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

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 [14], 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 connections become more popular and affordable. The Alexandria project [3, 26], 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, as they are expected to become information repositories which support extensive collections of large quantities of data items and are accessed by millions of users per day. Popular Web sites such as Alta Vista and IBM Olympic site [6] cope with increasing user demands by deploying multiprocessor Web servers. A DL server, which involves much more intensive I/O and heterogeneous CPU activities, can become a bottleneck in delivering digitized documents over Internet when there are many concurrent user accesses. An effective approach to avoiding server bottleneck and providing scalable performance is server clustering.

In this paper, we investigate optimization strategies in clustering a number of workstations together to create an ADL server group in order to improve the scalability of the ADL system. One challenge to be addressed is the potential non-uniformity of nodes in the cluster. The goal of our optimization is to assign each incoming request to the server that leads to the minimum response time. 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 on a LAN and therefore have a uniform Internet delay to the client, we only consider the time spent by the server when speaking of the response time of a request.

The ADL server is currently implemented as a Web-based information system, therefore multiple clustered ADL servers can be viewed as a web server group. Previous work has addressed issues related to building web server clusters. The first workstation based Web server cluster [18] used a DNS round-robin rotation to assign requests to the server nodes. The SWEB project [4] further optimized such a system by redirecting requests based on dynamic system loads. Web request redirection based on server CPU loads has also been recently studied in [8, 24]. Previous load balancing research (e.g [25]) normally used only one load index such as CPU usage or ready queue length and assumed uniform node capabilities. In Web systems such as the ADL server cluster, however, 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 considers the availability of multiple system resources, is needed.

Our previous work with the SWEB project 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 profile as well as the CPU and I/O load on each node. For file retrieval and image-processing requests [4, 5] 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. [19]), 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. We also 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 architecture. Section 3 discusses the organization of our server cluster and its scheduling and resource monitoring strategies. Section 4 provides a theoretical analysis on performance bounds of our server cluster. Section 5 presents experimental results to examine the effectiveness of the proposed techniques for the cluster. Section 6 concludes the paper.

2 The Alexandria Digital Library

The primary research objective 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 fall 1999, ADL will be publically accessible through California Digital Library (www.cdlib.org), a co-library of all nine campuses in University of California. The ADL operation will be supported by the Davidson Library at the University of California at Santa Barbara.

As illustrated in Figure 1, the high level view of the Alexandria Digital Library testbed system contains three major logical components: user interface, ADL server, and a set of the associated databases.

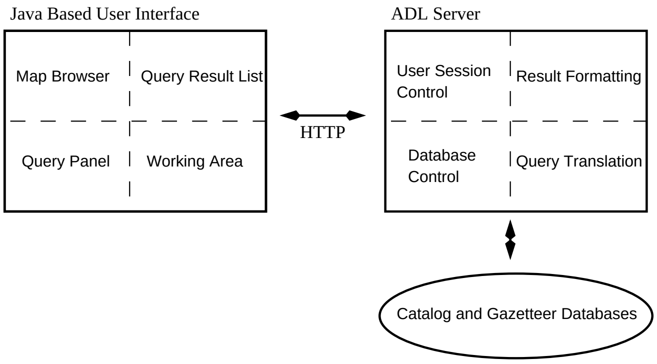

Figure 1: The architecture of the ADL system.

ADL provides two type of user interfaces. One is distributed as a Java application and another is HTML based. The Java application version has many more functions for working with both data and maps. The HTML based version provides easy but limited access to the ADL. Both user interfaces contains a map browser, a query panel, and a result display window. The Java client has an additional result set 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, delivery 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 middleware component is embedded within an HTTP server, using the server's extension functions. The major functions of the middleware 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 may need 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 selecting 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 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. Currently the ADL system uses Informix Illustra as its database server and AOLserver2.2 [1] as its Web server. Each node also runs a load monitor which assesses 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 server nodes via DNS rotation [4, 18] (or via a single-IP switch as discussed in Section 6) and then redistributed within the cluster based on the resource usage on each node.

