A Scheduling Framework for Web Server Clusters with Intensive Dynamic Content Processing

Huican Zhu, Ben Smith, Tao Yang
Deptartment of Computer Science
University of California
Santa Barbara, CA 93106

Abstract

Clustering support with a single-system image view for large-scale Web servers is important to improve the system scalability in processing a large number of concurrent requests from Internet, especially when dynamic content generation using CGI or other protocols becomes increasingly popular. This paper studies a two-level scheduling framework with a master/slave architecture for clustering Web servers. Such an architecture has advantages in dynamic resource recruitment, fail-over management and it can also improve server performance compared to a flat architecture. The key methods we propose to make this architecture efficient are the separation of static and dynamic content processing, low overhead remote execution, and reservation-based scheduling which considers both I/O and CPU utilization. This paper provides a comparison of several scheduling approaches using experimental evaluation and analytic modeling. The results show that proper optimization in resource management can lead to over 65% performance improvement for a fixed number of nodes, and can achieve more substantial improvement when considering idle resource recruitment.

1 Introduction

There are two types of information serviced at a Web site: static data, whose service is implemented as a simple file fetch, or dynamic data, which involves information construction at the server before users' requests are answered. Recently more Web sites generate dynamic content because it enables many new services such as electronic commerce, database searching, personalized information presentation, and scientific/engineering computing [7, 23]. Since dynamic content generation places greater I/O and CPU demands on the server, the server bottleneck becomes more critical compared to the network bottleneck and it limits the scalability of such servers in processing large numbers of simultaneous client requests. Examples of such systems can be found in IBM's Atlanta Olympics Web server and the Alexandria Digital Library system [3, 4, 20].

Clustering with a single-system image view is the most commonly used approach to increase throughput of a Web site. With a server cluster, multiple servers behave as a single host, from clients' perspective. NCSA [22] first proposed a clustering technique that uses DNS rotation to balance load among cluster nodes. Research has demonstrated that DNS round-robin rotation does not evenly distribute the load among servers, due to non-uniform resource demands of requests and DNS entry caching. A number of projects [5, 9] 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 the IP address of this machine is cached at the client's site. Hiding server failures is critical, especially when global customers depend on a Web site to get vital updated news and information. To address this issue, one solution is to maintain a hot-standby machine for each host [35]. Another technique for clustering Web servers with load balancing and fault tolerance support is load balancing switching products from Cisco, Foundry Networks and F5Labs [8, 14, 13]. These switching products assign a single address to a group of servers and distribute incoming requests among the live nodes, therefore effectively preventing users from accessing dead nodes. Switches use simple load balancing schemes which may not be sufficient for intensive dynamic content. Neither DNS nor switch based solutions provide a convenient way to dynamically recruit idle resources in handling peak load.

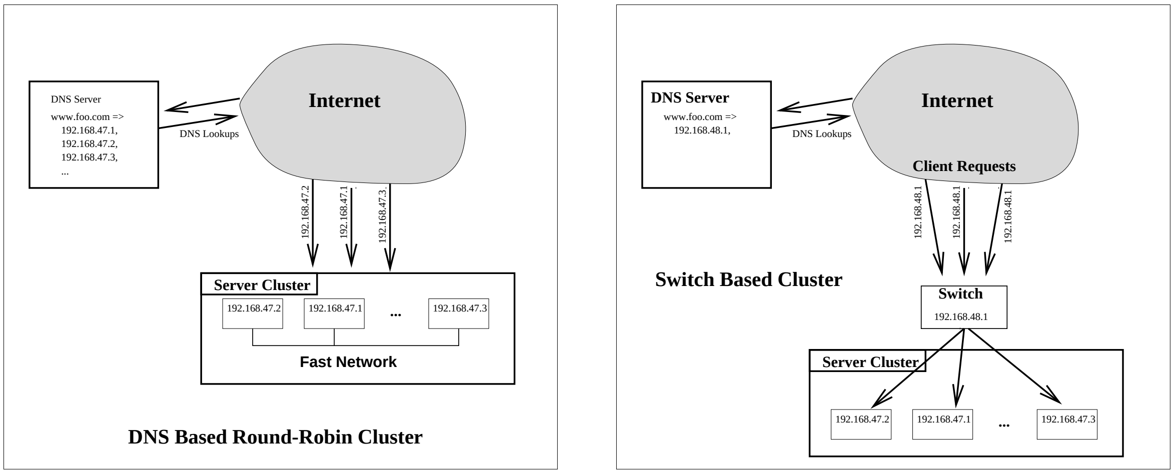
In this paper, we study a scheduling framework with a master/slave architecture for clustering Web servers. The architecture organizes server nodes in two levels. The master level accepts and processes both dynamic and static content requests. The slave level is only used to process dynamic contents upon masters' requests. This layered architecture has advantages in recruiting non-dedicated resources and in fail-over management, when integrated with switching or hot-standby techniques. It is also capable of delivering better performance in terms of system throughput than letting all nodes process both static and dynamic content requests, given the same hardware. The key methods we propose to make this architecture efficient are the separation of static and dynamic content processing, low overhead remote execution, and reservation-based scheduling which considers both I/O and CPU utilization. We illustrate its effectiveness with the CGI protocol and call our scheme remote CGI (RCGI). We show that RCGI can effectively achieve load re-balancing and its overhead is negligible and even smaller than standard local CGI execution. We do not use HTTP redirection [5] for request re-scheduling because it adds client round-trip latency for every rescheduled request and also exposes IP addresses of server nodes. Instances of M/S architecture can be found in current industry Web sites such as Web searching sites at Inktomi [19] and AltaVista [12]. A generalization for network services including proxy servers is made in [15]. Our contribution is to develop a generalized scheme for clustering Web servers with necessary optimization which considers characteristics of Web workloads. Another contribution is that we provide a detailed comparison and evaluation of different solutions, which benefits other Web application domains.

The rest of the paper is organized as follows: Section 2 gives a background overview of Web clustering. Section 3 gives the detailed argument for an M/S architecture. Section 4 provides a theoretical analysis to give insights and conditions when the M/S architecture can outperform a flat architecture. Section 5 presents design details of the M/S architecture. Section 6 describes the workload we use for tests and the experimental results that demonstrate the effectiveness of our scheduling scheme based on RCGI. Section 7 discusses related work and section 8 concludes this paper.

2 Backgrounds and Assumptions

In the World Wide Web environment, clients access an Internet server using the Hypertext Transfer Protocol (HTTP). To create dynamic content in response to an HTTP request, most servers implement the Common Gateway Interface (CGI), created by NCSA [31]. The CGI interface requires that the Web server initialize an environment containing the script parameters, and then fork and execute the script. As a result, every CGI request requires the creation of a new process. To reduce the fork overhead of CGI execution, many Web servers implement a proprietary interface for dynamic content generation. For example, Netscape provides the Netscape API (NSAPI) which allows developers to write a shared library that is loaded into the server's address space when a Netscape server starts to run. Routines from the developer's library are called when requests arrive. One of the primary drawbacks of this approach is that new scripts cannot be loaded without rebooting the entire server. Also the stability of a Web server becomes dependent on the stability of the plug-ins that developers create. A poorly written library can take down the whole Web server with it. Portability is also a concern, because a script written for one vendor's Web server will not work with another vendor's Web server. FastCGI, an extension of CGI developed by OpenMarket, attempts to circumvent some of the limitations of both proprietary interfaces and CGI [33]. A permanent connection is established between the server and the FastCGI application. Individual requests are forwarded from the server to the application, removing the need for process creation for each request. Currently only Apache [6] and Zeus [40] implement this interface. While this paper demonstrates the proposed techniques for CGI due to its popularity these techniques could be applied to other dynamic content generation protocols.

Figure 1(a) shows a DNS-based Web server cluster composed of a set of nodes connected with a fast network. The cluster is presented to the Internet as a single logical server. Each server node has a unique Internet Protocol (IP) address. The name entry for the logical Web server in the DNS system maps to the IP addresses of the set of nodes that define the cluster. A client will pick one address from the set and then send its request to the selected server node. If the DNS server cyclically rotates possible IP addresses in response to IP address queries, then client requests may be distributed [5, 22]. Since IP addresses of a Web site are cached at client sites, clients at one site may actually access the same server node until a cached address expires. Thus load imbalance may be caused by address caching. Moreover, a client site cannot be aware if a machine is no longer in service and may be denied service if that client tries to use a cached IP address to access a dead node.



(a) (b)

Figure 1: Two Web server clustering solutions

A solution for masking server failures is called "failover" using hot-standby techniques [35]. This solution dictates that one computer be used to monitor another one and it is activated to take over if the monitored computer crashes. Industry has sold such a solution for many years and cost of such a solution is expensive since the machine that monitors is not utilized if there is no failure. Another solution as shown in Figure 1(b) is to use recently developed load balancing switching products from Cisco, Foundry Networks, and F5Labs. A switch can cluster a logical community of servers represented by a single logical IP address. These products also provide load balancing schemes to distribute application traffic to server nodes. Typical strategies include: Round Robin, which assigns connections sequentially among servers in the cluster; Least Connections, which assigns a connection to the server with the least number of open connections. In addition to these two load balancing methods, many products have an option available to allow network managers to assign a weight to each server node. Switches also provide sub-second failure detection to eliminate a dead node from the server pool. This ensures that traffic continues to flow and services are still available for processing new client requests.

Clustering based on DNS rotation or load balancing switches still cannot deliver satisfactory performance for Web sites with intensive dynamic content processing for the following reasons:

- Adding nodes to the DNS list or to a load balancing switch requires a manual change of system configuration, which is not convenient and efficient for recruiting non-dedicated computer resources. It is widely recognized that idle computers can be found easily in a corporation or institutional environment [1, 2] and can be recruited to respond to bursty situations.

- The capability of load balancing provided in a switch is still limited because a switch must forward packets as fast possible. The study in [41] shows that scheduling based on the I/O and CPU demands of CGI requests can result in substantial performance gains.

