The Journal of Supercomputing, 35, 301–320, 2006

Tulip: A New Hash Based Cooperative Web Caching Architecture

ZHIYONG XU zxu@mcs.suffolk.edu
CS Department, Suffolk University

LAXMI BHUYAN bhuyan@cs.ucr.edu
CSE Department, UC, Riverside

YIMING HU yhu@ececs.uc.edu
ECECS Department, University of Cincinnati

Abstract. With the exponential growth of WWW traffic, web proxy caching becomes a critical technique for Internet web services. Well-organized proxy caching systems with multiple servers can greatly reduce the user perceived latency and decrease the network bandwidth consumption. Thus, many research papers focused on improving web caching performance with the efficient coordination algorithms among multiple servers. Hash based algorithm is the most widely used server coordination mechanism, however, there's still a lot of technical issues need to be addressed. In this paper, we propose a new hash based web caching architecture, Tulip. Tulip aggregates web objects that are likely to be accessed together into object clusters and uses object clusters as the primary access units. Tulip extends the locality-based algorithm in UCFS to hash based web proxy systems and proposes a simple algorithm to reduce the data grouping overhead. It takes into consideration the access speed dispatch between memory and disk and replaces expensive small disk I/O with less large ones. In case a client request cannot be fulfilled by the server in the memory, the system fetches the whole cluster which contains the required object into memory, the future requests for other objects in the same cluster can be satisfied directly from memory and slow disk I/Os are avoided. It also introduces a simple and efficient data dupllication algorithm, few maintenance work need to be done in case of server join/leave or server failure. Along with the local caching strategy, Tulip achieves better fault tolerance and load balance capability with the minimal cost. Our simulation results show Tulip has better performance than previous approaches.

1. Introduction

Internet experienced an exponential growth in the past decade. Web proxy caching (hereafter referred as web caching) is an important technique to deal with the fast increasing Internet traffic. Web caching systems usually locate between the end users and the original web servers. Proxy servers can quickly satisfy the local client requests with cached web objects and avoid expensive accesses to the remote original servers. It is an effective mechanism to achieve server load balance, reduce network traffic and decrease the client perceived latencies.

Since web caching techniques are widely used in organizations (universities, government agencies, corporations and ISPs), the number of proxy servers deployed on the Internet increases dramatically in recent years. However, the enormous web documents

make it difficult to achieve good performance, especially with a single proxy server configuration. A single proxy server can easily become a bottleneck or cause a single point of failure problem. As the number of clients increases, the workload on the proxy server rises and the server performance degrades. Many research papers have suggested the coordination among multiple servers to improve the web caching performance [8, 13, 15, 19, 20, 23, 24].

Existing cooperative web caching systems can be categorized into two types. The first approach is hierarchical architecture, it was first introduced in Harvest Cache [4]. In Harvest Cache, a series of proxy servers create a hierarchical structure arranged in a tree-like structure. A child server can query its parent server or other child servers in the same layer for a certain web object. Internet Caching Protocol (ICP) [21] is used as the basic mechanism for intercache communication. However, this approach is not scalable, it has a serious problem: in case the server cannot find the requested object in its cache, it does not know which server in the same layer might have the data, thus it must broadcast the client request to all the other servers in the same layer which generates a lot of query traffic.

The second approach is hash based web caching system [16]. It uses hash based allocation algorithm for the content dissemination. Each cached web object has exactly one copy in a pre-defined location within a proxy array (a group of proxy servers). Cache Array Routing Protocol (CARP) [5] is used as the basic protocol. In this approach, no search algorithm is needed since the location of all web objects are fixed, thus it is an ideal approach to resolve the previous problem in hierarchical architecture. Though this approach comes with other issues, we will give the details in the following sections.

In this paper, we propose a new solution Tulip, to address the existing problems in current hash based web caching systems. Tulip groups topologically close proxy cache servers into cache arrays as other hash based approaches. However, Tulip has several differences. It groups the web objects that are likely to be accessed together into object clusters and use object clusters as the primary transfer units between the memory and the disk. In case a client requests one object, the server fetches the whole cluster from the disk into memory, only one large disk I/O is required. Further requests for other objects in the same cluster can be satisfied from the memory directly, several expensive disk I/Os can be avoided. We also introduce an efficient data duplication mechanism to deal with server join/leave problem and achieve load balance. In case a server join or server leave/failure occurs, Tulip system can remain stable with the minimal overhead introduced. Tulip aims to be robust, efficient and fault tolerant.

The rest of the paper is organized as follows. We discuss the issues in today's web caching systems and the motivations of Tulip in Section 2. In Section 3, we describe the Tulip system design in details. We evaluate Tulip system performance in Section 4. Section 5 discusses the related works and finally, we draw the conclusion and give the future work in Section 6.

2. Motivations

The goal of Tulip system is to design a efficient hash based web caching system with minimal disk access overhead, optimal server load balance, robust fault tolerance and

efficient server join/leave operations. In this section, we analyze the problems in current hash based web caching systems and the motivations for our project.

The most popular technique for proxy cache cooperation is creating a cache hierarchy. However, it is not scalable and has several drawbacks [17]. First, every hierarchy layer introduces additional delay; second, a lot of redundant document copies are stored at every hierarchy level, and third, higher level caches tend to become bottlenecks. The fourth and the most serious issue is the high cost of query message broadcasting in case of the proxy cache server cannot satisfy the local client request.

For hash based web caching systems, a hash function is used to create a fixed mapping relation between an object and the proxy server which is used to store this object, the hash value is also used as the key for the searching procedure. However, server join/leave operations are very expensive, nearly all the cached web objects must be moved from one server to another.

Another issue in hash based caching system is the "hot spot" or "single point of failure" problem. Since each object only has one copy within a cache array, the requests for a certain object can only be satisfied by one specific proxy server and that server might become overloaded in case a large volume of requests are coming simultaneously. Also, in case of a server failure, all the requests for the web objects previously cached on that server will fail. Thus, it will result in a great degradation in web caching performance.

The above problems are under heavy attacks in recent years, different solutions are proposed to solve or relieve these problems. Tulip tries to address these two problems with low cost mechanisms. Using a simple and efficient data duplication mechanism, Tulip can easily deal with them without introducing too much extra cost.

Tulip also addresses other issues which are not fully considered in the previous web caching system researches. First is the spatial localities among different web objects. We define web access spatial locality as following: If object A is requested by a client N, in most cases, object B will also be requested by N in the following requests, then we define the spatial locality exists between object A and B. In Tulip, we exploit spatial locality by grouping the web objects which are likely to be accessed together into object clusters.

