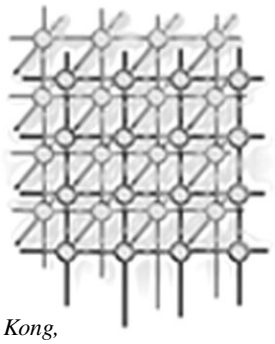
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2003; **15**:681–705 (DOI: 10.1002/cpe.723)

p-Jigsaw: a cluster-based Web server with cooperative caching support[‡]



Ge Chen[*,†], Cho-Li Wang and Francis C. M. Lau

Department of Computer Science and Information Systems, The University of Hong Kong, Hong Kong

SUMMARY

Clustering provides a viable approach to building scalable Web systems with increased computing power and abundant storage space for data and contents. In this paper, we present a pure-Java-based parallel Web server system, *p-Jigsaw*, which operates on a cluster and uses the technique of *cooperative caching* to achieve high performance. We introduce the design of an in-memory cache layer, called *Global Object Space* (GOS), for dynamic caching of frequently requested Web objects. The GOS provides a unified view of cluster-wide memory resources for achieving location-transparent Web object accesses. The GOS relies on cooperative caching to minimize disk accesses. A requested Web object can be fetched from a server node's local cache or a peer node's local cache, with the disk serving only as the last resort. A prototype system based on the W3C 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. Copyright © 2003 John Wiley & Sons, Ltd.

KEY WORDS: 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. The booming popularity of broadband connection adds fuel to the flame, creating a

*Correspondence to: Ge Chen, Department of Computer Science and Information Systems, The University of Hong Kong, Hong Kong.
†E-mail: gechen@csis.hku.hk

‡A preliminary version of this appeared in *Proceedings of IEEE International Conference on Cluster Computing (Cluster 2001)*, Newport Beach, CA, October 2001, under the title 'Building a scalable Web server with Global Object Space support on heterogeneous clusters'.

Contract/grant sponsor: The University of Hong Kong; contract/grant number: 10203944

Received 15 December 2001
Revised 1 August 2002

great need for building more powerful Web servers that can handle large volumes of HTTP requests efficiently.

There are generally two approaches to a more powerful Web server: (1) to deploy a single, cutting-edge server machine with advanced hardware support and optimized server software; (2) to rely on the aggregate power of parallel machines.

Several pioneering projects in Web server architecture have developed sophisticated technologies aiming to increase the performance of a single Web server [1–3]. The limiting factor, however, is with the hardware. With the rapid advances in hardware technologies, a powerful server today can easily become obsolete tomorrow, and major upgrades of the server hardware tend to be expensive and require substantial effort. A more sustainable solution is to employ multiple servers, in the form of a server farm (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 [4–7], with focuses on issues such as load distribution or balancing [8–10], scalability [11,12], and high availability [13,14]. These projects have shown that clustering is a promising approach to building scalable and high-performance Web servers [4,15].

One of the key techniques to increasing Web serving performance is Web object caching. Web caching can be adopted at different stages in the processing of a Web request. On the client side, a Web browser can serve the request using a cached copy directly from its local memory or on disk without going to the network. A proxy server situated between a client and the Web server can intercept and fulfill the request with cached copies held in memory due to previous requests by other clients. On the server side, the Web server can return the requested pages from its memory cache to avoid disk accesses. All three scenarios can help achieve faster response time as perceived by the user. Some recent surveys of Web caching can be found in [16–18].

Even though substantial work has been done on Web object caching in a single Web server or among cooperating proxy servers in a WAN environment, not so much work has been targeted at caching in cluster-based Web server systems. Because of the distributed nature of a cluster, a simple caching algorithm will probably not be able to fully utilize the power of a cluster, which is founded upon a fast interconnecting network, giant aggregate memory space, and parallel I/O.

In this paper, we present the design and implementation of a cluster-based Web server system, *p-Jigsaw*, which uses a *cooperative caching* mechanism for efficient sharing of Web objects among the participating Web server nodes. The server nodes share the processing of client requests in a single-queue, multiple-server fashion. The cooperative caching mechanism creates a logical, cluster-wide cache layer containing objects that can be accessed by any server node. The layer integrates all the deployable physical memory in each node to store the cacheable part of the site's contents. A *cache forwarding* operation is invoked whenever necessary to transfer cached Web objects from one server node to another. The 'cooperation' happens when there is a local cache miss—instead of fetching the object from disk, a request for a copy of the object is sent to a peer node. This is feasible because of the very fast transfers of objects and more accurate updating of cache state made possible by the high-speed LAN interconnecting the nodes. This is in contrast to cooperative caching support in geographically distributed Web server systems that work over long distance connections, where very sophisticated protocols are needed to achieve coarse-grained object sharing [19–21].

As dealing with objects in memory is many times faster than objects on disk, the cluster nodes can respond quickly to incoming requests and can therefore collectively handle more simultaneous

requests. With cooperative caching, we can obtain a global view of resource utilization, and can therefore devise means to optimize on the use of system-wide resources. Because we are able to fetch an object from a peer node's cache, a higher global cache hit rate can then be achieved as each node sees and accesses the same giant cache memory spanning all the server nodes. In comparison with any similar system without cooperative caching, this solution can effectively avoid excessive disk accesses, thus resulting in an increased overall system throughput.

The idea of cooperative caching was explored in distributed file system design. The purpose was to try to reduce accesses to the file server by allowing file blocks to be cached in a system-wide memory cache. 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 [20–24]. Web servers in fact can rely on the underlying file system for implicit object caching at the file block level. Web objects, however, have several features that make file system caching solutions not applicable [25], as will be discussed in Section 2.

Cooperative caching in our design is supported by a *Global Object Space* (GOS) layer that integrates multiple disparate memory chunks into a single logical memory space. In each cluster node, a per-node object cache manager is responsible for Web object caching as well as implementing the cooperation protocol. With cooperative caching, a Web object being requested can be served from a server's local cache or from a peer node's cache. Only failing both will the file system be accessed. The location where a cached object resides is transparent to the requesting client. Without cooperative caching, a cache miss will inevitably result in a disk access to the file system. The GOS provides a uniform cache space that extends over all the cluster nodes, and a mechanism for cluster nodes to cooperatively maintain useful global information to facilitate execution of effective cache placement and replacement policies.