- The class of CGI requests normally require much more computing and I/O resources, compared to static file retrieval requests. Mixing static and dynamic content processing can slow down simple static request processing.

Given a number of nodes plus a set of non-dedicated machines, we are interested in studying how a cluster should be organized to achieve better efficiency in terms of load balancing and high throughput, higher availability in terms of failover management, and easy expandability in handling bursty situations. We assume that a CGI script can run on any node, and nodes do not depend on each other; if one node fails, the failure will not affect other nodes. This can be realized if each server node has its local disk storage system and Web data is replicated to all nodes, or if they use a shared file system. Replication is common in systems requiring high availability [35]. We also assume that a non-dedicated node can be used only when it becomes idle. This policy is used in [1, 27].

If one examines popular Web sites such as Microsoft, Inktomi and Altavista, clustering with a certain degree of data replication is fairly common. In the Inktomi design [19], a few front-end hosts handle user HTTP requests. If these requests are for Internet searches, requests are forwarded and processed by a cluster of SUN Ultra workstations. A similar setup is used at Altavista where DEC SMPs are employed as cluster nodes. A generalization from the Inktomi server to a layered structure is studied in [15] for network services including proxy distillation. Their work is focused on system modeling in organizing a cluster for general network services and there is no detailed study and performance evaluation on issues specific to Web servers. This motivates us to study a scheduling framework with a master/slave architecture for Web server clusters (M/S for short). The questions we are asking are: why is such a design feasible and more beneficial compared to a flat architecture? What are roles of nodes? What software optimization is necessary to support this architecture? How many nodes should be allocated at the master level? Our goal is to develop an optimized cluster architecture with necessary software support to achieve better efficiency, easier expandability, and higher availability, leveraging with single-IP switches or DNS rotation. In this way, Web server software developers and application users can benefit from our studies.

3 Design Justification for an M/S Architecture

An M/S architecture contains two levels. Master nodes sit at level I. They can either be linked to a load balancing switch with a single logical IP address, or they can be attached to hot-standby nodes for fault tolerance, with requests distributed by DNS. Static requests are processed locally at masters. CGI requests may be processed locally at masters, or redirected to a slave node or another master. Slave nodes are at level II and are only used to process CGI requests. They may be non-dedicated and recruited dynamically when they become idle. If a slave node fails, a master node may need to restart a CGI process on another node. Section 5 describes the components of this architecture in more detail.

We contrast M/S with a flat architecture, where all nodes are equal and process both static and CGI requests. In the rest of this section, we argue why M/S suffers no inherent performance penalties compared to a flat architecture, and describe the advantages we can gain through such a configuration.

3.1 Why M/S is feasible

In the M/S model, a cluster has fewer nodes for processing static content requests compared to a flat architecture. This does not limit performance because one or a few static content nodes can saturate the outgoing network bandwidth. According to data submitted to SPEC in the third quarter of 1998, high performance Web servers can process file fetch requests at a rate of up to 13000 requests per second based on the SPECWeb96 benchmark [37]. Because Web traffic for static content is dominated by small file access, the average response size is 15Kbytes in SPECWeb96. This indicates that the delivery rate of such a node approaches 1.6Gbits/second. Considering that most institutions and companies are connected to the Internet with multiple T1 (1.5Mbits/s) or T3 (45 Mbits/second) connections, a high-end Web server (or a few low-end workstations) has sufficient throughput to deliver static content at a rate greater than what the outgoing link can handle. Notice that the NGI initiative [30] (Next Generation Internet) addresses end-to-end connectivity at speeds from 100 Mbps up to 1 Gbps. Although some networks have already achieved OC-12 speeds (622 Mbps) on their backbone links and some experimental links are running at over 1 Gbps, end-to-end usable connectivity is typically limited to less than 10 Mbps because of bottlenecks or incompatibilities in switches, routers, local area networks, and workstations. We expect in next few years the end-to-end bandwidth will increase substantially, but still a few servers should be sufficient to deliver enough data to saturate the outgoing link.

M/S requires that masters forward CGI results from requests executed on slave nodes. This is a concern if result copying creates internal network contention which reduces system throughput. Actually, it is unlikely that this problem would occur in practice. There are three reasons. First, part of the motivation of M/S is to shield slaves from the outside world and the masters' link to the Internet can be configured to be separate from the network that masters and slaves use to communicate. Second, LAN connectivity is typically at least an order of magnitude faster than WAN connectivity. As a result, an application would only consume at most 10% of the LAN bandwidth to saturate the outgoing WAN link. In our implementation, we pipeline the results between the remote server, the master and the client. Flow control places an upper bound on the amount of LAN traffic from CGI requests. In the worst case, given a LAN that is an order of magnitude faster than the WAN link, remote execution of CGIs would increase the LAN traffic by 10%. If the LAN is not already overloaded, the increase should not cause significant contention. Finally, our log analysis of four Web sites in Section 6 shows that their average CGI result sizes are small, varying from 2KB to 7.5KB. Also notice that the average processing rates of CGI requests are one or two orders of magnitude slower than static requests [20]. This suggests that the bandwidth demands of CGI request should be fairly small, and would not cause contention in the cluster network.

3.2 Why M/S can be better

The previous subsection shows that M/S does not inherently suffer any disadvantages over a standard flat architecture. This subsection shows that it can offer significant advantages over a standard DNS or switch based flat architecture.

- **Better expandability.** M/S can easily recruit non-dedicated nodes dynamically in response to load bursts and release them easily because their identity has not been exposed to the outside world. These recruited nodes can appear and disappear dynamically from the cluster, providing a larger pool of resources than if these nodes had to be dedicated.

  If we just use a load balancing switch, we would have to manually monitor, add and delete non-dedicated nodes. If we use DNS-based request routing, a deleted node may still receive requests from clients due to IP addresses caching.

- **Better efficiency.** M/S separates dynamic and static content processing, so long running, resource intensive CGI scripts will not slow down static content processing. Static requests require little processing time, but running them on the same node with dynamic requests may significantly increase their response time. In a UNIX time-sharing environment, quantum based priority and round-robin scheduling is used. Let q be the size of the quantum. Let t be an average server processing time of a static request. And Let g be an average server processing time of a CGI. Assume that there is one static request along with n CGI requests waiting in an OS queue. This static request could wait nq units of time before being processed, compared to nt if all those requests were static. Typically, t is in an order of one millisecond since most static requests access small files, q is 0.01 or 0.1 second [26, 32], and g is in an order of 0.1 or few seconds. If n=100, a static request could wait $100q$ seconds, which is an order of seconds before delivery. In today's Web sites, pages are increasingly complicated, containing a number of small files (e.g. buttons, icons, images) per page. Minimizing latency for simple file access at server sites becomes important for such Web pages in reducing load time. Another disvantage of mixing CGI with static content processing is that resource-intensive CGI requests tend to use a large amount of memory, which decreases space available for file system caching, further decreasing static request performance.

One more advantage M/S has is that a master can assign a CGI request to a node based on an optimized scheduling decision related to the characteristics of the CGI, and resource availability in masters, dedicated slaves, and non-dedicated slaves.

- **Better availability.** Fault tolerance for CGI scripts is also cheap to implement. If a slave server dies during CGI execution, the master can mask this failure by restarting the request on a different slave node. This advantage is discussed in [15] for their layered network service. If DNS is used to direct requests to masters, the M/S setting reduces the number of server IP addresses exposed to clients compared to a flat architecture, thereby reducing the number of nodes that need to be monitored by hot-standby backup nodes. The IP addresses of dynamically recruited nodes are never exposed, which does not create a concern for failover management. If a switch-based solution is used, a switch can only detect the failure of a master and will not route *future* requests to this node. Using the M/S scheme, failure of a *current* CGI execution can also be recovered. Thus M/S adds fault-tolerance beyond a switch or DNS based solutions.

The need for proper design and optimization. The main need for additional Web server nodes in a cluster is to improve performance of CGI-type dynamic content generation. M/S is designed to meet such a need, while maintaining competitive performance for static content requests. The experiment results in Section 6 show that without proper resource management, M/S can perform worse than an optimized flat architecture for a fixed number of nodes (without considering fault tolerance and idle resource recruitment). The remaining issues that need to be addressed are:

- Given p dedicated nodes in a cluster, how many nodes should be assigned as masters?
- Will remote CGI execution add significant overhead compared to local execution?
- How should requests be scheduled to proper masters and slaves so that overall response performance can be better than the flat architecture?

4 Analytic Modeling

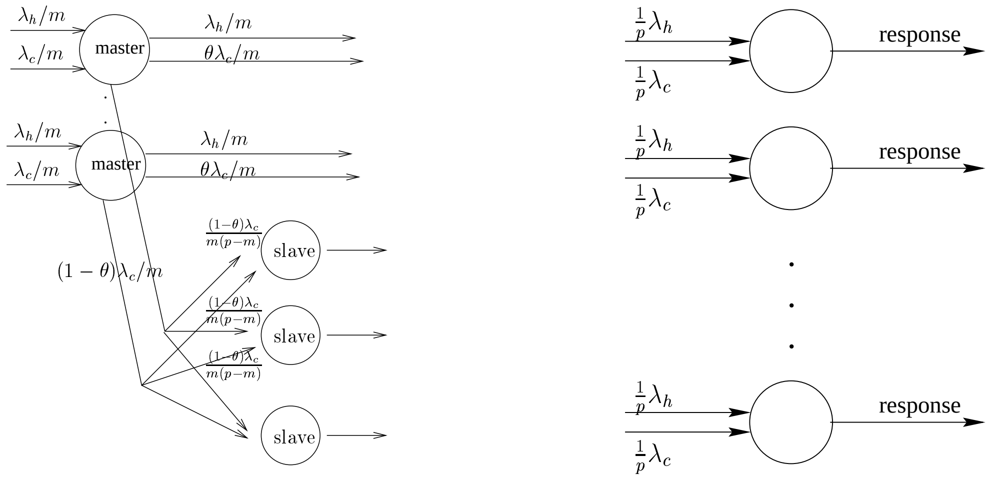
We have conducted analytic modeling using queuing theory and have derived conditions under which the M/S architecture outperforms the flat architecture. Although the model we use here is relatively simple and certain practical aspects are ignored, our objective is to gain insights that can be used in the next section to guide the design of our scheduling policy and determine the number of master nodes that should be allocated.