The second issue is the access speed dispatch between the memory and the disk. Generally, disk accesses are hundreds of times slower than memory accesses. The typical disk access time is around 5 ms while memory access time is only 50 ns. Studies have shown that 75% or more web documents in web proxy traffic are less than 8 KB [14], and the conventional file systems cannot handle small files efficiently. The frequent accesses to the files on the disk introduce expensive disk seeking and rotational latencies. However, the current web caching coordination solutions do not address this problem very well. We relieve this problem by replace many small disk I/Os into big ones to reduce the disk access overhead.

3. Tulip system design

Tulip is a hash based cooperative web caching system. As other hash based systems, in Tulip, several proxy servers are grouped together in a cache array to provide caching service to all the clients within this array. A fileid is generated for each file using a hash

function and used as the key for searching process. The whole hash space is divided into several disjointed zones with each server responsible for caching the objects whose hash key mapped into the corresponding zone. If a client request cannot be fulfilled by any of the servers, the responsible server will fetch the file from the remote original web server and store the file in its cache. The goal of Tulip system is to create a well-organized proxy cache server coordination model, provide higher caching service, better load balance and fault tolerance than the current approaches with minimal extra overheads. In this section, we describe the Tulip system design in detail and we concentrate our discussion on differences from other systems only.

3.1. Basic architecture

As shown in Figure 1, in Tulip, the cache space on each server is explicitly divided into two types: a *Buffer Cache*, which locates in a server's memory and a *Disk Cache* which locates on the hard disk. Current hash based caching systems do not divide the cache space, they view the cache space on a proxy server as a single entity and concentrate on improving cache hit rate, no matter the data locating in the memory or on the disk. However, because of the large access dispatch between memory and disk I/Os, the overheads of fetching cached objects from hard disk cannot be ignored.

Tulip focuses on reducing the average user perceived latency by increasing the buffer cache hit rate and reduce the number of small disk I/Os. In Tulip, correlated files that are likely to be accessed together are grouped into clusters and clusters are used as the basic transfer units between the memory and the disk. If a file is requested by a client and the cluster which contains this file is stored on the disk, Tulip reads the whole cluster into the buffer cache with only one big disk I/O. If a high percentage of other files in this cluster will be accessed by the following requests, the buffer cache hit rate can be improved, many small disk I/Os are avoided. Thus, the user preceived latency will decrease.

Assume a 64 KB data can contain 164 KB files. For modern disk systems, the typical time to read a 64 KB data is 7.1 ms while for a 4 KB data is 6.1 ms. There's only 1ms difference. This is because in a disk I/O, the rotational latency and the seek time

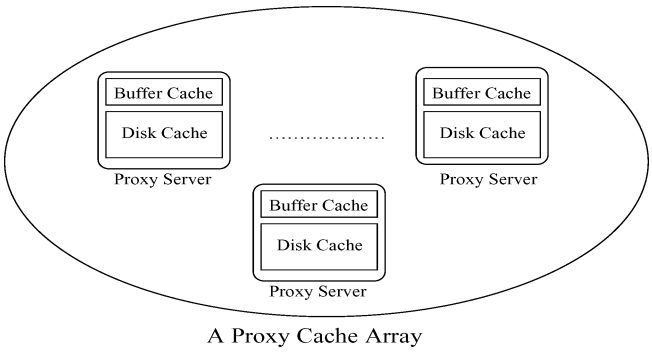

Figure 1. Tulip web caching system architecture.

dominate the whole access time, real data transfer operation only takes a small portion. If all these 16 files are fetched from disk individually, the total disk I/O time is $16 * 6.1$ (equals 97.6 ms). Even only 4 files are actually required in the near future, it still needs to take $4 * 6.1$ (equals 24.4 ms). Clearly, Tulip can greatly reduce the disk access overhead.

In Tulip, we use a file's original hostname to determine which server to cache this file. All the files from the same original web server are cached on the same proxy server. The whole hash space is divided into several disjoined zones and each server is responsible for one zone. Each client has a designated proxy server and it sends its requests to this server. In case of searching a file, the designated server calculates a hash key with the hostname in the file's URL string and then it forwards the request to the server which the hash key mapped into the corresponding zone. Then that server fetches the file from its cache or the original server. If no local caching is enabled, it returns the cluster containing that file to the client directly. If local caching is enabled, it sends the cluster to the designated server, the designated server returns the file to the client and stores the cluster in its local cache. Clearly, using hostname to generate the hash key is suitable for our purpose, the files from the same original server have higher possibility to be accessed together than files from different original servers.

We use two different mechanisms to exploit spatial localities: a dynamic mechanism and a static mechanism. The first one is extended from UCFS [22], it is complex but accurate. The second one is simple with the minimal overhead, however, it is not as accurate as the first mechanism.

3.2. Dynamic data grouping mechanism

The dynamic data grouping mechanism is adopted from UCFS [22]. We extend UCFS from a single proxy server to a distributed web caching system. We also migrate Cluster File System (CFS) from UCFS. A file can be uniquely identified by its URL string. However, the URL strings are too long and require significant storage space to store. As UCFS, MD5 algorithm is used to generate an unique key for each cached file in Tulip.

As shown in Figure 2, in the dynamic mechanism, the following data structures are used:

1. *An In-memory File Lookup Table*. this table is very important, it contains the location information for all cached files, it is used to track all files on the server. It is located in the memory to avoid slow disk I/O overheads. For a 4 million entries in this table, the total size is less than 96 MB, compare to the typical 512 MB or even 1 GB memory on a proxy server, this is affordable.
2. *A Buffer Cache*. it is divided into cold buffer, medium buffer and hot buffer. Files are located in different buffers according to their access frequency. When the files are fetched from disk, they are put into medium buffer first. In case the medium buffer is full, the frequently accessed objects are moved to hot buffer, and the least frequent used objects are moved to cold buffer.
3. *A Disk Cluster Table*. records the status of all disk clusters. In case of the RAM Buffer Cache are full, system searches this table and groups several files into a object cluster

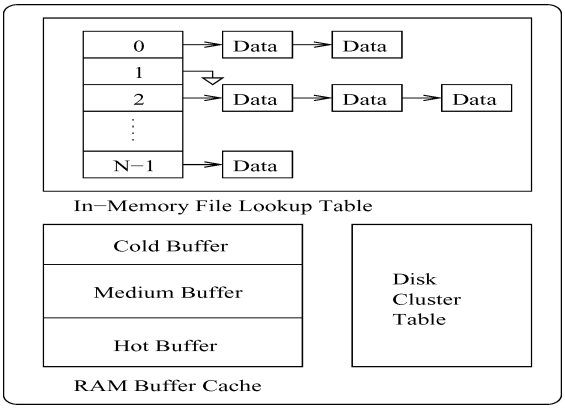

Figure 2. The In-memory data structure in the dynamic mechanism.

and write it to the disk within a one big write. The above three data structures are located in memory.