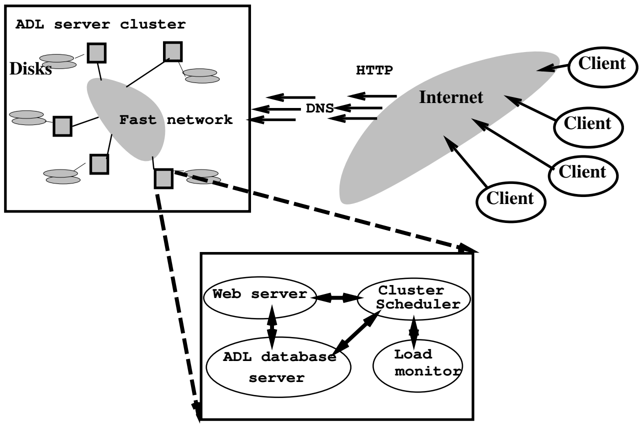

Figure 2: The architecture of the clustered ADL servers.

When a server node receives a request, the scheduler decides either to process the request locally, or to reassign the request to another server in the cluster. If the request is handled locally, the server further processes the request by delivering an HTML file/image or by calling a procedure (or creating a process) to generate dynamic content. Otherwise, the scheduler does the reassignment by remote script processing [29, 28].

In our current implementation, CGI [20] is used to generate dynamic content. Remote CGI can be implemented quite efficiently without imposing a significant amount of overhead for the remote execution. The CGI specification requires that a process be forked for each incoming CGI request to execute the script. In our implementation, when a node decides to execute a CGI request at a remote node, it opens a TCP connection to the remote node and passes the script name and a constructed environment. The remote node forks a new process to handle this request. This forked process redirects I/O to the TCP connection and executes the script with the environment as a parameter. The original node reads from the connected socket and forwards the result back to the client. Because we pass results to the client as they are received at the current node, flow control within TCP places an upper bound on the amount of LAN traffic, preventing the remote server from sending more data than the outgoing link can handle and thereby bounding the amount of local network traffic. This pipelining also limits the network cost of the remote CGI to the connection setup and network latency.

We do not currently use HTTP redirection [4] for request re-scheduling because it adds an additional client round-trip latency for every rescheduled request. Another approach for implementing re-assignment is socket forwarding or TCP/IP hand-off [2, 22]. Both of these methods require changes in the OS kernel or network interface drivers. We avoid making such changes for compatibility reasons.

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 re-balancing after the DNS rotation [1].

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

[1]If a single-IP switch solution discussed in Section 6 is used to do request assignments, load re-balancing is still needed [29].

1. Text and image file retrievals. The server resource demand is relatively small for this class of requests, compared to the Internet delay and the other type 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 requests, Internet delay is the main performance bottleneck and the DNS-based rotation (or a single-IP switch-based solution) provides reasonable load balancing. Our current strategy for this class is to not redirect such requests. For image processing requests, the processing time is relatively easy 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. Term t_{data} is the time to transfer the required data from the disk drive. Term 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. 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 [19], 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 take I/O capability into consideration instead of making scheduling decisions from 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. This means that 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 off-line sampling. We then use this value to estimate the *stretch factor* of each given query on a node according to the following formula:

$$SF = \omega * \frac{BaseCPUSpeed}{CPUSpeed * CPUIdleRatio} + (1-\omega) * \frac{BaseDiskBandwidth}{DiskBandwidth * DiskAvailRatio}. \quad (1)$$

The terms in Equation 1 are defined as follows.

- Symbol ω is the average percentage of the cost contributed by CPU for a query. The average percentage of the cost contributed by I/O is approximated by "$1-\omega$". We call the node with the lowest disk bandwidth in the cluster the *base node*. The value of ω is obtained by sampling the previous ADL trace on the unloaded base node.

