Adaptive Load Sharing
for Clustered Digital Library Servers

Huican Zhu Tao Yang Qi Zheng
David Watson Oscar H. Ibarra Terence Smith
Department of Computer Science
University of California
Santa Barbara, CA 93106
{hczhu, tyang, zheng, david, ibarra, smithtr}@cs.ucsb.edu

Abstract

This paper investigates load balancing strategies for clustered Alexandria digital library (ADL) servers. The ADL system, which provides on-line information searching and browsing of spatially-referenced materials through the World Wide Web, involves intensive database I/O and heterogeneous CPU activities. Clustering servers can improve the scalability of the ADL system in response to a large number of simultaneous access requests. One difficulty addressed is that clustered workstation nodes may be non-uniform in terms of CPU and I/O speeds. An optimization scheme is proposed in this paper to dynamically monitor the resource availability, use a low-cost communication strategy for updating load information among nodes, and schedule requests based on both I/O and computation load indices. Since the accurate cost estimation for processing database-searching requests is difficult, a sampling and prediction scheme is used to identify the relative efficiency of nodes for satisfying I/O and CPU demands of these requests. A set of experiments using the ADL traces have been conducted to verify the effectiveness of the proposed strategies.

1 Introduction

Network information systems are becoming increasingly important with the advent of Internet technology. Digital library (DL) systems[8], which provide on-line retrieval and processing of digitized documents through the Internet, have increasingly turned into a topic of national importance. This trend will continue as digital storage costs dramatically decrease relative to the cost of traditional library shelf-space and as high performance electronic services and Internet connection become more popular and affordable. The Alexandria project [1, 14], supported by the national digital library initiatives, is focused on the design, implementation, and deployment of a distributed digital library for spatially-indexed information. The collections of the library currently involve geographically-referenced materials, such as maps, satellite images, digitized aerial photographs, and associated metadata. Many collection items in the ADL have sizes ranging from megabytes to gigabytes and may require extensive processing for certain applications.

Performance issues must be resolved for DLs that support extensive collections with large data items. Considering that popular Web sites such as Yahoo already receive millions of accesses per day, a DL server, which involves much more intensive I/O and heterogeneous CPU activities, will become a bottleneck in delivering digitized documents over high-speed Internet.

In this paper, we investigate optimization strategies in clustering a number of workstations together as an ADL server group to improve the scalability of the ADL system. One challenge that needs to be addressed is that nodes in the cluster may be non-uniform. Assigning an incoming request to a proper server that leads to the minimum response time is the goal of our optimization. The response time for a request is normally defined as the length of time from when the transaction is initiated until all requested information arrives at the client. Since all the nodes in our cluster are located in a LAN and therefore have a uniform internet delay, we only consider the time spent by the server when speaking of the response time of a request.

ADL server is currently implemented as a Web-based information system, therefore, multiple clustered ADL servers can be viewed as a web server group. Issues related to building web server clusters have been addressed in the previous work. The first workstation Web server cluster [9] used a DNS round-robin rotation to assign requests to a server node. The SWEB project [2] further optimized such a system by redirecting requests based on dynamic system loads. Web request redirection based on the CPU loads has been also recently studied in [5, 12]. Previous load balancing research (e.g [13]) normally used one load index such as CPU usage or ready queue length and assumed uniform node capabilities. But in Web systems such as the ADL server cluster, requests may involve database searching and image processing in which computation and I/O demands are mixed and varying, and the nodes may have different I/O and CPU capabilities. A more sophisticated scheduling scheme, which can consider the availability of multiple system resources, is needed.

Our previous work with the SWEB project has developed a prediction based scheduling scheme for assigning Web requests. The scheme attempts to predict the absolute processing time for each request based on its computation and I/O complexity as well as the CPU and I/O load on each node. For file retrieval and image-processing requests [2, 3] which have easy-to-identify CPU and I/O usage patterns, this scheme has been shown to be very effective. It is difficult, however, to predict the cost of database queries which dominate ADL request activities. Although some research work on predicting the result size of a query is available (e.g. [10]), no result exists on how to accurately predict the actual processing time of a general database query based on the current CPU and I/O resource availability. To cope with this problem, We propose a scheme to sample the I/O and CPU demands of typical ADL queries, and identify a near-optimal server node using its relative resource availability.