4. *A Cluster-Structured File System (CFS).* Since clusters are used as the basic units for disk I/Os, we cannot use the current file systems which does not support this situation very well. CFS is similar to LFS by using its raw disk I/Os, and unlike LFS, CFS does not require high file system consistency, the maintenance overheads are greatly reduced.

In dynamic algorithm, a locality-based algorithm called *two-level URL resolution grouping algorithm* which is firstly introduced in UCFS is used to group files. To start a group, the system first chooses a file to be swapped out and allocates a write buffer for this grouping, then system chooses other candidate files based on their locality association with the first evicted file. The detailed description and data operations on clusters can be found in [22].

3.3. Static data grouping mechanism

As shown in UCFS, the locality-based algorithm can effectively group files with spatial localities into the same cluster. However, it also introduces considerable overheads. A lot of data structures need to be maintained. In Tulip, we introduce another data grouping mechanism as well. The idea is very simple: The files that have more identical contents in their URL strings have more possibility to be accessed sequentially. This is because an HTML file often has many embedded files such as image files and audio/video files that will be accessed together. Moreover, the HTML file may also contain hypertext links to other HTML files, which are likely to be accessed in the near future by the same user. For example, three objects c.htm, d.jpg and e.gif all have the same contents in their URL strings "http://hostnameA/directoryB/XXXX", these objects are very likely to be accessed together and should be grouped into the same cluster. Since we already

have URL stirng information for each file, we can simplely group the files in the same directory into the same cluster before any access history information obtained.

In our static mechanism, the cluster creation procedure is as following:

1. Choosing an original web server hostname;
2. Start from the lowest directory on this web server. The files in the same directory are grouped into one or several clusters. In case the files within one directory cannot fill one cluster, the files in upper layer directory and sibling directories or files from hypertext links can be grouped together. In case the size of a file is larger than the size of the cluster, it can be divided into several clusters. We do not group files bigger than 256 KB;
3. After a new cluster is created, the individual files are removed. System goes back to the second step again for remaining files;
4. After all the files on the same original server are grouped into clusters, system goes to step 1 to start another process on a new web server.

Figure 3 shows the disk layout for our system. For both dynamic and static algorithms, we have the similar disk layout. However, in dynamic algorithm, the files grouped into the same cluster are generated according the client access history, while in static algorithm, the files in the same cluster are the files which reside in the same directory or adjacent directories from the same original web server.

Static mechanism has several advantages compared to the dynamic mechanism. First, it is very simple, it avoids the expensive spatial locality analysis procedure and the maintenance overheads. Second, we can use the URL string based data grouping (such as http://hostA/directoryB) instead of the hostname as the hash key and distribute files on the same original server to different proxy servers. Thus it can relieve the "hotspot" problem on a specific server. A drawback of the static mechanism is the contents in

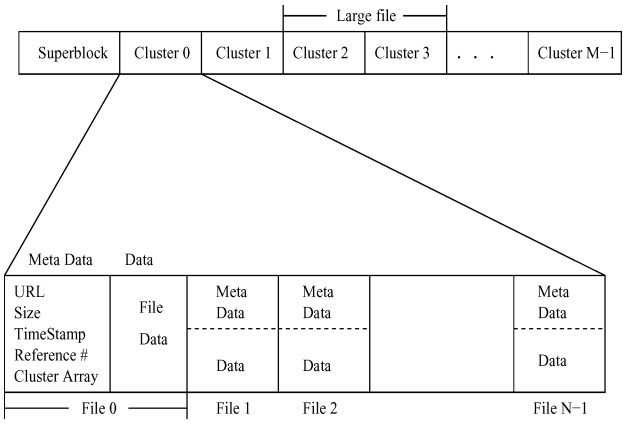

Figure 3. Disk layout table in our system.

a cluster do not change as the time passes. However, this problem can be relieved by periodically refreshing the cache contents on each server. Our simulation results show it can also achieve very good performance, although it is not as good as the dynamic mechanism.

3.4. Duplication strategy

For the standard hash based web caching system, each file has a fixed mapping relation with the server which caches it. The system forwards a client request to a proxy server according to the requested file's hash value. It neither needs to query a neighbor server nor exchange object lists among servers. This scenarios may easily cause a "hotspot" or "a single point of failure" problem. Another problem is in case of a server failure or a new server being added into the cache array, rebuilding the hash function is necessary and may result in a large volume of file migrating from one server to another. Robust Hashing [19, 20] solved the file migration problem by using multiple hash functions and caching a file on the server with the highest value. Only $1/n$ (n is the number of the files) files need to relocate when a new server is added and no operation needed in case of a server failure. Duplicated Hashing [13] created a second copy for each object by using the largest two hash values. It can effectively relieve the first problem. However, in these two systems, each time when a user request comes, the designated server must calculate multiple hash values before the forwarding decision is made. Thus, they generate considerable computation overheads, especially when the number of servers in a cache array is large.

Tulip system uses a simple and efficient duplication mechanism, it can achieve minimal data migration with the lowest computation overhead. In Tulip, each server duplicates half of its neighbor servers' contents. A simple illustration of our duplication strategy is in Figure 4. In this sample system, there are four proxy servers in the cache array. Assuming the hash space is [0,1023], and each server holds the files whose hash keys are within [0,255], [256,511], [512,767] and [768,1023] respectively. We divide all cached files into 8 subsets: D0 through D7. The size of each subset is 128. Each server caches files in 4 subsets. Server 0 holds subsets: D0, D1, D7 and D2. It is called the *primary cache server* for subsets D0 and D1 (for the files whose hash key is between [0,127] and [128,255]) and the *secondary cache server* for subsets D7 and D2.

With this file duplication strategy, the server can provide robust service and relieve "hotspot" and "single point of failure" problems. Since the system knows exactly which server has the copy for a certainsubsets, it can easily distribute client requests to either server and achieve better load balance. It can also forward the request to the remaining alive server if one of the two servers crashes. Tulip achieves this without extra computation overhead. fault tolerance. With two copies existed in two different servers simultaneously, if either of the servers fails, Tulip system still can provide caching service for the files on the other server. Thus the single point of failure problem is solved. Since the system knows exactly which servers contain the required data, a cached file in one server is also cached in the other two other servers. No multiple hash keys need to be calculated. File duplications can also be used for load balance purpose.