We model the M/S and flat architectures as multiple class, open queuing networks. The two classes of customers to these two queuing systems are the static request class and the CGI request class. The servers in the two queuing networks are assumed to be homogeneous. Figure 2 depicts the two queuing network models. p is the number of dedicated servers in each cluster. m is the number of master nodes in the M/S architecture. And θ is the percentage of CGI requests processed locally at master nodes.

For the flat architecture, requests are randomly dispatched to nodes in the cluster with a uniform distribution. For the M/S system, requests are first randomly distributed among master nodes. The master nodes will process all static requests and redirect a portion of the CGI requests to slave nodes. We assume that the overhead for executing the remote CGI is negligible, which is justified in Section 5.

Workload characterization. The following terms describe the workload:

- λ_h, λ_c are the mean arrival rates of the two classes of customers, namely, static and CGI requests. For the flat architecture, requests are assumed to be evenly routed to nodes and the mean arrival rates of static requests



(a) The M/S model. (b) The flat model.

Figure 2: Two clustering models.

and CGI requests to each server are λ_h/p and λ_c/p respectively. We use $\lambda = \lambda_h + \lambda_c$ to represent the total request arrival rate.

- μ_h and μ_c are the average service rates of static and CGI requests on each server. Note that service demand of a request is defined as the processing time of a request without resource contention from other requests. Service rate is defined as the reciprocal of service demand.

- Let $r = \frac{\mu_c}{\mu_h}$, and $a = \frac{\lambda_c}{\lambda_h}$. For most Web servers with extensive dynamic content generation, it is expected that $r << 1$.

To analyze these two models, we do not require specific request arrival patterns or request processing time distributions. For ease of analysis, we assume that the request queue length seen on average by an arriving customer must be equal to the time averaged queue length [25]. This assumption is satisfied by requests following Poisson distributions for arrival rates and for service times with exponential distribution. Requests can be processed in the First Come First Serve (FCFS) manner or time sharing manner. The results of this analysis are used in our system design (see next section). In section 6 we will examine the usefulness of such a design in processing traces which may violate these assumptions.

Stretch factors for evaluation. We use stretch factor as the major performance comparison metric. Stretch factor is defined as the ratio of response times of a sequence of requests over the service demands of these requests. More specifically, if request response times are $t_1, t_2, \cdots, t_n$, and their actual service demands are $d_1, d_2, \cdots, d_n$, then the average stretch factor is

$$\frac{\sum_{i=1}^{n} t_i/d_i}{n}.$$

Stretch factor is more suited for describing system performance than average response time because it relates a customer's waiting time to service demand. In a system with highly variable task sizes, users may be willing to wait longer for large tasks to complete, but expect that small tasks should complete quickly. Reducing average response time does not by itself insure that this will be the case[10]. Furthermore, stretch factor reveals the load of a system. A system with a high stretch factor is obviously overloaded, but one with high response time may not be so because the high value may be due to actual long task service demand.

We further define the following terms:

- Let S_F be the stretch factor in the flat model. $S_{F,c}$ is the stretch factor for CGI requests only. $S_{F,h}$ is the stretch factor for static requests only.
- Let S_M be the stretch factor in the M/S model. $S_{M,c1}$ is the average stretch factor for CGI requests processed at the master nodes. $S_{M,c2}$ is the average stretch factor for CGI requests processed at slave nodes. $S_{M,h}$ is the average stretch factor for static requests only.

We are interested in choosing appropriate θ's and m's such that $S_M \leq S_F$. By applying the queuing network analysis technique, the following equations can be established for the M/S and flat architectures:

$$\begin{array}{rclrcl} \frac{\frac{1}{\mu_h}}{1-\frac{\lambda_h}{m\mu_h}-\frac{a\theta\lambda_h}{mr\mu_h}} & = & S_{M,h} \times \frac{1}{\mu_h}, & & & \\ \frac{\frac{1}{r\mu_h}}{1-\frac{\lambda_h}{m\mu_h}-\frac{a\theta\lambda_h}{mr\mu_h}} & = & S_{M,c1} \times \frac{1}{r\mu_h}, & \frac{\frac{1}{r\mu_h}}{1-\frac{a(1-\theta)\lambda_h}{r(p-m)\mu_h}} & = & S_{M,c2} \times \frac{1}{r\mu_h}, \\ \frac{\frac{1}{\mu_h}}{1-\frac{\lambda_h}{p\mu_h}-\frac{a\lambda_h}{rp\mu_h}} & = & S_{F,h} \times \frac{1}{\mu_h}, & \frac{\frac{1}{\mu_c}}{1-\frac{\lambda_h}{p\mu_h}-\frac{a\lambda_h}{rp\mu_h}} & = & S_{F,c} \times \frac{1}{\mu_c}. \end{array} \tag{1}$$

S_M and S_F can be computed as:

$$\begin{array}{l} S_M = \frac{\lambda_h S_{M,h}+\lambda_c\theta S_{M,c1}+\lambda_c(1-\theta)S_{M,c2}}{\lambda_h+\lambda_c} = \frac{(1+a\theta)S_{M,h}+a(1-\theta)S_{M,c2}}{1+a}, \\ S_F = \frac{\lambda_h S_{F,h}+\lambda_c S_{F,c}}{\lambda_h+\lambda_c} = S_{F,h} = S_{F,c}. \end{array} \tag{2}$$

It is quite complicated to derive the range of θ such that $S_M \leq S_F$ using the above equations. However, We were able to observe that if we let $S_F = S_{M,h} = S_{M,c2}$, then $S_M = S_F$. Moreover, condition $S_F = S_{M,h} = S_{M,c2}$ can be satisfied by setting $\theta = \frac{m}{p} - \frac{r(p-m)}{ap}$. Equipped with this result, we can further derive the following theorem.

Theorem 1 *Let $\rho = \frac{\lambda_h}{\mu_h}$, $\theta_2 = \frac{m}{p} - \frac{r(p-m)}{ap}$, and $\theta_1 = \frac{m(a+1)+a\rho/r-a\rho-p}{ap-a\rho+a\rho/r}$. If $m \geq \frac{rp}{a+r}$ and $r < 1$, then $\theta_1 \leq \theta_2 \leq 1$, $0 \leq \theta_2$ and $\forall\theta \in [\theta_1, \theta_2]$, $S_M \leq S_F$. The minimum value of S_M is achieved when $\theta_m = \max((\theta_1+\theta_2)/2, 0)$ and the best m can be derived by numerically computing the following minimization problem:*

$$\min\{\frac{1+a\theta_m}{1-\frac{\rho}{m}-\frac{a\rho}{rm}} + \frac{a(1-\theta_m)}{1-\frac{a(1-\theta_m)\rho}{(p-m)r}}\}, \quad 1 \leq m \leq p.$$

The proof for Theorem 1 is given in Appendix A.

Theorem 1 gives a condition when M/S outperforms than the flat architecture. It also gives a means for determining the best number of master nodes if average request arrival and processing rates can be sampled in advance. We will apply these results in our system deign and will evaluate its effectiveness and sensitivity over prediction errors in Section 6. The above result can also be extended for a heterogeneous system with non-uniform nodes.

5 Component design of M/S

The components of the M/S architecture are depicted in Figure 3 and are discussed in detail as follows:

- **Master nodes.** In addition to the components needed for a regular Web server such as request parsing, we add two modules to a master node: *load manager* and *RCGI manager*. The load manager collects CPU and I/O load information from CGI slave nodes and other master nodes, and checks liveness of those nodes. A node is considered *dead* if the load manager has not received load information or a heartbeat message from that node after a period of time, or if it receives an "unavailable" message from that node.

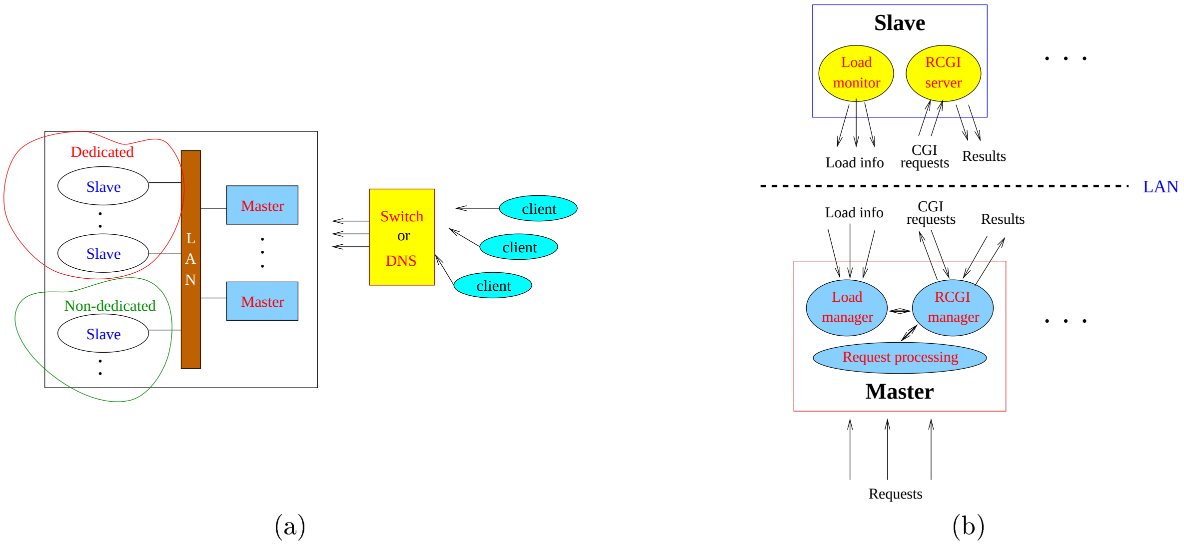


Figure 3: (a) The M/S architecture. (b) Its components.