Since the performance of scheduling is sensitive to the accuracy of the global load information available to each node, we further evaluate two popular schemes for load information collection and we propose a new solution which involves smaller overhead and delivers better performance.

This paper is organized as follows. Section 2 briefly describes the ADL system and its functionalities. Section 3 discusses the organization of our server cluster and its scheduling and resource monitoring strategies. Section 4 presents experimental results to examine the effectiveness of the proposed techniques. Section 5 concludes the paper. We have also conducted an analysis on performance bounds of our cluster. The results can be found in [16].

2 The Alexandria Digital Library

The primary research of the ADL is to provide users with spatially-referenced access to large classes of distributed library holdings through the Internet. Current ADL collections contain more than 6.7 million entries of maps, satellite images, text documents, scientific data sets, and information on geographical features. Each entry has geographical location information represented as a minimum bounding box using longitude and latitude. All entries are spatially indexed and stored in multiple databases.

In a high level view, the Alexandria Digital Library Testbed system has three major logical components: user interface, ADL server and a set of the associated databases.

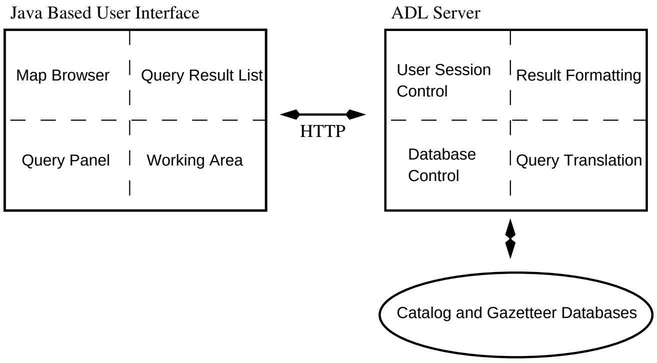

Figure 1. The architecture of the ADL system.

The Java based user interface contains a map browser, a query panel, a result display window and a working area. The map browser allows users to define spatial queries on the regions of interest and displays spatial information on a library item. The query panel is used to formulate non-spatial query parts such as the type of an item, delivering format, coverage time period, originator, and descriptive text. The result window displays query results and sorts the result list based on user requests. The working area is used to store user selected results from multiple queries (query history), and to invoke external processing programs. The user interface component communicates with the ADL server through HTTP calls.

The ADL server is a middleware component in the testbed and is embedded with an HTTP server using the HTTP server function extension. The major functions of the ADL server are query translation, database selection, result formating, and user session control. Since each query may require a few search methods (spatial, numerical, etc.) to find the set of matching library items, the server needs to merge partial results to produce the final query hits.

The set of databases can be divided into two large classes: catalog and gazetteer. The catalog database serves as a traditional library catalog system and has all metadata information about the ADL holdings. Gazetteer databases have information for geographic features such as cities, schools, rivers, and parks in the world. Each database has its own data management schema. Complexity of schema ranges from a single table with 10 attributes up to 80 tables with more than 500 attributes. The attributes exposed by the database interface for user query include location (geographic polygon), type of library holding (classification such as maps, image, book, report, etc), data format (text, GIF, TIFF, etc.), beginning of coverage date, ending coverage date, originators, keywords, descriptive information, and identifiers (such as ISBN, ISSN).

A typical ADL testbed operation path starts with a user to select a region of interest by zooming and panning the map browser in the user interface. Then the user can issue boolean queries for problems such as "Find out all NASA satellite images in GIF format from Nov. 1994 to Jan. 1996 which cover my selected region" or "Show me all airports in my selected region". Based on the query values, the ADL server will select a number of databases which may contain user requested objects, and translate the boolean query into database access statements (SQL or lower level access function calls) for each selected database. The server will merge and format query results from each database and return them back to user interface for displaying.