A prototype based on the proposed design has been implemented by modifying W3C's Jigsaw Web server [26] to serve static Web contents. We chose Jigsaw because of its pure-Java implementation, which is easy to modify, and the resulting Web server can operate in a heterogeneous environment. The caching mechanism of the prototype considers several key issues that affect the Web server performance, including object size and frequency of reference. Unlike experiments by many others that are based on trace-driven simulations [27–29], 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. 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 information 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 GOS. Section 5 presents benchmarking results for 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 in many areas in systems building, such as operating systems and networking systems. The chief benefit of caching if properly deployed is

increased performance in certain system operations. We discuss in this section existing solutions for caching in several kinds of systems, including distributed file systems, proxy servers, and Web servers, and argue that Web object caching in cluster-based Web servers requires a design that is different in certain respects from other systems.

2.1. File caching versus Web object caching

Several distributed file system projects have incorporated the idea of cooperative caching in their design. xFS is a distributed file system developed as a part of Berkeley's NOW project [30]. xFS pioneered the idea of using remote memory as caching space, which became feasible with the advent of fast network communication in switched local area networks. xFS is a serverless system where participating workstations as peers cooperate to provide all the needed file system services.

Hint-based cooperative caching file systems [22] propose to rely more on local state information in managing the cache, thus leading to simplicity in design and less overhead. In hint-based cooperative caching, information about possible locations of the file blocks is sent to the client. This information is derived from local knowledge about the global state, which is not guaranteed to be accurate—hence the name 'hint'. When a local cache miss on a certain file block occurs, the client will refer to the hints it has, and try to request the block from another location according to the hints; the client will get either the block from the peer or an updated hint about the block's current location. Hint-based cooperative caching achieves performance comparable 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.

Block-level file caching cannot be used as a substitute for Web object caching, for the following reasons. 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. Secondly, caching is not obligatory in Web servers or proxy cache servers, which is unlike the file system buffer that always places those recently requested data blocks in the cache. Since cache space is relatively scarce, Web caching solutions need to strive a balance between caching more small objects or fewer large objects. Thirdly, Web servers seldom experience a burst of requests for the same object from a single client because the requested object would likely be already cached in the client's local cache, or in some intermediary proxy. Thus, a single object request should not be a strong enough reason for cache placement or replacement since there might not be any following correlated accesses. On the other hand, multiple accesses to the same object from different clients in a short period of time do indicate high popularity of a document. Hence, Web caching requires a design that is different from that for file caching.

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 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 disk operation to bring 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. Web object caching can be very effective if 'hot' objects can all be cached. Therefore, traditional LRU replacement algorithms used in file caches are not suitable for Web document caching. Instead, frequency-based replacement algorithms, such as Least Frequently Used (LFU) and its variants, have been shown to be most effective for Web caching [27,31–34].

As mentioned, 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). In the design of a Web server system, it is possible to break up a large Web file into small pieces, in the same way that a file system does to files, and to cache just some of the pieces, but to our knowledge no research so far has indicated that this is practical and profitable. Variable-sized Web objects complicate the design of the memory management function in Web server caches.

2.2. Web server caching versus proxy caching

Web object caching can be adopted at different points along the path connecting the client to the Web server: the Web server, the proxy server, and the client.

Web browsers are examples of end-user clients and they cache recently visited Web pages in their local memory or on disk in order to shorten future visits to these pages. Although client caching can improve the access time for visits to the same page by the client in the future, it offers no help to other requests issued by other clients, because their caches are not shared.

Proxy servers can be 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 in a proxy server, it will be used to serve the client, thus reducing the client's perceived response time, and saving network bandwidth between the proxy and the Web server. A cache miss at the proxy will cause the requested object to be served from the Web server via the proxy. Proxy servers are usually placed near the edge 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 users in the same organization or community can lead to significant bandwidth savings for external Web access, improved response time, and increased availability of static data and objects [17].

Reverse proxy servers are proxy servers located on the Web server's side instead of close to the clients. A reverse proxy server can cache Web objects that originate from the Web server (or multiple such servers) connected to 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 [35–37].

Proxy servers aim at being able to handle all possible objects requested by their clients. The number of objects that a proxy server needs to handle is conceivably far greater than that of a reverse proxy server; the latter only needs to cache objects from a single Web site or several Web sites. Therefore, proxy servers require ample caching space to make the caching mechanism effective. Such a large space can come from cooperative caching spanning multiple servers in a LAN or a WAN [38–40]. Sophisticated mechanisms have been proposed for individual proxy servers to lookup objects, to exchange object information, and to do other necessary operations across WAN connections. In the following, we discuss two well-known cooperative proxy cache systems. A good survey of various Web caching schemes can be found in [16].

- *Harvest and ICP*. Harvest cache [39] 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 has evolved into the Internet Cache Protocol (ICP) [41]. 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 offering the lowest latency. In the 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 [40] 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.

Web object caching can be used in both proxy servers and Web servers (reverse proxy). The object access pattern and caching behavior of the two cases bear a certain resemblance. Several important differences, however, make existing cooperative caching mechanisms used by proxy servers not suitable for cluster-based Web servers.

- *Client and request characteristics.* Proxy servers, being at the edge of the Internet, tend to meet with 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 remote proxy servers. As proxy servers and clients have their own caching, requests for a certain object from a Web server tend to decline in number faster than the same 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 for a proxy server.
- *Scale of the object space.* A Web server handles objects hosted within a single site, while a proxy deals with requests that can target theoretically all reachable Web sites in the world.
- *Network connections.* Cooperative proxy servers usually work in a WAN environment whereas cluster-based Web servers consist of multiple server nodes located 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; it is not practical to adopt complicated caching algorithms because of the heavy overheads of network communications [40]. In contrast, for a cluster-based Web server in a high-speed LAN environment, it is possible, and in fact necessary, to obtain more accurate and fine-grained global state information, in order to locate cached objects and coordinate object replacement decisions.
- *Disk cache versus RAM cache.* The Web server can rely on a large in-memory cache to improve the performance. The use of in-memory cache may not have an impact on the proxy's performance, as the memory available in a proxy is but a tiny fraction of the total object space needed to accommodate all potential client requests. In addition, it is more meaningful for a cluster-based Web server system to exploit in-memory cooperative caching mechanism in a LAN environment because of the high-speed networking support. On the other hand, the latency

difference for accessing an object stored in a remote proxy's memory versus one that is on disk is insignificant due to the long Internet delay.

2.3. Software distributed shared memory

One of the most popular parallel programming support paradigms is software distributed shared memory (DSM), which offers an abstraction of a globally shared memory across physically distributed memory machines. DSM implementations also make use of caching for more efficient accesses to shared objects.

To a certain 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 [42].

