Building a Scalable Web Server with Global Object Space Support on Heterogeneous Clusters[1]

Ge Chen Cho-Li Wang Francis C.M. Lau
Department of Computer Science and Information Systems
The University of Hong Kong
Email: {gechen, clwang, fcmlau}@csis.hku.hk

Abstract

Clustering provides a viable approach to building a scalable Web server system. Many existing cluster-based Web servers, however, do not fully utilize the underlying features of the cluster environment, and most parallel web servers are designed for homogeneous clusters. In this paper, we present a pure-Java-implemented parallel Web server that can run on heterogeneous clusters. The core of the proposed system is an application-level "global object space", which is an integration of the available physical memory of the cluster nodes for storing frequently requested objects. The global object space provides a unified view of cluster-wide memory resources, and allows transparent accesses to cached objects. Using a technique known as cooperative caching, a requested Web object can be fetched from a node's local memory cache or a peer node's memory cache to avoid hot spots and excessive disk operations. A preliminary prototype system has been implemented by modifying the W3C's Jigsaw Web server. We obtained good speedups in the benchmark tests, indicating that clustering with cooperative caching can greatly improve the performance of a Web server system.

1. Introduction

The Internet's ever-increasing user popularity and rapid developments in broadband networking are demanding that Web sites be able to handle large amount of requests. This has created an urgent need for a more powerful Web server architecture.

A cluster-based Web server system consists of multiple Web servers connected together by a high-speed LAN. The Web servers work cooperatively to handle the Web requests. This multi-server architecture is one of the promising solutions to meeting the heavy demands on Web services. Many research and industry projects have been conducted on the design of cluster-based Web servers [2,4,5,11], which aimed mainly at issues related to load balancing [7,8,9], scalability [2,18], and high availability [15,25].

Web caching has been recognized as one of the effective ways to accelerate the speed of Web access. Various Web caching techniques developed for different levels in the Web hierarchy have been discussed [1,2,3,12,15,19,20]. For example, in Web proxy server design, popular objects are cached close to the clients to alleviate server bottlenecks and reduce the network traffic over the Internet, thereby minimizing user access latencies [16].

Cooperative caching is an enhancement of traditional caching mechanisms, which can reduce access latency and the workload of the main server by fetching objects from the server's peer nodes [10]. It has been widely discussed in the context of cooperative Web proxy caching systems [12,18] and distributed file systems [1,17,22].

The design goal of cooperative Web proxy systems essentially is to reduce the communication cost for propagating directory updates and the space overhead for directories, and to improve the cache hit rate using various cache replacement algorithms. Network latency, network reliability, and cache routing are the main areas for consideration in caching policy design, as cooperative Web proxy is usually employed in a wide area network. Unfortunately, there is always a lack of global location information of cached objects because each proxy server can afford only to maintain a directory of objects cached in its vicinity. In addition, it is impractical to adopt more complicated or aggressive caching algorithms due to long network latencies in wide area networks.

Cooperative caching has also been used in the design of distributed file systems [10,17,22]. The basic idea is to use the client's memory as a global file cache. By using cooperative caching, the requested file content can be forwarded from a peer node instead of the master server [22]. Caching is achieved on a per-block basis (e.g., NFS and Berkeley's xFS) rather than the whole file as one

[1] This research was supported by the Hong Kong RGC Grant HKU 7032/98E and HKU CRGC Grant 10203009.

object. The cache consistency protocol is usually sophisticated as multiple clients can update any variable stored in a data block. As the file access modes (e.g., readable, writable, or executable) are known in advance, consistency protocols can be designed accordingly. Because of the diverse implementations of file systems, however, none of the well-known distributed file systems is implemented in a heterogeneous environment.

Studies show that Web systems differ from distributed file systems in their access pattern, their scale, and that there is a single point of updates for Web objects [19]. In [1], a cluster-based file system with cooperative caching was designed for supporting a parallel Web server system. However, the file system there lacks the knowledge about the Web accesses, such as the access frequency and the relationship between the accessed Web pages. Therefore, it is nearly impossible to apply more efficient caching replacement algorithms. Moreover, the file system competes with the Web server on the use of the memory, which could result in poor memory utilization.