It is easy to see that the speed of the ADL server plays a major factor in the overall system response time. In our initial testbed system, the average database query response time was about 110 seconds. After a server redesign to provide multi-threaded access and database performance tuning, the average response time is about 40 seconds using a single ADL server. We will soon reduce this number to around 10-20 seconds by database reorganization including data partition, creating table views, and stripping data over disks and database servers. But the server bottleneck problem will be persistent if we only use a single server. Facing an increasing number of concurrent user accesses, the server bottleneck problem can only get worse. One way to reduce the congestion in the ADL server is to use multiple servers and form a server cluster.

3 Organization of the ADL server cluster

Our server cluster consists of a set of workstation nodes connected with a fast network as illustrated in Figure 2. Each server node in the cluster has a local disk which contains a fully replicated ADL database collection, and is running a complete single-machine version of the ADL system which consists of a Web server and a database server. Besides, each node also runs a load monitor which accesses the resource availability of local and remote nodes and a scheduler which identifies the best node for processing a given request. The cluster is presented as a single logical server to the Internet. User requests are first evenly routed to processors via DNS rotation [2, 9] and then redistributed within the cluster based on the resource usage on each node.

Currently the ADL system uses Illustra as its database server and AOLserver2.2 as its Web server, primarily because of the internal support for Illustra database access by AOLserver. Unavailability of source code for these commercial products has posed a problem to the integration of load balancing functionality into the system. We ad-

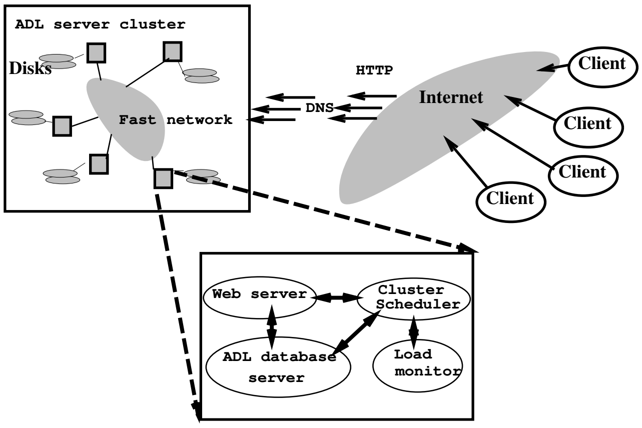

Figure 2. The architecture of the clustered ADL servers.

dressed this issue by using the "filter" API provided by the AOLserver. With that, the processing of a request proceeds in the following way: After receiving a request, the server first executes control access authorization, and then "filters" this request to the scheduler. If the scheduler decides to execute the request locally, it returns to the server for further processing of the request; Otherwise the scheduler reassigns the request to another server in the cluster.

The scheduler does the reassignment by URL redirection, i.e., by composing an HTTP response header "302: Document Temporarily Moved" and sending it back to the requesting client agent (browser) along with the location of the new node capable of processing the request. The client agent then sends the same request to the new location transparently without user awareness. To avoid having requests ping-pong between server nodes, the new server is not allowed to redirect this request again. Note that this reassignment method introduces a small redirection overhead. To avoid the dominance of such overhead, we include the redirection overhead in estimating the overall cost and a reassignment is made only if the redirection overhead is smaller than the predicted time savings. For example, if a request involves a very small file retrieval, no redirection usually will occur. HTTP redirection technique has been used in SWEB [2] and a commercial product called "distributed director" from CISCO [4].

3.1 Design of the load sharing strategy

As the DNS assigns requests to server nodes in a round-robin fashion, a uniform distribution of requests among all the nodes is expected. However the following situations complicate the load distribution.

1. The computation and I/O demands of users' requests vary substantially.
2. Clustered nodes are non-uniform in terms of CPU power and disk I/O speeds.
3. Hot spots may be caused by IP address caching which occurs in the DNS system.
4. The nodes in a cluster may not be dedicated. Some activities other than ADL request processing may be conducted concurrently. This is true in our testing environment since the workstations are shared by other users.

Each of these items may lead to serious load imbalance, which necessitates load rebalancing after the DNS rotation.