- *Granularity.* In DSM systems, the granularity of caching or other operations is usually determined by the implementation. A frequently used grain size is the memory page. Software DSM solutions favor coarse grain sizes due to performance and complexity considerations. In Web object caching, the granularity of caching is usually an object, and Web objects can have various sizes, from a few kilobytes to several megabytes or larger.
- *Replacement algorithms.* In DSM systems, 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. However, 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 system needs to support many different complicated reference and memory sharing models used by all kinds of applications running on top of it. Newer DSM systems increasingly adopt the multiple reader/multiple writer approach where participating computation nodes in a cluster are on equal ground and can issue changes to data concurrently [42]. However, changes to Web objects usually come from a single master site, and so the consistency is less of an issue.
- *Locality types.* In DSM systems, spatial locality and temporal locality are usually observed to an advantage. In Web object caching, there is little spatial locality because links in a document can refer to any object in the system, and not necessarily to pages located nearby in any sense. Temporal locality is not a feature in Web object caching either, because a single user visiting a Web page does not imply the same user will visit the page again, let alone that other users will visit the page. However, a burst of concurrent requests for the same page does 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 shares certain similarities with cooperative caching in distributed file systems, proxy servers, and DSM. For example, the general rules on the choice of 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 embodies 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-based Web server implementing our proposed mechanism can offer to its clients. With a giant cache

space that spans the entire cluster, higher cache hit rate can be achieved, which is an important indicator of success. With the fast and reliable underlying LAN connection, cluster nodes can exchange information about access patterns and caching situations efficiently and with relatively little overhead; such information is crucial in the system's ability to achieve optimal performance. Cooperative caching reduces substantially accesses to the disk, leading to better throughput of the Web server, more requests being served at the same time, and higher availability as seen by the clients.

3. THE p-Jigsaw SYSTEM ARCHITECTURE

We first present an overview of the design of the GOS and the major building blocks of our cluster-based Web server system. The GOS presents a giant cache space that spans the entire cluster to the running programs. Several cache replacement algorithms, which serve as an important policy module for the cooperative caching mechanism to achieve higher cache hit rate, will be discussed.

3.1. Overview

The p-Jigsaw Web server uses the GOS for caching frequently accessed static Web objects. The GOS is built from the physical memory of all the participating cluster nodes. Each node contributes a chunk, resulting in a very large system-wide memory cache. Cooperative object caching is the core mechanism used in the GOS, which upon a local cache miss triggers the forwarding of an object from a peer node instead of fetching the object from the disk. The amount of memory set aside per node for GOS is a parameter to be tuned when the system is deployed in reality. The use of cooperative object caching in a cluster-based Web server system is justifiable because of today's high-speed networking technologies that are readily available in a cluster environment. With such high-speed networking support, to access a data item in a peer node's memory would take less time than to access a disk-bound data item. A performance comparison between remote memory access and local disk access is provided in Section 5.2.

Figure 1 shows the basic architecture of our cluster-based Web server design. In the system, each server node sets aside and maintains a special memory segment, called the *hot object cache* (HOC), for Web object caching. The collection of HOCs coalesces into a globally shared object cache. A HOC is also referred to as a node's local object cache. All Web objects stored in the GOS 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, known also as the node's cache manager, is responsible for managing the node's HOC, as well as for working cooperatively with all the other nodes to provide location-transparent object accesses. With the GOS support, a Web server node can refer to any Web object in the system without having to know its physical address, which is similar to the function of a software DSM system.

Each cached object has a fixed *home node*, which is the node that keeps the most updated copy and approximated global access information of the cached object. We allow a popular object that has become 'hot' to be present in multiple HOCs. An object's home node is responsible for keeping track of the location information of all the copies of the object and their access statistics.

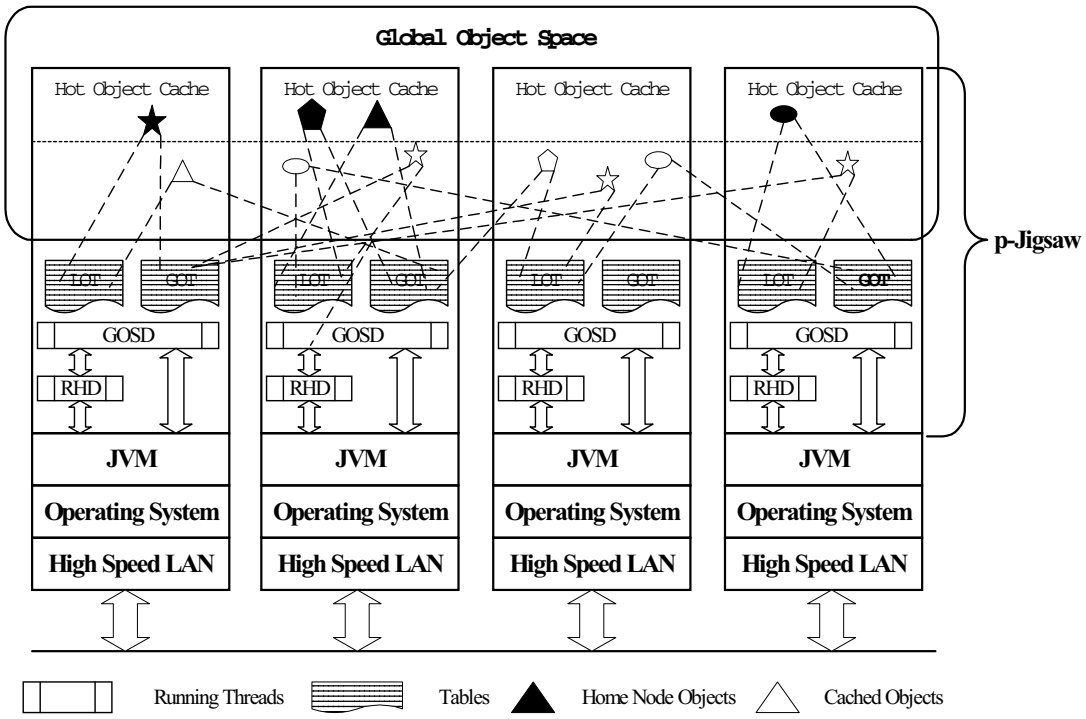


Figure 1. p-Jigsaw system architecture.

During execution, the RHD listens on the TCP port and is prepared to handle any incoming HTTP request. It parses and analyzes the incoming request and then forwards the result to the GOSD for processing. The GOSD can serve the request from its local HOC. If it is a cache miss, the request is then passed on to the home node of the object where the location of the requested object will be determined.

3.2. Hot objects

Much research effort has been directed at the characterization of Web workloads and the performance implications they have for Web servers [20,43,44]. One of the major results is the notion of *concentration* [43], which explains 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, our system 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.

In p-Jigsaw, a hot object can be replicated and cached in multiple HOCs in order to achieve good cache hit rates. Because of the distributed nature of the system, the accurate count of how many times an object has been accessed 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. Support for persistent connection

HTTP/1.1 [45] 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 setup time, resulting in improved latency and performance [46]. In our proposed system, because all the objects in the system are accessible via the GOS, a request-handling node can keep persistent connection with the client by fetching the necessary objects from other nodes via the GOS. Without a GOS to allow object sharing 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.

