ELSEVIER

Computer Networks 32 (2000) 261–275

COMPUTER NETWORKS

www.elsevier.com/locate/comnet

Adaptive-level memory caches on World Wide Web servers

Da-Wei Chang [a], Hao-Ren Ke [b,*], Ruei-Chuan Chang [a]

[a] *Department of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan, ROC*
[b] *Library, National Chiao Tung University, 1001 Ta-Hsueh Road, Hsinchu, Taiwan, ROC*

Abstract

Owing to the fast growth of World Wide Web (WWW), web traffic has become a major component of Internet traffics. Consequently, the reduction of document retrieval latency on WWW becomes more and more important. The latency can be reduced in two ways: reduction of network delay and improvement of web servers' throughput. Our research aims at improving a web server's throughput by keeping a memory cache in a web server's address space.

In this paper, we focus on the design and implementation of a memory cache scheme. We propose a novel web cache management policy named the *adaptive-level* policy that either caches the whole file content or only a portion of it, according to the file size. The experimental results show three things. First, our memory cache is beneficial since, under our experimental workloads, the throughput improvement can achieve 32.7%. Second, our cache management policy is suitable for current web traffic. Third, with the increasing popularity of multimedia files, our policy will outperform others currently used in WWW.

Keywords: Web server; Memory cache; Cache management policy; Adaptive-level policy

1. Introduction

With the popularity of World Wide Web (WWW), web traffic has become the fastest growing component among all kinds of Internet traffics. However, large volumes of traffic causes the document retrieval latency perceived by web users becomes longer. Many researchers notice the problem and have made efforts on improving WWW latency.

Generally speaking, WWW latency comes from two sources.

1. *Network delay*. In HTTP 1.0, retrieving a file from a web server needs at least two round trips between the client and the server.
2. *Request processing time*. For each requested file, a web server has to read it from its disks into a memory buffer and then sends the buffer to the client.

Thus, to reduce document retrieval latency, we must either reduce network delay or server's request processing time.

Researches that aim at the reduction of network delay focus on improvement of HTTP and development of caching strategies on proxy servers and clients. Other researches improve the web servers' throughput in the following ways:

1. *Cooperative servers* [10]. Instead of a single web server, multiple cooperative servers can be used to serve users' requests.
2. *Caching documents at server side* [12]. In order to cope with the high request rate, Markatos [12] suggests caching documents in a web

* Corresponding author.

1389-1286/00/$ - see front matter
PII: S1389-1286(99)00131-0

server's address space (which is called memory caching).

In this paper, we focus on the design and implementation of memory caches in web servers. We propose a novel web cache management policy that is named *adaptive-level* policy (i.e. file-level as well as chunk-level). All of the previous web cache management policies use file-level caches. That is, given a file, these kinds of cache policies keep all its content or nothing of it. However, we consider the file size while we are caching it. If a file is smaller than a chunk (i.e. a block with a predefined size), it can totally be cached. Otherwise, only a chunk of the file will be cached. We show that memory caching in web servers is beneficial since it improves throughput of web servers. In addition, we show that our cache management policy is suitable for current web traffic and outperforms other existing policies when accesses to large files become more and more popular.

The rest of this paper is organized as follows. In Section 2 we discuss the related works. Section 3 presents the motivation of the adaptive-level memory cache. Section 4 shows the design and implementation. The experiment results are presented in Section 5. Conclusions and future works are given in Section 6.

2. Related works

There are lots of research efforts focusing on improving WWW latency. In general, these efforts fall into three categories: improvement of HTTP, prefetching, and caching. In this section, we will briefly discuss the works on these three categories.

2.1. HTTP improvement

HTTP is a simple protocol layered over TCP. However, as shown in [16], several features of HTTP interact badly with TCP. Two such features are that HTTP establishes a new connection for each request and HTTP transfers only one object per request. Mogul [13,14] also noticed the problems and proposed two mechanisms, *long-lived connections* and *request pipelining*, to improve HTTP latency.