Our server-side caching solution is developed based on the *global object space* (GOS) concept. The global object space is built on the physical memory of all the cluster nodes and is used for the containment of all cached Web objects in order to alleviate excessive file accesses from the disks. All Web objects stored in the global object space are visible and accessible by all Web server nodes. In each node of the cluster, a Web server is responsible for managing its local object caches as well as cooperating with the rest of the collection of Web servers to create a unified and consistent view of a globally shared cache space. GOS implements a cooperative object caching mechanism. A requested Web object can be obtained from a server's local cache, a peer node's cache, the server's local file system, or other server node's file system based on various load balancing policies. The location where an object resides is transparent to the client request.

We use the concept of "hot objects" in the global object space to refer to those objects that are frequently requested. Hot objects can be replicated and stored at multiple server nodes in order to improve hit rates, and to avoid excessive remote fetching of hot objects from other nodes.

A prototype system has been implemented by modifying the W3C's Jigsaw Web server [24], a Web server written in Java. The application-level object space together with the pure-Java implementation makes it possible for the Web server system to operate in a heterogeneous cluster with nodes running different OSes on different hardware architectures.

The rest of the paper is organized as follows. Section 2 gives an overview of the Web server's system architecture. In Section 3, we discuss the basic concept and implementation details of the global object space. Section 4 presents some benchmarking results of the prototype system. Some related works are discussed in Section 5. We summarize our experiences and propose some future research items in Section 6.

2. System Overview

Figure 1 shows the overview of our Web server architecture. Each server node sets aside and maintains a special memory segment, called *single node memory space* (SNMS), for caching Web objects. An SNMS serves as a *hot object cache (HOC)*. All the SNMSs are combined into a globally shared object cache, called the *global object space (GOS)*.

To the clients, the Web server system appears to be a single system, with a single URL and IP address. A dispatcher node at the entrance of the system is employed to direct a client request to a selected server node for service. The decision of which node will serve an incoming request can be based on a simple round-robin scheme, or the workload situation across all server nodes.

Each node operates two daemons, the *global object space service daemon* (GOSD) and the *request handle daemon* (RHD). The GOSD is responsible for managing the node's HOC, as well as cooperating with all the other nodes to provide a global lookup mechanism for locating and accessing cached Web objects. The RHD listens on the TCP port that is used for communicating with the requesting client. It submits the parsed and analyzed requests to the GOSD which handles object requests from the local RHD as well as those from peer GOSDs.

Each cached object has a single *home node*. The home node is the node in which the original persistent copy of the cached object is located. Since a popular object that has become "hot" can appear in multiple SNMEs, 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. The system keeps track of the number of times the object is accessed over a certain period. This is recorded in a counter 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.

The workflow of the proposed Web server system is described as follows. Every time a new object request arrives, the GOSD will try to find the requested object in its local object cache. If a cached copy is found, the object's local counter is incremented, and the cached copy is sent to the client. If this very first search fails, the GOSD will send an object request message to the object's home node. The home node will return the request with a cached copy from its local object cache, if any, or reply to the requesting server with a disk copy if a copy is not found in the cache. The home node could request a cached