To effectively redirect requests, we classify ADL requests into the following three classes according to their CPU and I/O requirements and use different scheduling algorithms for each class:

1. Text and image file retrievals. CPU and I/O usage is small for this class of requests.
2. Database searches. This class requires a large amount of I/O time and varying amounts of CPU time.
3. Image processing for progressive delivery of image data. A moderate amount of CPU and I/O time is required.

For file retrieval and image processing requests, the complexity is relatively easier to estimate. The server-site cost for processing such a request at a node is approximated as:

$$t_s = t_{redirection} + t_{data} + t_{CPU}$$

where $t_{redirection}$ is the cost to redirect this request to another node, if required. t_{data} is the time to transfer the required data from the disk drive. t_{CPU} is the time to fork a process/thread, perform disk reading, plus any known associated computational cost if the request requires a server application to complete the request. For file retrieval and image processing requests, these values can be predicted with relative accuracy using the prediction model from earlier work [2, 3]. A detailed description of the formula is in [2, 3]. The scheduler uses function t_s to predict the cost on different nodes, and select the node with the smallest value to process.

For a database searching query, it is difficult to predict the exact cost [10], especially for queries which may involve spatial data searching, such as those in the ADL system. To address this issue, we obtained the ADL access log from Nov 15, 1997 to Jan 8, 1998. From this log we observed that the ADL database query requests have the following characteristics:

- Long processing time. This means that request scheduling and redirection overhead can be ignored.
- Intensive I/O activities and small usage of CPU. This means that we need to consider I/O capability to make scheduling decisions instead of only using CPU ready queue length or CPU idle percentage.
- Varying processing time. Some queries may take only a few seconds while others may take more than one minute even though they have very similar structures. Therefore it is difficult to accurately predict the processing time of each query.

Based on these observations, our strategy is to obtain the average I/O and CPU time percentage spent for processing the ADL database queries through offline sampling. We then use this value to estimate the *relative server-site response cost* (RSRC) of each given query on a node according to the following formula:

$$RSRC = \omega * \frac{BaseCPUSpeed}{CPUSpeed * IdleRatio} +$$
$$(1-\omega) * \frac{BaseDiskBandwidth}{DiskBandwidth * DiskAvailRatio}.$$

We explain the above terms as follows.

- ω is the average percentage of the cost contributed by CPU for a query. $1-\omega$ is used to approximate the average percentage of the cost contributed by I/O. ω is obtained by sampling the previous ADL trace on the unloaded node with the lowest disk bandwidth. This node is called the *base node*. In our trace analysis, $\omega \approx 0.08$ as explained in Section 4.
- $CPUSpeed$ is the aggregated CPU speed of this node. If a node has multiple CPUs, the CPU speed of this node is multiplied by the number of processors. $BaseCPUSpeed$ is the CPU speed of the base node. As we mentioned in Section 1, the CPU speeds of nodes in the cluster may differ.
- $DiskBandwidth$ is the aggregated bandwidth of the storage system linked to this node. $BaseDiskBandwidth$ is the bandwidth of the base node. Notice that I/O speeds of nodes in the cluster may differ.
- $IdleRatio$ is the percentage of idle CPU time available for usage in this node. $DiskAvailRatio$ is the available ratio of disk bandwidth in this node. These two numbers dynamically change based on the system loads and are supplied at runtime to compute the RSRC. In our implementation, we use the Unix rstat() function to collect the load information on each node.

Notice that the RSRC measurement does not give the actual processing time but gives an index for the availability of aggregated resources on a node relative to the unloaded base node. With this formula, a node with the smallest RSRC value is the best node for processing a given request. Since redirection of a request costs certain overhead, before redirecting a request to a remote node, we compare the difference between the RSRC values on the remote node and the local node. The redirection is done only when the difference is larger than a threshold value(currently chosen to be 0.1). Using this threshold ensures that the redirection cost is insignificant compared to processing time saving.

3.2 Policies for node selection and load collection

In order for each processor to make a decision in a distributed manner and choose the best server node for redirection, two issues need to be addressed. Which nodes are to be considered for redirection? How should the load information of a node be propagated to other nodes? Node selection and load collection policies have been studied in [6, 7, 15] and we list the two most popular strategies below.