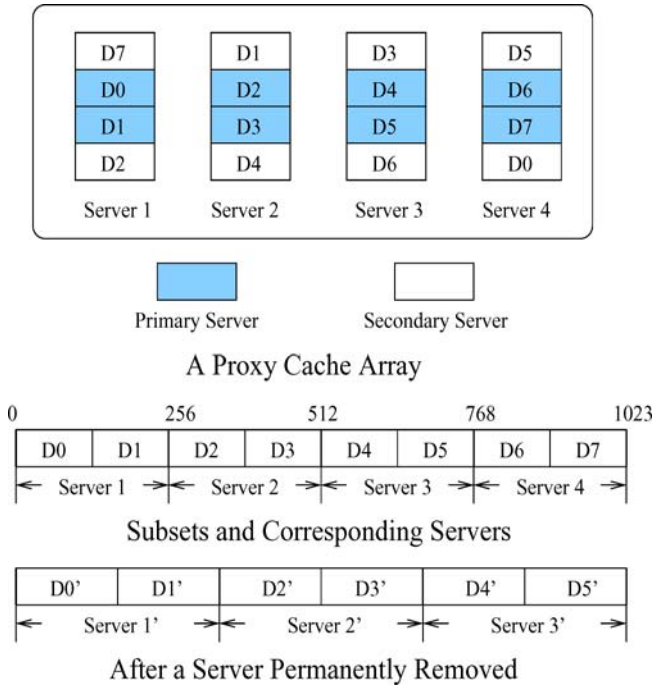

Figure 4. Tulip duplication strategy, 4-servers Cache array, Hash Space [0,1023] with 8 Subsets.

In case of the primary server hasheavy workload, clients can send their new requests to the secondary server. This is especially useful tosolve "hotspot" issue.

3.5. Server join/leave and failure

An important design principle in Tulip system is to reduce the impacts on system caching performance in case of environment change. In this section, we give the detailed discussion.

In Tulip, when a server failure occurs, the workload on this server can be automatically taken by its two neighboring servers. The system performance degrades gracefully. For example, in Figure 4, if server 2 fails, the workloads previously forwarded to server 2 are now automatically transferred to server 1 and server 3 according to the hash key of the requested file. Now, server 1 is responsible for cached files in 3 subsets: D0, D1 and D2. Server 3 is responsible for subsets D3, D4 and D5. However, server 4 is still responsible for two subsets D6 and D7 only. Thus, after a server failure, different servers maintain different number of subsets. However, this does not mean a server covering more subsets definitely has heavier workloads than a server covering fewer subsets. This is because different files have different access frequency. By using local caching technique together, Tulip still achieves good load balance with unevenly distributed subsets among servers. We choose this strategy because of its simpleness. No files need to be relocated.

In above mechanism, we suppose the failed server will soon recover after the failure is detected. However, if the server is permanently removed from the cache array, there are only three servers. Tulip needs to do a lot of file migration. The whole hash space need to be re-divided into 6 subsets and each server covers 4. However, there's a large overlap between the previous subsets and the new subsets as shown in Figure 4. Since the remaining servers can still provide caching services for all the subsets, the reconstruction procedure can be done gradually. The file migration operations are taken in the time when the client demand is low and each time only deal with one server. Thus, it does not affect system caching performance too much.

In case there are two servers out of service simultaneously, if they are not neighboring servers, the system can still provide service for all subsets with the remaining two servers. If they are two neighboring servers, files in certain subsets are lost, the system caching performance is degraded dramatically when clients request these files. However, since proxy servers are relatively stable, such kind of problem rarely happens.

When the number of client requests increases, the service provider may add new proxy servers into the cache array. Tulip uses two different strategies in case of a new server being added. One is a simple strategy which only affects two existing servers. As shown in Figure 5, the new server 1' is added and Tulip puts it between server 1 and 2. Then, only subsets D0 through D3 are affected. The files will be re-divided into 6 new subsets: D0' through D5' and they are redistributed on server 1, 1' and 2. The only modification on server 3 is it is the secondary server for subset D5' but not D3. Since D5' is only part of original D3, no file migration are needed. Server 3 can simply remove the files not belonging to D5'. For server 4, the same situation happens for subset D0'. As we mentioned earlier, uneven subset distribution does not necessarily cause uneven workload. The other one is a standard strategy, the hash space will be re-divided into 10 subsets: D0' through D9', each server still caches files in 4 subsets. However, it will generate considerable overheads. As a server permanently removed from the cache array,

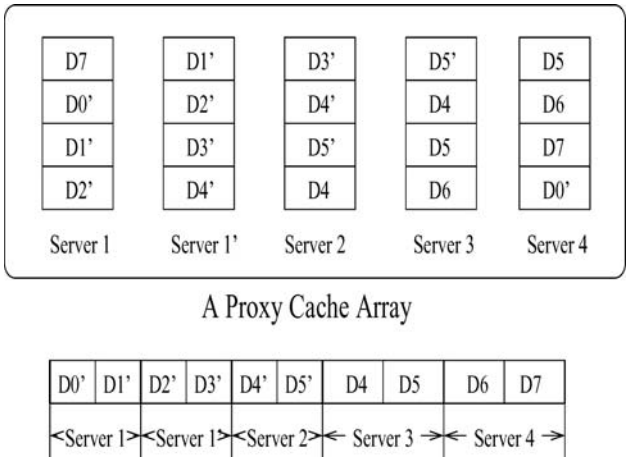

Figure 5. Cache array after a new server joins.

this operation can also be done when the service demand is low and dealing with only one server at one time.

We choose these two strategies based on two facts. First, proxy servers are much more stable than the ordinary client computers. After the deployment of proxy servers, they keep stable for a long time, the server location and the number of proxy servers in a cache array are seldom changed. The second fact is even hash space division does not result in equal workloads on each server. Some hot files have higher request rate than other files. By forwarding requests to both servers, this problem can be relieved even with an uneven division of hash space among proxy servers. Using local caching technique to solve the local requests can further relieve the workload on specific servers. We will discuss it in Section 3.7.

3.6. Duplication maintenance

While files are duplicated on two servers, the primary server must notify the secondary server the content change on some subsets. This will introduce new maintenance overhead. It is unaffordable to keep synchronous notification. To reduce the network traffic caused by the file copies, the primary and secondary servers need not to contain identical contents all the time. In Tulip, we use periodical noification mechanism. The primary server records a log of the changed contents since the last notification and sends them together to the secondary server from time to time. The interval of notifications can be varied according to the current client request rate and network traffic condition.