  In our trace analysis, $\omega \approx 0.08$ as will be explained in Section 5. In an operational ADL environment, the value of ω may need to be adjusted periodically since software implementation and user access patterns may gradually change.

- $CPUSpeed$ 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.

- $CPUIdleRatio$ 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 SF. In our implementation, we use the Unix rstat() function to collect the load information on each node.

The above stretch factor (SF) measurement represents the ratio of average response time with resource-sharing among requests to that without sharing [17]. Notice that the SF 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 SF value is the best node for processing a given request. Since redirection of a request adds some overhead, before redirecting a request to a remote node, we compare the difference between the SF 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 saved processing time.

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. First, it needs to be decided which nodes are to be considered for redirection. Second, it needs to be decided how the load information of a node should be propagated to other nodes. Node selection and load collection policies have been studied in [10, 11, 27] 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 SF values of all nodes using the latest load information.
- **Random polling**. When a request arrives at a node, this node polls a fixed number (L) of other nodes and compares the SF 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 a 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. This delay can become substantial and cause load information to become outdated very quickly. Another disadvantage of this strategy is that one node may be considered the "lightest" by all other nodes. This can cause many requests to be simultaneously redirected to it, causing *overloading* on that node. The second strategy (random polling) has the advantage of avoiding excessive broadcast operations between nodes. This is good because there is a high probability that an ideal node can be found when the entire server group is relatively lightly loaded. This strategy suffers from a synchronization problem however, especially when under heavy loads. Polling leads to more synchronization overhead than 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 the node is already over-worked 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 broadcasting load information to all other nodes, a node broadcasts to L randomly selected nodes. We expect this strategy to work well in heavily loaded situations since it informs load information with a reasonable cost and has less possibility of overloading a node by requests directed to it simultaneously from many other nodes.
- **Hybrid approach**. This scheme combines the ideas of "random multicast" and "random polling". 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 polling should be used:

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

This aggregated index combines both I/O and CPU load information for each node. If this value at a node is below a threshold, the node is considered lightly loaded and it will randomly poll L nodes and select the best among them. Otherwise, it will not poll, but 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 in 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 is evaluated in Section 5, which shows that the hybrid approach has the best performance for our setting.

4 System performance bound analysis

In this section, we use queuing models (M/M/1 and M/M/p) to estimate the expected maximum and minimum average response times for our system. To take into account the vast variability of request inter-arrival times and request processing times, we assume that requests are fed to our server cluster following Poisson processes and that the processing time of requests on each node has an exponential distribution. Under such an assumption, if λ is the overall average request arrival rate to this cluster, then in any time interval Δt, the number of arrivals (N) with value i to the cluster has a probability of:

$$P(N = i) = \frac{(\lambda \times \Delta t)^i}{i!} e^{-\lambda \times \Delta t}.$$

Table 1 lists some terms which are used throughout the rest of the paper.

Term	Meaning
p	The number of nodes in the cluster
μ_i	Processing rate of node i
T_i	Average request processing time on node i without resource contention
I	Average inter-arrival time of requests to the cluster
$Q^j(I)$	Average waiting queue length per node for a j node cluster and average inter-arrival time I
$R^j(I)$	Average request response time for a j node cluster and average inter-arrival time I
$Q_i(I)$	Average waiting queue length on node i for average inter-arrival time I
$R_i(I)$	Average response time on node i for average inter-arrival time I

Table 1: Terms used for performance analysis and measurement

The *expected maximum average response time* for a request is approximated by using the M/M/1 model [21]. By "maximum" we mean that this value is achieved by using a simple static scheduling method which is normally inferior to our dynamic load balancing scheme. This static method assigns requests to server nodes in proportion to their statically determined processing rates. We call this method *proportional scheduling.* Given the processing rates of all the server nodes and the request arrival rate to the cluster, the arrival rate on node i scheduled by such a simple scheduling scheme is

$$\lambda_i = \frac{\mu_i}{\mu}\lambda, \text{ where } \mu = \sum_{i=1}^{p} \mu_i.$$

It should be noted that the distribution of requests assigned to each node by the DNS rotation is almost even and request re-assignment is needed by the above proportional scheduling method in order for each node to get their share proportional to their processing rate. We omit request redirection overhead in the following analysis because for most of the ADL requests, redirection overhead is insignificant compared to their long processing times.