2.2. Prefetching

Instead of reducing WWW latency between clients and servers, Padmanabhan and Mogul [15] use prefetching to "hide" the latency. The benefit of prefetching comes from that there is a period of time between two adjacent requests from the same user, and this time can be used to prefetch next document that the user wants to read. Thus, with prefetching, when a client requests a document, it may be already in client's local cache.

2.3. Caching

The most common technique on reduction of data retrieval time is caching. Researches on WWW caching can be divided into three classes: the client-side caching, the network caching (i.e. proxy caching), and the server-side caching. In the following, we shall briefly describe these WWW caching efforts.

2.3.1. Client-side caching

Cunha et al. [7] analyzed the access patterns of individual users, and he found two access patterns. The first is that WWW clients tend to access small files. The second is that, keeping small files in client's cache is better than keeping large ones since the former has a larger latency-savings-per-byte. Bestavros et al. [4] compared three caching levels at the client side: the session level, the host level, and the LAN level. According to their experiments, the LAN-level caching is the most cost-effective policy among the three policies. This is not a surprising result. Given a specific document, a LAN cache keeps at most one copy of it. However, each session or host cache may keep one copy of the document in its own cache, and therefore waste the cache space.

2.3.2. Proxy caching

Abrams et al. [1] evaluated the limitations and potentials of proxy caches and showed that the upper bound of hit ratio of a proxy cache, regardless the cache replacement policy, is only about 30–50%. Since a basic idea for caching is to move the data that clients need closer to them. Some proxy-cache researches [3,9] take geographic distribution of client requests into account.

While many researchers proposed their own proxy caching policies, Williams et al. [19] provided a performance comparison among these policies and showed that the widely-used WWW caching policy, LRU, results in poor performance. In addition to the analysis of the traffic of WWW proxy servers and the invention of proxy cache strategies, there are some proxy cache implementations, such as the CERN cache [8], the Lagoon cache [6], the Harvest cache [5], and the Squid cache [17].

2.3.3. Server-side caching

While the purpose of caching at client-sides and proxies is to reduce data retrieval latency caused by the network, server-side caching strategies focus on the reduction of servers' response time and the improvement of servers' throughput.

Arlitt and Williamson [2] analyzed access logs from different web servers and identified ten invariance among these workloads. Kwan et al. [10,11] described the research efforts on web caching made by NCSA that use AFS to cache documents in web servers' local disks. In contrast to the work of NCSA, Markatos [12] proposed the notion of memory caching of web documents. The benefit of memory caching comes mainly from that it reduces the number of disk accesses. By keeping most frequently accessed files in main memory, many of the requests can be served without touching the file system, and therefore the access latency is reduced. Markatos also presented a cache replacement policy. However, he only showed the performance of his policy in terms of hit ratio by trace-driven simulation and no implementation results. The improvement of servers' throughput contributed by memory caching is still left unknown.

Our work differs from others in that, we purpose and implement an adaptive-level memory cache policy, and we also measure the performance improvement by a popular benchmark, WebStone [20].

3. Motivation

In this section, we describe the motivation of memory caching in web servers and the adaptive-level cache management policy.

3.1. Motivation of memory caching in web servers

There are fewer researches aiming at improving web servers' throughput than reducing the network delay. The reason is simple: the network delay currently dominates the document retrieval latency perceived by web users and thus reducing network delay effectively shortens the document retrieval latency. However, in contrast to most of the related researches, we focus on improving a web server's throughput because of the following reasons.

1. *High request rate*. Even with client-side and proxy caches that reduce the amount of traffic coming to a web server, a previous research [12] shows that the peak request rate for a busy server remains high.
2. *Improvement of network speed*. With the fast improvement of network speed, the situation that network delay dominates the document retrieval latency is changing. It is very possible that web servers will become the bottleneck in web document retrieval in the near future.
3. *Ratio of local to remote requests*. Preliminary data [7,12] present that the ratio of local to remote requests is about 1:4. This means that there are still 20% of the total requests coming from local sites. For these requests, the clients and the server are almost connected by high-speed network which has short delay and high bandwidth. Therefore, improving the server's throughput will cause a noticeable reduction in document retrieval latency.

