# A Scalable Cluster-based Web Server with Cooperative Caching Support[1]

G. Chen[2] C.L. Wang F.C.M. Lau
Department of Computer Science and Information Systems
The University of Hong Kong
Hong Kong
Email: {gechen,clwang,fcmlau@csis.hku.hk}

## Abstract

Clustering provides a viable approach to building scalable Web systems with increased computing power and abundant storage space for data and contents. We present a pure-Java-based parallel Web server with load balancing features for a heterogeneous cluster environment. Our design introduces a special in-memory cache layer, called the *global object space* (GOS), which is an integration of specially set-aside physical memory of individual participating cluster nodes. The GOS provides a unified view of cluster-wide memory resources and allows transparent accesses to cached objects independent of where they are located physically. The core of the GOS is a mechanism that performs dynamic in-memory caching of frequently requested Web objects. Using a technique known as *cooperative caching*, a requested Web object can be fetched from a node's local cache or a peer server's local cache, with the disk serving only as the last resort. A prototype system based on the W3C's Jigsaw server has been implemented on a 16-node PC cluster. Three cluster-aware cache replacement algorithms were tested and evaluated. The benchmark results show good speedups with a real-life access log, proving that cooperative caching can have significant positive impacts on the performance of cluster-based parallel Web servers.

---

[1] A preliminary version of this paper appeared in *Proceedings of IEEE International Conference on Cluster Computing (Cluster 2001)*, Newport Beach, California, October 2001, under the title "Building a Scalable Web Server with Global Object Space Support on Heterogeneous Clusters".

[2] Corresponding author: G. Chen, Department of Computer Science & Information Systems, The University of Hong Kong, Hong Kong / Email: gechen@csis.hku.hk / Fax: (+852) 2559 8447.

**Keywords:** Web object cache, cooperative caching, Global Object Space, hot object, replacement algorithms, cluster, the World Wide Web.

# 1 Introduction

The World Wide Web (WWW) has experienced explosive growth in both traffic and contents since its inception. Such a growth will continue for many years to come because of the growth in Internet users [1]. The increased availability of broadband connections is adding fuel to the flames [2]. More Internet users translates into more HTTP requests. Some study shows that HTTP requests generally account for 75-80 percent of all Internet traffic [3]. The whole picture points to challenges for researchers to come up with better technologies for and solutions to building more powerful Web servers.

There are generally two approaches to a more powerful Web server: to deploy a single, cutting-edge server machine with advanced hardware support and optimized server software, and to rely on the aggregate power of distributed machines. Several pioneering projects in Web server architecture have developed sophisticated technologies to increase the performance of a single Web server [4, 5, 6]. The limiting factor, however, is with the hardware. With the rapid advances in hardware technologies nowadays, a powerful server today will easily become obsolete tomorrow, and major upgrades of the server hardware tend to be expensive and require substantial effort.

The other solution is to employ multiple servers, in the form of a server farm [7] or a cluster-based Web server. A cluster-based Web server system consists of multiple Web servers connected by a high-speed LAN. The Web servers work cooperatively to handle the Web requests. Many research and industry projects have been conducted on the design of cluster-based Web servers [8,9,10,11], which aimed at such issues as load distribution or balancing [12,13,14], scalability [15,16], and high availability [17, 18]. These projects have shown that clustering is a promising approach to building scalable and high-performance Web servers [8,19].

One of the key techniques to increase Web serving performance is caching of Web objects. Web caching can be adopted at different levels. Clients, such as a Web browser, can cache recently visited pages in its local memory or on disk. A proxy

server sitting somewhere between a client and the Web server can intercept and fulfill the client's requests with local cached copies of remote pages. At the far end, the Web servers can return pages in the memory cache for faster response time. Some good surveys of Web caching can be found in [20, 21, 22].

Even though substantial work has been done on Web object caching in a single web server or within cooperative proxy servers in a WAN environment, not as much work has been targeted at caching in cluster-based Web server systems. The latter is different from caching in a single Web server because of the distributed nature of a cluster. A simple caching algorithm will likely not be able to fully utilize the underlying strength of a cluster, such as fast network communication speed, giant aggregated memory space, and parallel I/O system.

In this paper, we present the design and implementation of a cluster-based Web server system, focusing on its *cooperative caching* mechanism. The "cooperation" here happens within a LAN that connects the cluster nodes, which is different from cooperative caching in another popular context where proxy servers scattering in a wide area network work cooperatively over long distance connections. The latter requires protocols that are more sophisticated [23,24,25]. The nodes in our cluster-based server system work cooperatively, to fully utilize all the deployable physical memory in each node to form a cache layer for storing the site's contents. As handling objects in memory is many times faster than if they are on disk, the cluster nodes can respond quickly to incoming requests, and can handle more simultaneous requests, thus resulting in an increased overall system throughput. With cooperative caching, we are able to use the aggregate memory size to achieve higher cache hit rate than can be achieved on a single node, and to reduce excessive disk I/O accesses. The purpose of this paper is to demonstrate this fact.

Cooperative caching first appeared in the distributed file system design. The idea is that any node can access file blocks in the caches of other nodes to reduce accesses to the file server. The technique has been shown to be effective in improving the performance of distributed file systems or software RAID in a high-speed LAN environment [24,26,27,28]. The concept has been carried over to cluster-based Web server design. In fact, cooperative caching appears to be even more suitable for cluster-based Web severs because Web servers usually have larger active data set size than a normal distributed file system, which calls for an enlarged memory space in a system-wide cache.

Web servers can rely on caching provided by the underlying file system. Web objects, however, have several features that make file system caching solutions not necessarily applicable [29]. Firstly, Web objects have a different granularity than files. File system buffers are designed for fixed-size blocks of data, whereas Web caching treats files in their entirety. Therefore, since cache space is scarce, Web caching is faced with the issue of hitting a balance between caching more objects of small object sizes and caching fewer objects of large object size. Secondly, caching is not obligatory in proxy cache servers; that is, some documents may not be admitted at all by the cache, if by not admitting, the system would gain performance. This is unlike the file system buffer which always places those recently requested data blocks in the cache. Thirdly, Web servers seldom experience a burst of requests for the same object from a single client because the requested object would already be cached in the client's local cache. Thus a single access to an object should not be a reason for caching the object since there will not necessarily be any following correlated access. On the other hand, multiple accesses to the same object from different clients in a short time do indicate high popularity of a document. Hence, Web caching requires a design that is different from that of file caching.

Cooperative caching in our design is supported by a *Global Object Space* (GOS) implemented on the top of the physical memory of the cluster nodes. In each cluster node, a per-node object cache manager is responsible for caching Web objects as well as cooperating with object caching threads running in other nodes to create a globally shared cache space, the GOS. With cooperative caching, a Web object being requested can be served from a server's local memory cache or from a peer node's memory cache. Only failing both would the file system be accessed. The location where a cached object resides is transparent to the requesting client. Without cooperative caching, a cache miss invariably results in a disk access to the file system. The GOS not only supplies a uniform cache space that extends over all the cluster nodes, but is also equipped with a mechanism for cluster nodes to cooperatively maintain system-wide access information to facilitate accurate cache placement and replacement policies.

A prototype of the proposed design has been implemented by modifying W3C's Jigsaw Web server [30]. The caching mechanism of the prototype considers several key factors that are characteristic of Web object accesses, including object size and frequency of reference. Unlike most other results that are based on trace-driven

simulations [31,32,33], our benchmark results are collected from a real cluster environment with up to 16 server nodes. Our experimental results show significant performance improvement with the use of cooperative caching over the case of not using it. Several caching policies are tested and compared in order to evaluate the effects of cache policies on the overall performance.

The rest of the paper is organized as follows. Section 2 presents some background that is related to this project. Section 3 gives an overview of our proposed Web server architecture. Section 4 discusses the details of the core components and their functions for supporting the global object space. Section 5 presents some benchmarking results on the prototype system. Some related projects are discussed in Section 6. We conclude by summarizing our experience and proposing some future research work in Section 7.

# 2 Background

Caching is an age-old concept that has been applied to many areas in systems building, such as operating systems and networking systems. The chief benefit of caching is the increase in performance in certain system operations. Caching, however, requires the use of extra resources such as physical memory or disk space. In a good design, these resources could come as free because they are not needed by other operations. We discuss in this section design issues related to caching in distributed file systems, proxy servers, and Web servers, and argue that Web object caching in a cluster-based Web servers requires a design that is different in certain fundamental aspects from other systems.

## 2.1 Cooperative Caching in Distributed File Systems

File systems rely heavily on various caching mechanisms for increased performance. In a cluster environment, where the distributed file system has more workload to handle than in a single machine file system, caching plays a key role. A cluster, on the other hand, offers more resources on which the caching mechanism can be built. One proposal was *cooperative caching* that local memory in a network node can be made available to other nodes for caching purposes [28]. Several file system projects have incorporated the idea in their caching design.

### xFS

xFS is a distributed file system developed as a part of Berkeley's NOW project [34]. xFS pioneered the research on using remote memory as caching space, which became feasible because of the fast network communication in switched local area networks. It is a serverless system where participating workstations cooperate as peers to provide all the file system services.