This strategy is reasonable. Since the hot files which are frequently accessed by clients will stay in the cache, only the less frequently accessed files have high possibilities to be replaced by other files. Thus, even we define a relative long notification time interval, hot files are still exist in both primary and secondary servers, the content difference between the primary and the secondary servers are rarely accessed files which have minor effects on system overall caching performance.

3.7. Local caching

One significant drawback in current hash based web caching systems is the hot spot problem. In Tulip, we use file duplication to relieve this problem. To further improve load balance, Tulip creates a local cache on each proxy server and caches the hot files for its local clients. Each time, when a client sends a request to its designated server and that server fetches a file from another server in the cache array, the designated server caches the cluster which contains the request file in its local cache. The data replacement policy used in local cache is also LRU. By abosorbing a lot of local requests for the hot objects on the designated server, the workload can be more evenly distributed, the client can get quick response. Our simulation shows it is an effective approach for hotspot problem.

3.8. Overhead analysis

To group the related files together, we have introduced extra data structures and algorithms. In [22], we have analyzed the overhead and speedup in dynamic algorithm. From

our analysis, although the dynamic algorithm brings extra computational overhead, it can still efficiently improve the web caching performance. The extra storage overhead of maintaining the extra data structure is about 4 KB per 64 KB cluster, consider the cheap storage price and large storage space today, it is not a big issue.

For static algorithm, we do not have to record and use the file access history, it generates the file clusters according to the relative location of the files on the original server, thus it has much smaller computational overhead than the dynamic algorithm. Although its caching performance is a little lower than the dynamic algorithm, it has the best performance/cost ratio.

4. Performance evaluation

In this section, we conduct several trace-driven simulation experiments to evaluate Tulip performance and compare it with the standard hash based web caching system. We introduce the simulation environment and workload traces first. Then we analyze the simulation results.

4.1. Simulation environment

We develop our own simulator for Tulip system and use DiskSim [10] as the underlying disk simulator which is a widely used comprehensive disk simulator. In our simulation, when the system fetches a file or a cluster from the disk, DiskSim is called to calculate the disk access latency. We use Quantum Atlas 10 K 9.1 GB drive as the basic model (which is the largest model currently provided by DiskSim) and triple the sectors of each track to expand the capacity to 27.3 GB.

Unless specified, we use the standard server configuration which a cache array contains four servers in our simulation experiments. Each server contains 512 MB memory and a 27 GB hard disk. The size of the Buffer Cache is set to 396 MB and the object cluster is set to 64 KB. 128 bit MD5 algorithm is used for hash key generation. The standard hash based system is used as the baseline, no file clustering enforced in it and each server only fetches the requested file. For Tulip, clusters are the basic disk I/O units. We denote DM as using the dynamic mechanism and SM as using the static mechanism. LC means local caching is enabled and NLC means local caching is disabled.

For both baseline system and Tulip, a client request is sent to the designated server first. For each newly coming client, system randomly assigns a server as the designated server for this client. The access latency for fetching a file or a cluster from the memory on the designated server is set to 0. If the file or the cluster is fetched from another server, we add 5ms network latency. If the data is fetched from the hard disk, the additional disk access latency is calculated by DiskSim. If the file is neither in the memory nor on the disk, the file must be fetched from the original remote server, we add another 100 ms network latency.

We use web proxy logs obtained from the National Laboratory for Applied Network Research (NLANR) in our simulation. The trace data are collected from six individual servers: *pa, pb, rtp, st, sj, sb* between April 30, 2002 and May 6, 2002. The characteristics

Table 1. Characteristics of workload trace: pa

Date	Client number	File number	Size< 1 KB	Size< 10 KB	Size< 100 KB	Size> 100 KB
Apr 30	922	1987913	841727	840772	291251	14163
May 01	979	1857267	762441	817483	264594	12749
May 02	893	1872351	796309	798530	264480	13032
May 03	945	2007385	853894	848746	288238	16507
May 04	1023	1336279	541515	599779	183870	11115
May 05	856	1196386	492466	529373	165641	8906
May 06	943	2050214	890906	844679	300159	14470

Table 2. Characteristics of workload trace: pa (continued)

Date	Total size	Max size	Min size	Median size
Apr 30	21.4 GB	137 MB	0	12.3 KB
May 01	19.2 GB	132 MB	0	15.6 KB
May 02	19 GB	248 MB	0	7.85 KB
May 03	20.9 GB	101 MB	0	11.7 KB
May 04	13.8 GB	191 MB	0	8.3 KB
May 05	12.1 GB	102 MB	0	13.4 KB
May 06	19.9 GB	201 MB	0	8.63 KB

of workload trace: pa are shown in Tables 1 and 2. The other workload traces have the similar characters, thus, we did not show it here.

4.2. Average client perceived latency

The main purpose of web caching is to minimize the client perceived latency when fetching a web object, thus the average client perceived latency is the most important metric to evaluate web caching efficiency. Here we test the average user perceived latency in Tulip and compare it with baseline. We use four different Tulip configurations: Tulip (DM, NLC), Tulip (DM, LC), Tulip (SM, NLC) and Tulip (SM, LC). The result is shown in Figure 6. The baseline system has the worst performance, all the four Tulip configurations outperform it, the average latency reduction are 54.79, 68.87, 51.38 and 67.01% respectively. Clearly, performance is improved significantly when using clusters. This is because the spatial localities among files are utilized and the disk access overhead is decreased. By using dynamic mechanism, Tulip (DM, NLC) and Tulip (DM, LC) have better performance than corresponding configurations using static mechanism. However, the performance gain here is trivious, showing the static mechanism is good enough to achieve satisfactory results, providing the minimal maintenance cost compared with the dynamic mechanism. Comparing Tulip (DM, NLC) and Tulip (SM, NLC) with Tulip (DM, LC) and Tulip (SM, LC), we can find using local caching is also very important. By enabling local cachig, the user perceived latency reduces 30.49% and 31.29% on average. From this figure, we can get the conclusion that Tulip (DM, LC) has the best

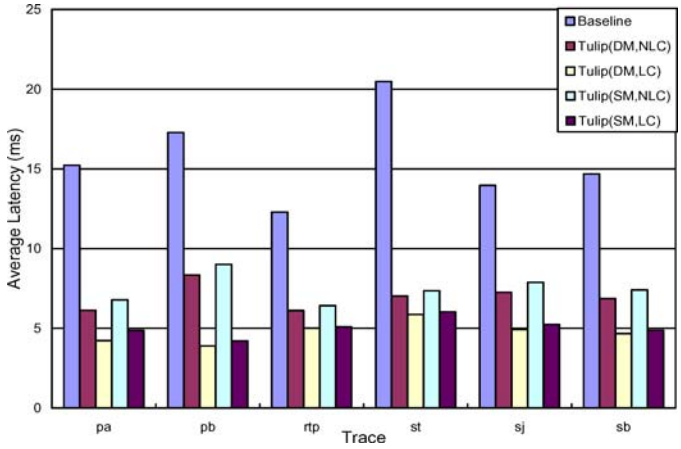

Figure 6. Average client perceived latency (ms).

performance, but Tulip (SM, LC) might be a better choice since it has near-optimal performance without introducing too much overheads.