When a client request is received by a master, it is parsed to see if it is a static request. If it is, the requesting file is retrieved and sent back to the client. Otherwise, the RCGI manager takes over, and queries the load manager to select the best server node. If processing should be done remotely, the RCGI manager sends a request to a remote CGI node, waits for the result, then forwards this response to the client. If the result does not come back from a remote node and this node is considered "dead", a new RCGI node is selected to execute the same request.

- **Slave nodes.** Each CGI slave server runs two daemons: *load monitor* and the *RCGI server*. The load monitor periodically broadcasts its load to the load manager on master nodes, and also informs them that this node is available for processing. The load index of a node includes the percentage of CPU idle time and available disk bandwidth. If this slave node is non-dedicated, the load monitor on this node also monitors activity from the mouse or keyboard and sends a "node unavailable" message to master nodes when any activity is detected.

  The RCGI server waits for CGI requests forwarded from masters. Upon receiving a CGI request, it forks a child to process the request and sends the result back to the master.

Our current implementation of M/S is based on the freely available Apache Web server version 1.3 source code and the Swala server with cooperative content caching support [18, 6]. To prevent malicious requests from being sent to the RCGI servers, access control and identity verification may be done by each RCGI server for each CGI request. This security check is not currently implemented in our testbed.

In the rest of this section we discuss the design and implementation of RCGI and scheduler.

5.1 An efficient implementation of RCGI

RCGI 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 the master node decides to execute a CGI request at a slave node it opens a TCP connection to the slave node and passes the script name and a constructed environment. The slave node forks a new process to handle this request. The forked process redirects I/O to the TCP connection and executes the script with the environment as a parameter. The master 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 master from the RCGI server, flow control within TCP places an upper bound on the amount of LAN traffic, preventing the RCGI server from sending more data than the outgoing link can handle and thereby bounding the amount of local network traffic. The master will not read data from the RCGI server faster than it can send data to the client. This pipelining also limits the network cost of the remote CGI to the connection setup and network latency.

Using our implementation, a remote null CGI execution can actually be faster than local execution with a standard Web server CGI implementation. Fork/exec are expensive and their run time depends on the size of the forked process, as well as the number of threads running. For example, on a 143Mhz Sun Ultra-1 running Solaris 2.5.1, the fork/exec call sequence with a negligible amount of memory allocated and no additional threads takes 32ms, on average. However, when the amount of memory allocated is 10 megabytes and the forking process has 20 threads running (a typical Web server configuration), the fork/exec call sequence takes 43ms. [1] The RCGI server is small, and has only one thread running, so it has the potential to more efficiently execute CGI requests. The added cost of executing remotely is TCP connection setup and network (and protocol) latency. Connection setup for TCP on the Sun Ultra-1 takes 0.852ms and TCP latency is 0.280ms on 100Base-T Ethernet, so the network will add approximately 1.5ms [2] to the execution time for a remote CGI invocation. [29] This is still faster than running the CGI locally. Experiments in section 6 support our claim that remote CGI can be more efficient than local CGI because of running thinner Web servers as slaves.

5.2 Prediction and Reservation based Scheduling

Our scheduler uses a prediction based model to determine which node is the best to run a CGI request, subject to resource reservations at master nodes to accommodate future static content requests. In selecting the best node for CGI processing, we use periodically-updated I/O and CPU load information to estimate the expected cost for processing a CGI request and select a node with the minimum cost. This is important since I/O and CPU demand for different request types can vary significantly. We also reserve a certain amount of CPU and I/O for static content processing on each master node. In this way, master nodes are not overloaded with CGI processing and simple, static content requests can be processed promptly. Our theoretical analysis in Section 4 provides a guideline in controlling the percentage of CGI requests that should be processed at master nodes.

Without knowing the CPU and I/O demand of each request, it would be hard to use I/O and CPU load information. Also, for some CGI requests such as database searching it is hard to provide an accurate formula to predict the cost. We use the following formula to estimate the *relative server-site response cost* (RSRC) of each CGI request on a node in a homogeneous cluster:

$$RSRC = \omega * \frac{1}{CPUIdleRatio} + (1-\omega) * \frac{1}{DiskAvailRatio}. \tag{3}$$

The meanings of these terms in Equation 3 are as follows. ω is the average percentage of the cost contributed by the CPU for a request. $1-\omega$ is used to approximate the average percentage of the cost contributed by I/O. ω is obtained by off-line sampling, approximating the I/O and CPU demand of the request on an unloaded system. If a value for ω cannot be obtained, we assume $\omega = 0.5$. $CPUIdleRatio$ 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 change dynamically based on the system load 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.

[1] Note: Solaris 2.5.1 does *not* do copy-on-write for fork() and fork1(). vfork() implements copy-on-write, but Sun claims that vfork() is not MT-safe. Other operating systems (for example, Linux) may use copy-on-write for fork, and will not see this performance difference. Informal experiments on Linux support this conclusion.

[2] The complete network cost would be the cost of connection setup (0.852ms), plus the cost of the initial request to the client ($tcp.latency + msgsize/bandwidth = 0.280ms + 1024bytes/9.5MB/s = 0.382ms$), plus the latency of the return request ($tcp.latency = 0.280ms$), for a total network overhead of 1.51ms.

It should be noted that disk caching is not considered in our model because it is difficult to model such a factor. However, our prediction provides a reasonable indication of resource demands. If nodes in a cluster are non-uniform, the relative speed in accessing CPU and disk I/O resource needs to be considered. This issue is not addressed in this paper.

Ideally, the percentage of CGI requests processed at masters should be θ_m from Theorem 1 of Section 4. In order to be more effective, this term requires an accurate estimation of service rates of CGI and static requests during a short period of time, which is difficult. In our scheme, we set a limit on the percentage of CGI requests that should be processed at masters. Since the upper bound for M/S to outperform the flat architecture is θ_2 as defined in Theorem 1 of Section 4, we set the reservation ratio at $\theta'_m = \frac{m}{p} - \frac{r(p-m)}{ap}$. During execution, the percentage of CGI requests scheduled to masters may not reach this limit. Note that this limit only depends on the *relative* service rates of CGI and static requests (r), and the relative arrival rates of CGI and static requests (a). We monitor the arrival rates for calculating a. However, it is not easy to provide an accurate online estimation of request service rates. As a compromise, we use current *relative response times* of static and CGI requests to approximate r. The load managers on master nodes update θ'_m periodically by collecting response times of requests on different servers.

Note that the adjustment of θ'_m is self-stabilizing in the sense that θ'_m will converge to a specific value if the system itself is stable, no matter what the initial value of θ'_m is. For example, if the initial value of θ'_m is too low, very few CGI requests will be processed by masters. Then r will decrease, θ'_m will increase and more CGI requests will be executed on masters. When θ'_m becomes too large, too many CGI requests are executed at master nodes, which slows down processing of static requests, thus r will increase which will lower θ'_m. The value of θ'_m is also adjusted according to ratio a. With more static requests compared to CGI requests, the ratio a and θ'_m will also decrease. Thus, more resources are reserved for static processing at master nodes.

6 Experiments

The main purposes of the experiments are to study if the M/S architecture provides better performance over switch or DNS-based flat architectures, demonstrate the effectiveness of recruiting idle resource, and assess the benefits of our several optimization strategies used in this framework. We examine the performance sensitivity to inaccurate parameters for determining the number of masters. We study consistency between the analytic results based on the queuing theory presented in section 4 and the simulation results. We also measure the cost of remote CGI execution.

Currently the number of dedicated cluster nodes available to us to run tests is small. Therefore, we have also developed a simulator which emulates clustered Web servers; we can run experiments on this simulator to examine performance of large clusters in processing heavy server loads. To ensure the quality of simulation, we have also conducted a validation on a SUN cluster to compare simulated and experimental results.

We discuss the workload in our trace-driven experiments, the method to replay the traces, and present experimental results.

6.1 Workload description and evaluation methodology

Table 1 summarizes the main characteristics of the traces we have obtained. The UCB log [17] was gathered from the Home IP service offered by UC Berkeley to its modem pool users. The KSU log was recorded by the Web server for online library service at Kansas State University. The ADL log is from the testbed of Alexandria Digital Library [3], a digital library for spatially referenced data. For these logs we can distinguish dynamic and static requests, and can extract the completion time of each request and the response size. We also have a DEC trace [24] available. For both UCB and DEC traces we cannot recognize URLs and their parameters because they are scrambled for privacy concern. Since the CGI percentage of the DEC trace is similar to the UCB log, we decided not to use that trace. To keep trace replay time reasonable, we extracted a segment of the UCB log for use in our tests. The extracted segment consists of 128668 requests and has a time span of 4 hours.

Log name	Year	No. requests	CGI percentage	Avg. interval	HTML size	CGI size
UCB	1996	9.2 million	11.2%	0.139 sec	7519 bytes	4591 bytes
DEC	1996	24.5 million	8.7%	0.09	8821	5735
KSU	1998	47364	29.1%	18.486	482	8730
ADL	1997	73610	44.3%	22.418	2186	2027

Table 1: Characteristics of four Web traces.

Our tests are performed by resubmitting all the requests in each log to our server cluster. To replay requests, two major issues are considered:

- **Request arrival intervals.** If we issue requests to our server cluster with the same rates as recorded in the logs, the load will be too light for a tested cluster. To show the benefits of clustering in processing heavy loads, we scale intervals among requests so that requests in each log are issued to the cluster at various fast rates to represent different degrees of loads. In this way, we can study the behavior of popular, CGI-intensive Web sites. [3]

- **Replaying an individual request**. If a request is to retrieve a file, replaying it is relatively simple since the recorded result size is the file size. Since our research is targeted at Web sites with intensive CGI processing, variation in static file access patterns will not dramatically affect overall system behavior. We replace all file fetches from the logs with the 40 representative files from SPECWeb96. For each file request in the log, the file in this set with the closest size is returned.

