least one node left in each tier, and decide if the reconfiguration should be done immediately (by moving existing requests to the neighbor nodes in the same tier) or if it should wait until all existing requests finish. Finally Step 5 does the reconfiguration.

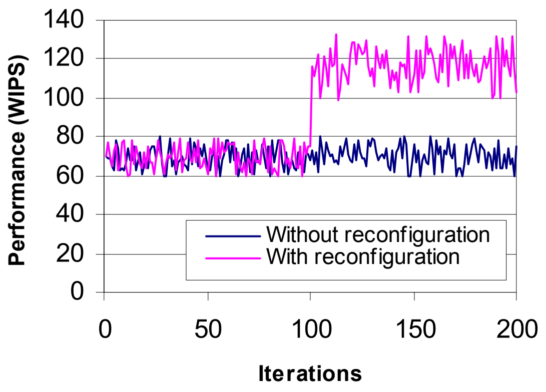
$$F + N_k \times M_{km} - N_k \times A_k \qquad (1)$$

When the result of the equation (1) for the selected node *k* in the Step 4(c) is non-negative, the Active Harmony system will not reconfigure the node *k* until all the jobs on it are finished. This is because it will be more cost-effective to wait than to reconfigure the node *k* immediately. On the other hand, when the result of the equation is non-positive, the Active Harmony system will reconfigure the node *k* immediately. This is because the cost for immediate reconfiguration will be less than waiting for the system to be idle to reconfigure.

[3] The degree of urgency for each node depends on the characteristics of the application. It may very from case to case. For example, over loading the CPU may cause bigger problem than utilizing all the network bandwidth for some applications. Therefore, nodes with over-loaded CPU will have higher priority than nodes whose network bandwidth is highly utilized.



(a) One node moved from the proxy server tier to the application server tier
(Workload changes from browsing to ordering)



(b) One node moved from the application server tier to the proxy server tier
(Browsing workload)
Figure 7: Reconfiguration experiment results

The Active Harmony can automatically perform node reconfiguration without taking the system down. While one node is being reconfigured from one tier to another, all the remaining nodes in the system are still serving requests normally. This helps the system to provide uninterrupted service.

Figure 7 shows the experimental results when applying the reconfiguration algorithm. The initial configuration for Figure 7(a) has four nodes serving the proxy tier and another two nodes for the application tier; all six nodes are homogeneous. The experiment starts with browsing workload and changes to ordering workload after the 90th iteration (The performance gains between 90th and 100th iterations are due to different workloads). We forced the Active Harmony system do the dynamic adjustment checking exactly once right after the 100th iteration of the tuning process. Figure 7(a) shows the performance improvement when the Active Harmony decides to move a node from the proxy server tier to the application server tier based on the algorithm. This is expected since when the system has a workload dominated by ordering, it requires more application servers to handle the dynamic data from the database. On the other hand, most browsing workload requires static data that can be served from the proxy servers. Before the adjustment, the application servers are highly loaded (CPU utilization is always close to 100%) and some proxy servers are idling most of the time (CPU utilization is close to 0% and very few network or disk I/O requests). After the adjustment, the average utilization of the application servers is lowered while the average loading for the proxy servers increases a little. The bottleneck of the whole system is relived and the system performance is improved about 62%.

Figure 7(b) shows the performance improvement when given a different configuration at the beginning. There are six nodes, two of them serving as the proxy servers and four serving as application nodes. However, the proxy servers are highly utilized under the browsing workload. After the dynamic adjustment checking after the 100th iteration, it moved a node from the application server tier to the proxy server tier for the adjustment automatically. The CPU and disk I/O are highly loaded on the proxy servers before the adjustment and some application servers are idling most of the time. After the adjustment, the average load on all proxy servers is lowered, the average utilization on the remaining application servers is increased and the system performance is improved for about 70%.

Since the cases in Figure 7(a) and (b) are dual of each other, it shows the importance of automatic tuning of node roles.

V. Discussion

To tune existing software such as the Squid proxy server, we needed to make some minimal modifications to add calls to the Active Harmony API. However, some variables are only referenced once after the program starts execution (i.e., those variables read from the configuration script file). Rather than make more extensive changes to the program, the Active Harmony system restarts the server for each of the tuning iterations automatically.

Another issue is the hard coded (compile time) limits in the applications. In order to make the system tunable, some limits had to be increased. Again, a more significant coding effort could have been used to convert these hard-coded limits into ones that could be changed at runtime. For example, to increase the number of files opened simultaneously, the value in the */proc/sys/fs/file-max* on Linux needed to be increased. Otherwise the number of files opened simultaneously would be limited. The Active Harmony tuning server will not be able to have the system open files more than this number to improve the performance. This may not be good when the system has extra resources. In this case, recompilation of the linux kernel would be necessary. Besides the kernel, the linux operating system also imposes similar constraints in the */etc/security/limits.conf* and */etc/sysctl.conf* .