- **Broadcast**. Every t seconds, each node broadcasts its load (CPU idle ratio and disk bandwidth availability) to other nodes. When a request arrives at a node, this node compares the RSRC values of all nodes using the latest load information.

- **Random poll**. When a request arrives at a node, this node polls a fixed number (L) of other nodes and compares the RSRC values of those nodes.

The first strategy keeps all nodes informed of recent load information, which helps each node make an accurate decision. However it has a disadvantage, especially in our setting where DNS has already evenly distributed requests. If a node is heavily loaded, there is a high probability that other nodes are also heavily loaded. The benefits of having all nodes keep accurate load information may be outweighed by the delay of broadcasting on a heavily loaded system, which can become substantial and cause the fast aging of load information. Another disadvantage of this strategy is that many requests may be simultaneously redirected to one node considered as "lightest" by all other nodes, which causes load explosion on that node. The second strategy (random poll) has the advantage of avoiding excessive broadcast operations between nodes, and there is a high probability that an ideal node can be found when the entire server group is relatively lightly loaded. However it suffers from a synchronization problem. Polling leads to more synchronization overhead compared to single-side sending, since a node has to wait for responses after polling and cannot make a decision until all selected nodes reply. When the system workload is heavy, this synchronization affects the system performance since this node is already over-busy handling other requests. Thus we propose the following two strategies for the ADL cluster.

- **Random multicast**. This is the same as the broadcast strategy except that instead of a node broadcasting load information to all other nodes, the node broadcasts to L randomly selected nodes. We expect this strategy to work well in the heavily loaded situations since it informs load information with a reasonable cost.

- **A hybrid approach**. This scheme combines the ideas of "random multicast" and "random poll". Under this scheme, load information at each node is multicasted to L randomly selected nodes every t seconds In processing an incoming request, we use the following aggregated load index to check if random poll should be used:

  $$\gamma(1 - IdleRatio) + (1 - \gamma)(1 - DiskAvailRatio).$$

  This aggregated index combines both I/O and CPU load information for each node. If this value at a node is below a threshold, this node is considered lightly loaded and it randomly polls L nodes and selects the best among them. Otherwise, it will not poll other nodes, and will instead use the most recent load information multicasted by others to select a node. Note that when doing polling, the node may also receive loads multicasted from other nodes and use them for making a decision.

  In our experiment, we use a threshold of 0.5 and set $\gamma = 0.20$ since the majority of our current ADL accesses involve I/O. We use t=2 seconds and $L = 2$ since our tested cluster contains at most 8 nodes.

The effectiveness of the above four policies are evaluated in Section 4, which shows that the hybrid approach has the best performance for our setting.

4 Experiments

We implemented our system on a cluster of six Sun 147Mhz UltraSparc 1 workstations and two Ultra 2 workstations with dual 167Mhz processors connected via fast Ethernet. The achievable bandwidth of database disks on these nodes are: 14Mbits/sec on the two Ultra 2's, 10Mb/sec on four of the Ultra I's and 8 Mb/sec on the other two Ultra 1's. The Ultra 2's have a processing rate about 2 times that of the slowest nodes for the ADL requests we tested.

4.1 Experiment setup

We obtained two sets of test requests from the ADL. One set, $S1$, consists of 490 database requests and has been used by the ADL implementation team as sample queries for the ADL database performance tuning. The other set, $S2$, consists of 880 queries and was obtained from the access logs of the ADL Web server spanning from Nov 15, 1997 to Jan 8, 1998. The average processing time of all the requests on all the nodes is 14 seconds for set $S1$ and 11.5 seconds for set $S2$.

$S1$ and $S2$ contain solely database queries. Considering that the next version of the ADL testbed system will incorporate a progressive subregion image browsing scheme [3, 11], we mixed $S2$ with 800 sample requests representing subregion image accessing/browsing with an average processing time of 1.5 seconds and 800 image file retrievals with an average processing time of 0.8 seconds. We call this new set $S3$.