  For CGI requests, replaying is more difficult. For the KSU log, we do not have its proprietary library data and binary, and we cannot determine the request service time from the log data. For the UCB log, we do not know what each scrambled CGI request is doing, and we cannot determine the request service time either. We will need to replace those CGI requests with real operations in order to run experiments. In doing such a replacement, we generate synthetic loads which represent CPU and I/O intensive or mixed situations. For the UCB trace, we use a CGI script from the WebSTONE benchmark [38]. This script receives a file size as a parameter and dynamically generates a file of that size and returns it to the client. We modified this script so that we can control the running time of the script in order to generate different loads by CPU busy-spinning. As a result, these CGI requests are CPU intensive. For the KSU library-searching requests, we created a database with approximately 10000 items using the WebGlimpse software [28] and replaced the CGI library requests with WebGlimpse commands. These requests have mixed CPU and I/O demands, but on average 90% of service time is spent searching index information in memory. For the ADL trace, we replicated a small ADL catalog database in our local cluster and ran an ADL server at each node. This workload is I/O intensive with about 90% of the servicing time consumed by disk accesses.

To study the system performance on a large cluster with fast nodes and a very high arrival rate, we simulate a Web server cluster. We have developed a simulator of a Web server cluster which approximates the behavior of OS management for CPU, memory and disk storage. This simulator consists of a UNIX-like job scheduler, an access scheduler for storage and a virtual memory manager. Each Web request will be provided with service time, I/O and CPU demand distribution. Each request job will be modeled as a sequence of CPU bursts and I/O bursts, submitted to the CPU queue and I/O queue. CPU scheduling is based on the UNIX BSD 4.3 strategy [26]. The process ready queue is a multilevel feedback queue divided into multiple lists according to process priority. Processes are scheduled based on priority and may be preempted following quantum expiration. The I/O queue also maintains a set of I/O processes and is scheduled using round-robin. The memory management maintains a set of free pages

[3]Currently Yahoo receives 115 million page views per day in June 1998 [39] which is around 1331 hits per second. We expect the hit ratios will increase substantially over next few years.

and allocates a number of pages to a new process. For each request, a memory size requirement is provided and the system generates working-set oriented access patterns to stress the demand-based paging scheme. The above setting represents fairly simple OS functions in managing CPU, memory and I/O. Network contention within a cluster is not modeled and the reasons are discussed in Section 3. Other factors such as disk caching are also not considered. We believe that such a simulator is representative of the behavior of modern OS systems in processing Web workloads. In section 6.2.2, we compare simulated results and experimental results. The difference is reasonably small.

6.2 Experimental results

In our experiments, we evaluate the performance using stretch factors defined in Section 4 and compare M/S with the following two solutions where the number of cluster nodes is fixed and no idle resources are available:

- **Flat-R**. User requests are evenly distributed to nodes. This reflects a switch-based flat architecture with round-robin scheduling. It also can be a DNS-based solution in a perfect situation, i.e. clients are uniformly distributed geographically and IP address caching does not create a problem. The study in [9] shows that its performance is near-optimal for a DNS setting, better than the other methods proposed in [9] with this uniform-distribution assumption.
- **Flat-C**. Requests are scheduled to a node with the least number of outstanding connections. This is achieved in our simulation by selecting a node with the minimum queue length.

We also examine the reduction of stretch factors after idle resources are recruited dynamically in M/S. We further assess the effectiveness of the proposed optimization for M/S and isolate improvement ratios due to the use of cost sampling, master reservation, and separation of static and dynamic content processing. Since the number of master nodes in M/S is based on sampled arrival and service ratios of static and dynamic content requests, we also examine the performance sensitivity of our setting when these rates vary dramatically during execution. Another result we present is the consistency between simulated results and the analytic model in Section 4.

6.2.1 Simulation results

Parameter setting. The numbers of nodes in the simulated clusters are 32 and 128 which represent medium and large cluster sizes for current and future Web workloads. Each node has the capability to handle 1200 requests/second for SPECWeb96 based on the results submitted to SPEC from 1996 to 1998. Those rates may be higher than what average workstation servers can deliver; however, these numbers are reasonable for a commercial Web site or for future machines, considering advances in CPU, disk, memory and networking technology. The system overhead charged in the simulation is based on current high-end server performance [29]. The CPU quantum is 10milliseconds. The priority update period is 100 milliseconds. The context switch overhead is 50 microseconds. The fork overhead is 3 milliseconds. The remote CGI latency (excluding fork) is 1 millisecond, representing TCP connection time. The page size is 8KB. The I/O burst for accessing a page is an average of 2 milliseconds. Notice that disk seek time for a sector is longer than 2 milliseconds; however, considering that current storage speed can range from 5 to 80MB/second, with support of file caching and block transfer, it is reasonable to use this average number.

Table 2 lists the workload parameters examined for the three traces. The average ratio of the CGI arrival rates to static request rates (a) is fixed, based on the log data. Arrival rates (λ) are scaled in replaying to reflect various workloads. Arrival rates considered vary from 500 to 8000 requests/second, to reflect a current or future popular Web site. Since KSU and ADL traces have much more CGI activity than UCB, and since 128 nodes can handle four time more load than 32 nodes, the arrival rates we have examined for each trace are listed in Table 2 and such a setting creates reasonable loads for a corresponding cluster to handle. Otherwise, the load would be too light or too heavy. The average ratio of CGI processing rate to static request rate, r, is chosen to be: $1/20, 1/40, 1/80, 1/160$ to represent a wide range of CGI resource demands. This choice is based on the previous studies [21, 18, 41].

	a	Arrival rates	r
UCB	0.12	1K, 2K for 32 nodes. 4K, 8K for 128 nodes.	$\frac{1}{20}, \frac{1}{40}, \frac{1}{80}, \frac{1}{160}$
KSU	0.42	0.5K,1K for 32 nodes. 2K, 4K for 128 nodes.	$\frac{1}{20}, \frac{1}{40}, \frac{1}{80}, \frac{1}{160}$
ADL	0.78	0.5K,1K for 32 nodes. 2K, 4K for 128 nodes.	$\frac{1}{20}, \frac{1}{40}, \frac{1}{80}, \frac{1}{160}$

Table 2: Workload parameters examined.

A comparison of MS, Flat-R, and Flat-C. Figure 4 shows stretch factors of the three schemes as well as the percentage of improvement of MS over Flat-R and Flat-C which is defined as: $(\frac{Strech\_factor(\text{Flat-R or Flat-C})}{Strech\_factor(M/S)} - 1) \times 100\%$. The number of cluster nodes is 32 or 128. The arrival rates for each case are listed in the figure. The number of master nodes for M/S is computed by rate sampling. The figure shows that the improvement of M/S over Flat-C and Flat-R is significant, with a maximum of 66%. The average improvement is about 23% over Flat-C and 36% over Flat-R. The figure also shows that as the ratio r decreases, CGI activity becomes more intensive and the advantage of M/S becomes more clear. This is because as the average processing time of CGI requests increases, optimizing CGI performance becomes more critical. Also, there will be more CGI requests queued at each node, which increases the waiting time of static requests.

Benefits of recruiting idle resources. The load managers of masters monitor a set of non-dedicated resources and recruit them if they are idle. Obviously adding more nodes dynamically can reduce stretch factors dramatically in handling bursty traffic, compared to the Flat-R or Flat-C method. We are interested in studying how such resource addition improves the system performance under different loads. Figure 5 shows the stretch factors for the three traces when the arrival rate λ is 2000. The number of non-dedicated nodes varies from 8 to 32. We assume that those nodes are available 50% of time during the test. The X axis marks the average number of idle nodes recruited. This assumption is consistent with the previous studies on idle machine availability [2, 1]. The result shows that resource recruiting becomes more effective when CGI load is significant ($r = 1/160, 1/80$ for UCB. $r = 1/40, 1/80, 1/160$ for KSU and ADL). With $r = 1/160$ and an average of 16 extra nodes (50% of 32 nodes), the number of total nodes in this cluster increases by 50% and the improvement is 116% for the UCB trace. This shows that adding extra nodes can be very beneficial when there is severe resource contention in a heavily loaded system.

Effectiveness of the proposed optimization. We compare M/S with the following alternative solutions to examine the effectiveness of each optimization strategy used in M/S: 1) **M/S-ns.** No sampling is used to assess I/O and CPU demands. We set $\omega = 0.5$, assuming that I/O and CPU are of equal importance for CGI requests. Performance gain reflects the benefits of demand sampling. 2) **M/S-nr.** No reservation is used to keep a portion of master resources available for static content processing. Performance gain reflects benefits of resource reservation in avoiding overloading of master nodes. 3) **M/S-1.** This setting treats all nodes as master nodes. The scheduling algorithm is not changed. Performance gain reflects the benefits of separating static and CGI requests. Again we report the percentage of improvement: $(\frac{Strech\_factor(M/S-?)}{Strech\_factor(M/S)} - 1) \times 100\%$. Figure 6 shows the results of the comparison. The arrival rates of requests are set to be the same as in Figure 4.

Figure 6 shows that M/S has significant improvements over M/S-nr (up to 68% reduction), which underscores the importance of reserving resources for static requests. The figure also shows that M/S-1 could have up to 26% performance degradation, which reveals the benefits of separating static requests from CGI requests. The performance improvement of M/S over M/S-ns ranges from 5% to 22% and the average is 14%, which shows that sampling characteristics of CGI requests is useful.