An *N-chance* mechanism is built in xFS to increase the cache hit rate of the cooperative caching. With N-chance, an evicted file block having only one copy in the globally managed caching space, called a *singlet*, will be forwarded to a randomly chosen client. A recirculation count (initialized to N) is associated with each forwarded singlet, and is decremented every time it is forwarded. Only those singlets with a positive recirculation count will be forwarded, thus giving file blocks more chances to stay in the system after they have been chosen for replacement. The idea was prompted by the fact that knowledge in a distributed system sometimes does not propagate fast enough and replacement decisions are therefore not always accurate. By using the idle memory in remote peer nodes, xFS tries to reduce the number of disk access, alleviate I/O burden, and improve user visible performance.

### Hint-based Cooperative Caching File System

Hint-based cooperative caching file systems propose to use more local state information in decision making, thus leading to simplicity in design and less overhead [26]. In hint-based cooperative caching, information about possible locations of the file blocks needed by some client is sent to the client. This information represents local, not necessarily accurate knowledge of the state of affairs – hence the name "hint". When a local cache miss on certain file block occurs, the client will first refer to the hints it has. By requesting the block from the some cooperative client according to the hints, the client will get either the block from the peer or an updated hint about the block's location.

Hints are maintained in a client's own local cache. By using hints, the clients can avoid contacting the central manager for every local cache miss. In a hint-based cooperative caching file system, a master copy, which is the first cached copy that came directly from the file server, is associated with any cached block. For simplicity, only location hints of the master copies are recorded, but not other copies. During cache replacement, only master copies are forwarded for possible further caching by

clients using N-chance, which avoids duplication of the same file block. Hint-based cooperative caching achieves comparable performance to that of using tightly coordinated algorithms. The latter use accurate global state information, which is at the expense of more overhead than the simple hint-based approach.

## 2.2 Web Object Caching

Web object caching is widely adopted at different levels of the Web hierarchy, from Web servers, proxy servers, to end-user clients.

End-user clients, such as Web browsers, cache recently visited Web pages in their local memory to shorten future visits of these pages. Although client caching can improve the access time for the same page visited by the client in the future, client caching offers no help to other, perhaps nearby clients because clients cannot access each other's cache.

Proxy servers are widely deployed at locations between the clients and the Web servers, to intercept requests from the clients and service them on behalf of the Web servers. If a requested object is cached locally in a proxy server, it will be used to serve the client, thus reducing the client's perceived response time, and saving some network bandwidth. Failing that, the requested object will be served from the Web server through the proxy, which may or may not cache the object. Proxy servers are usually placed at the edges of the Internet, such as in corporate gateways or firewalls, or in a community's ISP servers. Using a proxy server for a large number of internal users can lead to wide-area bandwidth savings, improved response time, and increased availability of static data and objects [21].

Reverse proxy servers are proxy servers sitting at the other side of the connection, near the Web servers. A reverse proxy server caches Web objects that originate from the Web server behind it, thus alleviating the burden on the server and generating faster responses to client requests.

Much research effort has been devoted to studying the caching behavior of single Web servers, and the focus generally is on the performance impacts of different caching replacement policies [35,36,37].

## 2.3 Web Caching vs. File System Caching

Web caching is different from traditional file system caching in various aspects. An obvious difference is that most file system buffers store fixed-size blocks of data,

while Web servers always read and cache entire files. The sizes of Web objects range from a few kilobytes (such as HTML and image files) to extremely large ones such as video files. It is of course possible to break up a large Web file into small pieces and cache just some of the pieces, but to our best knowledge, no research so far has indicated that this is practical and beneficial. Variable-sized Web objects complicate the memory management in Web server caches.

File access patterns generally exhibit strong temporal locality of references, that is, objects that were recently referenced are likely to be referenced again in the near future. Therefore, most file system caches employ the Least Recently Used (LRU) replacement policy or some derivative of LRU to manage the file blocks in the cache. A file system buffer manager always requires that the requested data block be in the cache in order to fulfill a request. Thus, a cache miss on a requested file block invariably results in a cache operation for bringing the requested file block into the cache, and possibly a series of cache replacement operations if the cache is full. That is, caching of a requested block is obligatory. This is not the case for Web object caching.

Unlike file accesses, access patterns observed by Web servers do not usually exhibit high temporal locality. In general, the same user rarely requests the same Web object twice from a Web server in a short time interval (except in unusual circumstances where a reload is necessary), since the requested object should already be in the Web browser cache, the client OS cache, or the proxy cache. Therefore, traditional LRU replacement algorithms that are so popular in file caches are not suitable for Web document caching. On the other hand, multiple accesses from different clients in a short time interval do indicate high popularity ("hotness") of a document. Frequency-based replacement algorithms, such as Least Frequently Used (LFU) and its variants have been shown to be more suitable for Web caching [31,38,39,40,41]. A good survey of various Web cache solutions can be found in [20].

Moreover, accesses to Web caches exhibit "focus of interests", where bursts of requests are not uncommon when many clients in some locality have a common interest on certain hot issues. This observation suggests that Web object caching can be very effective if those hot objects can be all cached.

## 2.4 Cooperative Caching in Proxy Servers

Proxy servers aim to cope with all possible objects requested by their clients. The number of objects that a proxy server will handle is conceivably far greater than that of a reverse proxy server or a single Web server, which only need to cache objects from a single Web site. Therefore, proxy servers need ample caching space to make the caching mechanism effective. Such a large space can come from cooperative caching spanning multiple servers in WANs [42,43,44]. These projects have developed sophisticated mechanisms for individual proxy servers to lookup objects, to exchange object information, and to do other necessary operations across WAN connections.

### Harvest and ICP

Harvest cache [43] exploits hierarchical Internet object caching. The participating cache servers are organized as parents and siblings, and they work cooperatively by using a cache access protocol, which evolved into the *Internet Cache Protocol (ICP)* [45]. By using ICP, a cache server queries its parents, siblings, and possibly the origin Web server simultaneously for a cached copy in case of a cache miss. The cache server will retrieve the object from the site with the lowest latency. In case where the answers from its first level parents and siblings are all negative, the cache server will further query up the hierarchy until a cached copy is found, or is returned by the origin Web server.

### Summary Cache

Summary cache [44] is a scalable wide-area Web cache sharing protocol. Unlike the multicast-based inter-proxy object query approach of ICP, Summary cache maintains a compact summary of the cache directory of every participating proxy server. When a client request results in a local cache miss, the proxy server checks the summaries to identify potential proxy servers to query. Bloom filters are used to keep the summaries small. It is claimed that summary cache can significantly reduce the number of inter-cache protocol messages, bandwidth consumption, and CPU overhead while it achieves almost the same cache hit rate as ICP.

## 2.5 Cooperative Caching in Proxy server vs. Web Server

Web object caching happens in both proxy server and Web server (reverse proxy). The object access pattern and caching behavior of the two cases have similarities in general. Several important differences, however, make existing cooperative caching mechanisms employed in proxy servers not well-suited for cluster-based Web servers.

- Client and request characteristics: Proxy servers, being at the edge of the Internet, tend to face closely related clients such as those in the same community or of the same ISP. These users have strong common interests. Web servers on the contrary are far from the end-user, and their requests come mostly from proxy servers. As proxy servers and clients have their own caching, requests for a certain object tend to decline in number faster than that for a proxy server. Moreover, Web servers face a wider client population than proxy servers, so the change of interests tends to be faster than that for a proxy server.
- Scale of the object space: Note that a proxy server has to deal with far more possible objects than a Web server which serves a single site. The community that a proxy server services can potentially visit the whole Web.
- Network connections: Cooperative proxy servers usually work in a WAN environment while Web servers serving a single site are more close to each other, such as in a LAN environment. The design of cooperative caching protocols for proxy servers therefore needs to focus on how to reduce the communication costs for propagating directory updates, and how to improve the cache hit rate of a distributed global cache that spans a wide area. In such systems, it is not practical to adopt complicated caching algorithms because of the heavy overheads of network communications [44]. In contrast, for a cluster-based Web server in a high-speed LAN environment, it is possible, and in fact necessary, to obtain accurate and fine-grained global state information, in order to locate cached objects and coordinate object replacement decisions.

## 2.6 Software Distributed Shared Memory

*Distributed shared memory* (DSM) is a model for interprocess communication in distributed systems. Processes running on separate hosts can access a shared address space through normal load and store operations. DSM can be implemented in software, hardware, or in some combination of the two. There are three key issues in

the construction of DSMs: DSM algorithms, implementation level of the DSM mechanism, and memory consistency model [46]. DSM implementations also make use of caching for more efficient accesses to shared objects.

To some extent, cooperative Web object caching share some similar goals with caching in software DSM. There are, however, some significant differences between DSM caching and Web object caching [47].

- Granularity: In DSM systems, the granularity of caching or other operations is usually determined by the implementation. A common grain size is the memory page. Software DSM solutions favor coarse grain sizes for performance and complexity considerations. In Web object caching, the granularity of caching is usually at object level, and Web objects can have various sizes, from a few kilo-bytes to several mega-bytes or even larger.
- Replacement algorithms: There is always insufficient space in a cache to accommodate all the data that deserve a place in the cache. A replacement strategy is necessary. In DSMs, the LRU and its derivatives are the common choices of a replacement strategy because of their simplicity, and they seem to work well with fixed-size grains. Whereas in Web object caching, many other parameters need to be considered, such as reference frequency, object size, and so on.
- Sharing model and consistency: A DSM needs to support the different complicated reference and memory sharing models of all the possible applications running on top of it. Newer DSMs increasingly adopt the multiple reader/multiple writer approach because participating computation nodes in a cluster are on equal ground and can change the data concurrently [47]. Whereas changes to Web objects usually come from a single master site, and so the consistency issue is much simpler.
- Locality types: In DSMs, spatial locality and temporal locality are usually observed. In Web object caching, spatial locality is not a feature because links in a document can refer to anywhere in the Web, and not necessarily to pages located nearby. Temporal locality is not necessarily a feature in Web object caching neither because a single user visiting of a Web page does not imply the same user will visit it again, nor others will visit it too. But a few concurrent requests for the same page do mean that the page is popular and will have a good chance of being visited again in the near future.