4.3. Cache hit rate

In this experiment, we compare the cache hit rate of Tulip and baseline system, Duplicated Hashing system is also evaluated. The results are shown in Figure 7, the cache hit rate is overall hit rate including both the memory and disk cache hit rate. The baseline system has the highest overall hit rate, the cache hit rate in both Duplicated Hashing and the Tulip configurations are lower than in the baseline. The reason is in baseline system, there's no

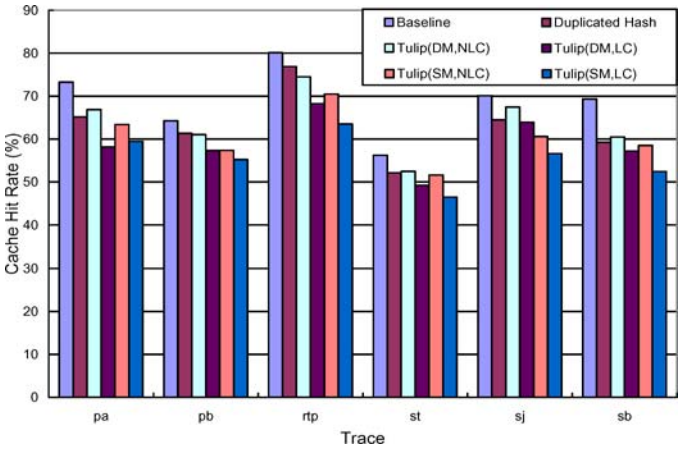

Figure 7. Overall Cache hit rate comparison.

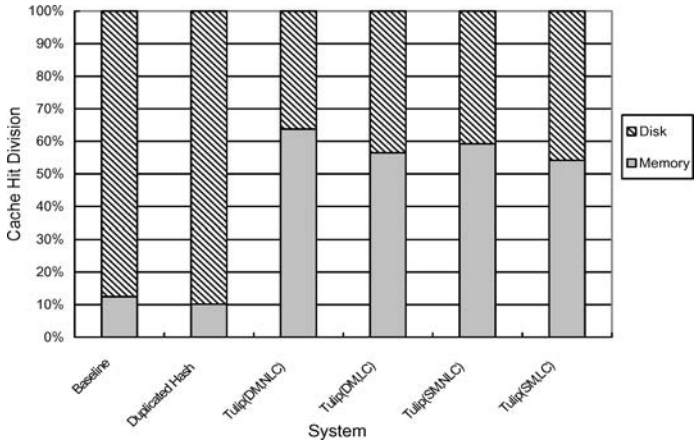

Figure 8. Cache hit distribution.

file duplication enforced, system has the highest storage space efficiency. Although the baseline system can cache more files than the other two, these files are rarely accessed and do not contribute too much for cache hit rate. Thus, the hit rate difference between the baseline and other systems are not very significant.

Duplicated Hashing, Tulip (DM, NLC) and Tulip (SM, NLC) use the similar duplication strategy: each file has at most two copies in the system. From the simulation results, these systems have similar cache hit rate as well. In Tulip (DM, LC) and Tulip (SM, LC), with the local caching enabled, each file may have more than two copies distributed in a proxy server's local cache, their system cache hit rate is slightly lower than systems without local caching. This is because part of the storage space is allocated for local caching (in which all the files are duplications). However, the hit rate reduction is small, which is between 3.68% and 12.98%.

We can find contradiction existing in the results of Figures 6 and 7. Although the baseline system has the highest overall cache hit rate, it has the highest average client perceived latency. To figure out the reason, we divide the cache hit rate into two types: memory and disk. Figure 8 shows the cache hit distribution for all six traces. For Baseline system, most cache hits are disk cache hits which introduce significant overheads. Duplicated Hashing system has similar characteristic. This is because they do not consider the spatial locality among file accesses. In all Tulip configurations, most cache hits are low cost memory cache hits. Though Tulip can not achieve a high overall cache hit rate, they can achieve lower client perceived latency. Dynamic mechanism is the best choice. Both Tulip (DM, NLC) and Tulip (DM, LC) have higher memory cache hit rate than their corresponding configurations with the static mechanism. However, the difference is trivious (which is only 11.43 and 8.58%). We can also find when local caching is enabled, the memory cache hit percentage is a little lower. However, since local memory cache hit has the lowest cost (the smallest latency), Tulip system with local cache enabled still has the smallest latency as shown in Figure 6.

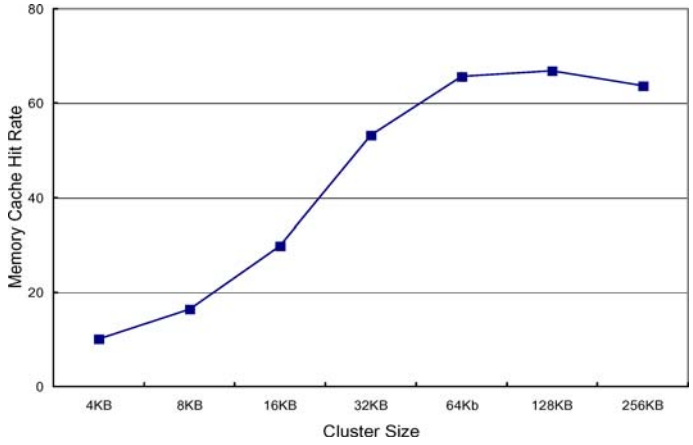

Figure 9. Memory Cache hit with different cluster sizes, Tulip (DM,NLC).