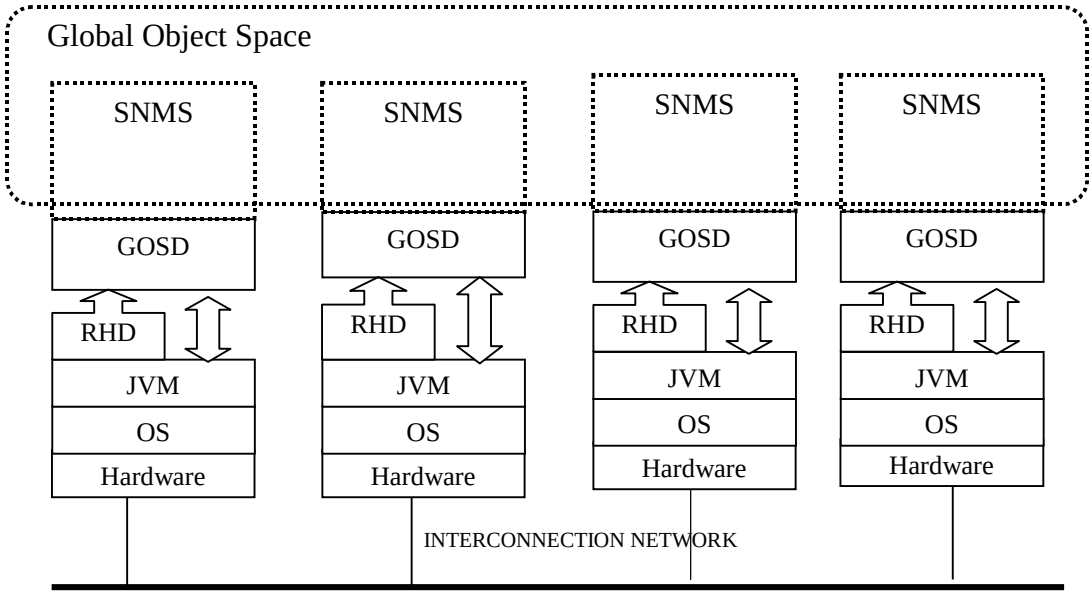

Figure 1. System architecture (RHD: Request Handle Daemon, GOSD: Global Object Space Service Daemon, SNMS: Single Node Memory Space)

copy, if one exists, from another node. This feature is switched off in the current implementation because we would like to concentrate on the basic features and their performance in this paper.

To build and maintain the global object space, two tables are defined and managed by each GOSD: (1) *local object table* (LOT), and (2) *global object table* (GOT). The GOT contains system-wide information for all objects of which the node in question is their home node; this information includes the server locations where copies of the object are placed and the system-wide access count (global access count). The LOT contains basic access records of objects stored in the node's HOC, including the objects' cache addresses, object sizes, local access counts, approximated global access counts, and the objects' home node locations. The approximated global access count stored in the LOT for an object is mainly used for cache replacement decisions. The computation of the local access count and the global access count will be explained in Section 3.1.

A preliminary prototype system has been implemented by modifying the W3C's Java Web Server, Jigsaw [24]. Since Jigsaw is written in Java, and runs on top of the Java Virtual Machine, our system can support Web services on a heterogeneous cluster. The JVM hides all the hardware and OS differences.

3. Object Caching in Global Object Space

3.1. Hot Object Caching

Much research effort has been directed to the workload characterization and the performance implication of Web servers [3,6,21]. One of the major results is the notion of *concentration* [3], which is that documents on a Web server are not equally accessed. Some of them are extremely "hot" and popular, and are accessed frequently at short intervals by many clients from many sites. Other documents are accessed rarely, if at all. Analyses show that around 10% of the distinct documents are responsible for 80-95% of all requests received by the server [3]. This result applies to both requests and byte throughput.

Based on the above research conclusion, we adopted the *hot object* concept in designing caching policies for the global object space. Hot objects, by our definition, are those popular Web objects that receive multiple requests from different clients within a short interval or that are accessed frequently. They have a higher priority to be cached in the global object space. In addition, because the set of hot objects only accounts for a small portion of the whole objects in a Web site, we allow hot objects to be cached at more than one node. Thus, bursty hot object requests could be handled simultaneously by different server nodes which reply to the requests directly from their caches for speeding up the processing time and sharing the workload.

To determine the "hotness" of an object, the home node is responsible for keeping track of the object's global access count. When an object is cached in or swapped out of a server's HOC, this change will be immediately reflected in its local access count. Periodically, all nodes send the local access counts of the cached objects to their home nodes to update the global access counts stored in the home nodes' GOT and LOT. The local access counts in the nodes holding the cached copies are then reset to zero. In the meantime, the updated

global access count is sent in a reply to update those servers' approximated global access count stored in their LOT. Thus, we ensure that all server nodes keep the most up-to-date global access count in their LOT, in order to make a more accurate cache replacement decision. Because we only periodically update the global access count in the LOT and the GOT, this value could be slightly different from the actual global access count. However, this approach causes less overhead and is practical in a cluster environment since to maintain a correct global access count on every server is costly.

As the hot object cache provides only limited cache space, an LFU-Aging cache replacement algorithm is adopted at each node. When there is no free cache space available in a server node's HOC, a cached object with the least approximated global access count will be replaced. At the end of every pre-specified time interval, the approximated global access count is divided by two to simulate the aging effect.