From the above discussions, we can see that cooperative caching in a cluster-based Web server has certain similarities to cooperative caching in distributed file systems, proxy servers, and distributed shared memory. For example, the general rules on cache replacement algorithms resulting from previous proxy cache studies should be applicable to Web server caching because of their similar access patterns and object granularities. Nevertheless, none of the above three caching scenarios represents a satisfactory solution for adoption by a cluster-based Web server because of the differences discussed.

We therefore propose, in the remaining sections, a cooperative caching mechanism that is a good fit for a cluster-based Web server. We will explore the potential benefits of caching that a cluster can provide to a cluster-based Web server equipped with our proposed mechanism. With a giant cache space that spans the entire cluster, higher cache hit rate is achieved. With the fast and reliable underlying LAN connection, cluster nodes can exchange information about access patterns and cache situations efficiently and with relatively little overhead. Cooperative caching reduces accesses to the disk, leading to better throughput at the Web server, more requests being served at the same time, and higher availability as seen by the clients.

# 3 Proposed System Architecture

Our answer to improved Web server performance is to combine the processing power and storage space of all the nodes in a cluster using a cooperative caching mechanism that is most fitting for a LAN-connected cluster. This section presents an overview of the design of that mechanism, called the *Global Object Space*, and the major building blocks of our cluster-based Web server.

## 3.1 System Overview

The Global Object Space (GOS) is built from the physical memory of all the cluster nodes. The amount of memory set aside per node for this is a parameter to be tuned when the system is deployed in reality. Each node can contribute a large chunk, resulting in a gigantic system-wide memory space for caching. The design can be so flexible that nodes can choose to or not to participate in the global caching system before or during runtime. We assume for simplicity of discussion that all nodes in the cluster participate. The GOS is used as a large cache for storing all cached Web objects and all Web objects are treated as equal, that is, with no distinction based on

which original nodes they are from. Cooperative object caching is the core mechanism used in the GOS, which upon a local cache miss enables triggering the forwarding of an object from a peer server node instead of fetching the object from the disk. This is justifiable because of the high-speed networking power available in a cluster environment, where to access a data item in a peer node's memory would take far less time than to access a disk-bound data item. The purpose of this paper is to prove that this in fact is the case with today's clustering and networking technologies.

Figure 1 gives a high-level view of our Web server architecture. In the system, each server node sets aside and maintains a special memory segment, called the Hot Object Cache (HOC), for Web object caching. All the HOCs coalesce into a globally shared object cache. An HOC is but a node's local object cache. All Web objects stored in the global object space are visible and accessible by all the nodes at all times through some specially designed lookup mechanism, to be discussed in detail in Section 4.

Each node operates two server processes (daemons), the global object space service daemon (GOSD) and the request handling daemon (RHD). The GOSD is responsible for managing the node's HOC, as well as to work cooperatively with all the other nodes to provide a location-transparent service for accessing cached Web objects. The illusion offered by the GOS is that a Web server node can refer to any Web object in the system without having to know its physical address. This is similar to the idea of software DSM systems. During execution, the RHD listens on the TCP port for incoming HTTP requests. It forwards parsed and analyzed requests to the

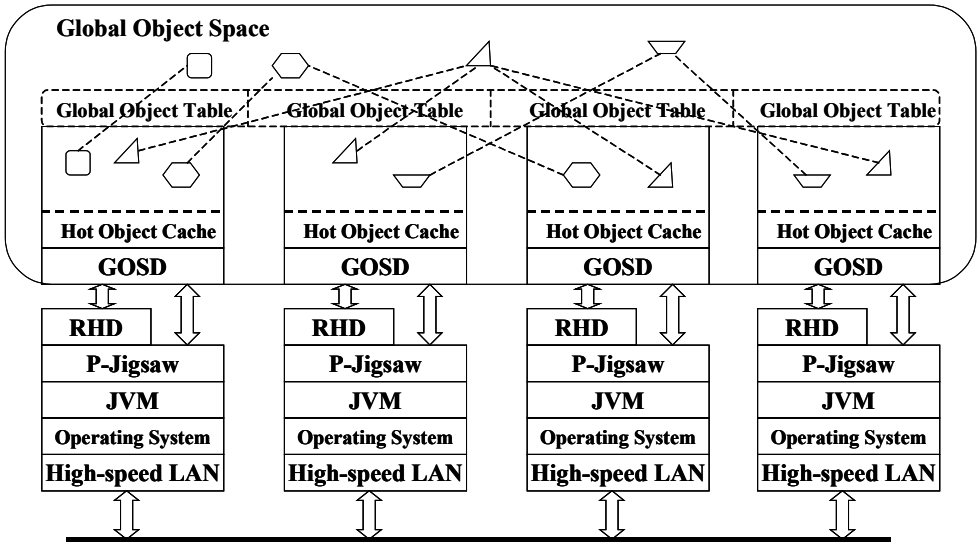


**Figure 1. System Architecture Overview**

GOSD which handles both requests from the local RHD as well as those from peer GOSDs.

Each cached object has a unique *home node*. The home node is the node in which the original persistent storage copy of the cached object is located. As a popular object that has become "hot" can be present in multiple HOCs, an object's home node is assigned the responsibility to keep track of the location information of all the copies of the object and their access statistics.

## 3.2 Hot Objects

Much research effort has been directed to the characterization of Web workloads and the performance implications they have for Web servers [24,48,49]. One of the major results is the notion of *concentration* [48], which points to the phenomenon that documents on a Web server are not equally accessed. Some of them are extremely popular, and are accessed frequently during short intervals by many clients from many sites. Other documents are accessed rarely, if at all. In our system, we use "*hot object*" to refer to a Web object that has been requested many times in the recent past by clients and therefore is expected to be frequently requested in the immediate future. The "hotness" of an object should not be treated as a static property because of the quickly changing nature of the Web. Therefore using a fixed threshold to measure hotness is not appropriate. Instead, the system in our design keeps track of the number of times the object is accessed over a certain period. This is recorded in a counter variable associated with the object. Because of the distributed nature of the system, where multiple copies of the same object could exist, the accurate count for an object could be elusive. A mechanism has been built into the system to make sure that at least a good approximation of the actual count can be obtained when an object needs to be placed or replaced, which will be discussed in Section 4.

## 3.3 Requests Distribution

The focus of this paper is on the cooperative caching mechanism and its effectiveness. Therefore, the part of the system that is exposed to all potential clients out in the Web is not of concern here. This is the part that determines whether a single URL or multiple URLs are to be used for the service; similarly for the IP address. A simple and common design is to present to the clients a single system, with a single URL and IP address. The IP address can actually be a "virtual" address. A dispatcher node at

the entrance of the system would rewrite this address in an incoming packet using the real IP address of the selected server node. The effect is that every client request is directed to one of the server nodes, to be served by that node. The decision on which node to select can be based on a simple round-robin scheme, or on the load situation across all the nodes. If dynamic load information can be made available to the dispatcher, load balancing can be applied to the server nodes, resulting in better overall performance. The cost would be that individual nodes need to inform the dispatcher node of their workload level periodically or on demand. Design issues and alternatives for load-balancing Web servers are discussed in [12,14,16].

# 4 Global Object Space with Cooperative Caching

The global object space is the core part of our proposed system. The GOS provides the service of cooperative object caching, remote object access, and persistent connection support. In this section, we will present details about the various aspects of the global object space.

## 4.1 Tables and Counters

To build and maintain the GOS, two tables are defined and managed by the local *Global Space Service Daemon* (GOSD): (1) local object table (LOT) and (2) global object table (GOT).

There are two counter variables associated with each object cached in the global object space:

- Approximated global access counter (AGAC): This counter reflects the approximated total number of accesses received by all the participating nodes for the object since the counter was initialized.
- Local access counter (LAC): This counter, which belongs to some node, stores the number of accesses received by the node for the object since the last reporting of the LAC value to the home node of the object.

The LOT contains the local access information and records of objects cached in the node's hot object cache, including the AGACs, the LACs, and home node numbers of the cached objects.

The GOT maintains two kinds of information. One is the mapping of object IDs (URLs in the current implementation) to home node numbers. The other one is global information for those objects whose home node is the current node. The global

information includes, for each object, the AGAC and node number(s) of existing cached copies.

With the support of the LOT, the GOT, and other information, cooperative caching is enabled in the GOS.

## 4.2 Workflow of the GOS

Figure 2 depicts the workflow to obtain an object from the GOS when an object request is received by a node's GOSD. Upon receiving a new request, the GOSD will search for the requested object in the local hot object cache. If this very first search fails, the GOSD will forward the request to the object's home node. The home node will respond to the request with a cached copy from its local object cache, if one exists, or a disk copy if there is not a cached copy. The home node also can respond with a message indicating where a cached copy may be obtained, if it is already too overloaded and there is a cached copy in a peer node, or there is no cached copy in its local hot object cache but there is one in a peer node. Allowing other nodes to serve the request can avoid overloading the home node, especially when the home node is home to many hot objects. Note that the original server node that first received the request performs the actual serving of the request, and all the help from the other nodes is for getting a copy of the requested object to be forwarded to this node.