Due to the reasons mentioned above, we know that it is effective to reduce document retrieval latency by improving a server's throughput. To improve a server's throughput, we focus on reducing the number of disk accesses by maintaining the most frequently accessed documents in a web server's address space. Generally speaking, the operating system underlying the web server maintains a buffer cache and hence prevents some file system requests from really going to the disk. However, we additionally maintain a separate cache instead of only using the original buffer cache provided by the operating system. The

reason is that, as shown in a previous research [12], approaches that manage buffer caches are not suitable for web server traffic. Furthermore, even with buffer caches, web servers also need to call *read( )* system calls to read data from the buffer caches. It costs at least one kernel-to-user data copy.

3.2. Motivation of adaptive-level caching

Previous cache implementations for WWW use file-level caches because the accesses for web documents are all-or-none accesses. That is, if a user makes a request for a file, he or she will receive the total content of it (if success) or nothing of it (if some errors occur). However, there is a drawback for this kind of caching mechanism. Since the cache size is limited, putting a whole large file in the cache may cause many other small files be flushed out. The situation is worse especially for web caches, because web users tend to access small files and therefore the files flushed out are usually popular ones. This will result in poor cache performance.

To overcome the drawback, file-level caches always equip with a threshold and never cache files larger than this threshold. It avoids the situation mentioned above. However, it causes another problem. A large file will never be cached although it may be the most popular one. This will also degrade the performance.

Due to the above reasons, we propose the idea of adaptive-level caching. When a file is to be cached, we consider its size. If it is smaller than a threshold, we cache its total content. Otherwise, only the first chunk of it is cached. Clearly, it avoids the first problem since caching only a chunk of a large file does not require too much room. Therefore, fewer small but popular files will be flushed out. At the same time, if a large file is popular, it is better to cache a portion of it than do nothing at all. In addition, caching the first chunk enables a server to respond to its clients quickly since the first chunk can be sent immediately from the cache without involving disk accesses and the server can read the rest of the file from the disk at the same time.

4. Design and implementation of the memory cache

The design goals of our memory cache are to improve the throughput of a WWW server so that it can handle more requests concurrently and to minimize the overhead due to the maintenance of the memory cache. In the following two sections, we shall present the design and implementation of the memory cache.

4.1. Design of the memory cache

4.1.1. MIME header caching

For each client request, the web server sends a response to the client. The response normally contains a MIME header followed by the requested file. This MIME header contains the return code, the information related to that server, and the attributes of the requested file. Generally, MIME headers are generated on the fly, which means the server has to reconstruct the MIME header of a file each time when it is requested. This imposes overheads on a busy web server. However, this overhead can be reduced by caching MIME headers in advance, since they seldom change due to the following reasons. First, most of the fields (except for the file attributes) do not change. Second, most web pages are seldom modified. Therefore, the file attributes usually remain unchanged, too. Third, WWW is a weakly consistent access system.

With MIME header caching, the server does not need any more to spend its time on dynamically constructing a file's MIME header. Instead, it can send the cached MIME header directly to the client. This reduces some overheads that are imposed to the server. However, if a file is updated, the server should recognize it and invalidate the cached copy of that file as well as its MIME header. This can be achieved by attaching a timestamp to each MIME header to record the time that the file is inserted into the cache. After doing that, the server can ensure the freshness of the cached copy of a file by comparing the timestamp with the last-modified time of that file. If the cached copy is not fresh, the server can just discard it or re-load the file to into the cache. It should be noted that, since WWW is a weakly consistent

system, the timestamp-comparing task does not have to be performed frequently. Instead, servers can perform this task during their idle periods so that it will not degrade the performance of servers.

Currently, we do not incorporate such a timestamp-comparing task into our server. The reason is that we only concern the benefit of a memory cache at normal and heavy server loads.