3.2. Scalable Global Object Table Design

When the system scales up, a full mapping of object IDs to resource locations will make the mapping table extremely large and will consume a lot of memory space. 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 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 whole request handling time in a single 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 global object table, we propose a partitioning mechanism that optimizes the use of the available memory space. This design originates from the observation that in a Web server, the file organization of a Web site usually follows the tree structure. In a cluster-based Web server system, a centralized NFS server, for example, may create access bottlenecks. Thus, it is wise to partition the whole file hierarchy into a few sub-trees and distribute them among the cluster nodes for balancing the load caused by the file accesses. This is sometimes inevitable for a large Web site where the whole document set cannot fit on a single disk.

Based on our partitioning mechanism, it is not necessary that every object has an entry in the table. In our system, a hash table is used as the global object space table. 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 the objects under the directory will share a same entry key. For example, in a tree structured Web site, if the HTML files under directory /root/dir1/dir12/ are stored in the same node, all the files under this directory will share the same entry in the GOS table. There's only one key, "/root/dir1/dir12/", to all the files under dir12. This will greatly reduce the mapping table size if the website is well organized.

3.3. Load Balancing for Persistent Connections

Support of persistent HTTP connections [23] is usually available in modern Web server. The use of persistent HTTP connections can greatly improve performance, specially when multiple HTTP requests from different clients are routed to a single proxy and then dispatched as a whole to the Web server. Our Web server system supports persistent HTTP connections with the help of the global object space. To avoid the hot-spot problem, where all the HTTP requests of a session jam at a single Web server, a redirection function is implemented on top of the global object space.

During normal execution of our Web server system, after a client makes a request to some server node and the objects are not found in the local cache, the server would keep the persistent HTTP connection. In the meantime, it collects all the requested objects from other nodes' object caches through the GOS. As such, the server in question can easily become a hot spot since it has to handle all the HTTP requests and object caching.

Since our system runs at the Java application level, low-level approaches such as *packet rewriting* are not suitable for our system. Our system uses a load-weighted HTTP redirection approach to distribute the incoming HTTP requests embedded in a single connection among the server nodes. When a persistent HTTP connection is requested, the server will first parse the request and start fetching the requested object and deliver the object to the client. When the server node finds out that it is overloaded, it will stop the service and find out the home node of the requested object. Then it will send an HTTP redirection reply to the client browser to redirect the client to the home node of the requested object to continue the service. To achieve the best performance, all URLs are represented by relative path format in our Web server. Thus, the redirected client can continue to send subsequent requests to the new server node.

4. Performance Results

A preliminary prototype system has been implemented by modifying the W3C's Jigsaw server, version 2.0.5 [24]. The global object space layer is added to Jigsaw to provide cluster-wide cached object sharing.

4.1. Experimental Setup

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

All the 16 clients run a Web server benchmark program, which is a modified version of httperf [13]. Httperf performs stress test on the designated Web server based on a collected Web server log. The main characteristics of the data set and the log file are summarized in Table 1 and Table 2. All files are evenly partitioned into disjoint document sets, and stored in each server node's local disk.

Table 1. 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 2. Summery of access log characteristics

Number of requests	~640,000
Data transferred	~ 35 Gbytes
Distinct files requested	52,347

Httperf supports customized workload generation based on a workload file. We modified the collected access log file to make it work for httperf. Requests are generated by httperf according to this modified workload file.

Similar tests were also carried out on a 4-node heterogeneous cluster composed of a Celeron PC running Microsoft Windows 2000 Professional, an SMP PC with two Pentium Pro CPUs running RedHat Linux 6.1, and two PCs each with a Pentium II CPU running RedHat Linux 6.2. Our Web server runs well on this heterogeneous cluster. As the performance behavior is similar to the 4-node homogeneous cluster, we focus on the performance of the homogeneous cluster in the following.

4.2. Effects of Scaling the Cluster Size

Figure 2 and Figure 3 show the requests and bytes throughput obtained for the 2-node, 4-node, 8-node, and 16-node configuration respectively with an object cache size of 56M bytes enabled at each server node.

The curves show the impact of the hot object caching on the overall system performance. With the hot object cache enabled, the performance increases almost linearly as we increase the number of server nodes since we can cache most of the frequently requested objects in the global object space. With the hot object caching support, the speedup is 4.77 when the system scales from two nodes to sixteen nodes, while that for the case without hot object caching support is only 2.02.