### Support for Persistent Connection

HTTP/1.1 [50] specification defines the persistent connection, which allows multiple HTTP requests between the same pair of client and server to be pipelined via a single TCP connection. The persistent connection eliminates unnecessary TCP connection

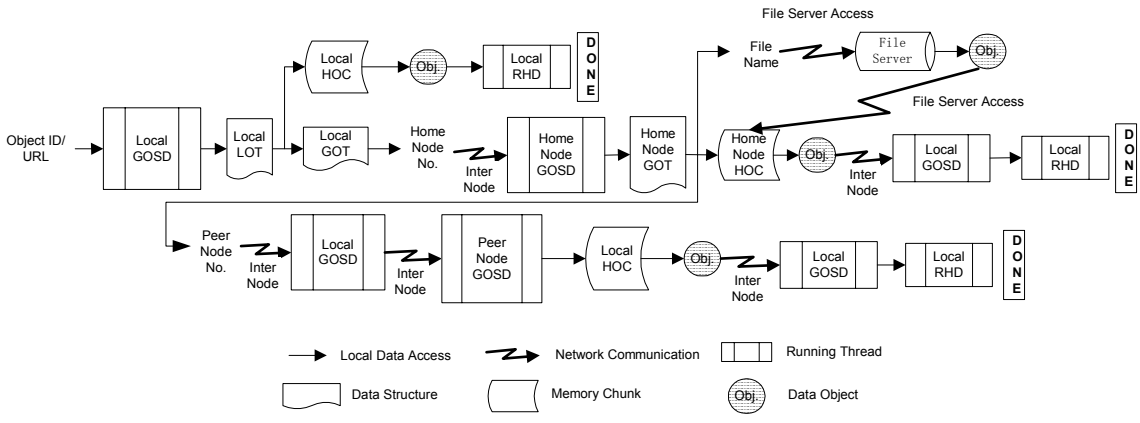


**Figure 2. Workflow of GOS**

setup time, resulting in improved user perceived latency and performance [51]. In our proposed system, because all the objects in the system are accessible via the GOS service, a request-handling node can keep persistent connection with the client, while fetching object from other nodes via GOS. Without a GOS where objects can be shared on a global basis, it is very difficult if not impossible to deal with multiple requests in a single TCP connection for objects that are physically scattered.

## 4.3 Updating of Access Counters

The two access counters, AGAC and LAC, associated with each cached object are essential for the cache replacement algorithm. The AGAC value is used as a key in the cache replacement process. The LAC helps count accesses that contribute to the total number of accesses to the object. The following describes how the access counters for a certain object cached in the global object space are updated.

The home node of the object holds the object's AGAC. For each cached copy of the object in any other node, a copy of the AGAC and the LAC are recorded in the LOT. Whenever a cache hit of the object occurs in a caching node, the LAC for the object is incremented by one. Periodically, each node will send the LAC value of every cached object in its local HOC to the object's home node. The interval for such reporting is a parameter to be set by the administrator or automatically by some policy module. When the home node gets the report from its peer node, it will add the received LAC value to the object's AGAC. The new AGAC value is then sent to the reporting node. After receiving the new AGAC value for the object, the reporting node will clear its LAC to zero, and start a new round of counting. Because requests for Web objects tend to come in bursts, a threshold is set for an object's LAC value, which is used to trigger a reporting to the home node when the threshold is reached. This makes sure that rapid changes to the access count of an object get reflected sufficiently quickly, and objects receiving such bursting requests will be cached in other nodes more promptly.

Because the participating nodes only exchange object access information at regular intervals (unless the LAC threshold value is reached), the AGAC in the home node of an object contains at best an approximation to the real value. The copies of the AGAC in various other nodes is therefore an approximation to an approximation. This is something unavoidable in a distributed environment due to the costs of maintaining global information. Nevertheless, a reasonably good approximation should work well

for our purpose. Our performance benchmark data show that this is true for our system.

## 4.4 Cache Replacement Algorithm

Cache replacement algorithms are an important component of any cache system. Cache replacement algorithms in a single Web server or proxy server have been well studied [29,31,32,33], but relatively little has been done for the same in a cluster-based Web serving environment. The design of and results from using our cache replacement algorithm presented in the following can serve as a reference for future studies of cache replacement algorithms in a distributed context.

In our system, the cache replacement decisions are made by individual nodes with help from the approximated global access counters. Every time a new object is referenced, locally or from a remote node, the GOSD will try to insert the object into the local hot object cache. The approximated global access counter records how many times the object is accessed recently by all nodes, thus indicating how popular the object is. The approximated global access counter is update periodically according to the scheme discussed in Section 4.3. Objects with the largest AGAC values are regarded as hot objects, and are given higher priority for a place in the hot object cache. That is, the AGAC serves as the key for the cache replacement algorithm. During our design, we tried two algorithms based on the Least-Frequently-Used (LFU) criterion, LFU-Aging and Weighted-LFU, to be executed in every server node.

- *LFU-Aging*: The aging part is due to the fact that any information about the popularity of an object, if not updated, becomes less and less reliable as time passes [39]. This is particularly true for Web object caching, since as mentioned in Section 2.5, requests for the same object declines faster in Web server than in ordinary file systems. In our design, we apply aging to the AGAC: the AGAC values in objects' home nodes are halved at the end of a pre-set time interval. Halving is not the necessarily optimal aging operator, but our experiments show that it is a reasonable choice. Updated AGAC values are propagated to all concerned server nodes through the feed-back message during LAC reporting.
- *Weighted-LFU*: A large object takes up a substantial amount of real estate, and when it leaves the system, it creates room for many small objects. This algorithm considers both the object size and the access frequency. Replacement decisions are based on a simple formula that divides the global counter value by the file size

in log scale. The log scale is used because memory size is generally much larger than object sizes, and so we do not need to be too fine-grained. With this algorithm, if two objects of different sizes have the same access frequency, the larger one will lose out if one has to be replaced.

All the objects in the HOC are sorted by their counter or weighted counter values. The last object in the queue can be viewed as the most unpopular one in the node and is the first to be discarded from the HOC during a replacement operation.

In addition to the above two LFU-based algorithms, an LRU (Least-Recently-Used) algorithm is also implemented and tested in order for us to compare the impact of difference caching replacement algorithms on Web object caching performance. The LRU algorithm is one of the simplest cache replacement algorithms to implement, and it and its variances are widely adopted in existing file system caches. As we will show in next chapter, LRU performs much poorer than LFU for Web object caching.

## 4.5 N-chance Forwarding

Our discussion above has revealed that we only use an approximated global count for the replacement algorithm because updating the access count at the home node and other nodes for every single client request would result in intensive inter-node communication in the cluster. Since we only periodically update the global counter, or when the local count reaches the threshold value, many server nodes would not be in sync with the exact global counter value. We could solve the problem by broadcasting every update as immediately as we could, but this has negative impact on the scalability of the system. In the worst scenario, an object that suddenly receives a large number of hits right after a global update could be replaced by a less popular object because the local server nodes still keep the now obsolete hit count value. To counteract such an anomalous situation, we could give the object "a second chance".

With the second-chance provision, when a node's replacement algorithm decides to swap an object out of its HOC, rather than discarding the object right away, the node sets a special variable of the object, called recirculation count, to N (N is set to 2 in our current implementation, for "second" chance). It forwards the object along with its LAC to a random peer, and informs the home node of the object about its new location. The peer that receives the copy of object uses its cache replacement algorithm to judge whether it should keep the copy or not. That is, the object gets a chance to be re-examined for its popularity. If the result is not favorable, the

recirculation count is decremented and the copy of object is forwarded again until either the count drops to zero or a node accepts it as a cache entry. The node that accepts the copy would inform the home node. Because in our implementation, N is set to two, the forwarding traffic is kept to the minimum.

This method is similar to the N-chance technique used in xFS [26,28]. The difference is that the N-chance forwarding there attempts to avoid discarding singlets (objects having a single copy across the whole system) from the client memory, and so before a block is forwarded, the node has to check to see if that block is the last copy cached by any client. In our algorithm, a copy of the object is forwarded regardless of whether the copy is the last copy or not.

## 4.6 Cache Consistency

HTTP/1.1 specification provides a simple expiration model to help Web object caches to maintain cache consistency [50]. In our system, every cached copy carries an expiration timestamp provided by the home node. The validity of a cached object is checked in compliance with the HTTP specification when the node replies to the client request. Objects that are deemed invalid according to the timestamp will be expunged from the local hot object cache, and re-fetched from the home node. HTTP cannot ensure strong consistency, however, and there is a real potential for data to be cached too long (perhaps because the timestamp is too loosely set). Strong consistency refers to having all copies of an object synchronized at all times or most of the time. A home-node-driven invalidation mechanism is therefore built into the system to allow objects' home nodes to invalidate the cached copies to keep the cache consistent. When the home node of an object finds the cached copy to be no longer valid, it will send invalidation messages to the other nodes as indicated by its records. The nodes receiving the message will expunge the expired object from its local hot object cache.

# 5 Performance Evaluation

A prototype system has been implemented by modifying W3C's Java Web Server, Jigsaw, version 2.0.5 [30]. The global object cache is added to Jigsaw in order to support cooperative caching. Three different cache replacement algorithms, LFU-Aging, Weighted LFU, and LRU were implemented and tested in a Linux PC cluster

to evaluate the effects of cooperative caching with different cache replacement policies.