4.1.2. Adaptive-level caching

As we describe earlier, we use an adaptive-level cache. We use a predefined threshold, say MAX_FILE_SIZE, to determine if we should put the total content of a file into the cache. If the total size of a file and its MIME header is less than MAX_FILE_SIZE, we cache its total content and its MIME header (this is called completely cached). Otherwise, we allocate a cache space with size MAX_FILE_SIZE and put the MIME header and a portion of the file into the cache space (this is called incompletely cached).

Fig. 1 illustrates the cases of completely and incompletely cached. In the case of incompletely cached, we cache the first k bytes of the file where k equals to the MAX_FILE_SIZE minus the size of the file's MIME header.

4.1.3. Cache replacement policy

In addition to decide whether to cache a file completely or not, we still have to determine which files should be removed from the cache when the room of the cache is not enough. That is, we must choose a cache replacement policy.

We use least frequently used (LFU) as our cache replacement policy. The reasons are as follows. First, due to the fact that some web pages are popular (i.e. frequently referenced) and others are not, it is straightforward to use LFU as the cache replacement policy. Second, previous researches [2] show that about one-third of files and bytes recorded in a web server's access log file are accessed only once. These files are seldom accessed and therefore keeping them in cache gains almost nothing at all. By using LFU, these files will not be cached even they are referenced recently, because the access frequencies of these files will generally smaller than those of the cached files. Third, its implementation is straightforward. By keeping a reference count for each file in the local web site, we can easily implement the policy.

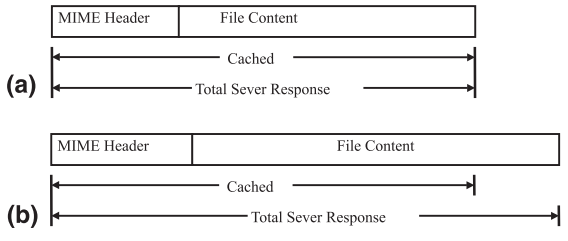


Fig. 1. (a) The case of completely cached, and (b) the case of incompletely cached.

4.1.4. Cache lookup

For each request, the server has to find whether the requested file is in the cache. This overhead should be minimized since it happens frequently. To achieve this, we use a hash table to locate files in the cache. Each time when a file is requested, its name is used as input of the hash function to locate the cached data.

4.1.5. Free list handling

In order to save memory space, cache buffers are allocated on demand. That is, cache buffers are allocated only when we need to insert data into the cache. When we remove a file from the cache, its buffer is released immediately. However, it is possible that a file is requested and inserted into the cache soon after it has just been removed. In this case, if we free the buffer immediately, we have to allocate the buffer, copy in the file content and link the buffer into the cache again. This causes unnecessary overheads when a file is needed immediately after it has been removed.

Therefore, we use an alternative approach. We maintain a free list to hold the corresponding buffers for files removed from the cache. When we remove a file, we unlink the buffer from the cache and link it into the free list. On the other hand, when we try to insert a file into the cache, we search for the free list first. If the file is in the free list, we unlink it from the list and re-link it into the cache. Otherwise, the file can only be read from the file system.

Because a free list is only an auxiliary tool that gives files a second chance from freeing their buffers, it should not impose too much pressure to a server. Specifically, it should not consume too much memory space. In order to achieve this, we flush the entire free list periodically.

4.2. Implementation of the memory cache

In this section we shall describe the implementation of the memory cache. Instead of developing a whole new web server, we modify an existing one. We choose a tiny web server, named Thttpd [18], which is a freeware developed by *ACME Laboratories* [18], and, for simplicity, we only handle the "GET" requests. That is, the server does not provide any other services except for sending files to the clients.

In the following subsections, we shall first describe Thttpd. And then, we present our modifications to it, including the data structures and the cache maintenance procedures.

4.2.1. Overview of Thttpd

Thttpd is a tiny web server that can run on many UNIX variants. It is a single-processed web server and uses the *select( )* system call to multiplex concurrent requests (i.e. connections). To handle multiple connections concurrently, a fixed-size *connection table* is maintained in it. Each non-free entry represents an active connection and records the information about that connection. The information includes the socket file descriptor for that connection, the file descriptor of the requested file, total bytes to be sent, total bytes have been sent, etc.