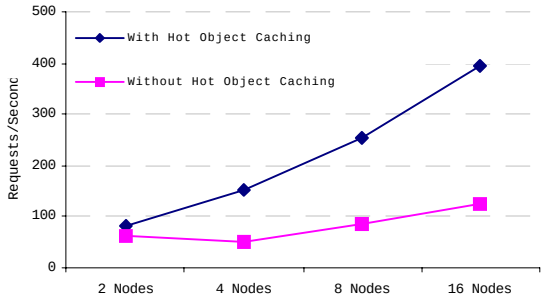

Figure 2. Comparison of requests service rate

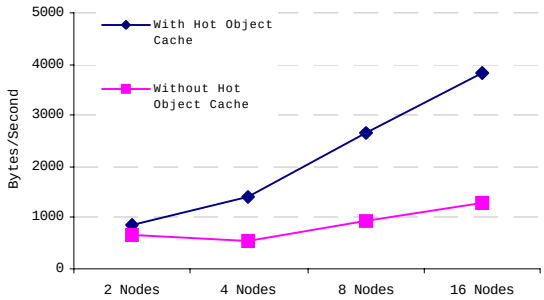

Figure 3. Comparison of bytes service rate

4.3. Effects of Scaling the Cache Size

Figure 4 shows the global cache hit rate with different hot object cache sizes allocated on each node. Figure 5 and Figure 6 show the requests and bytes throughput with different cache sizes. We tested the cache hit rate and the overall system performance with different hot object cache sizes from 7M bytes to 56M bytes per node.

Given an object data set that will be stored in a cluster Web server system, we are interested to know what is the appropriate hot object cache size that should be used for each node in order to achieve a high cache hit rate. The global cache hit rate increases from around 45% to 90% when the cache size scales from two nodes with a 7M bytes cache on each node to sixteen nodes with a 56M bytes cache on each node. The largest total cache size in the tests is the 16-node configuration with 56M bytes on each node. That is, the total cache size is 896M bytes. It is about 13% of the data set size, but the cache hit rate reaches around 90%. This observation confirms earlier research results about the nature of the access patterns for

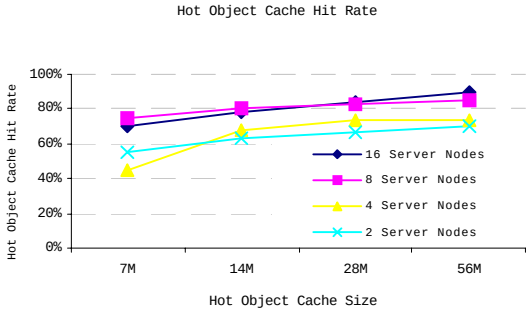

Figure 4. Global cache hit rate with different cache size

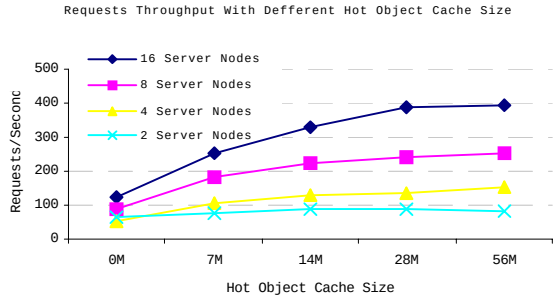

Figure 5. Requests throughput with different cache size

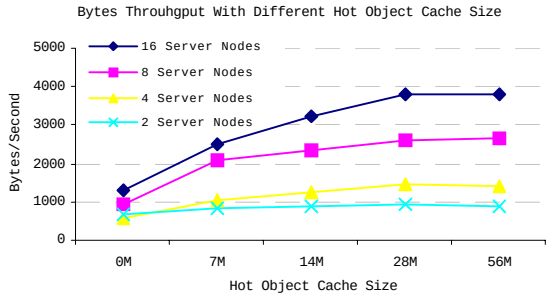

Figure 6. Bytes throughput with different cache size

hot objects [3] as stated in Section 3.1. With a relatively small amount of memory in each node used for caching hot objects, we are able to obtain a high cache hit rate which increases the whole system's performance considerably.