## 5.1 Experimental Environment

We measured the performance of our cluster-based Web server on a 32-node PC cluster. Each node consists of a 733 MHz Pentium III running Linux 2.2.4. These nodes are connected through an 80-port Cisco Catalyst 2980G Fast Ethernet switch. During the tests, 16 nodes acted as clients and the rest as Web servers. Each of the server nodes has 392M bytes of memory.

The benchmark program is a modified version of the httperf program [52]. The httperf program performs stress test on the designated Web server based on the Web server log from a well-known academic website.

The main characteristics of the data set and the log file are summarized in Table 1 and Table 2, respectively. Because the data set is from an academic department's website, there are a number of files with size larger than a few megabytes. These are academic papers for visitors to download. Such files are not common in a normal website and should be the traffic of an ftp server, and so we filtered away some of these files the make the test data more reasonable. httperf supports customized workload generation based on a given workload file. We modified the collected access log file to make it work for httperf. Requests are generated by httperf according to this modified access log file.

| | |
|---|---|
| Total size | 6.756 Gbytes |
| No. of files | 89,689 |
| Average file size | 80,912 bytes |

**Table 1. Summary of Data Set Characteristics (Raw Data Set)**

| | |
|---|---|
| Number of requests | ~640,000 |
| Data transferred | ~ 35 Gbytes |
| Distinct files requested | 52,347 |
| Total size of distinct files requested | ~2.73G |

**Table 2. Summary of Access Log Characteristics**

The website is stored in an NFS server mounted across all the server nodes. The NFS server is a dedicated SMP-PC server with two 450MHz Pentium III CPUs running Linux 2.4.2. The home node of each Web object is decided by a predefined

partition list. The NFS server is connected to the Cisco switch via a dedicated Gigabit Ethernet link.

In the following figures and discussions, "with CC" means the cooperative caching mechanism is enabled. When there's a local hot object cache miss, a server node will first try to fetch the object copy from a peer node's hot object cache, before resorting to the file server. "Without CC" means the cooperative caching mechanism is disabled, such that when there's a local hot object cache miss, the server node will contact the file server directly.

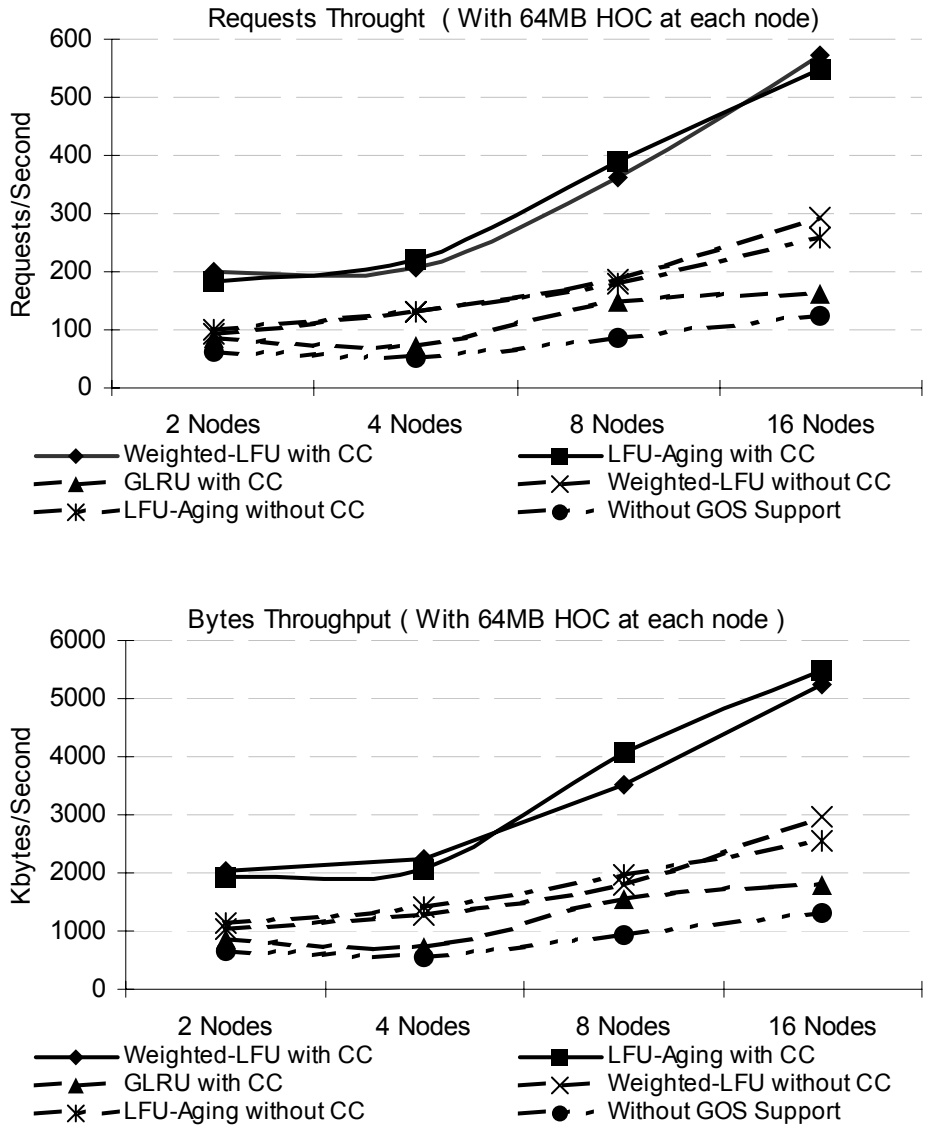


**Figure 3. Effects of Scaling Cluster Size**

## 5.1 Effects of Scaling Cluster Size

Figure 3 shows the request and byte throughput of the system with cluster size ranging from two nodes to sixteen nodes. 64 Mbytes per node is allocated for caching.

The curves clearly show that the GOS can substantially improve the system's performance. For the 16-node case, when using Weighted-LFU with CC, the request throughput is around 4.56 times of that without GOS support. For the 2-node case, it is 3.12 times. The gap is larger when the system scales up.

The figure also shows that the cooperative caching mechanism has good scalability against scaling cluster size. Without GOS support, the speed up of requests throughput is only around 2.02 when system expands from two to sixteen nodes, while that with Weight-LFU with CC is around 2.89. This is still not most satisfactory as the ideal speedup should probably be close to 8 times – something for future investigation or improvement.

From the figures, we can find that LFU-based algorithms have better performance and are more scalable than LRU-based algorithms. For example, when the cluster size scales from two nodes to sixteen nodes, the speedup for GLRU with CC is around 1.71, while it is around 2.89 for Weighted-LFU with CC.

## 5.2 Effects of Scaling Cache Size

We tested the *global cache hit rate* on a 16-node server configuration by scaling the size of hot object cache at each node. A global cache hit means that the requested object is found in the request handling node's local hot object cache or a peer node's hot object cache. In the experiment, the size of cache memory at each node scaled from 8 Mbytes to 64 Mbytes. The aggregated cache size from all the nodes combined corresponds to around 1.8%, 3.6%, 7.2%, and 14.4% of the size of the data set, and is referred to as the *relative cache size* (RCS). The RCS is a meaningful indicator on the cache size for applications with various test data sizes.

Figure 4 shows the global cache hit rates and request throughputs obtained from the test. We omit the curves for byte throughput, which are very similar to those for the request throughput.

The figure shows that LFU-based algorithms perform much better than LRU-based algorithms in general, especially when the cache space is relatively small relative to