We used $S1$ as a training set to sample the I/O and CPU demands of database queries on our base node and obtained $\omega = 0.08$ for estimating the RSRC value discussed in Section 3.1. All experiments are conducted using sets $S2$ and $S3$.

Each experiment is conducted in the following way: There is a central request generator located on a different subnet which reads requests from the test set and sends them randomly to one of the nodes in the cluster. Given a request arrival rate, the generator controls the inter-arrival rate (I) among requests following a Poisson process, a standard model for evaluating client server applications [6, 7]. We ran our experiments during night hours to minimize external load activities. We repeated each experiment four times with the same set of parameters (request arrival rate, ω value and load collection and node selection policy). The average response time of all the requests in an experiment is used for comparison.

We report three sets of experiments. The first set studies the speedups of our system when the number of nodes in the cluster increases. The second set of experiments examines the advantages of our hybrid policy for node selection and load updating compared to the other strategies. The last set of experiments examines the effectiveness of our method in predicting the relative efficiency of non-uniform nodes and selection of the best node for redirection. When reporting experiment results, besides listing the average inter-arrival time, we also list the expected waiting queue length Q_p on each node in order to give readers some impression of system load. Q_p is calculated by assuming that the request processing time is exponentially distributed [16].

4.2 Multi-node performance

This subsection examines the performance of the system when the number of nodes in the cluster is varied. The node selection and load collection policy used here is the hybrid method, combining random poll and multicast. Given an average request arriving interval I for the entire cluster, we measure the relative speedup on our non-uniform cluster as: $\frac{R_1(I)}{R_p(I)}$ where $R_1(I)$ is the average response time on the fastest single node (Ultra 2) with an arrival rate of $1/I$ to this node. $R_p(I)$ is the average response time with p nodes and a total average arrival rate of $1/I$. Note that for each node, the average arrival rate is about $\frac{1}{pI}$ after the DNS rotation, although each varies slightly.

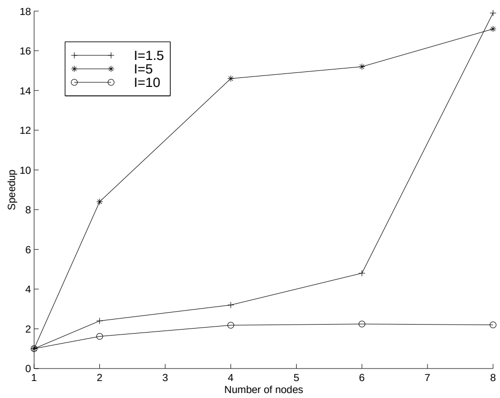

Figure 3. Speedups for different arrival rates.

Figure 3 depicts the speedups under different workloads and different numbers of nodes for processing trace set $S2$. The speedup result for $S3$ is similar. The values we chose for the inter-arrival time I are $1.5, 5$ and 10, which represents a range of workloads, from heavy to light. We used the following cluster configurations: 1 node (Ultra 2), 2 nodes (Ultra 1, Ultra 2), 4 nodes (two Ultra 1 and two Ultra 2), 6 nodes (four Ultra 1, two Ultra 2), and 8 nodes (six Ultra 1 and two Ultra 2).

The following two cases should be noted from Figure 3.

- For average inter-arrival times of $I = 1.5$ and 5, which are smaller than the average request processing time on one node, superlinear speedups are achieved (as much as 17.9 with 8 nodes). In these cases, request resource demand exceeds system capacity, a request cannot be processed in time, and its waiting time adds to its response time. This phenomenon reveals the importance of adding more processing nodes to an overloaded system.
- For $I = 10$, there is not much speedup. This is expected because the total workload is very light even

for one node in this case. There is really no need to use more nodes. No matter how many more nodes are used, we cannot expect to see the average processing time be improved.

Figure 4 shows the ideal and sustainable throughput of the system when the number of nodes increases. The throughput is defined as the number of requests processed per second. We explain how we compute the ideal and sustainable throughput as follows.