The following results hold on each node for proportional scheduling:

$$T_i = \frac{1}{\mu_i}, \quad Q_i(I) = \frac{T_i}{pI - T_i},$$

$$R_i(I) = (Q_i(I) + 1) \times T_i = \frac{pI \times T_i}{pI - T_i} = \frac{1}{\mu_i - \lambda_i} = \frac{\mu}{\mu_i(\mu - \lambda)}.$$

Thus the expected maximum average response time based on such a scheduling method is

$$R_H(I) = \frac{\sum_{i=1}^{p} R_i(I)}{p} = \sum_{i=1}^{p} \frac{1}{p(\mu_i - \lambda_i)}. \tag{2}$$

Note that μ_i should be larger than λ_i, or, equivalently, μ should be larger than λ. Otherwise the workload is too heavy for the system to handle.

To approximate the optimal load distribution method for estimating the *minimum average response time*, we view the system as an M/M/p queuing system with variant node processing rates by assuming there were a central request queue in the system responsible for feeding requests to all the server nodes. This model optimizes the system performance in the sense that each node always redirects a request to a node with the lowest queue waiting time. Note that this scheduling method can not be implemented in real systems because processing times of requests are unknown until they are processed.

For such a theoretical scheduling model, the minimum average response time can be estimated as:

$$R_L(I) = \frac{1}{\mu_{max}} + \frac{\frac{\rho^C}{C!} \times \frac{\rho \times C}{(\rho - C)^2}}{\sum_{i=0}^{C-1} \frac{\rho^i}{i!} + \frac{\rho^C}{C!} \times \frac{C}{C-\rho}}. \tag{3}$$

where

$$\mu_{max} = \max(\mu_1, \mu_2, \cdots, \mu_p), \quad Z = \frac{\sum_{i=1}^{p} \mu_i}{\mu_{max}}, C = \lceil Z \rceil, \quad \text{and } \rho = \frac{\lambda}{\mu_{max}}.$$

Note that if all the nodes have the same processing rates, Equation 3 reduces to the formula for calculating average response time of an M/M/p queue with uniform processing rates [21].

Equations 2 and 3 will be used in Section 5 to bound the scheduling performance and help us study experimentally if our algorithm is competitive to the optimal scheduling method.

5 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 1'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.

5.1 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 only database queries. Considering that the next version of the ADL testbed system will incorporate a progressive subregion image browsing scheme [5, 23], 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 $S3$. Notice that those 800 subregion requests are not from the current trace and are simulated to represent potential user access patterns.

We used $S1$ as a training set to sample the I/O and CPU demands of database queries on our base node and obtained $\omega = 0.08$, which was used for estimating the SF value discussed in Section 3.1. All experiments were 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 [10, 11]. 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.

Our experiments are reported in the following three subsections. The first subsection examines the scalability of our system from three aspects: speedup, average request response time and system sustained throughput. The second subsection examines the advantages of our hybrid method for node selection and load updating compared to the other strategies. The last subsection 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 I, we also list the expected waiting queue length $Q^p(I)$ on each node in order to give readers some impression of system load.

5.2 System performance scalability

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 polling and multicast.

Response time speedup. Given an average request arrival 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.

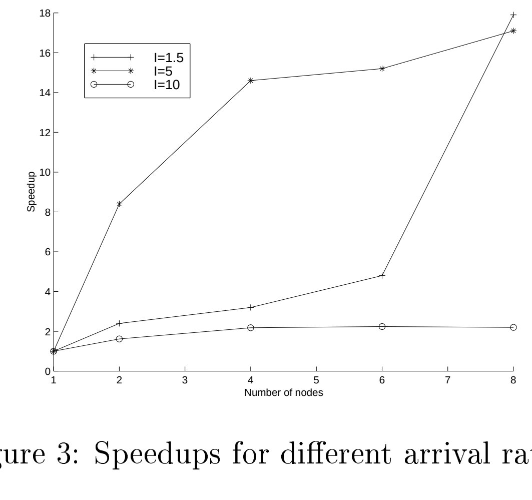

Figure 3: Speedups for different arrival rates.