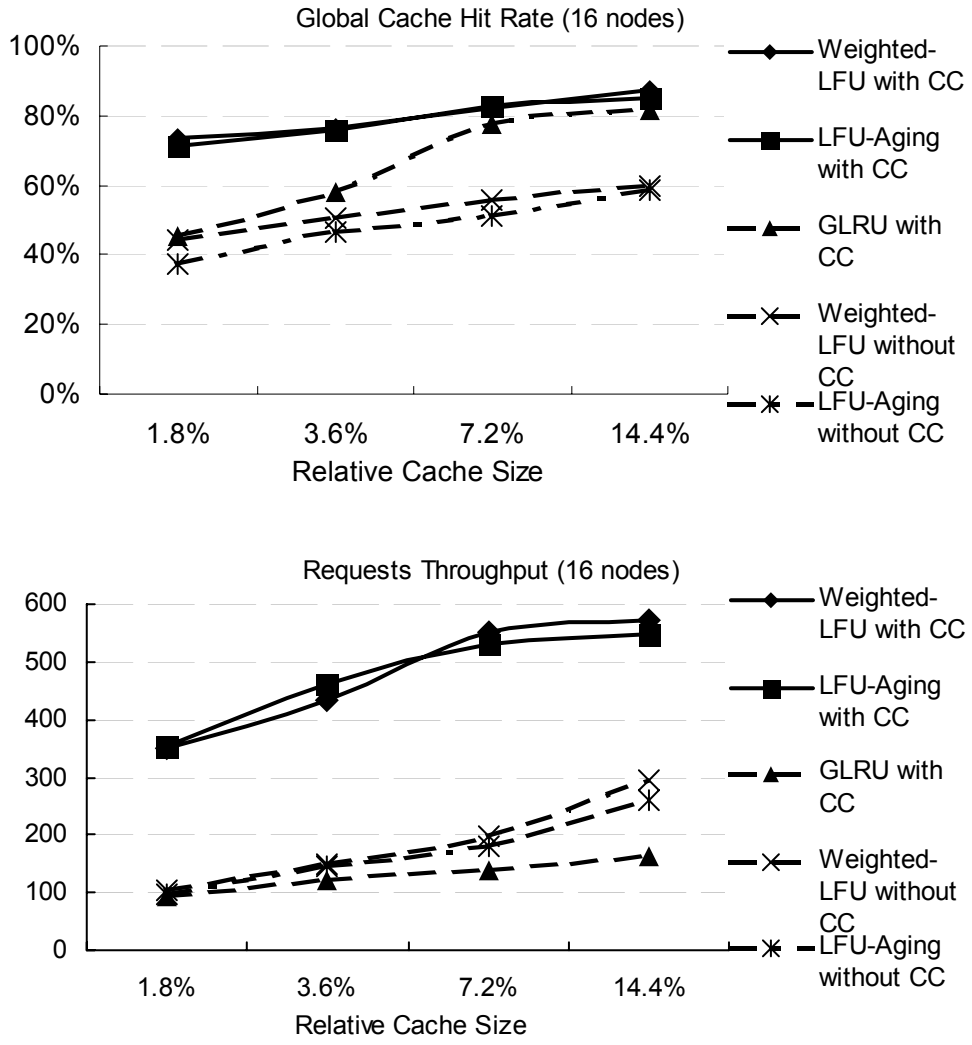


**Figure 4. Effects of Scaling Cache Size**

the data set size. With an RCS of 1.8%, the global hit rate can reach as high as 73%, which is the case when using Weighted-LFU with CC.

From the global hit rate curves, we can see that cooperative caching utilizes the caching space provided by a cluster efficiently, and can improve on the global cache hit rate considerably, even with a small RCS. With the same cache replacement algorithm, the global hit rate of Weighted-LFU increases from around 44% to around 73% when cooperative caching is enabled.

The results also indicate that Weighted-LFU can achieve a higher global cache hit rate than LFU-Aging when cooperative caching is disabled. It is especially true for the smaller RCSs. It is because Weighted-LFU will favor small objects during cache replacement operations, resulting in more small objects being placed in the cache, and if the cache space is small, the effect will be more standout.

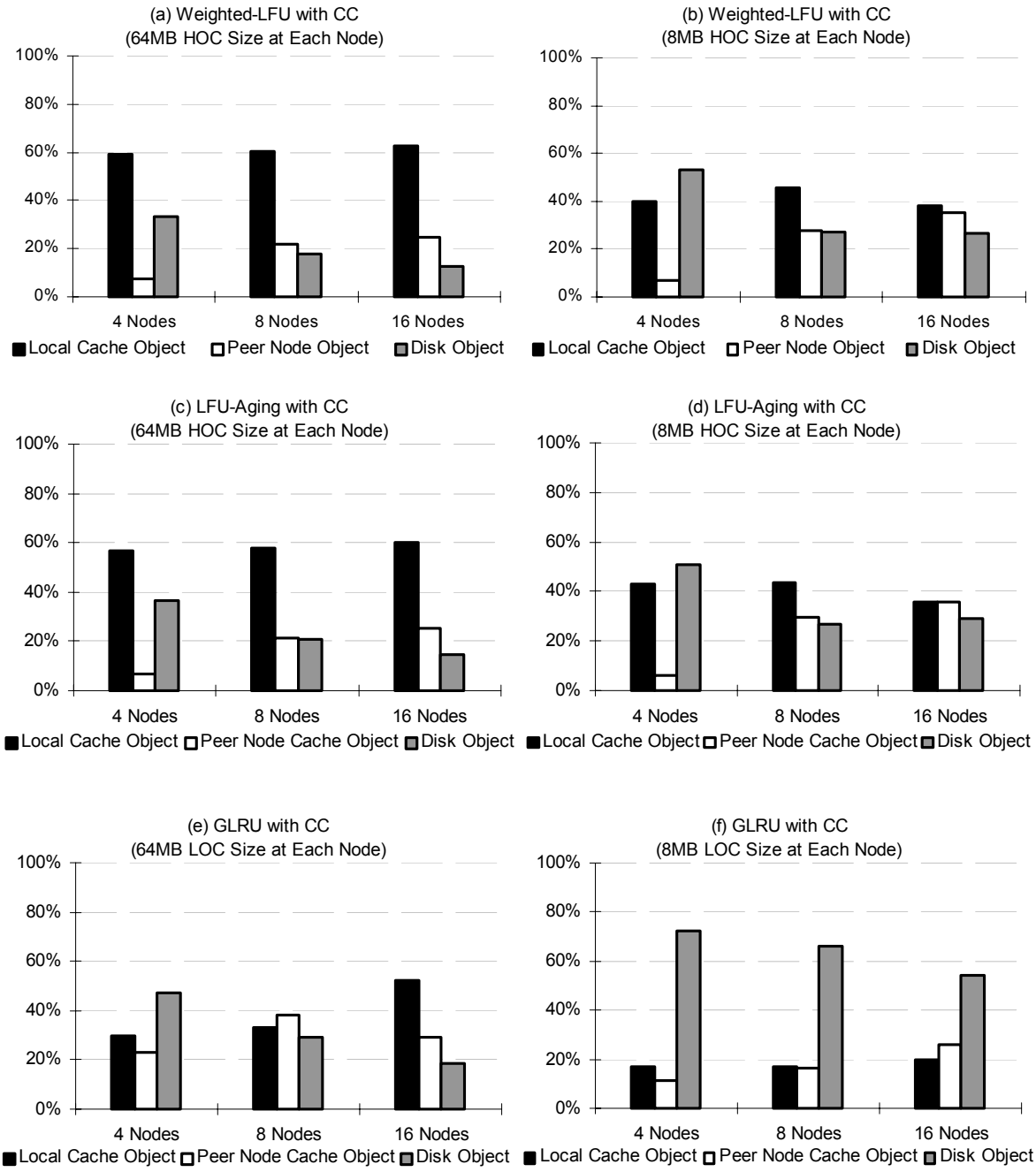


**Figure 5. Breakdown of Request Handling Pattern**

## 5.3 Breakdown of Request Handle Pattern

From the request throughput curves in Figure 4, we can see that the throughput is closely related to the global cache hit rate. One exception is that GLRU with CC has a higher global cache hit rate than that of Weighted-LFU without CC and LFU-Aging without CC, but its request throughput is worse than those of the two LFU-based

algorithms. This can be explained by delving into how the global hit rate is obtained. We designed and conducted a detailed study for this purpose.

We classify the returned object into three categories based on where object came from:

- *Local Cache Object*: The server that receives the request has the requested object in its local hot object cache. The object is fetched from the cache and served immediately. This type of object access has the shortest access latency.
- *Peer Node Cache Object*: The server that receives the request does not have the requested object in its local hot object cache, but there are cached copies of the object in the cache of the object's home node or other peer nodes. Therefore, the object is fetched from either the home node or a peer node.
- *Disk Object*: The requested object is not in the global object space, and has to be fetched from the file server. This has the longest serving time.

Figure 5 presents the percentages of client requests that resulted in each type of the above three categories. In the figure, both the local cache object and peer node cache object is considered and counted as a cache hit in the global object space.

The figures for LFU-based algorithms with 64 Mbytes of hot object cache in each node show high local cache hit rates. The local cache hit rates are around 60% for both Weighted-LFU and LFU-Aging, suggesting that LFU-based algorithms are indeed very effective in predicating hot objects. A good local cache hit rate reduces the costly remote object fetching from peer nodes, thus improving the system's performance greatly. The test results also show that, for the 16-node case, with 64 Mbytes hot object cache size in each node, around 75% of the top 15% most requested objects have their hit in the hot object cache of the node serving the request, and another 14% in the hot object cache of a peer node. Totally, 89% of the top 15% of the most requested objects got a hit in the GOS. This figure is in line with the previous research result that around 10% of the documents of a website accounts for around 90% of the requests that the website would receive [48]. Even with only an 8-Mbytes hot object cache in each node, the local hot object cache can still reach around 50%, and 20% for peer node HOC hits. It shows that our mechanism to approximate the global access count is effective in identifying the most probable popular objects.

The figures for LFU-based algorithms with 8 Mbytes of hot object cache in each node show that the number of peer node cache object increases from around 6.7% with 4 nodes to around 35.2% with 16 nodes, while the number of disk objects drops

from around 50% to around 25%. This confirms that the cooperative cache can greatly reduce the costly file server disk accesses, which is a common bottleneck for a website. The effect of cooperative caching is more pronounced when the cluster size scale up, suggesting that cooperative caching should be a feature of all large-size clusters.

Concerning the GLRU algorithm, we can see a much lower local cache hit rate when compared to the LFU-based algorithms. This provides an explanation to the question we had about the phenomenon of lower throughput but high global cache hit rate with using GLRU with CC. The GLRU does achieve nearly the same global cache hit rate as the LFU-based algorithms when the cache space is plentiful (Figure 4), but unfortunately a large portion of those hits are non-local hits, which explains why the lower throughput. Figure 5 also indicates that the local cache hit rate for LRU-based algorithms drops much faster than that for LFU-based ones with decreasing cache size. For example, for the 16-node case, when the hot object cache size in each node decreases from 64 Mbytes to 8 Mbytes, the local cache hit rate for GLRU drops from around 52% to around 20%, while that for Weighted-LFU with CC only drops from around 60% to around 40%.