4.4. Cluster size effect

The size of the object cluster also affects Tulip system performance. With an increased cluster size, more spatial localities among file accesses can be exploited thus system can achieve better performance. On the other hand, it may group unrelated files into the same cluster, system can store fewer clusters, some relatively hot files might be evicted from the memory which reduces the system efficiency. Thus, an appropriate cluster size is very important for Tulip to gain the optimal performance. We vary the cluster size from 4 KB to 256 KB. Figure 9 shows the results. In this simulation, Tulip (DM, NLC) is used, we also evaluate other Tulip configurations and get the similar results. As the size of the cluster increased from 4 KB to 128 KB, the memory cache hit rate increases accordingly. With a 4 KB cluster size, the system memory cache hit rate is nearly the same as the baseline system which means the system can not utilize spatial localities efficiently. As the cluster size increased to 8 KB, 16 KB and 32 KB, the memory cache hit rate increases sharply. However, after the size reaches 64 KB, the hit rate increases slowly. As the cluster size increased to 256 KB, the memory cache hit rate even reduces, the reason is mentioned above.

4.5. Server failure

Tulip aims to improve system fault tolerance capability. In this experiment, we test its performance in case a server failure occurs. Since the possibility of two servers in a cache array breaking down simultaneously is very low, we only consider one server failure problem. Figure 10 shows the results. In baseline system, when a server failure occurs, a large portion of files previously cached on the failed server are lost, the cache hit rate reduces dramatically. While for both Duplicated Hashing and all the Tulip configurations, the cache hit rate does not decrease too much in case of server failure because nearly all the files on failed server have copies on other servers, the workloads previously taken by

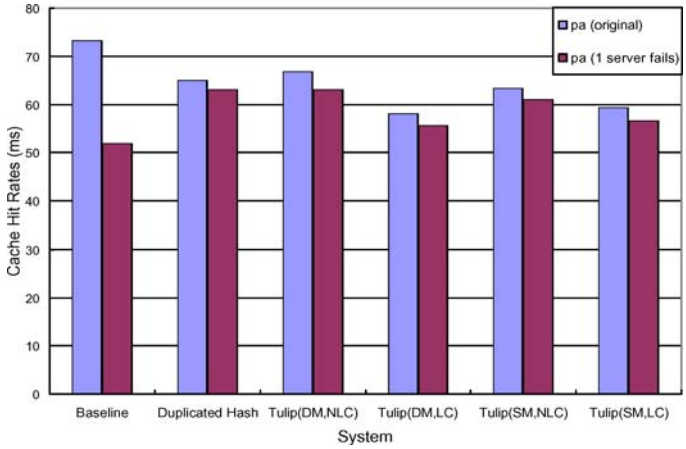

Figure 10. Cache hit rates in case of a server failure.

the failed server are migrated to other servers as well. Giving the same fault tolerance capability, Tulip systems has lower overhead than Duplicated Hashing system, it only needs one hash function while Duplicated Hashing system needs to calculate multiple hash values before satisfying a new request.

4.6. Load balance

Tulip can achieve better load balance. In this experiment, we use a synthetic traces with a Zipf distribution, a small number of files have much higher request frequency than other files. Table 3 shows the results. Without local caching, the workloads on each sever are skewed, baseline system and all the Tulip configurations can not achieve good load balance, the number of requests satisfied by server 1 is much higher than all the other three servers. By using local caching, the workloads are nearly evenly distributed among proxy servers. Clearly, local cache can greatly improve system load balance.

Table 3. Tulip load balance property

	Server 1 (%)	Server 2 (%)	Server 3 (%)	Server 4 (%)
Baseline	45.12	21.13	19.10	14.65
Tulip (DM, NLC)	46.22	20.03	17.12	16.63
Tulip (DM, LC)	28.38	25.19	22.48	23.95
Tulip (SM, NLC)	47.21	21.38	20.37	11.04
Tulip (SM, LC)	24.16	23.47	28.01	24.36

5. Related works

Web caching techniques have gained great popularity on the Internet [1–3, 18]. Since the number of proxy servers on the Internet increases rapidly, developing efficient and scalable cooperative web caching systems becomes an active research topic in recent years.

In Summary Cache [8], each proxy server keeps a summary of the cache directory of each participating server and checks these summaries for potential hits before sending any queries. It can reduce the number of query messages in hierarchical cache organization, however, it is not scalable, system generates considerable summary maintenance overhead, especially when there's a large number of servers. Cache Digest [15] uses similar strategy as Summary Cache.

Robust Hashing [19, 20] uses a very interesting strategy to reduce the maintenance overhead in case of server join/leave. In Robust Hashing, for each sibling cache, the URL and the sibling cache name are used together to generate a hash value or score, the object is then mapped to the sibling cache with the highest score. More specifically, let $h(u, c)$ be a hash function which maps a URI u and a cache server name c to an ordered hash space. For a given URL u, robust hashing calculates the score $h(u, c_1)$ ..... $h(u,c_n)$ for each of the n sibling caches, it routes the URL u to the sibling cache m that has the highest score. However, the system must calculate multiple hash scores which introduces extra runtime computation overhead.

Kawai and Yamaguchi proposed a duplicated hash routing algorithm [13], which is an extension of Robust Hashing System. Duplicated hash routing solves one of the drawbacks in hash routing: the lack of robustness against failure. Because WWW becomes a vital service on the Internet, the fault tolerance of systems that provide the WWW service is important. The algorithm introduces minimum redundancy to keep system performance when some caching nodes are crashed. In addition, it optionally allows each node to cache objects requested by its local clients (local caching) as Tulip, however, it does not consider server join/leave problem and the duplicated algorithm is not well defined.

Adaptive Web Caching [24] addresses incremental deployment issue. The general architecture of the envisioned adaptive web caching system would be comprised of many cache servers which self-organize themselves into a tight mesh of overlapping multicast groups and adapt themselves as necessary to changing conditions. This mesh of overlapping groups forms a scalable, implicit hierarchy that is used to efficiently diffuse popular web content towards the demand.

In [23], a proxy server connecting to a group of networked clients maintains an index file of data objects in all clients' browser caches. If a user request misses in both its local browser cache and the proxy cache, the browsers-aware proxy server will search the index file attempting to find it in another client's browser cache before sending the request to an upper level proxy or the web server. If there's a hit in a client, this client will directly forward the data object to the requesting client; or the proxy server loads the data object from this client and then sends it to the requesting client. It is an interesting approach to utilize the client resources, however, it introduce great maintenance overhead.