3.4. Requests distribution

For completeness of the description, we discuss the request distribution mechanism used in p-Jigsaw. In p-Jigsaw, a Web switch or load balancer sits at the entrance of the cluster and distributes the requests to the Web server nodes by a Layer-4 dispatching mechanism. All the Web server nodes share a single virtual cluster IP address. The distribution decision is made based on a simple round-robin scheme or on the load situation across all the nodes. The dispatcher does not accept connections from the clients. It only routes packets to the chosen node, which establishes the TCP connection directly with the client. Since TCP is a connection-oriented protocol, the endpoints of a connection will not change. The dispatcher has to route the packets to the same Web server node for the same connection. The server can send the replies to the clients directly without passing through the dispatcher. Load-balancing Web servers work in a similar way, but use load information for where to dispatch a request to [8,10,12].

4. COOPERATIVE CACHING IN THE GOS

The GOS is the core of our design. The use of the GOS can reduce accesses to the disk, leading to better throughput of the Web server, more requests being served at the same time, and higher availability as seen by the clients. The effectiveness of the GOS relies on the fast and reliable underlying LAN connection for fast exchange of object access patterns and status information with minimal overhead. In this section, we present details of the design of the cooperative caching mechanism which is the most crucial component of the GOS.

4.1. Tables and counters

To build and maintain the GOS, two tables are defined and managed by the local GOSD: (1) the *local object table* (LOT) and (2) the *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, managed by the home node of the object, contains an approximation to the total number of accesses received by all the participating nodes for the object since the counter was initialized.
- *Local access counter (LAC)*. This counter, managed by each individual server node having a cached copy of the object, 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 of a node contains local access information and records of objects cached in the node's hot object cache, including the AGAC, the LAC, and the home node address of each locally cached object. The GOT maintains two kinds of information: a mapping between object IDs (URLs in the current implementation) and home node numbers for all objects in the system; and global information related to those objects whose home node is this node. The global information includes, for each object, the AGAC and node addresses of all existing cached copies. All information is updated dynamically during runtime.

All Web servers require a file mapping table to convert a request string to an object ID that refers to a file in the file system. When the system scales up, a full mapping of object IDs to locations will make the mapping table extremely large and will consume a lot of memory. Such a large in-memory table will compete for memory space with the object caches, leading possibly to poor cache performance. Furthermore, as the mapping table scales, the lookup time will increase, which will result in longer response time. For example, in Jigsaw's lookup process, where an object ID is mapped to a file path, when the lookup table becomes larger than some predefined size, the system will be busy swapping entries of the lookup table to the file system. This results in heavy file I/O traffic. Our test shows that, the lookup time may account for as much as 70% of the request handling time using a Jigsaw Web server**.** Therefore, a scalable system should try to keep the table as small as possible.

In order to reduce the size of the GOT, we propose a partitioning mechanism that optimizes on the use of memory. This design originates from the observation that in a Web server the file organization of a Web site usually follows the tree structure. Our partitioning mechanism makes it unnecessary to keep an entry in the table for every object in the Web site. A hash table is used as the GOT. All the objects having the same common part in its object ID will share the same entry in the table. If all the objects' home nodes under a directory are in fact the same server node, then all these objects will share a common entry key. For example, in a tree structured Web site, if the HTML files under directory '/root/dir1/dir12/' have the same home node, all the files under this directory will share the a single entry in the GOT table. There is only one key, '/root/dir1/dir12/', to all the files under 'dir12'. This can greatly reduce the mapping table size if the Web site is well organized.

4.2. Workflow of the GOS

Figure 2 depicts the workflow for obtaining an object from the GOS when an object request is received by a node's GOSD. The GOSD will search for the requested object in the local hot object cache. If this

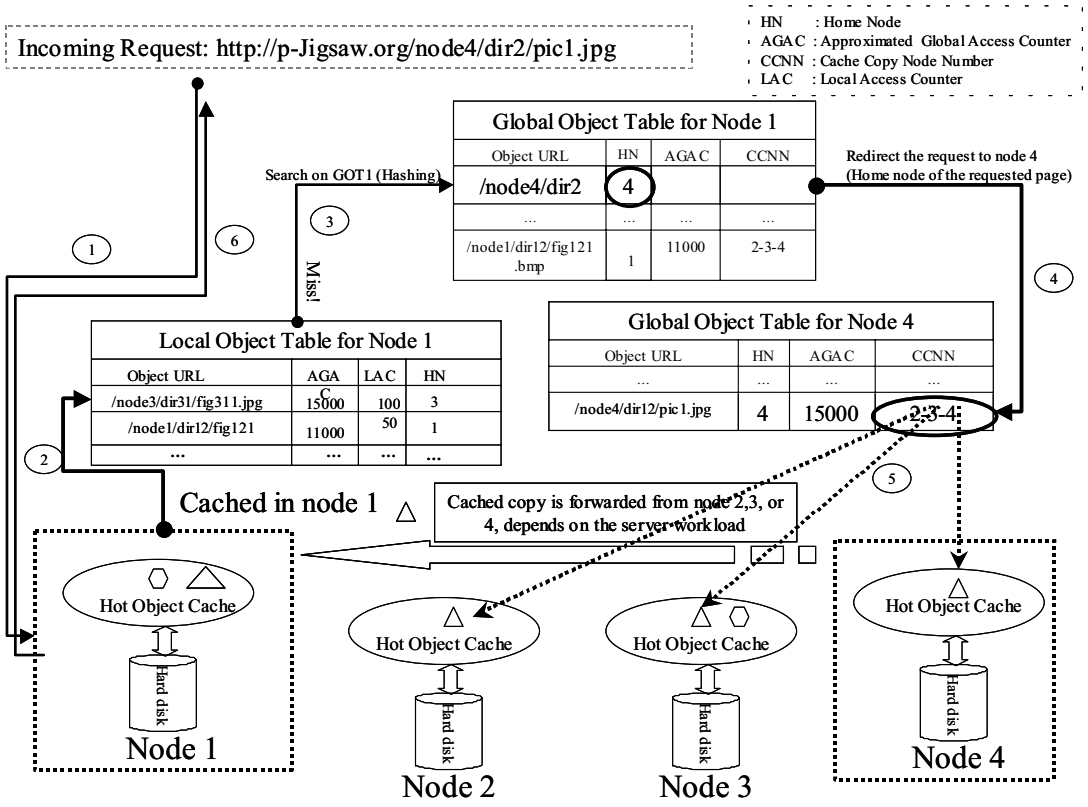


Figure 2. Workflow of GOS.

very first search fails, the GOSD will forward the request to the object's home node based on the hashing mechanism discussed in Section 4.1. The home node will respond to the request with a cached copy from its local object cache, if one exists. If the home node does not have a cached copy, or to achieve load balancing (i.e. the home node is already overloaded), the home node would respond with a message indicating where a cached copy may be obtained. Allowing other nodes to serve the request can avoid overloading the home node, especially when the home node is home for many hot objects. Note that the original server node that first received the request continues to be the one that 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.