A similar phenomenon also happens in the application code. For example, the Apache [2] 1.3.26 HTTP server allows the system administrator to set the maximum number of clients who can simultaneously connect. However, there is another hard limit set by HARD_SERVER_LIMIT inside httpd.h source code. To allow Harmony maximum flexibility, we had to increase that limit, too.

The Active Harmony helps the cluster-based web service adapt itself when facing different workloads. It shows the ability to tune a large-scale system simultaneously and automatically. The tuning includes the parameters adjustment inside each machine and the explicit configuration change for the load imbalance issue. This performance improvement is difficult to achieve by tuning each single machine independently since it is extremely difficult to decide the contribution for each individual machine to the performance of the whole system.

VI. Related Work

There are several projects that are trying to develop techniques to allow applications to be responsive to their available resources or that allow them to be tuned at runtime. The Falcon project [9] focuses on computational steering. It provides a way for users to alter the behavior of an application under execution. The execution results are also changed based on the steering mechanism. For example, adding a particle to a simulation, as part of a problem-solving environment will change the experiment result. The Active Harmony project also allows user to alter the configuration during execution but it is focusing on performance tuning rather than the experiment result.

The Autopilot project [16, 17] allows applications to be adapted in an automated way. It uses sensors to extract quantitative and qualitative performance data from executing applications, and provides the requisite data for decision-making. The kernel of the decision process for the Autopilot is fuzzy logic. Their actuators execute the decision by changing parameter values of applications or resource management policies of underlying system. The Active Harmony project differs from the Autopilot project in that it tries to coordinate the use of resources by multiple libraries and applications rather than focusing on a single application.

The AppLes project [6] and the Odyssey project [15] focus on the resource awareness in the application level. In those systems, applications are informed of resource changes and provided with a list of available resource sets. Then, each application allocates the resources based upon a customized scheduling to maximize its own performance.

The ATLAS [21] project has developed automatically tuned linear algebra libraries. They develop a methodology for the automatic generation of high efficient basic linear algebra routine for a given microprocessor. By using a code generator that probes and searches the system for an optimal set of parameters, it can produce highly optimized matrix multiply for a wide range of architectures. The difference between our work and ATLAS is that our work focuses on general applications that use program libraries rather than that of a specific library.

The Nimrod/O project [5] tries to reduce the search space for engineering design. It applies multiple tuning algorithms including Simplex, P-BFGS, Divide and Conquer, Simulated Annealing. The design for the aerofoil may need to search for the global optima instead the local optima. The Active Harmony project focuses on the performance issue. Therefore, operating points on local optima are still acceptable in most of the cases since they are also good enough from the performance point of view.

Another TPC-W benchmark implementation available from academic institute is from the DynaServer project [19]. The project studies the design of scalable, high-performance and highly available e-business servers.

Others have discussed cluster-based web service with different performance metrics. Joel L. Wolf's work [22] proposed a scheme, which attempts to optimally balance the load on the servers of a clustered Web farm. They try to solve the performance problem by achieving minimal average response time for customer requests. Thus ultimately achieve maximal customer throughput.

ADAPTLOAD [18] developed by Riska, A., et al. models clustered web server as a front-end dispatcher and back-end nodes. They use an online algorithm to decide the share of the total workload for each node to achieve load balance. They treat back-end nodes to be static while the Active Harmony tries to configure the clustered system properly to achieve better performance.

Chen, S., et al. [8] use a reconfiguration mechanism to improve the throughput of a clustered system. Their focus is to avoid letting a small number of running jobs with unexpectedly large memory allocation block the execution of the majority jobs in the cluster. The Active Harmony focuses on a general mechanism to improve overall system performance by several means.

Kalogeraki, V., et al. [11] migrate objects or jobs from hotspots in the cluster to improve the performance. Their goal is to achieve load balance while the Active Harmony focuses on performance improvement.

Gage [13] focuses on load distribution to provide the performance guarantee for cluster-based Internet services. This involves support from network level while the Active Harmony only tries to tune the system to achieve better performance.

VII. Conclusion

Active Harmony is a general tuning system that has no domain specific information while tuning. It improves the system performance iteratively by changing the parameter values and observing the result.

This paper shows the usefulness of using the Active Harmony system to improve the performance of a cluster-based web service system. We applied the Active Harmony to a real-world large-scale system and evaluated the result using a practical benchmark. The performance improvement is difficult to achieve when tuning individual components of the system separately. Since no single universal configuration is good for all kinds of workloads, the cluster based web service system needs a tuning mechanism like the Active Harmony.