Figure 3 depicts the speedups under different workloads and different numbers of nodes for processing of 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.

The above performance observation is expected. Our analysis below further points out that for underloaded situations, any reasonable load sharing method should perform well, but for overloaded situations, good optimization can lead to substantial improvements. This is important for a Web site such as ADL, since clients tend to expect the server to sustain its performance, even during the busiest usage periods.

A comparison with theoretical bounds. Table 2 compares the average response times obtained by our scheme with the theoretical maximum and minimum values derived from Equation 2 and Equation 3. The performance data used in this table are based on set $S2$. Let $R_H(I)$ and $R_L(I)$ be the expected maximum and minimum average response times for requests in $S2$ with an average inter-arrival time I. In Table 2, columns marked "H" show the percentage of difference between the measured average response times and the theoretically predicted maximum values. The difference is calculated by $1 - \frac{R^p(I)}{R_H(I)}$. Columns marked "L" in Table 2 show the percentage of difference between the measured average response times and the theoretically predicted minimum values. The difference is calculated by $1 - \frac{R_L(I)}{R^p(I)}$. An "N/A" entry in Table 2 means that the workload under such a situation is so heavy that theoretically the system with the corresponding number of nodes can not handle it if this load persists over a long time. Since test set $S2$ for this experiment only has a limited number of requests, our system is able to handle it, but gives relatively long response times.

We can draw two conclusions from the result in Table 2. 1) When the system load is light (e.g. I=15), the difference between H and L is relatively small and any reasonable method would perform well in such a situation. When the system load is heavy (e.g. I=1.5), the difference between H and L is large and good optimization can lead to substantial performance improvement. 2) The performance of our scheme is somewhere close to the middle of the lower and upper bounds. Notice that the optimal solution approximated in Section 4 assumes that the processing time of each request is known in advance, which is not feasible in practice. Thus there may be some room for further improvement of our scheme in heavy load situations; however we do not expect a large improvement.

I=	1.5		2.75		5		10		15	
	H	L	H	L	H	L	H	L	H	L
p=2	N/A	N/A	N/A	N/A	N/A	N/A	36%	50%	22%	13%
p=4	N/A	N/A	N/A	N/A	34%	40%	18%	19%	13%	12%
p=6	N/A	N/A	54%	49%	20%	34%	14%	13%	9%	9%
p=8	70%	56%	31%	32%	17%	15%	7%	7%	3%	7%

Table 2: Response time comparison with the theoretical maximum and minimum values.

System throughput. Figure 4 shows the ideal and sustained throughput of the system when the number of nodes increases. The throughput is defined as the number of requests processed per second. We compute the ideal and sustained throughput as follows:

- We measure the average processing rate 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:

$$\sum_{i=1}^{p} \mu_i.$$

- The sustained 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 set to be $2/\bar{\mu}$ where $\bar{\mu} = \sum_{i=1}^{p} \mu_i/p$. The sustained throughput of a specific number of nodes is measured 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 sustained throughput of our system, without sacrificing average response time, is very close to the ideal throughput. This indicates that our load balancing strategies are very effective.

5.3 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 polling and multicast. We compare this strategy with the “load broadcast”,

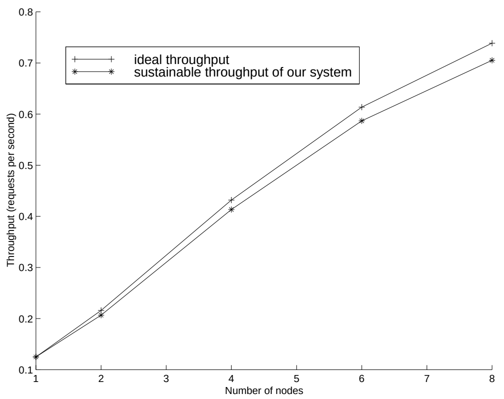

Figure 4: The ideal and sustained throughput of the system.