4.3. Approximated global object access statistics

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 keeps a count of accesses that would contribute to the total number of accesses to the object. The following describes how the access counters for a certain object cached in the GOS 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 contained 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 a peer node, it will add the received LAC value to the object's AGAC. The new AGAC value is then fed back to the reporting node. For cost reasons, this update operation is done by each node asynchronously without a global synchronization among or broadcast to all server nodes. 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 can be reflected quickly, and objects receiving such bursts of 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 are past approximated values obtained from the home node. This is unavoidable in a distributed environment due to the costs of maintaining global information. Nevertheless, a reasonably good approximation should work well for our purpose.

4.4. Cache replacement algorithm

In our system, cache replacement decisions are made by individual nodes with help from the AGACs. Every time a new object is referenced, locally or by another server node, the GOSD will try to insert the object into the local hot object cache. The AGAC records how many times the object has been accessed recently by all nodes, thus indicating how popular the object is; the counter is updated asynchronously 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. We have tried two algorithms based on the LFU criterion, LFU-Aging and Weighted-LFU.

- *LFU-Aging*. The aging part is due to the fact that any information about the popularity of an object, if not updated, becomes less of a reliable indicator of the object's popularity as time passes [32]. This is particularly true for Web object caching, since, as mentioned in Section 2, requests for the same object decline faster in Web servers 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 necessarily the 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 feedback 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 a log scale. The log scale is used because 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 only one can be accommodated.

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 and is the first to be discarded from the HOC during a replacement operation.

In addition to the above two LFU-based algorithms, a approximated Global Least-Recently-Used (GLRU) algorithm is also implemented and tested in order that we can 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 its variants are widely adopted in existing file system caches. With the GLRU algorithm, all the nodes are synchronized with a start time as the base time for future LRU information recording. The LRU information is exchanged asynchronously as in the LFU algorithms.

4.5. Cache consistency

HTTP/1.1 specification provides a simple expiration model to help Web object caches to maintain cache consistency [45]. 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 Jigsaw Java Web server, version 2.0.5 [26]. The GOS is added to Jigsaw in order to support cooperative caching. Three cache replacement algorithms, LFU-Aging, Weighted LFU, and GLRU were implemented and tested in a Linux PC cluster to evaluate the effects of cooperative caching with different cache replacement policies. A benchmarking of the network RAM access speed and sustained disk access speed was also performed to provide support for the appropriateness of cooperative caching.

Table I. Summary of data set characteristics (raw data set).

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

Table II. Summary of access log characteristics.

No. of requests	~640 000
Data transferred	~35 Gbytes
Distinct files requested	52 347
Total size of distinct files requested	~2.73 Gbytes

5.1. Experimental environment

We measured the performance of the p-Jigsaw 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 server nodes. Each of the server nodes has 392 Mbytes of memory.

The benchmark program is a modified version of the httperf program [47]. The httperf program performs stress tests on the designated Web server using a Web server log from the Computer Science Division, EECS Department, of UC Berkeley [48].

The main characteristics of the data set and the log file are summarized in Tables I and II, respectively. Because the data set is from an academic department's Web site, there are a number of files with size larger than a few megabytes. These are academic papers for visitors to download. Such files are common in FTP traffic, but not so common in a normal Web site, and so we filtered away some of these files.

The Web site documents are stored in an NFS server accessible by all server nodes. The NFS server is a dedicated PC server with two 450 MHz 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.

5.2. Network RAM speed versus disk access speed

Our design assumes that fetching data from a remote node's memory requires less time than fetching the same from the local disk. We conducted experiments to verify that this is the case. We used a benchmark programme to compare the performance of remote memory access and local disk access.

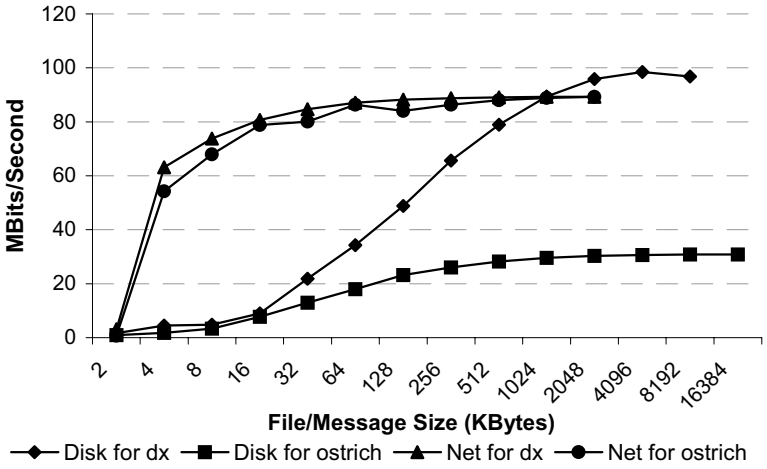


Figure 3. Network RAM speed versus disk access speed.

The program is written in pure Java, and we ran it on two sets of machines using the same JVM as we used for testing p-Jigsaw. The 'dx' machines (Figure 3) are four-way SMP PIII PC servers with Fast Ethernet adapters and SCSI hard disks. The 'ostrich' machines are PIII PCs with Fast Ethernet adapters and IDE hard disks. The experiments went through a period to warm up the system buffer by a long-time random reading of large file blocks, totaling over 500 Mbytes.

Figure 3 shows the sustained remote memory access performance and local disk access performance. From the figure, we can find out that, when the object size is small, remote memory access out-performs both local IDE and SCSI disk accesses. When the object size is large, local SCSI disk access will out-perform remote memory access, but the remote memory access is still much faster than that local IDE hard disk access. As the sizes of the Web objects generally are smaller than 100 kbytes, given the situation as shown in Figure 3, we can conclude that the technique of cooperative caching when used in cluster-based Web server systems could be effective in improving the overall performance. We will show benchmarking results in the following sections to demonstrate this.

It should be noted that, in our design, the transfer of cached objects between peer servers is the major cost in cooperative caching and works best with small-size objects. For larger objects, reading from the local SCSI hard disk is probably more efficient as the SCSI disk has a higher sustained transfer rate.