## 5.4 Effects of N-chance

N-chance can improve the overall performance of cooperative caching in distributed file systems considerably by increasing the global cache hit rate without significantly reducing the local cache hit rate [28]. It is natural to think that N-chance would likewise have a positive impact on cooperative caching in cluster-based Web servers. We tested N-chance in our prototype system. We use a recirculation counter having a value of 2, which is value suggested in [28]. A value of 2 means that an evicted object will be given two more chances to be cached by another node. For the test, we put on each cached object a tag to indicate whether the object is cached after an N-chance forwarding.

Our test results show that N-chance has practically zero impact on our LFU-based cooperative caching mechanism. Specifically, with Weighted-LFU with CC, objects that have been “saved” by N-chance contributed less than 0.1% of the total global cache hits. The figure is similar for LFU-Aging with CC. The explanation is that because of the effectiveness of our approximated global frequency based caching replacement algorithms as demonstrated through the other tests, most evicted objects

did already have a really low system-wide reference count, and it was hard for them to be cached again in other nodes through N-chance forwarding. The results agree with the fact that Web objects have a faster changing degree of popularity than ordinary files. Once an object becomes "cold", it will have very little chance of being referenced in the near future.

# 6 Related Work

The design and implementation of cluster-based Web servers is among the hot topics of Web research. Much of the research work, however, is concerned with request dispatching part of a Web server cluster [14,53,54,55,56]. Only a few research projects have put a focus on cooperative operations and the caching component. We discuss some of these directly or closely related projects below.

## DC-Apache

The *Distributed Cooperative Apache* (DC-Apache) Web server [11] dynamically modifies the hyperlinks in HTML documents to redirect client requests to multiple cooperating Web servers. A Web server in the pack will dynamically migrate or replicate documents to remote cooperating Web servers. To redirect future requests for migrated documents to the new servers, the origin server needs to modify all the hyperlinks pointing to the migrated documents.

Although the DC-Apache approach can achieve application-level request redirection without the help of any front-end dispatcher, the overhead of parsing HTML documents could be significant, especially when the number of documents is large. This puts a constraint on the ability to dynamically migrate documents. The other problem is that when a document is migrated to a new node, all the documents, and not just those in one particular server node, containing a hyperlink to this document need to be modified. The result could be increased network traffic, CPU time, and memory (for the data structure to keep track of the documents' relationships), and the use of other system resources. It does not seem to be a scalable approach for a website with thousands of documents. Furthermore, frequently redirecting client requests to a new server will result in constant TCP connection setups and breakdowns, making it difficult to implement HTTP/1.1 persistent connections to improve the system performance.

### LARD

*Location Aware Request Distribution* (LARD) [9,56] is a mechanism to distribute requests among backend server nodes in a cluster environment. The prototype system uses a frontend server as a request dispatcher. The dispatcher maintains a location mapping of objects to backend servers. Incoming requests will be distributed to the backend server node according to this mapping. Restricted backend server cooperation is allowed by using a mechanism called backend request forwarding, which is similar to our remote fetching of object from home nodes. LARD uses a similar location-aware approach to ours to locate the resources. The main difference is that LARD relies on centralized information, whereas ours is a distributed solution. The obvious potential problem is that the frontend can easily turn into a bottleneck when the system scales up. Because only fetching from the designated node (equivalent to our home node) for an object is allowed, a serving node will become a bottleneck if there many requests happen to target at objects in that server. In addition, in order to keep the centralized LARD information updated, the backend servers need to communication with the frontend server constantly to exchange information, which adds further overhead to the frontend server.

### KAIST's Work

A distributed file system with cooperative caching to support cluster-based Web servers that use LARD request distribution policy is proposed by researchers at KAIST [57]. The proposed file system is based on the hint-based cooperative caching idea for distributed file systems. It uses a Duplicate-copy First Replacement (DFR) policy to avoid the eviction of valuable master copies from the cache. This policy will result in improved global cache hit rates because it avoids excessive duplication of objects in the cache, leaving more room to accommodate more different objects. Its weakness, however, is that, by replacing duplicate copies first, local cache hit rates will decrease, which will result in more inter-node fetching of cached copies. Although current LAN speeds are increasing rapidly, inter-node communication is still a major overhead for cooperative caching. As we have analyzed in the previous section, low local hit rates will lead to poor overall system performance even when the recorded global hit rate is high.

Furthermore, although this approach might work well at the distributed file system level, as discussed in Section 2, it has characteristics that are not shared by Web

object caching systems. File systems operate with file block granularity rather than object granularity. File system accesses usually need only a part of a file, and thus the block approach can help save cache space. On the other hand, Web requests always ask for a document in its entirety, and therefore block caching is not necessary or redundant. If an object's file blocks are scattered in multiple nodes, a single object request will result in much inter-node communications, which will add burden to the network and the nodes involved. File system level caching also lacks global access information, which is necessary for constructing aggressive caching algorithms like our approximated global LFU algorithms.

### WhizzBee

Whizz Technology's WhizzBee Web Server [58] is a cluster-based Web server that realizes cooperative caching. A WhizzBee cluster consists of a number of Web server nodes and a dispatcher. The dispatcher accepts incoming Web requests from the clients and evenly distributes the requests to the Web server nodes based on their individual CPU load and network utilization.

One of the innovative features of WhizzBee is its Extensible Cooperative Caching Engine (EC-Cache Engine). Each Web server node contributes a portion of its physical memory to the cluster's global, cooperative cache. A file fetched from the I/O device will be cached in the physical memory. When another Web server node needs the same file, it will ask its peer to perform a cache-forwarding operation, instead of initiating an I/O access itself.

WhizzBee runs a centralized Cache-Load Server (CLSERV) at the dispatcher to coordinate the access of the cooperative cache. When a Web server node receives an HTTP request, it first queries the CLSERV to see whether the requested file has been cached in other Web server nodes. If the file is cached in one or more Web server nodes, CLSERV will recommend the best available Web server to perform the cache-forwarding operation based on the dynamic CPU load information. Otherwise, the Web server will retrieve the file from the I/O device, and inform CLSERV that this file has been cached.

Although WhizzBee and our Web server share a number of common features, there are three major differences between them. First, WhizzBee uses a centralized CLSERV to keep track of all the cached objects in the cluster. While this approach is good enough for small- to medium-sized clusters, it may not be an ideal choice for

large clusters as the CLSERV could become a performance bottleneck. Second, the access counter of a cached object in a WhizzBee server node is replicated to other server nodes during the cache-forwarding operations, while in our Web server this is done by periodic updates to the home node of each cached object. WhizzBee's approach ensures higher accuracy of the access counters, which in turn can improve the effectiveness of cache replacement. The price however is that it consumes more network bandwidth for synchronizing the access counters. This is actually a tradeoff between the effectiveness of the cache replacement strategy and the efficiency of the synchronization protocol. From the experimental results as presented in the previous section, our less expensive approximation-based approach, which is an advantage when considering scalability, seems to yield a performance level that is reasonably acceptable.

# 7 Conclusion and Future Work

Our experience with the global object space shows that the use of physical memory as the first-level cache can lead to improved Web server performance and scalability. Even with just a relatively small portion of the memory at each node that is set aside for object caching, a high cache hit rate can be achieved if the right cache policies are used. Cooperative caching will further increase the global cache rate because of more efficient use of the collective cache space. By using global access information, cache managers can make more accurate cache replacement decisions. Approximated global access information has been shown to be good enough for identifying the hot objects.

Our experimental results also indicate that, between replicating more hot objects and squeezing more different objects into the global object space, the choice should be to replicate more hot objects. These hot objects account for most of the requests, and they tend to come in bursts, and what bursts need is fast and parallel handling by the servers. This also explains why a higher local cache hit rate is more important than a higher global cache hit rate in a cluster-based Web server system. Thus, the benefit that cooperative caching can bring to a cluster-based Web server is not only the increased global cache hit rate, but also, perhaps more importantly, the gain in local access times due to the global sharing of object access patterns.

Our future work includes a re-examination of the overheads of the global object space. The cooperative caching mechanism has room for further improvement, such

as a smart forwarding scheme where the home node of an object will ask a peer node holding a cached copy to forward the copy to the requesting node directly.

Currently, our prototype system is designed to handle mainly static contents. As the use of dynamic contents is being more and more commonplace, it is imperative that the global object space be eventually extended to support dynamically generated contents. The extension requires a more intelligent and rigorous mechanism for handling caching consistency.

Finally, although our prototype system is built on top of the Jigsaw Web server, the global object space implementation can be easily ported to other Web servers. The advantage with using the Java-based Jigsaw Web server of course is the cross-platform capabilities provided by Java. The cooperative caching mechanism is not only suitable for cluster-based Web servers, but also cluster-based proxy servers which share much similarities with Web servers.

## Acknowledgements

Special thanks go to Roy S.C. Ho for providing information on the WhizzBee Web server, and the referees for their useful comments on a preliminary version of this paper.