- We measure the average processing time of all requests on each node when each request is processed exclusively and mark the results on all p nodes as $\mu_1, \mu_2, \cdots, \mu_p$. The ideal throughput of the system is defined as:
$$\sum_{i=1}^{p} \frac{1}{\mu_i}.$$
- The sustainable throughput of our system is defined as the maximum number of requests per second that the system can handle while the average response time is within a predefined limit. In our experiments, the limit is $1.5\bar{\mu}$ where $\bar{\mu} = \sum_{i=1}^{p} \mu_i/p$. We measure the sustainable throughput on a specific number of nodes by gradually increasing the request arrival rate until the average response time exceeds the limit.

It is difficult to reach the ideal throughput since this measurement does not consider any system overhead for scheduling and assumes perfect load balancing. From Figure 4 we can see that the sustainable throughput of our system is very close to the ideal throughput. This indicates that our load balancing strategies are very effective.

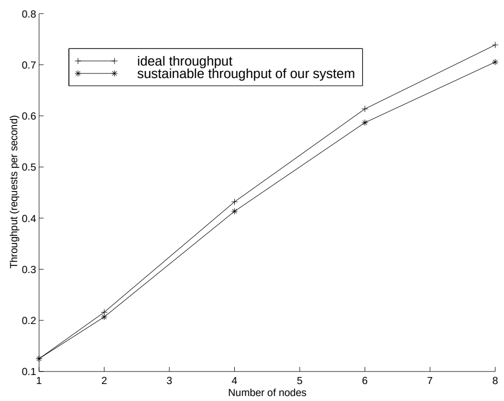

Figure 4. The ideal and sustainable throughput of the system.

4.3 A comparison of different scheduling and load collection policies

In this subsection we assess the effectiveness of our hybrid scheduling scheme by comparing it with several other methods. One of them is the DNS round-robin assignment which does not utilize load information. We call it "DNS even assignment". As discussed in Section 3.2, the hybrid approach combines random poll and multicast. We compare this strategy with those using "load broadcast", "random poll", and "random multicast". The improvement ratios of our method over the others are listed in Table 1 for processing request set $S2$. The results for $S3$ are similar.

Q_p	0.5	1	2	3
I	4.5	3	2.2	2
DNS even assignment	48%	40%	38%	39%
Broadcast	17%	26%	31%	25%
Random poll	0%	4%	17%	20%
Random multicast	8%	11%	7%	5%

Table 1. Response time improvement of the hybrid approach over the other policies.

The results show that our method outperforms the DNS assignment substantially by around 40%. It is also consistently better than the "Broadcast" strategy by up to 31%. When the workload is heavy with $Q_p = 2$ and 3, the performance of "Random multicast" is pretty close to ours. When the workload is relatively light with $Q_p = 0.5$ and 1, "Random poll" performs reasonably well. The overall performance of our hybrid method is the best since it takes advantage of "random multicast" for heavy workloads, and that of "random poll" for light workloads.

4.4 Consideration of I/O and CPU load indices

We investigate the effectiveness of our RSRC function using weighted CPU/IO load indices and report performance improvement ratios using our scheme over four other simpler strategies listed below.

- Let $BaseCPUSpeed = CPUSpeed$ and $BaseDiskBandwidth = DiskBandwidth$. But let $\omega = 0.08$. This scheme considers all nodes uniformly.
- Let $\omega = 1$. In this case, only CPU factor is considered. A node with light CPU load will receive more requests to handle. We still consider nodes non-uniformly.

I (seconds)	1.5		2.125		2.75		3.375	
Q_p	23		2.3		1.2		0.8	
Request set	S2	S3	S2	S3	S2	S3	S2	S3
CPU load only	42%	61%	34%	52%	32%	43%	25%	39%
Half CPU, half I/O	36%	49%	33%	38%	25%	35%	19%	26%
I/O only	10%	22%	4%	17%	-3%	9%	2%	7%
Uniform nodes	33%	44%	29%	36%	20%	25%	27%	20%

Table 2. Improvement ratios using our parameter over other simpler strategies for set $S2$ and $S3$.

- Let $\omega = 0$. Only disk I/O factor is considered. A node with light I/O load will receive more requests to handle at any time step. We still consider that nodes are non-uniform.