5.3. Effects of scaling cluster size

In the remainder of Section 5, the label 'with CC' in figures means the cooperative caching mechanism is enabled. When there is a local hot object cache miss, a server node will first try to fetch a copy from a peer node's hot object cache, before resorting to the file server. 'Without CC' means that the cooperative caching mechanism is disabled, such that when there is a local hot object cache miss, the server node will contact the file server directly.

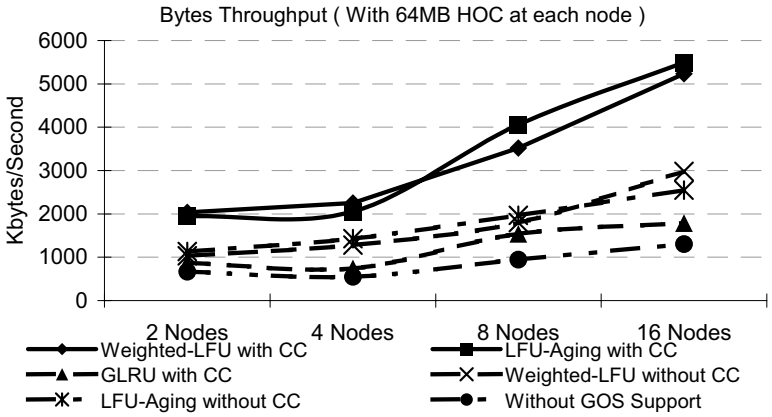


Figure 4. Effects of scaling cluster size.

Figure 4 shows the request and byte throughput of the system with cluster size ranging from two nodes to 16 nodes. At each server, 64 Mbytes 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 two-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 request throughput increases by around 2.02 times when the system expands from two to 16 nodes, while with Weight-LFU with CC, it is around 2.89 times. This is still not most satisfactory as the ideal increase should probably be close to 8 times—something for future investigation or improvement.

From the graphs, we 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 16 nodes, the increase of throughput for GLRU with CC is around 1.71 times, while it is around 2.89 times for Weighted-LFU with CC.

5.4. Effects of scaling cache size

We tested the *global cache hit rate* on a 16-node server configuration by scaling the size of the hot object cache at each node. A global cache hit means that the requested object is found either 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 ranged 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 are 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.

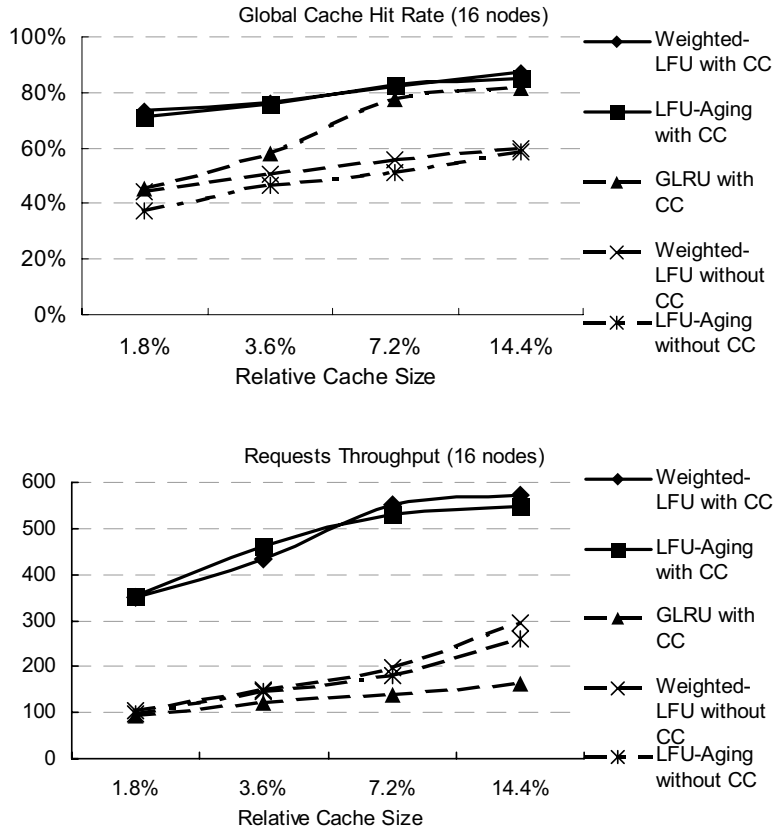


Figure 5. Effects of scaling cache size.

Figure 5 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 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 cache space provided by a cluster efficiently, and can improve 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. This 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 obvious.

5.5. Breakdown of request handle pattern

From the request throughput curves in Figure 5, 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 classified 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 GOS, and has to be fetched from the file server. This has the longest serving time.

Figure 6 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 are considered and counted as a cache hit in the GOS.

The graphs 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 amount of 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 had 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. In total, 89% of the top 15% of the most requested objects received a hit in the GOS. This figure is in line with previous research in that around 10% of the documents of a Web site accounts for around 90% of the requests that the Web site would receive [43]. Even with only an 8-Mbyte hot object cache in each node, the local cache hits can still reach around 50%, and 20% for peer node cache hits. This shows that our mechanism to approximate the global access count is effective which successfully identified the most probable popular objects.

The graphs for LFU-based algorithms with 8 Mbytes of hot object cache in each node show that the number of peer node cache objects increases from around 6.7% with four nodes to around 35.2% with 16 nodes, while the number of disk objects drops from around 50% to around 25%. This confirms the fact that the cooperative cache can substantially cut down on costly file server disk accesses, which are a common bottleneck for a Web site. The effect of cooperative caching is more pronounced when the cluster size scales up, suggesting that cooperative caching should be a feature of all large-size clusters.