Fig. 2 shows the control flow of Thttpd. After initialization, the server enters an infinite loop. In the loop, it first calls *select( )* to poll every connections.

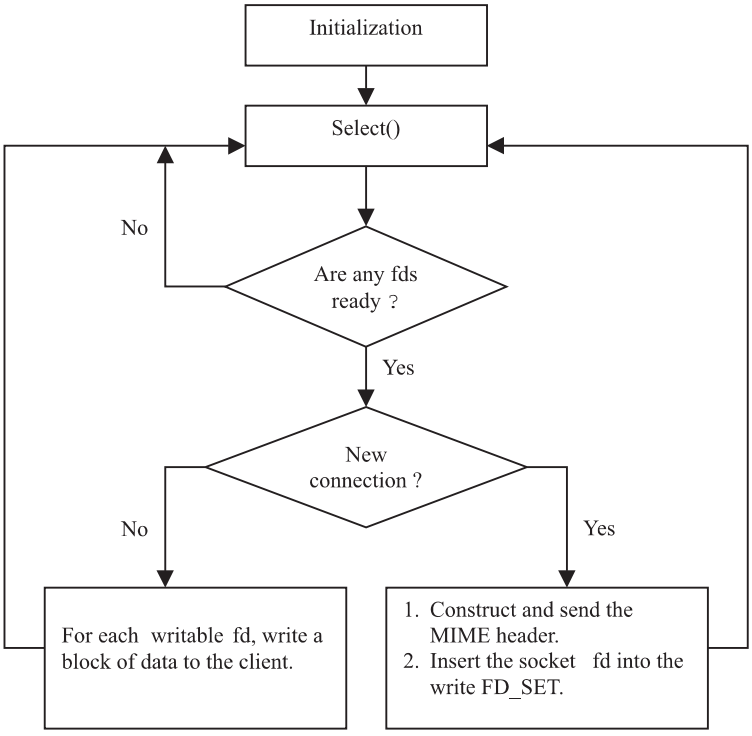


Fig. 2. The control flow of *Thttpd*.

There are three sets of file descriptors (i.e. FD_SETs) in *select()*, read, write and exception FD_SETs. The former two FD_SETs are used in Thttpd. The web server's socket descriptor (i.e. the WWW socket descriptor that usually listens at port 80) is the only file descriptor in the read FD_SET. It is used for accepting new connections. In other words, there are new connections coming to the server if the read FD_SET gets ready. For each active connection, the server has to send the file to the peer client by writing data to the socket descriptor of that connection. These socket descriptors are inserted into the write FD_SET. Therefore, if one of the file descriptors in the write FD_SET gets ready, the server can write data to that connection.

When a new connection arrives, the following steps are taken:

1. The server finds a free entry in the connection table to record the information about the new connection.
2. The MIME header for the requested file is constructed on the fly and sent to the client.
3. The socket descriptor for the connection is inserted into the write FD_SET.
4. The control flow goes back to *select()*.

Notice that none of the file content has been sent to the client yet.

After all the new connections are processed, the server starts to send data for existing connections. For each ready file descriptor in the write FD_SET, the server reads one block of the corresponding file into a memory buffer and then writes the buffer to the client. After all the ready write file descriptors are served, the control flow goes back to *select()*.

From the above description, we can see that the server gives a higher priority to accept new connections than to serve existing ones. The reason is that, for each connection which has finished the TCP's three-way handshaking procedure, the operating system inserts an entry corresponding to that connection into a queue. And the server process can dequeue the entries by calling *accept()* system calls. However, most operating systems impose a limit on the length of this queue and a new connection is dropped when the queue length exceeds the limit. Therefore, the server gives a higher priority to accept new connections so that it can process the new connections as quickly as possible and hence shortens the length of the queue.

4.2.2. The control flow of the modified web server

Fig. 3 shows the control flow of the web server after our memory cache is added. Similar to the original web server, the modified version gives a higher priority to accept new connections.