"random polling", and "random multicast" strategies. The improvement ratios of our method over the others are listed in Table 3 for processing request set $S2$. The results for $S3$ are similar.

$Q^p(I)$	0.5	1	2	3
$I(second)$	4.5	3	2.2	2
DNS even assignment	48%	40%	38%	39%
Broadcast	17%	26%	31%	25%
Random polling	0%	4%	17%	20%
Random multicast	8%	11%	7%	5%

Table 3: 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 $Q^p(I) = 2$ and 3, the performance of "Random multicast" is pretty close to ours. When the workload is relatively light, with $Q^p(I) = 0.5$ and 1, "Random polling" performs reasonably well. The overall performance of our hybrid method is the best since it acts like "random multicast" for heavy workloads, and like "random polling" for light workloads.

5.4 Consideration of I/O and CPU load indices

We investigate the effectiveness of our SF 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.

I (second)	1.5		2.125		2.75		3.375	
$Q^p(I)$	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 4: Improvement ratios using our parameter over other simpler strategies for sets $S2$ and $S3$.

- 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.
- 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 4 lists the improvement ratios, using 8 nodes, for request sets $S2$ and $S3$. 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. For $S3$, however, there are mixed computation and I/O activities, so the advantage of our method becomes more clear.

6 Concluding remarks and related work

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 load collection method which is more efficient than other existing methods, in this context. Our future work is to further evaluate and extend our scheme, e.g. deal with non-uniform memory sizes and updating weight value ω on-the-fly.

A number of projects [4, 16, 8] have proposed methods for more fairly distributing load among a group of servers, based on HTTP redirection or intelligent DNS rotation. The main weakness of a DNS-based server cluster is that the IP addresses of all nodes in a cluster are exposed to the Internet. When one server goes down for administrative reason or due to machine failure, a client may still attempt to access this machine because the DNS server is unaware of the server's status, or because the IP address of this machine is cached at the client's site. Another technique for clustering Web servers with load balancing and fault tolerance support is load balancing switching products from Cisco, Foundry Networks and F5Labs [7, 13, 12]. Those switches offer the single-IP capability, but they use simple load balancing schemes which are not sufficient for resource-intensive dynamic content. Another project in [9] proposes a routing solution with single IP. Our system can take advantage of the single IP approach to distribute load internally instead of using DNS rotation and re-scheduling can be conducted after this initial request distribution.

Previous work on general distributed load balancing [25] normally considers one load index. Our earlier SWEB work [4] uses multiple resource load indices and request redirection. The techniques proposed in this paper improves SWEB by using low-overhead remote dynamic content processing instead of request redirection. Also our new scheme employs simple cost approximation for request scheduling while SWEB requires exact cost prediction, which is not feasible for general Web requests such as database searching. Previous work in distributed system scheduling [10, 11] proposed sender-initiated or receiver-initiated methods to balance load by job migration. In our setting, once the scheduler makes a node assignment decision for an incoming request, this request is processed by that node without migration. This is done because migration causes significant overhead in process transferring.

Web caching and load sharing for static contents at a server cluster is considered in [22], though dynamic content processing is not their focus. Scalable Web service was studied recently in [16]; however they do not address optimization techniques for server clustering and they focus on caching dynamic content. The Swala system [15] addresses caching of dynamic content on a cluster based on the ADL trace data. These caching techniques are complementary to our load sharing strategies.

Acknowledgments

This work was supported in part by NSF CCR-9702640, IIS-9817432, IRI94-11330, and UC MICRO/SUN. We would like to thank anonymous referees for their helpful comments.