M/S-1 can actually be viewed as a flat architecture with remote CGI. If we compare M/S-1 and M/S-nr, it is clear that without resource reservation on master nodes, M/S can perform worse than a properly configured flat architecture. This means that without considering fault tolerance and idle resource recruitment, M/S may not be

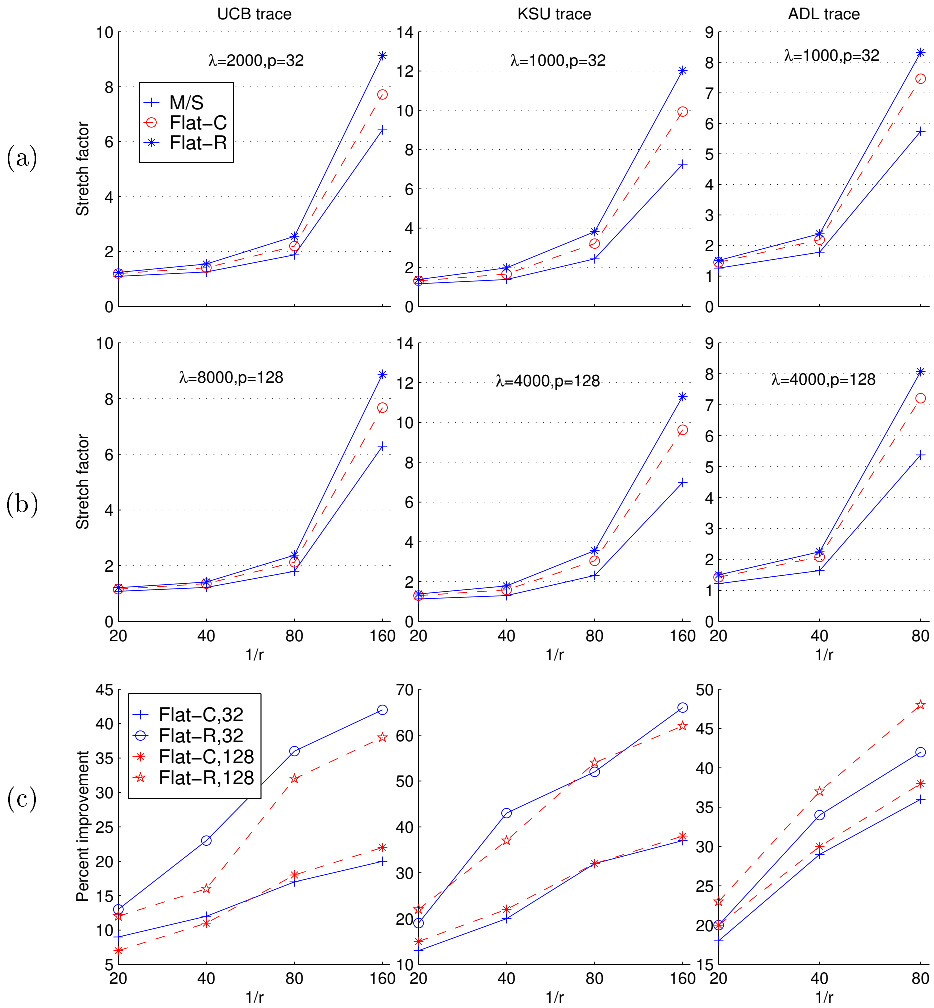


Figure 4: (a) Stretch factors when $p = 32$. (b) Stretch factors when $p = 128$. (c) Percentage of improvement over Flat-R and Flat-C.

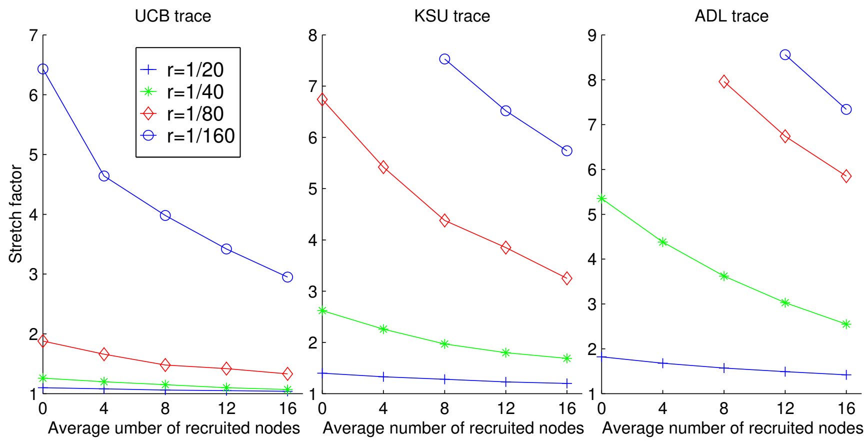


Figure 5: Stretch factors of a 32-node cluster with additional idle resources.

(a)

(b)

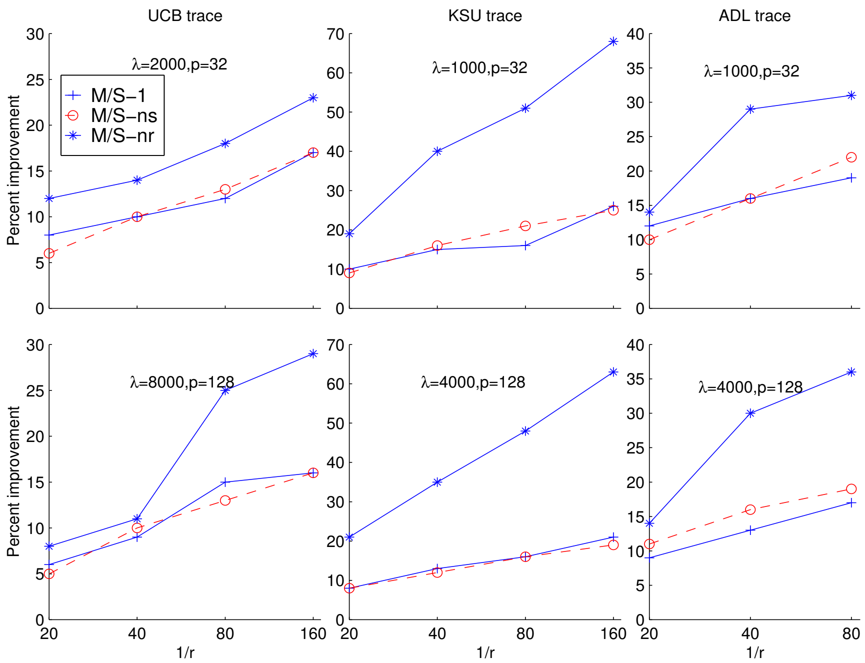


Figure 6: Percentage of improvement using different optimization strategies in M/S. (a) p=32. (b) p=128.

competitive to the flat architecture if optimization is not properly applied.

Sensitivity analysis on number of masters. Our method for determining the number of master nodes is based on sampled average service rates of CGI and static content requests and their arrival rates. Such a number may be stable for a certain period, e.g. on a weekly basis and the number of master nodes can be changed by administrators periodically. If fluctuations within request patterns are very high, parameter estimation may be poor. Ideally M/S should be able to choose the numbers of master nodes based on dynamically changing workload characteristics. In this experiment, we study the system performance sensitivity and fix the number of master nodes m decided by using parameter $r = 1/60$, $a = 0.44$, $\lambda = 750$ for 32 nodes and $\lambda = 3000$ for 128 nodes. The calculated number of masters is 6 for 32 nodes and 25 for 128 nodes. Then we use this configuration to handle the three traces in which r varies from 1/20 to 1/160, ratio a varies from 0.12 to 0.78, λ varies from 500 to 2000 for for 32 nodes, and from 2000 to 8000 for 128 nodes. Figure 7 shows increasing ratios of stretch factors of M/S using the fixed m compared to the one which adapts to runtime parameters change. We can see that the degradation ratio is at most 9% and the average is 4%. This means that performance of M/S with fixed m is fairly robust. M/S with a variable number of masters may be better; however, it requires dynamic configuration change.

A comparison of analytic model and simulation results. The theoretical analysis in Section 4 makes a number of assumptions which differ from the simulation. We are interested in checking the consistency between results derived by the analytic model and by simulation. We set up our experiment as follows. Since in section 4 requests are uniformly distributed among masters nodes and among slave nodes, in simulation we compare Flat-R with M/S which uniformly distributes requests among slave nodes and among master nodes, under reservation constraint. Figure 8 lists the percentage of improvement for the three traces with curves marked as "Simu". We also manually compute the improvement ratio using equation 2 in section 4 and the predicted improvement is depicted in figure 8 marked as "Theo".

The result shows that the analytic result overestimates the improvement. The discrepancy is small when CGI activity is not intensive, but increases when $r = 1/160$. The cause is that the theory cannot distinguish the impact of I/O and CPU activities, and quantum-based OS scheduling, and the request arrival pattern and cost distribution are different from what the three traces have. Nevertheless, the trend of improvement indicated by the analytic model

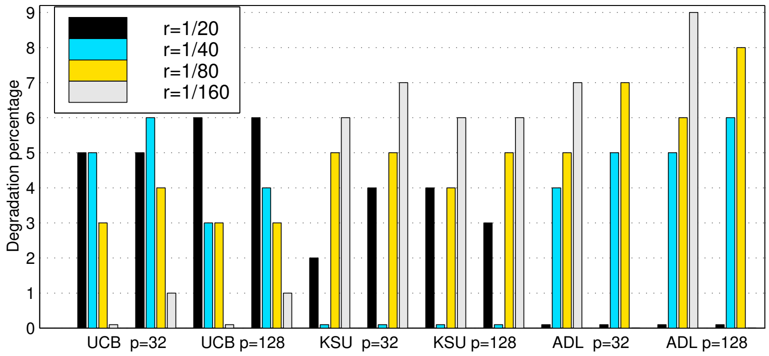


Figure 7: Performance degradation using a fixed number of masters. The arrival rates for the 12 bar groups are 1000/s, 2000/s, 4000/s, 8000/s, 500/s, 1000/s, 2000/s, 4000/s, 5000/s, 1000/s, 2000/s and 4000/s

is consistent with the simulation. As a result, the theory-guided reservation and computation of the master number are still very effective, as demonstrated in the previous experiments.