Active Harmony adjusts the tunable parameters based on the observed performance results to improve the overall system performance. The experiment results show that Active Harmony system improves the system performance from 5% to 16% depending on the workload. It is able to make applications sensitive to the external factors and parameters that characterize the environment in which they are executed.

Scalability becomes a critical issue when tuning large-scale systems with numerous parameters. We investigated two approaches for tuning – parameter replication and parameter partitioning. This is helpful to speed up the tuning process so the tuning results will not be out of date. Parameter duplication helps to speedup the tuning process while parameter partitioning makes the tuning process smoother with stable performance.

Dynamically adjusting the components of the cluster, the performance is improved by load balancing issue. In our experiments, the system throughput is improved up to 70%. All the results demonstrate that Active Harmony can bring significant performance improvement to the cluster-based web service system and permit new ways to adapt applications to dynamic environments.

References

1. *The Apache Jakarta Project http://jakarta.apache.org/.*
2. *The Apache Software Foundation http://www.apache.org*.
3. *MySQL Database Server*, MySQL AB *http://www.mysql.com.*
4. *Squid Web Proxy Cache http://www.squid-cache.org/.*
5. Abramson, D., et al. *An Automatic Design Optimization Tool and its Application to Computational Fluid Dynamics*. in *SC*. 2001. Denver.
6. Berman, F. and R. Wolski. *Scheduling from the perspective of the application*. in *Proceedings of 5th IEEE International Symposium on High Performance Distributed Computing*. 1996. Syracuse, NY, USA 6-9 Aug. 1996.
7. Bezenek, T., et al., *Java TPC-W Implementation Distribution http://www.ece.wisc.edu/~pharm/tpcw.shtml.*
8. Chen, S., L. Xiao, and X. Zhang. *Adaptive and Virtual Reconfigurations for Effective Dynamic Job Scheduling in Cluster Systems*. in *22 nd International Conference on Distributed Computing Systems (ICDCS'02)*. 2002. Vienna, Austria.
9. Gu, W., et al. *Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs*. in *Frontiers '95*. 1995. McLean, VA: IEEE Press.
10. Hollingsworth, J.K. and P.J. Keleher. *Prediction and Adaptation in Active Harmony*. in *The 7th International Symposium on High Performance Distributed Computing*. 1998. Chicago.
11. Kalogeraki, V., P.M. Melliar-Smith, and L.E. Moser. *Dynamic Migration Algorithms for Distributed Object Systems*. in *The 21st International Conference on Distributed Computing Systems*. 2001. Mesa, AZ.
12. Keleher, P.J., J.K. Hollingsworth, and D. Perkovic. *Exposing Application Alternatives*. in *ICDCS*. 1999. Austin, TX.
13. Li, C., et al. *Performance Guarantee for Cluster-Based Internet Services*. in *The 23rd IEEE International Conference on Distributed Computing Systems (ICDCS 2003)*. 2003. Providence, Rhode Island.
14. Nelder, J.A. and R. Mead, *A Simplex Methd for Function Minimization.* Comput. J., 1965. **7**(4): p. 308--313.
15. Noble, B.D., et al. *Agile Application-Aware Adaptation for Mobility*. in *16th ACM Symposium on Operating Systems Principals*. 1997.
16. Ribler, R.L., H. Simitci, and D.A. Reed, *The Autopilot Performance-Directed Adaptive Control System.* Future Generation Computer Systems, special issue (Performance Data Mining), 2001. **18**(1): p. 175-187.
17. Ribler, R.L., et al. *Autopilot: Adaptive Control of Distributed Applications*. in *High Performance Distributed Computing*. 1998. Chicago, IL.
18. Riska, A., et al. *ADAPTLOAD: Effective Balancing in Custered Web Servers Under Transient Load Conditions*. in *22 nd International Conference on Distributed Computing Systems (ICDCS'02)*. 2002.
19. Snavely, A., et al. *A Framework for Application Performance Modeling and Prediction*. in *Supercomputing 2002*. 2002. Baltimore, MD.
20. Tapus, C., I.-H. Chung, and J.K. Hollingsworth. *Active Harmony: Towards Automated Performance Tuning*. in *SC'02*. 2002. Baltimore, Maryland.
21. Whaley, R.C. and J.J. Dongarra. *Automatically tuned linear algebra software (ATLAS)*. in *Supercomputing*. 1998. Orlando, FL.
22. Wolf, J. and P.S. Yu, *On Balancing the Load in a Clustered Web Farm.* ACM Transactions on Internet Technology, 2001. **1**(2): p. 231-261.