4.4. Performance Analysis

Based on our caching mechanism, each client request may result in one of the three types of memory or disk objects:

(1) *Local Cache Object*: The client is replied with a copy of the requested object from a server's local object cache. This type of object access has the shortest access latency.

(2) *Home Node Cache Object*: The server that receives the request does not have the object in its cache. The server fetches a copy of the requested object from the local object cache of the object's home node and then replies to the client. Additional network delay is caused as compared with (1). This also consumes the cluster network bandwidth.

(3) *Home Node Disk Object*: The document is not in the global object cache at all. The server that receives the request fetches a copy of the requested object from the object's home node, and then replies to the client. This type of operation involves extra network delay and disk access at the home node. Thus, it requires the longest time to serve.

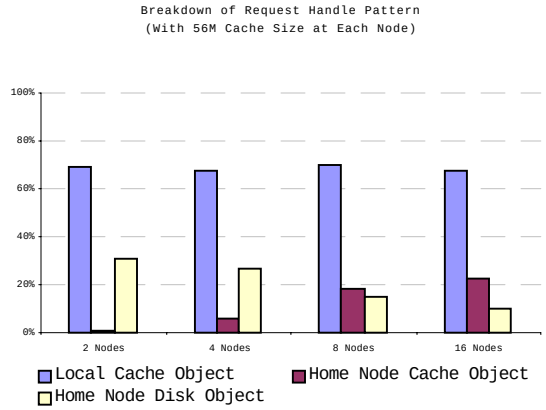

Figure 7. Analyses of request handle pattern (with 56M cache at each node)

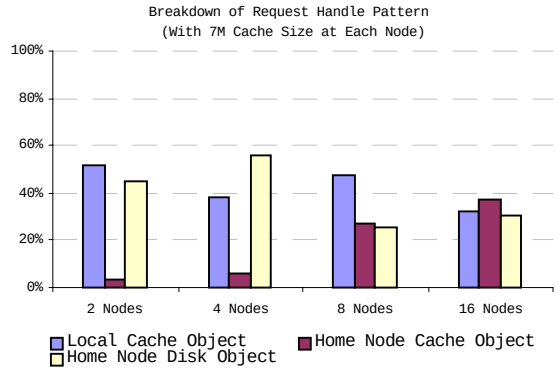

Figure 8. Analyses of request handle pattern (with 7M cache at each node)

Figure 7 shows the percentage of client requests that results in each type of the above objects with respect to different cluster server sizes, where each node has a 56M bytes cache. Figure 8 shows the case with a 7M bytes cache at each node. In the figure, a local cache object and a home node cache object both contribute a cache hit when calculating the global hit rate.

In Figure 7, the portion of home node disk objects decreases as the number of server nodes increases. This is because the size of the global object cache increases

linearly with the number of nodes. With more nodes, the larger aggregated cache size can cache more objects.

Figure 7 also indicates a high hit rate for the *local object cache*. The local object cache hit rate can reach as high as 70%. This proves that our approximated global LFU algorithm is very effective. It simulates well the global LFU algorithm so that the local object cache will cache the "hottest" objects in the system. Therefore, most of the incoming requests will be served with objects fetched from the local object cache. This reduces the high overhead of fetching an object from a remote cache or disk, and can greatly improve the throughput and performance.

For a 16-node case, we found that with only 1/8 of the cache size (7 MB vs. 56 MB), the disk access percentage only increases from around 10% to around 30%. As the number of cluster nodes increases, the home node cache objects account for a larger portion of the total global cache hit rate. This has shown the advantage of adopting the cooperative caching technique in our system for improving the performance by reducing the disk operations. Nevertheless, a more aggressive cooperative cache mechanism can be employed whereby the requested object can be fetched from any peer node holding a cached copy rather than only from the home node. This will alleviate the possible overload of the home node when many of the objects in the home node become hot and a great number of requests asking for an object copy come from the peer nodes at the same time. This will alleviate the possible overload of the home node when many of the objects in the home become hot and a great number of requests asking for an object copy would come from the peer nodes. This mechanism is under development and will be an integral part of the next version.

5. Related Works

Distributed Cooperative Web Servers (DCWS) is a cooperative Web server system developed by the Department of Computer Science, University of Arizona [6]. The DCWS project tries to explore application-level techniques for distributing Web contents. The approach is to dynamically manipulate the hyperlinks stored within the Web documents. All documents originally reside on a home server. It dynamically migrates documents to cooperative Web servers which are dedicated server nodes for sharing the load of the home server. By modifying the hyperlinks in the document, the Web server will distribute workload to cooperative nodes to achieve dynamic load balancing.