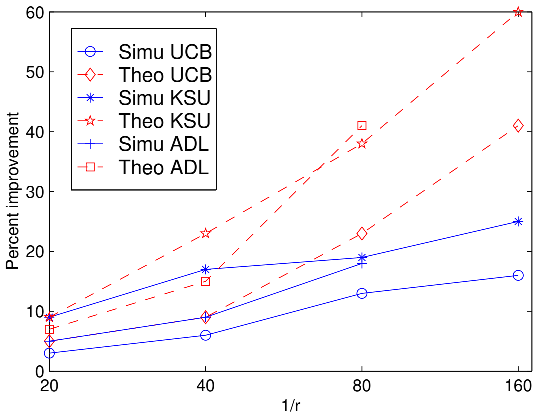


Figure 8: Reduction ratio of stretch factors obtained by simulation and by the analytic model.

6.2.2 Validation on a Sun cluster

We have also implemented and run our M/S Web server prototype on a Sun cluster (Solaris 2.5) containing six Sun Ultra I workstations connected by a Fast Ethernet switch to validate conclusions drawn from simulated results. With SPECWeb96 benchmark, each Ultra 1 node can process static requests at 110 requests/second.

The ADL trace is replayed with a database smaller than what is used at its real site due to storage constraints. The ratio r is about 1/40. For the UCB trace, we ran each CGI using a CPU-intensive script with an average ratio $r = 1/40$. We ran each KSU CGI using a WebGlimpse search script with $r = 1/40$. The arrival rates used are 20 requests/second and 40 requests/second. Requests are sent to servers in a round-robin fashion. The number of master nodes are 3, 1 and 1 for the UCB, KSU and ADL traces, respectively.

Simulated and actual stretch factors. Figure 9 compares the average stretch factors from experiments and from the simulation. The parameters we choose for the simulator are based on experimental specifications and performance indices of the Sun Ultra I [29]. The result shows that the difference between simulated and experimental results is less than 12%, with an average of 9%. This is very small, considering the simulator does not do detailed OS emulation.

Simulated results vs. actual improvement ratios. Table 3 shows the improvement ratio of the M/S method over

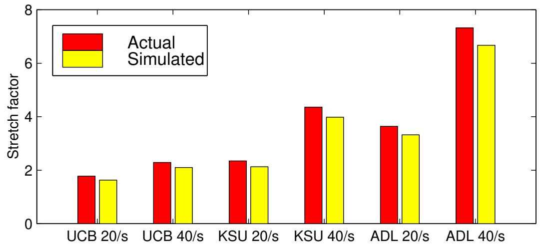


Figure 9: Simulated and actual stretch factors on a 6-node SUN Ultra-I cluster.

other methods derived by simulation and by actual execution. Again, the simulated performance and experimental results are very close for all cases and the average difference is around 3%. Simulated results are slightly optimistic, compared to actual performance. This is because the simulator does not consider background jobs running in the cluster and only captures approximated behavior of Solaris OS 2.5.

	M/S-1		M/S-ns		M/S-nr		Flat-C		Flat-R	
	Actual	Simu	Actual	Simu	Actual	Simu	Actual	Simu	Actual	Simu
UCB $\lambda = 20/s$	07%	05%	08%	09%	14%	12%	12%	14%	18%	20%
UCB $\lambda = 40/s$	12%	10%	13%	14%	18%	19%	18%	20%	26%	29%
KSU $\lambda = 20/s$	10%	12%	09%	12%	23%	27%	22%	26%	28%	31%
KSU $\lambda = 40/s$	17%	20%	19%	18%	32%	35%	30%	33%	41%	45%
ADL $\lambda = 20/s$	12%	13%	13%	15%	24%	27%	20%	22%	24%	27%
ADL $\lambda = 40/s$	17%	19%	16%	19%	37%	40%	27%	29%	35%	39%

Table 3: Performance improvement of M/S over other methods on a SUN cluster by actual running and simulation.

Remote CGI overhead. Finally, we also report an experiment on the remote CGI overhead conducted on a Sun Ultra-1 cluster and an Ultra-30 cluster. For the local CGI test, we started the Web server on one machine, and issued 2000 sequential null-CGI requests to that machine from a client on 100Base-T Ethernet. We measured the elapsed time to process all requests, and divided by the number of requests to arrive at the average response time. For the remote CGI tests, we started an additional RCGI server, and directed all requests from the Web server to the RCGI server. We used the same client and issued 2000 sequential null-CGI requests to the Web server and recorded the elapsed time. Table 4 lists the response times of local and remote null-CGI execution, which includes client-server delay. It also lists the cost of fork and exec. The result shows that local CGI is from 7 to 9 milliseconds slower than RCGI. We attribute this difference to the increased cost of forking a larger Web server. Both the local CGI and remote CGI are quite close to the theoretical minimum for CGI execution, given process size. This suggests that an efficient implementation of local CGI would include a separate process from the Web server to execute CGI scripts, in order to minimize the fork cost.

	Fork(1 thread and 0MB)	Fork(20threads and 10MB)	Local CGI	RCGI
Ultra-1(Solaris 2.5.1)	32.2	41.9	46.0	37.6
Ultra-30(Solaris 2.6)	20.0	27.8	27.9	20.3

Table 4: Response times in milliseconds for remote and local null-CGI execution and overhead of forking.

7 Related Work

The concept of a master/slave structure can be found in earlier parallel programming research [16] where a host and multiple slaves are used to exploit the fork-join type of parallelism. The feasibility and importance of recruiting idle resource in a clustered environment has been studied in parallel computing [2] and distributed job execution [27]. Our research is focused on clustering for Web servers and takes into consideration Web access characteristics. A layered architecture for network service is proposed in [15], which generalizes the structure of the Inktomi's searching server. Their work demonstrates its usefulness in the distillation proxy. Their work is focused on system modeling from the software engineering point of view, and does not have a well-defined environment for system implementation. Our study extends this work by providing a general but implementable solution for Web servers and proposing optimization support in order to make an layered architecture efficient. We provide a detailed study on optimization issues involved in clustering Web servers and performance evaluation comparison with other Web server clustering solutions. Our result in Section 6 supports the necessity of developing these optimization techniques. For example, there can be over 65% performance difference by varying the scheduling strategy for M/S.

In addition to the NCSA solution for Web clustering [22], our earlier SWEB project [5] addresses load balancing on a cluster of nodes and uses the cost prediction to assign requests to nodes using multiple resource load indices. Notice that previous work on general distributed load balancing [36] normally considers one load index. An improvement compared SWEB is that we currently use remote dynamic content processing with low overhead 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. An IBM group [20] proposes several methods to improve the DNS-based solution, by having the DNS server monitor the load of cluster nodes. They found that scheduling optimization is needed because IP address caching creates load imbalance. They also found that round-robin performs well if clients are uniformly distributed. We have compared our framework to the round-robin with uniform client distribution and the results show that for CGI intensive applications, our method can outperform it by as much as 66%, without considering idle resources. Another project in AT&T and Bell lab [11] proposes a routing solution with single IP, similar to the switching techniques used in [8, 14, 13]. They have also used a broadcast approach to make all nodes in a cluster aware of arrivals of each request. If broadcasting is efficient, the performance should be similar to a switching based solution.

Web caching for static contents at a server cluster is considered in [34] and dynamic content processing is not their focus. Web caching for dynamic content is possible if content is not changed frequently and this issue is studied in our Swala Web server [18]. Our work in this paper does not consider CGI caching. We actually have implemented our testbed on Swala and a simple extension to consider caching in our scheme can be incorporated. In terms of architecture design, the central server in [34] may become a performance bottleneck and thus in their scheme, a powerful machine such as an SMP is needed and a TCP/IP handoff protocol is used so that back-end servers can deliver results directly to clients. Using such a setting the dispatcher cannot detect if a back-end node completes a request successfully. Also the handoff protocol requires a change in the OS kernel library of front/back-end nodes and our work focuses on *user-level* system support, which can run on standard (unmodified) operating systems. Furthermore, multiple front-end machines are needed for fault tolerance and then this architecture becomes an instance of M/S.

Request scheduling for static content processing based on file sizes is studied in [10] where short jobs for accessing small file sizes are assigned to the lighted loaded nodes to avoid mixing with heavy jobs which access large files. While this work does not address dynamic contention generation, our work is reminiscent in the sense that short file access jobs are assigned to masters and long CGI jobs are mainly assigned to slaves.

8 Concluding remarks

In summary, we have studied a master/slave scheduling framework integrated with necessary optimization strategies for clustering Web servers. Table 5 summaries advantages of M/S over other existing solutions. This framework requires that the Web server code be modified to support node cooperation, while a switch or DNS based solution can use current Web server software without any change. However with such a change, M/S can gain advantages in resource recruiting and failover management. M/S outperforms the others substantially with the addition of dynamic

	M/S +switch or DNS	Flat+Switch	Flat+DNS
Software change	Yes	No	No
Dynamic recruitment	yes	no	no
Failover management	1	2	3
Performance (stretch factors)	1	2	3

Table 5: A comparison of different solutions. Numerical numbers represent ranking and the lower is better.

idle resources. Even without using idle resources, M/S can still gain up to 38% over a switch-based solution and 66% over the DNS-based solution by separating static content and CGI processing and by employing a reservation-based load balancing scheme which considers both I/O and CPU utilization. Within a two-level framework, proper optimization in resource management is still needed and our experiments show that the proposed methods can achieve over 65% performance improvement compared to the other alternatives when the number of nodes in a cluster is fixed.

The theoretical results that compare a M/S model and a flat model have made a number of simplifying assumptions, but they do provide useful insight on the behavior of two models and guide our scheduler design. The experimental results demonstrate the effectiveness of such a design. The presented results are focused on a homogeneous cluster and we are making an extension for managing heterogeneous nodes.