All the above solutions did not consider memory and disk access dispatch problem. In [11, 12], Hu and Yang proposed DCD and RAIPD Cache to improve file system

speed by aggregating several small expensive disk I/Os into large ones. UCFS [22] takes this approach and it is the first one to address slow disk I/O problem on the proxy servers. It presents a novel, user-space and custom-made file system called UCFS which can drastically improve I/O performance of a single proxy server. UCFS is a user-level software component of a proxy server which manages data on a raw disk or disk partition. Since the entire system runs in the user space, it is easy and inexpensive to implement. It also has good portability and maintainability. UCFS uses efficient in-memory meta-data tables to eliminate almost all I/O overheads of meta-data searches and updates. It also includes a novel file system called Cluster-structured File System (CFS). Tulip extends UCFS from a single proxy server to the hash based caching systems which contain multiple proxy servers and Tulip propose an efficient duplication mechanism to achieve better fault tolerance and load balance which does not addressed in UCFS.

Content Delivery Network (CDN) [6, 7, 9] is another major mechanism used in today's web services. Unlike web caching systems which use a reactive mechanism (the previously client accessed data are cached on proxy servers for future accesses), CDN takes a more proactive approach, the data are pushed to the content servers which are topologically close to the clients even before any accesses happened. CDN techniques may be used in combination with web caching systems to improve the performance. However, we do not address this problem in this paper.

6. Conclusions and future works

In this paper, we propose Tulip, a new hash based web caching architecture. It has three contributions. First, tulip exploits the spatial localities among file accesses and groups files that are likely to be accessed together into clusters. A dynamic and a static mechanism are introduced for grouping files. Dynamic mechanism can achieve best performance while static mechanism can achieve near optimal performance with the minimal cost. Second, Tulip can effectively increase the memory cache hit rates. By using clusters as the primary data transfer units, a large number of small disk I/Os are converted into less number of big ones, the slow disk I/O overhead problem is relieved. Third, Tulip uses a simple and efficient duplication algorithm to achieve fault tolerance and load balance. It introduces minimal maintenance overhead in case of server join/leave or server failure. Local caching technique is used to further reduce the client perceived latency and achieve even better load balance property. Our simulation results show Tulip outperforms the standard hash based caching systems.

We are currently investigating new algorithms to exploit spatial localities, a data grouping algorithm with the accuracy of dynamic mechanism with much lower overhead is desirable. We will also try to reduce the duplication maintenance overhead to further improve Tulip efficiency.

References

1. M. Baentsch, L. Baum, G. Molter, S. Rothkugel, and P. Sturm. World wide web caching: The application-level view of the internet. *IEEE Communications Magazine*, 170–178, June 1997.
2. G. Barish and K. Obraczka. World wide web caching: Trends and techniques May 2000.

3. A. Bestavros, R. L. Carter, M. E. Crovella, C. R. Cunha. A. Heddaya, and S. A. Mirdad. Application-level document caching in the Internet in *Proceedings of the 2nd International Workshop in Distributed and Networked Environments (IEEE SDNE '95)*, (Whistler, British Columbia), 1995.
4. A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell, A hierarchical internet object cache. in *Proceedings of USENIX Annual Technical Conference*, (Toronto, CA), pp. 153–164, June 1996.
5. V. V. J. Cohen, N. Phadnis, and K. Ross. Cache Array Routing Protocol v1.1." Internet Draft (draft-vinod-carp- vl-01.txt), September 1997.
6. G. T. M. Day, B. Cain, and P. Rzewski. A Model for Content Internetworking (CDI). Internet Draft (draft-ietf-cdi-model-01.txt), Feb. 2002.
7. I. C. F. Douglis and P. Rzewski. Known Mechanisms for Content Internetworking. IETF Draft (draft-douglis-cdi-known-mech-00.txt), June 2001.
8. L. Fan, P. Cao, J. Almeida, and A. Z. Broder, Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking*, 8 (3), pp. 281–293, 2000.
9. S. Gadde, J. S. Chase, and M. Rabinovich, Web caching and content distribution: A view from the interior. *Computer Communications*, 24 (2), 222–231, 2001.
10. G. R. Ganger, B. L. Worthington, and Y. N. Patt. The disksim simulation environment version 2.0 reference manual." http://citeseer.nj.nec.com/ganger99disksim.html, Dec. 1999.
11. Y. Hu and Q. Yang. DCD—disk caching disk: A new approach for boosting i/o performance in *Proceedings of the 23rd Annual International Symposium on Computer Architecture(ISCA), Philadelphia, PA*, pp. 169–178, May 1996.
12. Y. Hu, Q. Yang, and T. Nightingale. RAPID-Cache—a reliable and inexpensive write cache for disk I/O systems Tech. Rep. 1198-0001, Department of Electrical and Computer Engineering, University of Rhode Island, Nov. 1998.
13. E. Kawai, K. Osuga, and et al., Duplicated Hash Routing: A Robust Algorithm for a Distributed WWW Cache System. *The Institute of Electronics, Information and Communication Engineers Transactions*, 5, pp. 1039–1047, 2000.
14. C. Maltzahn, K. Richardson, and D. Grunwald. Reducing the Disk I/O of Web Proxy Server Caches. in *Proceedings of the USENIX Annual Technical Conference*, (Monterey, CA), june 1999.
15. A. Rousskov and D. Wessels, Cache Digest. in *the 3th International WWW Caching Workshop*, (Manchester, England), June 1998.
16. K. W. Ross. Hash-routing for collections of shared web caches. *IEEE Network Magazine*, Jan 1997.
17. P. Rodriguez, C. Spanner, and E. W. Biersack. Web Caching Architectures: Hierarchical and Distributed Caching March 1999.
18. L. Rizzo and L. Vicisano, Replacement policies for a proxy cache *IEEE/ACM Transactions on Networking*, 8 (2), 158–170, 2000.
19. B. Smith and V. Valloppillil, Personal communications February-June 1997.
20. D. G. Thaler and C. V. Ravishankar, Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking*, 6 (1), 1–14, 1998.
21. D. Wessels and K. Claffy, ICP, and the Squid Web cache *IEEE Journal on Selected Areas in Communication*, 16(3), 345–357, 1998.
22. J. Wang, R. Min, Y. Zhu, and Y. Hu. UCFS—A novel user-space, high performance, custom file system for web proxy servers. *IEEE Transactions on Computers*, 1056–1073, Sept 2002.
23. L. Xiao, X. Zhang, and Z. Xu, On reliable and scalable Peer-to-Peer web document sharing. in *Proceedings of International Parallel and Distributed Processing Symposium, (IPDPS'02)*, (Fort Lauderdale, FL), Apr 2002.
24. L. Zhang, S. Michel, K. Nguyen, A. Rosenstein, S. Floyd, and V. Jacobson, Adaptive web caching: Towards a new global caching architecture. in *3rd International WWW Caching Workshop*, June 1998.