Although it realizes application-level document distribution, it could suffer from excessive overheads from modifying the hyperlinks in the documents, because parsing of an HTML document is rather time consuming. The other problem is that if any of the cooperative nodes receives a request for a document it does not have, it will fetch it from the home node where the document originally locates. The document will be delivered (actually migrated) to the requesting node by the home node, and all the documents containing hyperlinks referring to this document have to be updated. This is a nontrivial task in terms of network traffic and the use of system resources such as CPU and memory, since it needs re-parsing and modifying all the documents in the system. The overhead is obviously enormous when the document set size is large. Moreover, DCWS does not deal well with hot objects, which are called *hot spots* in DCWS, because it only maintains one cache copy of a document; this could result in poor scalability when the hot spot problem occurs, according to the test conducted by the authors.

A prototype system of a cluster-based Web server developed by Department of Computer Science, Rice University uses an extended Location Aware Request Distribution (LARD) policy to distribute requests among the backend server nodes in a cluster environment [4]. The prototype system uses a front-end server as request dispatcher. An initial incoming request from a new client will be dispatched to one of the backend server nodes according to the LARD policy, and requests that follow will be handled by the assigned backend node, to implement persistent connection in HTTP/1.1. Non-local documents are fetched by using the TCP connection handoff protocol between the concerned backend nodes, which actually forwards the request to the home node of the requested non-local document.

It uses a centralized LARD information control. The potential problem is that the front-end will easily become a bottleneck when the system scales up. In addition, because all the requests for the same document will be dispatched by the front-end server and will ultimately be passed to a single node, with enough hot objects, that node will become a bottleneck too.

In [1], a cluster-based file system with cooperative caching was designed for supporting a parallel Web server system. The authors proposed an improved cooperative cache system based on some hint-based file system cooperative cache [17] for their cluster-based Web server's file system. Positive results are obtained from the simulation experiments indicating good load balancing and reduced disk access rates. This approach allows Web servers to transparently access the Web objects without worrying the underlying caching activities. However, the cost is a passive Web object caching policy since the file system is lack of the knowledge about the Web access patterns. More aggressive caching approaches, such as hot object caching, are generally impossible and the cache replacement algorithm is applied with restricted information. Moreover, the file system competes with the Web server on the memory resources, which could result in poor memory utilization.

6. 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 throughput and service availability. The hot object cache is very helpful in increasing system performance and scalability. With relatively small amount of memory dedicated for object content caching, we are able to achieve a very high hit rate. The approximated global LFU cache replacement algorithm works well for our goal of keeping the hottest objects in the relatively limited cache space. This will lead to improved server performance for most of the requests served by the Web server. By using cooperative caching among the cluster nodes, we can further improve the cache performance in terms of global cache hit rate. The fetching of hot objects from peer nodes' caches can also reduce the expensive disk I/O operations.

Our future work includes a study of the impact of hot object caching on the overall system performance for larger cluster sizes. More advanced replacement policies on hot object caching will also be studied, as well as more aggressive cooperative caching mechanisms that may achieve more efficient use of cluster-wide resources, such as memory space and network bandwidth.

Although our prototype system does not implement dynamic content caching, it is possible to provide the extension by implementing some efficient object consistency protocol within the global object space. Our cooperative caching solutions can be implemented on other non-Java Web server systems to improve their performance.

References:

[1] Woo Hyun Ahn, Sang Ho Park, and Daeuyeon Park, "Efficient Cooperative Caching for File Systems in Cluster-Based Web Servers," *Proc. of IEEE International Conference on Cluster Computing*, 2000.

[2] D. Andresen, Tao Yang, V. Holmedahl, O. H. Ibarra, "SWEB: Towards a Scalable World Wide Web Server on Multicomputers". *International Conference of Parallel Processing Symposium (IPPS)*, 1996, pp. 850 –856

[3] Martin F. Arlitt, and Carey L. Williamson, "Internet Web Servers: Workload Characterization and Performance Implications," *IEEE/ACM Transactions on Networking*, Vol. 5, No.5, October 1997, pp.631-645.