References

[1] America Online, Inc. *About AOL Server*. http://www.aolserver.com/server/index.html, 1995.

[2] E. Anderson, D. Patterson, and E. Brewer. The Magicrouter, an Application of Fast Packet Interposing. *Second Symposium on Operating System Design and Implementation (OSDI96)*, 1996.

[3] D. Andresen, L. Carver, R. Dolin, C. Fischer, J.Frew, M. Goodchild, O.Ibarra, R. Kothuri, M. Larsgaard, B. Manjunath, D. Nebert, J. Simpson, T. Smith, T. Yang, and Q. Zheng. The WWW Prototype of the Alexandria Digital Library. *Proceedings of ISDL'95: International Symposium on Digital Libraries*, Aug. 1995.

[4] 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.

[5] 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.

[6] J. Challenger, P. Dantzig, and A. Iyengar. A Scalable and Highly Available System for Serving Dynamic Data at Frequently Accessed Web Sites. In *Proceedings of the Supercmputing'98*, Nov. 1998.

[7] CISCO. *Local Director.* http://www.cisco.com/warp/public/751/lodir/index.shtml, 1997.

[8] 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.

[9] O. P. Damani, P. Chung, Y. Huang, C. Kintala, and Y.-M. Wang. One-IP: Techniques for Hosting a Service on a Cluster of Machines. In *Proceedings of the Sixth Int. World Wide Web Conference*, Apr. 1997.

[10] 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.

[11] 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.

[12] F5Labs. *BigIP.* http://www.f5.com/, 1997.

[13] Foundry Networks. *ServerIron Server Load Balancing Switch.* http://www.foundrynet.com, 1998.

[14] E. Fox, R. Akscyn, R. Furuta, and Leggett. Special Issue on Digital Libraries. *CACM*, Apr. 1995.

[15] V. Holmedahl, B. Smith, and T. Yang. Coperative Caching of Dynamic Content on a Distributed Web Server. *Procecedings of the Seventh High Performance Distributed Computing*, July 1998.

[16] A. Iyengar and J. Challenger. Improving Web Server Performance by Caching Dynamic Data. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Dec. 1997.

[17] R. Jain. *The Art of Computer Systems Performance* . Academic Press, 1992.

[18] E. Katz, M. Butler, and R. McGrath. A Scalable HTTP Server: the NCSA Prototype. *Computer Networks and ISDN Systems*, 27:155–164, 1994.

[19] R. Lipton and J. Naughton. Query Size Estimation by Adaptive Sampling. *J. Comput. Syst. Sci.*, 51:18–25, 1995.

[20] NCSA. *Common Gateway Interface*. http://hoohoo.ncsa.uiuc.edu/cgi/.

[21] R. Nelson. *Probability, Stochastic Processes, and Queueing theory*. Spring-Verlag, 1995.

[22] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network Service. In *Proceedings of ASPLOS-VIII*, Oct. 1998.

[23] A. Poulakidas, Srinivasan, O. Egecioglu, O. Ibarra, and T. Yang. A Compact Storage Scheme for Fast Wavelet-based Subregion Retrieval. *Proc. of 1997 International Computing and Combinatorics Conference (COCOON)*, Aug. 1997.

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

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

[26] T. Smith. A Digital Library for Geographically Referenced Materials. *IEEE Computer*, 29(5):54–60, 1996.

[27] S. Zhou. A Trace-driven Simulation Study of Dynamic Load Balancing. *IEEE Trans. Softw. Eng.*, 14(9):1327–1341, Sept. 1988.

[28] H. Zhu, B. Smith, and T. Yang. A Scheduling Framework for Web Server Clusters with Intensive Dynamic Content Processing. Technical Report TRCS-98-29, CS Dept, UCSB, Nov. 1998.

[29] H. Zhu, B. Smith, and T. Yang. Scheduling Optimization for Resource-Intensive Web Requests on Server Clusters. In *the Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'99)*, June 1999. Available from http://www.cs.ucsb.edu/research/rcgi.