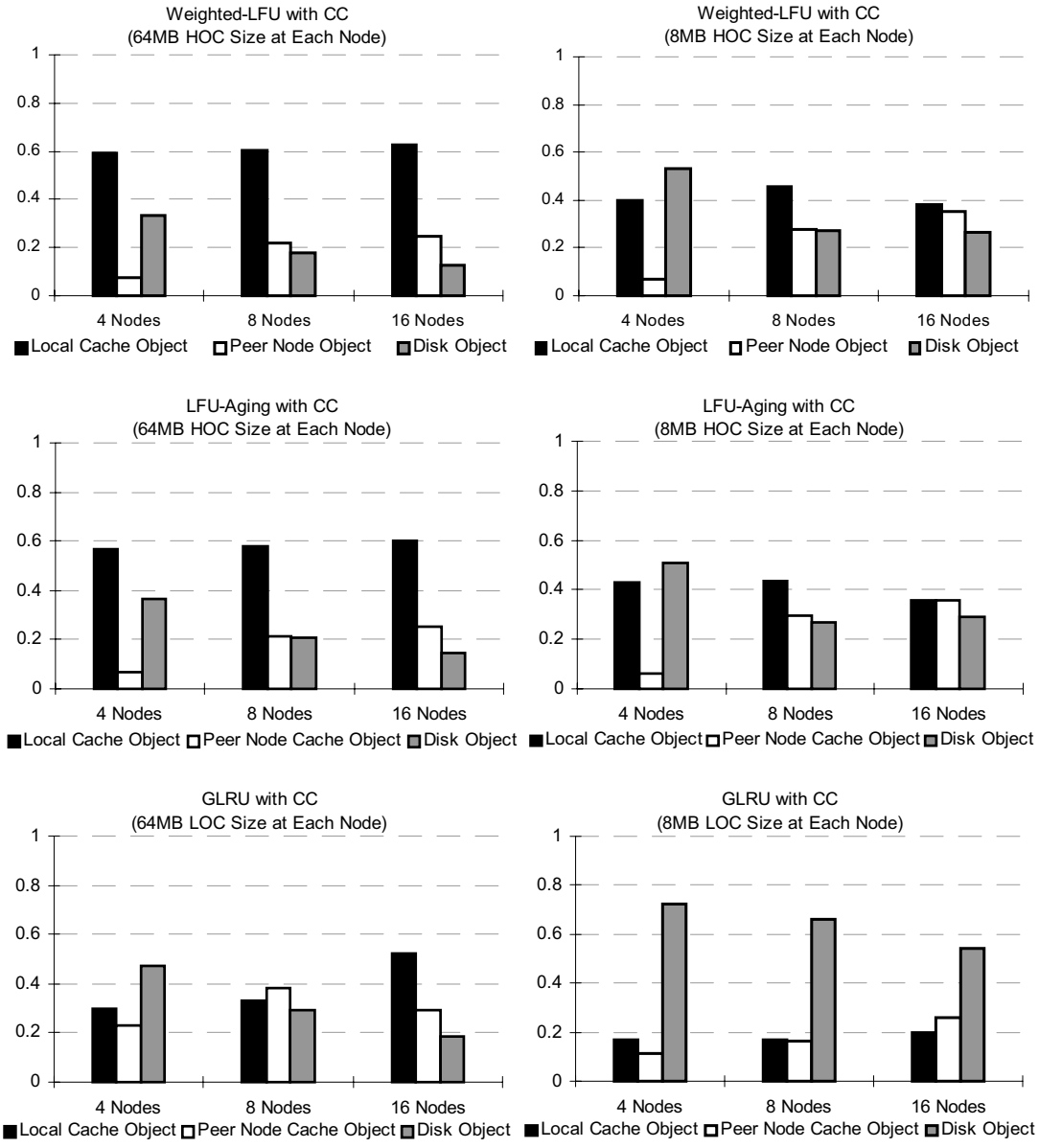


Figure 6. Breakdown of request handle pattern.

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 when using GLRU with CC. The GLRU does achieve nearly the same global cache hit rate as the LFU-based algorithms when cache space is plentiful (Figure 4), but unfortunately a large portion of those hits are non-local hits, which explains 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%.

6. RELATED WORK

The design and implementation of cluster-based Web servers is among the hot topics of Web-related research. Much of the research work, however, is concerned with request dispatching part of a Web server cluster [10,49–52]. Only a few research projects have focused on cooperative operations and the caching component. We discuss some of these directly or closely related projects below.

6.1. DC-Apache

The *Distributed Cooperative Apache* (DC-Apache) Web server [7] dynamically modifies the hyperlinks in HTML documents to redirect client requests to multiple cooperating Web servers. A Web server 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. This could potentially increase network traffic, CPU time, and memory usage (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 Web site 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.

6.2. LARD

Location Aware Request Distribution (LARD) [53] is a mechanism to distribute requests among back-end server nodes in a cluster environment. The prototype system uses a front-end server as a request dispatcher. The dispatcher maintains a location mapping of objects to back-end servers. Incoming requests will be distributed to the back-end server node according to this mapping. Restricted back-end server cooperation is allowed by using a mechanism called *back-end request forwarding*,

which is similar to our remote fetching of objects from home nodes. LARD uses a location-aware approach similar 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 front-end 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 many requests happen to target objects in that server. In addition, in order to keep the centralized LARD information updated, the back-end servers need to communication with the front-end server constantly to exchange information, which adds further overhead to the front-end server.

6.3. KAIST's work

A distributed file system with cooperative caching to support cluster-based Web servers that use the LARD request distribution policy is proposed by researchers at KAIST [52]. 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 other objects. Its weakness, however, is that, by replacing duplicated 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 non-trivial 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 somewhat 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.

6.4. WhizzBee

Whizz Technology's WhizzBee Web server [54] is a cluster-based Web server that realizes cooperative caching. One of the innovative features of WhizzBee is its *Extensible Cooperative Caching Engine* (EC-Cache Engine), which is similar to our cooperative caching mechanism.

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. If the file is not already in cache, the Web server will retrieve the file from the I/O device, and then inform CLSERV that this file is now cached.

Although WhizzBee and p-Jigsaw 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 p-Jigsaw 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 GOS 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 set aside in each node for object caching, a high cache hit rate can be achieved if the appropriate cache policies are used. Cooperative caching will further increase the global cache rate because of more efficient use of the aggregate cache space. By using global access information, local cache managers can make sufficiently 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 it is more profitable to replicate more hot objects than to pack more different objects into the GOS. These hot objects account for most of the requested objects, and their requests tend to come in bursts. These request bursts require 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 GOS. 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. We will consider more powerful mechanisms such as socket migration [55] which could be helpful in reducing the overhead in the GOS.

Currently, our prototype system is designed to handle mainly static contents. As the use of dynamic contents is increasingly common, it is imperative that the GOS be eventually extended to support dynamically generated contents. The extension requires a more dynamic and rigorous synchronization mechanism for handling cache replacement and object consistency. Part of our future work will focus on this aspect.

Finally, although our prototype system is built on top of the Jigsaw Web server, the GOS implementation can be easily ported to other Web servers. The advantage with using the Java-based Jigsaw Web server 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 many similarities with Web servers.