Our design and experiments assume that data of a Web server is either fully replicated or stored on a shared device and CGI scripts can run on any server. In practice, portions of the data may be replicated and some CGI scripts require specific servers to run. Scheduling design becomes more challenging and application-specific. Nevertheless our experiences demonstrate that multi-level scheduling with proper optimization for Web server clustering can maximize efficiency, expandability, and availability.

Acknowledgments

This work was supported in part by NSF CCR-9702640, CDA-9529418, IRI94-11330, and UC MICRO/SUN. We would like to thank Anurag Acharya for his helpful comments and suggestions, Dan Andresen and Tim McCune for providing us the KSU trace, and Vegard Holmedahl for implementing the Swala Web server.

A Proof of Theorem 1

Our goal is to find a range of θ which satisfies $S_M \leq S_F$. Using Equation 2, inequality $S_M \leq S_F$ can be transformed as:

$$(1+a\theta)S_{M,h} + a(1-\theta)S_{M,c2} \leq (1+a)S_{F,h}. \tag{4}$$

Using Equation 1, the above inequality can be furthered reduced to

$$(1+a\theta)\frac{1}{1-\frac{\lambda_h}{m\mu_h}-\frac{a\theta\lambda_h}{mr\mu_h}} + a(1-\theta)\frac{1}{1-\frac{a(1-\theta)\lambda_h}{r(p-m)\mu_h}} \leq (1+a)\frac{1}{1-\frac{\lambda_h}{p\mu_h}-\frac{a\lambda_h}{rp\mu_h}}. \tag{5}$$

By eliminating the denominators in both sides of Equation 5, we have:

$$A\theta^2 + B\theta + C \leq 0 \tag{6}$$

where

$$
\begin{aligned}
A &= a^2\rho p(rp - r\rho + \rho) \\
B &= a\rho(arm\rho - rm\rho - 2armp - am\rho + r^2p^2 - r^2pm - pr^2\rho \\
&\quad +r^2m\rho + rp\rho + rp^2 - rmp - ap\rho + apr\rho) \\
C &= r^2m^2\rho - armp\rho + arm^2\rho + a^2m\rho^2 - ar^2pm\rho + ar^2m^2\rho \\
&\quad +a^2rm^2\rho + ar^2m\rho^2 - ar^2m\rho^2 - a^2rm\rho^2 + r^2p^2\rho - 2r^2mp\rho - arp\rho^2.
\end{aligned}
$$

Thus the range of θ values that satisfy Inequality 4 can be obtained by solving the above inequality. Let θ_1 and θ_2 be the roots of equation $A\theta^2 + B\theta + C = 0$. Since $A > 0$, θ should be chosen in between θ_1 and θ_2.

Since coefficients A, B, and C are complicated, it is very difficult to derive θ_1 and θ_2 directly using the standard quadratic equation solving technique. We however notice that if $S_F = S_{M,h} = S_{M,c2}$, two sides of Inequality 4 become equal. The θ value for this case is $\frac{m}{p} - \frac{r(p-m)}{ap}$. Then we can let $\theta_2 = \frac{m}{p} - \frac{r(p-m)}{ap}$. Since $\theta_1 + \theta_2 = -\frac{B}{A}$, we have:

$$\theta_1 = -\frac{B}{A} - \theta_2 = \frac{m(a+1) - p + a\rho/r - a\rho}{ap - a\rho + a\rho/r}.$$

With the assumption that $m \geq \frac{rp}{a+r}$ and $r < 1$, it can be shown that $\theta_1, \theta_2 \leq 1$ and $0 \leq \theta_2$.

We can also prove that $\theta_1 \leq \theta_2$ by calculating the difference of θ_2 and θ_1. The difference $\theta_2 - \theta_1$ is

$$\frac{(p-m)(1-r)(p-\rho-a\rho/r)}{a(p-\rho+\rho/r)}$$

which is larger than or equal to 0 since $m \leq p, r < 1, \rho/r - \rho > 0$ and $1 - \rho/p - a\rho/(rp) > 0$. The last inequality holds because $S_{F,h} = 1/(1 - \rho/p - a\rho/(rp)) > 0$ (c.f. Equation 1).

To choose the optimal m, we first choose the optimal θ's for each m and then choose the (m,θ) pair that gives the smallest average stretch factor for S_M. The best θ for each m is $\theta' = (\theta_1 + \theta_2)/2$ which is the middle of the two roots. This value however may be less than 0 which is meaningless. Therefore we choose $\theta_m = \max(\theta', 0)$.

We cannot derive the closed form for calculating the optimal m. Instead, we calculate the stretch factor S_M numerically for each possible m value, then pick up the m which delivers the smallest stretch factor. Notice that the possible choices of m are integers between 1 and p.

References

[1] A. Acharya, G. Edjlali, and J. Saltz. The utility of exploiting idle workstations for parallel computation. *ACM SIGMETRICS'97*, pages 225–236, 1997.

[2] T. E. Anderson, D.E. Culler, D.A. Patterson, and the NOW Team. A case for NOW(networks of workstations). *IEEE Micro*, February 1995.

[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. In *Proceedings of ISDL'95: International Symposium on Digital Libraries*, August 1995. Revised version appeared in IEEE Computer, 1996, No.5.

[4] D. Andresen, T. Yang, O. Egecioglu, O. Ibarra, and T. Smith. Scalability Issues for High Performance Digital Libraries on the World Wide Web . In *Proc. of IEEE ADL '96 (Advances in Digital Libraries)*, pages 139–150, May 1996.

[5] 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, April 1996.

[6] Apache. *Apache HTTP Server Project.* http://www.apache.org, 1995.

[7] H. Casanova and J. Dongarra. Netsolve: a network server for solving computational science problems. *Proceedings of Supercomputing'96*, November 1996.

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

[9] Michele Colajanni, Philip S. Yu, and Daniel M. Dias. Analysis of task assignment policies in scalable distributed web-server systems. *IEEE Transactions on Parallel and Distributed Systems*, pages 585–598, June 1998.

[10] M. E Crovella and M. Harchol-Balter. Task assignment in a distributed system: Improving performance by unbalancing load. *ACM SIGMETRICS*, pages 268–269, July 1998.

[11] Om P. Damani, P.Emerald Chung, Yennun Huang, Chandra Kintala, and Yi-Min Wang. One-IP: Techniques for Hosting a Service on a Cluster of Machines. In *Proceedings of the Sixth Int. World Wide Web Conference*, April 1997.

[12] Digital Equipment Corporation. *About Alta Vista.* http://www.altavista.com/av/content/about.htm, 1995.

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

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

[15] A. Fox, S.D. Gribble, Y. Chawathe, E.A.Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, October 1997. fox_sosp_16.ps.

[16] Al Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine - A Users Guide and Tutorial for Network Parallel Computing.* MIT Press, 1994.

[17] S.D. Gribble. *UC Berkeley Home IP HTTP Traces.* http://www.acm.org/sigcomm/ITA/, 1997.

[18] V. Holmedahl, B. Smith, and T. Yang. Cooperative Caching of Dynamic Content on a Distributed Web Server . In *Proc. of Seventh IEEE International Symposium on High Performance Distributed Computing*, pages 243–250, July 1998.

[19] Inktomi Corporation. *The Inktomi Technology Behind HotBot, a White Paper.* http://www.inktomi.com, 1996.

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

[21] A. Iyengar, E. MacNair, and T. Nguyen. An analysis of web server performance. *Proceedings of IEEE GLOBECOM*, pages 1943–1947, November 1997.

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

[23] K.Dincer and G. C. Fox. Bulding a world-wide virtual machine based on Web and HPCC technologies . In *Proceedings of ACM/IEEE SuperComputing'96*, November 1996.

[24] T.M. Kroeger, J. Mogul, and C. Maltzahn. *Digital's Web Proxy Traces.* ftp://ftp.digital.com/pub/DEC/traces/proxy/webtraces.html, 1997.

[25] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models.* Prentice Hall, 1984.

[26] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Impelmentation of the 4.3BSD UNIX Operating System.* Addison Wesley, 1990.

[27] M. Litzkow, M. Livny, and W. Mutka. Condor - a hunter of idle workstations. *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, June 1988.

[28] U. Manber, M. Smith, and B. Gopal. Webglimpse – combining browsing and searching. *Proceedings of the Usenix Technical Conference*, January 1997.

[29] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. *Proceedings of the Usenix Technical Conference*, January 1996.

[30] National Coordination Office. *Next Generation Internet Initiative.* http://www.ccic.gov/ngi/, 1996.

[31] NCSA. *Common Gateway Interface.* http://booboo.ncsa.uiuc.edu/cgi/, 1995.

[32] Janon Nieh and Monica S. Lam. SMART unix svr4 support for multimedia applications. *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, June 1997.

[33] OpenMarket. *Fast CGI.* http://www.fastcgi.com/.

[34] Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich Nahum. Locality-Aware Request Distribution in Cluster-based Network Service. In *Proceedings of ASPLOS-VIII*, October 1998.

[35] G. F. Pfister. *In Search of Clusters.* Prentice Hall, 1998.

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

[37] SPEC. *SPECWeb96 Benchmark.* http://www.spec.org/osg/web96/, 1996.

[38] G. Trend and M. Sake. WebSTONE: The first generation in HTTP server benchmarking. *Silicon Graphics, Inc. whitepaper: http://www.sgi.com/*, February 1995.

[39] Yahoo! Inc. *Yahoo! Investor Relations Center.* http://www.yahoo.com/info/investor/, 1998.

[40] Zeus Technology. *Zeus Web Server v3.* http://www.zeustech.net/, 1998.

[41] H. Zhu, T. Yang, Q. Zheng, D. Watson, O. H. Ibarra, and T. Smith. Adaptive load sharing for clustered digital library servers. *Procecedings of the Seventh High Performance Distributed Computing*, pages 235–242, July 1998.