[4] Mohit Aron, Peter Druschel, and Willy Zwaenepoel, "Efficient Support for P-HTTP in Cluster-Based Web Servers," *Proc. of the 1999 Annual Usenix Technical Conference, Monterey*, CA, June 1999.

[5] Scott M. Baker, and Bongki Moon, "Scalable Web Server Design for Distributed Data Management,". *IEEE, 1999. Proc. of the 15th International Conference on Data Engineering, 1999 , Page(s): 96*

[6] Scott M. Baker, and Bongki Moon, "Distributed Cooperative Web Servers," *Proc. of WWW8 Conference*, http://www8.org/w8-apers/2abserver/distributed/distributed.html

[7] H. Bryhni, E. Klovning, O. Kure, "A Comparison of Load Balancing Techniques for Scalable Web Servers", *IEEE Network*, Volume: 14 Issue: 4, July-Aug. 2000, pp.58-64.

[8] Richard B. Bung, Derek L. Eager, Gregory M. Oster, Carey L. Williamson. "Achieving Load Balance and Effective Caching in Clustered Web Servers". *The 4th International Web Caching Workshop, 1999.*

[9] Valeria Cardellini, Michele Colajanni, and Philip S. Yu, "Dynamic Load Balancing on Web-Server Systems," *IEEE Internet Computing*, May-June 1999, pp. 28-39.

[10] M.D. Dahlin, R. Y. Wang, T.E. Anderson, and D.A. Pattern, "Cooperative Caching: Using Remote Client Memory to Improve File System Performance," *Proc. Of the 1st Symposium on Operating Systems Design and Implementation*, 1994, pp. 267-280.

[11] Arun Ivengar, et al., "High-Performance Web Site Design Techniques," *IEEE Internet Computing*, March-April 2000, pp.17-26.

[12] Jean-Marc Menaud, Valerie Issarny, and Micheal Banatre, "A scalable and Efficient Cooperative System for Web Caches," *IEEE Concurrency*, July-September 2000, pp.56-62.

[13] David Mosberger, Tai Jin, "httperf – A Tool for Measuring Web Server Performance", Proc. of 1998 Workshop on Internet Server Performance, Madison, Wisconsin, June 23, 1998.

[14] B. Narendran, S. Rangarajan, S. Yajnik, "Data Distribution Algorithms for Load Balanced Fault-Tolerant Web Access". *The Sixteenth International Symposium on Reliable Distributed Systems*, 1997, pp. 97 –106.

[15] Guillaume Pierre, et al. "Differentiated Strategies for Replicating Web Documents," *Proc. of The 5th International Web Caching and Content Delivery Workshop*, Lisbon, Portugal, 22-24 May, 2000.

[16] Michael Rabinovich, Jeff Chase, and Syam GaddeNot, ''All Hits Are Created Equal: Cooperative Proxy Caching Over a Wide-Area Network,'' *The Third International Web Caching Workshop*, 1998.

[17] Prasenjit Sarkar, John H. Hartman, "Hint-Based Cooperative Caching," *ACM Transactions on Computer Systems*, Vol. 18, No. 4, November 2000, pp.387-419.

[18] Trevor Schroeder, Steve Goddard, and Byrav Ramamurthy, "Scalable Web Server Clusering Technologies," *IEEE Network*, May/Jun 2000, pp.38-45.

[19] Jia Wang, "A Survey of Web Caching Schemes for the Internet," *ACM Computer Communication Review (CCR)*, Vol. 29, No. 5, October 1999.

[20] Duane Wessels et al. Squid Internet Object Cache, http://squid.nlanr.net/

[21] Alec Wolman, et al., "On the scale and performance of cooperative Web proxy caching," *Proc. of 17th ACM Symposium on Operating Systems Priciples*, Kiawah Island Resort, SC, USA, December, 1999, pp.16-31.

[22] Berkeley's Serverless File System xFS: http://now.cs.berkeley.edu/Xfs/xfs.html

[23] Hypertext Transfer Protocol – HTTP/1.1, URL: http://www.w3c.org/Protocols/

[24] Jigsaw Overview, http://www.w3c.org/Jigsaw

[25] SASHA http://www.zweknu.org/tech/src/lsmac2/paper/paper.html