ACKNOWLEDGEMENTS

This research was supported by The University of Hong Kong's seed funding program for basic and applied research under account code 10203944. 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. Hu J, Schmidt D. JAWS: A framework for high-performance Web servers. http://www.cs.wustl.edu/∼jxh/research/research.html [20 December 2001].
2. Pai V, Druschel P, Zwaenepoel W. Flash: an efficient and portable Web server. *Proceedings of the 1999 USENIX Annual Technical Conference*. Monterey, CA, June 1999.
3. kHTTPd Linux HTTP accelerator. http://www.fenrus.demon.nl [20 December 2001].
4. Iyengar A *et al*. High-performance Web site design techniques. *IEEE Internet Computing* 2000; **4**(2):17–26.
5. 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.
6. 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, July 1998.
7. Li Q, Moon B. Distributed cooperative apache Web server. *Proceedings of the Tenth World Wide Web Conference*. Hong Kong, May 2001.
8. Bryhni H, Klovning E, Kure O. A comparison of load balancing techniques for scalable Web servers. *IEEE Network* 2000; **14**(4):58–64.
9. Richard B *et al*. Achieving load balance and effective caching in clustered Web servers. *The Fourth International Web Caching Workshop*. San Diego, CA, 31 March–2 April 1999.
10. Cardellini V, Colajanni M, Yu P. Dynamic load balancing on Web-server systems. *IEEE Internet Computing* 1999; **3**(3): 28–39.
11. Andresen D *et al*. SWEB: Towards a scalable world wide Web server on multicomputers. *Proceedings of International Conference of Parallel Processing Symposium (IPPS)*. Honolulu, Hawaii, April 1996.
12. Schroeder T, Steve G, Byrav R. Scalable Web server clustering technologies. *IEEE Network* 2000; **14**(3):38–45.
13. David D, Shrivastava S, Panzieri F. Constructing dependable Web services. *IEEE Internet Computing* 2000; **4**(1):25–33.
14. Schroeder T, Goddard S. The SASHA cluster-based Web server. http://www.zweknu.org/tech/src/lsmac2/paper/paper.html [20 December 2001].
15. Brewer E. Lessons from giant-scale services. *IEEE Internet Computing* 2001; **5**(4):46–55.
16. Wang J. A survey of Web caching schemes for the internet. *ACM Computer Communication Review (CCR)* 1999; **29**(5):36–46.
17. Barish G, Obraczka K. World wide Web caching: trends and techniques. *IEEE Communications Magazine* 2000; **38**(5):178–184.
18. Davison B. A Web caching primer. *IEEE Internet Computing* 2001; **5**(4):38–45.
19. Menaud J, Issarny V, Banatre M. A scalable and efficient cooperative system for Web caches. *IEEE Concurrency* 2000; **8**(3):56–62.
20. Wolman A *et al*. On the scale and performance of cooperative Web proxy caching. *ACM Operating System Review* 1999; **34**(5):16–31.
21. Korupolu M, Dahlin M. Coordinated placement and replacement for large-scale distributed caches. *Proceedings of the 1999 IEEE Workshop on Internet Applications*. Jose, CA, July 1999.
22. Sarkar P, Hartman J. Hint-based cooperative caching. *ACM Transactions on Computer Systems* 2000; **18**(4):387–419.
23. 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.
24. Dahlin M *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.
25. Tatarinov I. Performance analysis of cache policies for Web servers. *Proceedings of 9th International Conference on Computing and Information*. Manitoba, Canada 1998.
26. Jigsaw-W3C's Server. http://www.w3c.org/Jigsaw [20 December 2001].

27. Arlitt M, Williamson C. Trace-driven simulation of document caching strategies for internet Web servers. *Simulations* 1997; **68**(1):23–33.
28. Arlitt M, Friedrich R, Jin T. Performance evaluation of Web proxy cache replacement policies. *Performance Evaluation* 2000; **39**(1-4):149–164.
29. Pierre G, *et al*. Differentiated strategies for replicating Web documents. *Proceedings of 5th International Web Caching and Content Delivery Workshop*. Lisbon, Portugal, May 2000.
30. Anderson T *et al*. Serverless network file systems. *ACM Transaction on Computer Systems* 1996; **14**(1):41–79.
31. Reddy N. Effectiveness of caching policies for a Web server. *Proceedings of the 4th International Conference on High Performance Computing*. Cambridge, MA, December 1997.
32. 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, CO, May 1990.
33. Williams S *et al*. Removal policies in network caches for world wide Web documents. *Proceedings of ACM SIGCOMM '96*. Stanford University, CA, August 1996.
34. 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.
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, 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, March 2001.
38. Gadde S, Chase J, Rabinovich M. A taste of crispy squid. *Proceedings of Workshop on Internet Server Performance (WISP98)*. Madison, WI, June 1998.
39. Chankhunthod A, Danzig P, Neerdaels C. A hierarchical internet object cache. *Proceedings of the USENIX 1996 Annual Technical Conference*. San Diego, CA, January 1996.
40. Fan L *et al*. Summery cache: a scalable wide-area Web cache sharing protocol. *IEEE/ACM Transactions on Networking* 2000; **8**(3):281–293.
41. Internet Cache Protocol. http://icp.ircache.net [20 December 2001].
42. Cano J *et al*. The difference between distributed shared memory caching and proxy caching. *IEEE Concurrency* 2000; **8**(3):45–47.
43. Arlitt M, Williamson C. Internet Web servers: workload characterization and performance implications. *IEEE/ACM Transactions on Networking* 1997; **5**(5):631–645.
44. Baker S, Moon B. Distributed cooperative Web servers. *Proceedings of the Eighth World Wide Web Conference*. Toronto, Canada, May 1999.
45. HTTP-Hypertext Transfer Protocol. http://www.w3c.org/Protocols [20 December 2001].
46. Mogul J. The case for persistent-connection HTTP. *Proceedings of the ACM SIGCOMM '95 Symposium*. Cambridge, USA 21 August–1 September 1995.
47. Mosberger D, Jin T. httperf—a tool for measuring Web server performance. *Proceedings of Workshop on Internet Server Performance (WISP98)*. Madison, WI, June 1998.
48. University of California, Berkeley. CS Access Log Files. http://www.cs.berkeley.edu/logs/ [10 July 2002].
49. Schroeder T, Goddard S, Ramamurthy B. Scalable Web server clustering technologies. *IEEE Network* 2000; **14**(3):38–45.
50. 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, CO, October 1999.
51. 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.
52. Ahn W, Park S, Park D. Efficient cooperative caching for file systems in cluster-based Web servers. *Proceedings IEEE International Conference on Cluster Computing*. Chemnitz, Germany, 28 November–1 December 2000.
53. Vivek SP *et al*. Locality-aware request distribution in cluster-based network servers. *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*. San Jose, CA, October 1998.
54. Whizz Technology. http://www.whizztech.com [20 December 2001].
55. Sit Y-F, Wang C-L, Lau F. Socket cloning for cluster-based Web server (CLUSTER 2002). *IEEE Fourth International Conference on Cluster Computing*, Chicago, IL, 23–26 September 2002.