- Let $\omega = 0.5$. CPU and I/O loads are considered evenly here. Nodes are still considered non-uniform.

Table 2 lists the improvement ratios for request set $S2$ and $S3$ on 8 nodes. We can see that our method substantially outperforms the other strategies. As the workload becomes heavier the improvement ratio gets larger since load balancing becomes more important than the cases where workload is light. The "I/O only" strategy is competitive to ours for $S2$ with light workloads because those requests are I/O intensive. However for $S3$, there are mixed computation and I/O activities and the advantage of our method becomes more clear.

5 Concluding remarks

In this paper we present a study on adaptive load balancing strategies for the ADL server on a non-uniform cluster of workstations. The experiments show that it is important to consider both I/O and CPU load factors in making scheduling decisions, and that our method for estimating the impact of these two factors is effective in processing ADL requests on this non-uniform cluster. We have also designed a hybrid node selection and collection policy which is more efficient than other existing methods in our context. Our future work is to further evaluate and extend our scheme (e.g. dealing with non-uniform memory sizes and dynamically updating the weight value ω), and study the use of a faster network such as SCI for the cluster.

Acknowledgments

This work was supported in part by NSF IRI94-11330, CCR-9702640, CDA-9529418, and UC MICRO/SUN. We thank Reagan Moore and the anonymous referees for their valuable comments.

References

[1] D. Andresen and et al. The WWW prototype of the alexandria digital library. *Proceedings of ISDL'95: International Symposium on Digital Libraries*, Aug. 1995.

[2] D. Andresen, T. Yang, V. Holmedahl, and O. Ibarra. Sweb: Towards a scalable WWW server on multicomputers. *Proc. of Intl. Symp. on Parallel Processing, IEEE*, pages 850–856, Apr. 1996.

[3] D. Andresen, T. Yang, D. Watson, and A. Poulakida. Dynamic processor scheduling with client resources for fast multi-resolution WWW image browsing. *Proc. of Intl. Symp. on Parallel Processing, IEEE*, Apr. 1997.

[4] CISCO. *Distributed Director*. http://www.cisco.com, 1997.

[5] M. Colajanni, P. Yu, and D. Dias. Scheduling algorithms for distributed web servers. *Proc. of Intl. Conf. on Distributed Computer Systems, IEEE*, pages 169–176, May 1997.

[6] S. Dandamudi. Performance impact of scheduling discipline on adaptive load sharing in homogeneous distributed systems. *Proc. of Intl. Conf. on Distributed Computer Systems, IEEE*, pages 484–492, 1995.

[7] D. L. Eager, E. D. Lazowska, and J. Zahojan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans on Software Engineering*, 12(5):662–675, 1986.

[8] E. Fox, R. Akscyn, R. Furuta, and Leggett. Special issue on digital libraries. *CACM*, Apr. 1995.

[9] E. Katz, M. Butler, and R. McGrath. A scalable HTTP server: the ncsa prototype. *Computer Networks and ISDN Systems*, 27:155–164, 1994.

[10] R. Lipton and J. Naughton. Query size estimation by adaptive sampling. *J. Comput. Syst. Sci.*, 51:18–25, 1995.

[11] A. Poulakidas and et al. A compact storage scheme for fast wavelet-based subregion retrieval. *Proc. of 1997 International Computing and Combinatorics Conference (COCOON)*, Aug. 1997.

[12] Resonate. *Dispatch*. http://www.resonate.com, 1997.

[13] B. A. Shirazi, A. R. Hurson, and K. M. Kavi, editors. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE CS Press, 1995.

[14] T. Smith. A digital library for geographically referenced materials. *IEEE Computer*, 29(5):54–60, 1996.

[15] S. Zhou. A trace-driven simulation study of dynamic load balancing. *IEEE Trans. Softw. Eng.*, 14(9):1327–1341, Sept. 1988.

[16] H. Zhu, T. Yang, Q. Zheng, D. Watson, O.H.Ibarra, and T. Smith. Adaptive load sharing for clustered digital library servers. *Technical Report, CS, UCSB*, 1998.