## References

1. The State of the Internet 2000. *http://www.usic.org*. Accessed at December 20, 2001.
2. Lake D. The need for speed. *http://www.idg.net/ic_527364_5054_1-3901.html*. Access at December 20, 2001.
3. Thompson K, Miller G, Wilder R. Wide-Area Internet Traffic Patterns and Characteristics. *IEEE Network* 2000; **11**(6):10-23.
4. Hu J, Schmidt D, JAWS: A Framework for High-performance Web Servers. *http://www.cs.wustl.edu/~jxh/research/research.html*. Accessed at December 20, 2001.
5. Pai V, Druschel P, Zwaenepoel W. Flash: An Efficient and Portable Web Server. *Proceedings of the 1999 USENIX Annual Technical Conference*, Monterey, CA, USA, June 1999.

6. kHTTPd Linux HTTP Accelerator. *http://www.fenrus.demon.nl*. Accessed at December 20, 2001.
7. Burns R, Rees R, Long D. Efficient Data Distribution in a Web Server Farm. *IEEE Internet Computing* 2001; **5**(4):56-65.
8. Iyengar A, Challenger J, et al. High-Performance Web Site Design Techniques. *IEEE Internet Computing* 2000; **4**(2):17-26.
9. Aron M, Druschel P, Zwaenepoel W. Efficient Support for P-HTTP in Cluster-Based Web Servers. *Proceedings of the 1999 USENIX Annual Technical Conference*, Monterey, CA, June 1999.
10. Holmedahl V, Smith B, Yang T. Cooperative Caching of Dynamic Content on a Distributed Web Server. *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC-7)*, Chicago, IL USA, July 1998.
11. Li Q, Moon B. Distributed Cooperative Apache Web Server. *Proceedings of the tenth World Wide Web Conference*, Hong Kong, May 2001.
12. Bryhni H, Klovning E, Kure O. A Comparison of Load Balancing Techniques for Scalable Web Servers, *IEEE Network* 2000; **14**(4):58-64.
13. Richard B, Derek L, et al. Achieving Load Balance and Effective Caching in Clustered Web Servers. *The fourth International Web Caching Workshop*, San Diego, CA, USA, March 31-April 2, 1999.
14. Cardellini V, Colajanni M, Yu P. Dynamic Load Balancing on Web-Server Systems. *IEEE Internet Computing* May-June 1999; **3**(3):28-39.
15. Andresen D, Yang T, et al.. SWEB: Towards a Scalable World Wide Web Server on Multicomputers. *Proceedings of International Conference of Parallel Processing Symposium (IPPS)*, Honolulu, Hawaii, USA, April 1996.
16. Schroeder T, Steve G, Byrav R. Scalable Web Server Clustering Technologies. *IEEE Network* 2000; **14**(3):38-45.
17. David D, Shrivastava S, Panzieri F. Constructing Dependable Web Services. *IEEE Internet Computing* 2000; **4**(1):25-33.
18. Schroeder T, Goddard S. The SASHA Cluster-Based Web Server. *http://www.zweknu.org/tech/src/lsmac2/paper/paper.html*. Accessed at December 20, 2001.
19. Brewer E. Lessons from Giant-Scale Services. *IEEE Internet Computing* 2001; **5**(4):46-55.

20. Wang J. A Survey of Web Caching Schemes for the Internet. *ACM Computer Communication Review (CCR)* 1999; **29**(5).
21. Barish G, Obraczka K. World Wide Web Caching: Trends and Techniques. *IEEE Communications Magazine* 2000; **38**(5):178-184.
22. Davison B. A Web Caching Primer. *IEEE Internet Computing* 2001; **5**(4):38-45.
23. Menaud J, Issarny V, Banatre M. A Scalable and Efficient Cooperative System for Web Caches. *IEEE Concurrency* 2000; **8**(3):56-62.
24. Wolman A, Voelker G, et al. On the scale and performance of cooperative Web Proxy Caching, *ACM Operating System Review* 1999; **34**(5):16-31.
25. Korupolu M, Dahlin M. Coordinated Placement and Replacement for Large-Scale Distributed Caches. *Proceedings of the 1999 IEEE Workshop on Internet Applications*, San Jose, CA, USA, July 1999.
26. Sarkar P, Hartman J. Hint-Based Cooperative Caching. *ACM Transactions on Computer Systems* 2000; **18**(4):387-419.
27. Cortes T, Girona S, Labarta J. PACA: a Cooperative File System Cache for Parallel Machines. *Proceedings of the Second International Euro-Par Conference*, Lyon, France, August 1996.
28. Dahlin M, Wang R, et al. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. *Proceeding of First Symposium on Operating Systems Design and Implementation*, November 1994.
29. Tatarinov I. Performance Analysis of Cache Policies for Web Servers. *Proceedings of 9th International Conference on Computing and Information*, Manitoba, Canada, 1998.
30. Jigsaw-W3C's Server. *http://www.w3c.org/Jigsaw*. Accessed at December 20, 2001.
31. Arlitt M, Williamson C. Trace-Driven Simulation of Document Caching Strategies for Internet Web Servers. *Simulations* 1997; **68**(1):23-33.
32. Arlitt M, Friedrich R, Jin T. Performance evaluation of Web proxy cache replacement policies, *Performance Evaluation* 2000; **39**(1-4):149-164.
33. Pierre G, Kuz I, et al. Differentiated Strategies for Replicating Web Documents. *Proceedings of 5th International Web Caching and Content Delivery Workshop*, Lisbon, Portugal, May 2000.
34. Anderson T, Dahlin M, et al. Serverless Network File Systems. ACM transaction on Computer Systems 1996; **14**(1):41-79.

35. Tatarinov I. Performance Analysis of Cache Policies for Web Servers. *Proceedings of 9th International Conference on Computing and Information*, Manitoba, Canada, June 1998.
36. Cohen E, Kaplan H. Exploiting Regularities in Web Traffic Patterns for Cache Replacement. *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing (STOC99)*, Atlanta, Georgia, USA, May, 1999.
37. Cohen E, Kaplan H. The Age Penalty and its Effects on Cache Performance. *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, San Francisco, CA, USA, March 2001.
38. Reddy N. Effectiveness of Caching Policies for a Web Server. *Proceedings of the 4th International Conference on High Performance Computing*, Cambridge, Massachusetts, USA, December 1997.
39. Robinson J, Devarakonda M. Data Cache Management Using Frequency-Based Replacement. *Proceedings of the 1990 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, Boulder, Colorado, USA, May 1990.
40. Williams S, Abrams M, et al. Removal Policies in Network Caches for World Wide Web Documents. *Proceedings of ACM SIGCOMM '96*, Stanford University, CA, USA, August 1996.
41. Rizzo L, Vicisano L. Replacement Policies for a Proxy Cache, *UCL-CS Research Note RN/98/13*, Department of Computer Science, University College London, 1998.
42. Gadde S, Chase J, Rabinovich M. A Taste of Crispy Squid. *Proceedings of Workshop on Internet Server Performance (WISP98)*, Madison, Wisconsin, USA, June 1998.
43. Chankhunthod A, Danzig P, Neerdaels C. A Hierarchical Internet Object Cache. *Proceedings of the USENIX 1996 Annual Technical Conference*, San Diego, California, January 1996.
44. Fan L, Cao P, et al. Summery Cache: A Scalable Wide-Area Web Cache Sharing Protocol, *IEEE/ACM Transactions on Networking* 2000; **8**(3):281-293.
45. Internet Cache Protocol. *http://icp.ircache.net*. Accessed at December 20, 2001.
46. Protic J, Tomasevic M, Milutinovic V. Distributed Shared Memory: Concepts and Systems. IEEE Parallel & Distributed Technology: Systems & Applications 1996; **4**(2):63-71.

47. Cano J, Pont A, et al. The Difference between Distributed Shared Memory Caching and Proxy Caching. *IEEE Concurrency* 2000; **8**(3):45-47.
48. Arlitt M, Williamson C. Internet Web Servers: Workload Characterization and Performance Implications. *IEEE/ACM Transactions on Networking* 1997; **5**(5):631-645.
49. Baker S, Moon B. Distributed Cooperative Web Servers. *Proceedings of the eighth World Wide Web Conference*, Toronto, Canada, May 1999.
50. HTTP-Hypertext Transfer Protocol. *http://www.w3c.org/Protocols*. Access at December 20, 2001.
51. Mogul J. The Case for Persistent-Connection HTTP. *Proceedings of the ACM SIGCOMM '95 Symposium*, Cambridge, USA, August 28-September 1, 1995.
52. Mosberger D, Jin T. httperf – A Tool for Measuring Web Server Performance. *Proceedings of Workshop on Internet Server Performance (WISP98)*, Madison, Wisconsin, USA, June 1998.
53. Schroeder T, Goddard S, Ramamurthy B. Scalable Web Server Clustering Technologies, *IEEE Network* 2000; **14**(3):38-45.
54. Crovella M, Frangioso R, Harchol-Balter M. Connection Scheduling in Web Servers. *Proceedings of the 1999 USENIX Symposium on Internet Technologies and Systems (USITS '99)*, Boulder, Colorado, October 1999.
55. Colajanni M, Yu P, Dias D. Analysis of Task Assignment Policies in Scalable Distributed Web-server Systems. *IEEE Transaction on Parallel and Distributed Systems* 1998; **9**(6):585-598.
56. Pai V, Aron M, et al. Locality-aware Request Distribution in Cluster-based Network Servers. *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, California, October 1998.
57. Ahn W, Park S, Park D. Efficient Cooperative Caching for File Systems in Cluster-Based Web Servers. *Proceedings of IEEE International Conference on Cluster Computing*, Chemnitz, Germany, November 28-December 1 2000.
58. Whizz Technology. http://www.whizztech.com. Accessed at December 20, 2001.