When a new connection arrives, the server first checks whether the requested file is cached. If it is not cached, the server attempts to insert it. In our current implementation, the cache space needed to insert a file is the total size of its MIME header and the file content or the predetermined threshold: MAX_FILE_SIZE, depending on which is smaller. If the available room of the cache is not smaller than the needed space, the file can be inserted without removing any other files out of the cache. Otherwise, the cache replacement procedure must be taken.

If the cache insertion (or cache replacement) procedure fails, which means that the file cannot be inserted, the following steps are taken. First, the MIME header is sent to the client. Second, the file's descriptor is inserted into the write FD_SET and then the control flow goes back to the *select()*. The data of the file will be sent in later iterations.

If the cache insertion procedure successes or the file is already cached, the server checks further to see if the file is completely cached or not. If it is completely cached, the server writes the cached data directly to the client, closes the connection (since the request is completely served), and the control flow goes back to the *select()*. Otherwise, the server first writes the cached data to the client. Then, it opens the file and arranges the read header of that file to the beginning of the rest of the file data by calling *lseek()* function. At last, the control flow goes back to the *select()* again. Thus, the rest of the data can be sent in later iterations.

4.2.3. The data structures of the memory cache

In this section we present the data structures used in the memory cache. There are four kinds of data structures in it: the file header, the hash table,

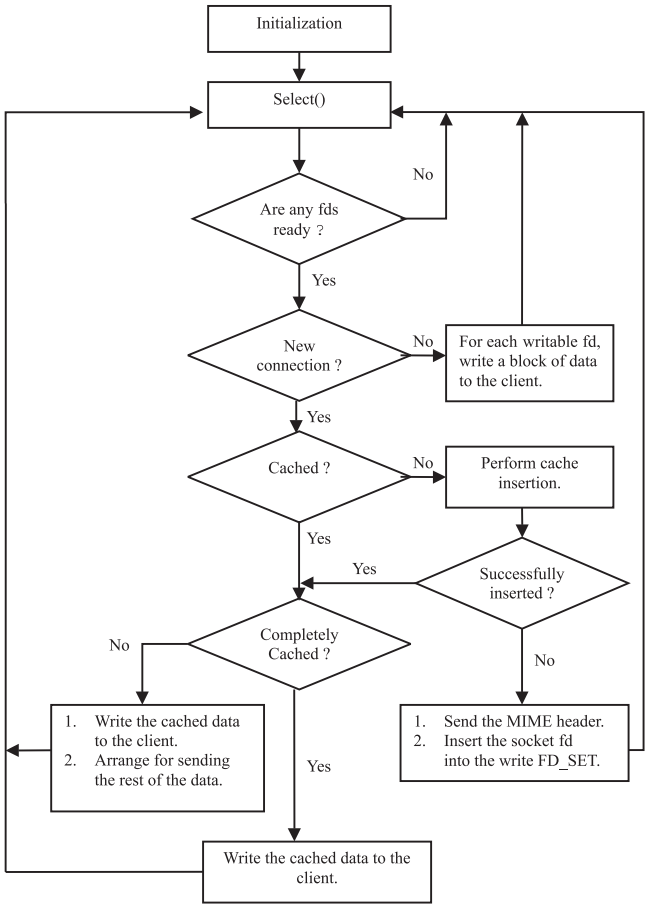


Fig. 3. The control flow of the modified *Thttpd*.

the frequency list, and the free list. Below we describe them in more detail.

4.2.3.1. The file header. The file header is the central data structure in our web cache implementation. Each file that can be served by the web sever has a corresponding file header which, in our current implementation, is loaded into the memory by the web cache initialization procedure. The fields of a file header are shown in Fig. 4. These include:

1. *state*. This field indicates whether the file is cached or not. The value of this field is either CACHED or NOT_CACHED.
2. *file_name*. The URL of the file. It is used as an ad hoc tool for locating the file header that corresponds to a given file.
3. *freq*. The access frequency (i.e. reference count) of the file. Each time when a file is requested, its access frequency is increased by 1.
4. *prev, next*. They are two pointers linking to the previous and the next file header in the frequency list or the free list. Because each file header is located in at most one of the two lists, only one pair of link fields is enough. The frequency list and the free list will be described later.
5. *hash_next*. A link to the next file header in the hash chain, which will be described later.

```
typedef struct file_header {
                        int             state ;         /* The state information */
/* Values of file_header.state */
#define NOT_CACHED      0                               /* the file is cached       */
#define CACHED          1                               /* the file is not cached   */
                        unsigned char   *file_name ;    /* name of the file         */
                        int             freq ;          /* reference count          */
                        struct file_header *prev ;      /* frequency list or        */
                        struct file_header *next ;      /* free list links          */
                        struct file_header *hash_next ; /* hash chain link          */
                        unsigned char   *mime_header;   /* mime header buffer       */
                        unsigned int    mime_length;    /* length of the mime header */
                        unsigned char   *file_data ;    /* data buffer              */
                        unsigned int    total_length;   /* file size + mime_length  */
} file_header ;
```

Fig. 4. The *file_header* structure.

6. *mime_header*. A pointer to a buffer that holds the MIME header of the file.
7. *mime_length*. The length of the MIME header corresponding to the file.
8. *file_data*. A pointer to a buffer that holds the MIME header and the content (either totally or partially) of the file. The buffer itself is the cache space of this file. That is, the actual operations to insert a file into the cache are allocating a buffer to hold the file's MIME header and content (may be partially), and linking the buffer into this field. The reason for putting the MIME header in the buffer is that the server can send the MIME header together with the file content in a single *write( )* system call. If we do not do so, the server has to send the MIME header and the file content in separate system calls, or use a more complicated interface, *writev( )* to send them in a single system call.
9. *total_length*. The total size of the file's MIME header and content. This field is used to determine whether a file is, or can be, completely cached or not. If it is larger than MAX_FILE_SIZE, the file cannot be completely cached. Otherwise, it can be cached completely.

Because each file served by the web server has a corresponding file header, its size should be small enough so as not to impose large space overheads to the web server. From Fig. 4, we can see that the size of a file_header entry is 40 bytes. If a web site has one thousand files, the total size used by the file header structures is 40 K, which is acceptable.

4.2.3.2. The hash table. Each time when a new request arrives, the first task of the web server is to see if the requested file is in the cache or not. It can be done by simply checking the *state* field of the corresponding file header. To locate the corresponding file header, a hash table is used. The key of the hash table is the name of a file. All file headers hashing to the same entry in the hash table are chained by their *hash_next* fields. Although they are in the same hash chain, they can still be identified easily according to their *file_name* fields.

Notice that the hash table is used to locate **all** the file headers in the web server, not just the file headers corresponding to the cached files. All the file headers are inserted into the hash table by the

initialization procedure and kept there all the time when the server executes.

4.2.3.3. The frequency list. The frequency list is an ordered list of the file headers corresponding to all the cached files. The first file header of the list has the least frequency (among all cached files) and the last has the most frequency. The reason for keeping a frequency order in the list is for the efficiency of the cache replacement procedure. Since we use LFU, the least frequently cached files will be removed from the cache first. Thus, by keeping a frequency order, we can easily find which files should be removed first when we perform the cache replacement procedure.

4.2.3.4. The free list. The free list is a doubly linked list that starts with a pointer: *free_start* and stops at another pointer: *free_end*. When a file is removed from the cache, the web server first removes its file header from the frequency list (since it is not cached any more). Then, the file header is appended into the free list. Notice that the cached data (i.e. pointed by the *file_data* field) is not released in case that the file may be re-inserted into the cache soon.

When a file is in the free list, to re-insert it into the cache is quite simple: removing the file header from the free list, inserting it to the frequency list and setting the *state* information of the field header as CACHED. It is also efficient since the overheads of allocating memory buffers and data copy are eliminated.

Fig. 5 gives a global view of the web cache data structures. All file headers can be located by the hash table keyed on file names. When a file is inserted into the cache, its file header is inserted into the frequency list. When it is removed from the cache, the file header is moved to the free list.

To prevent the free list from growing too large, we should flush the free list periodically. In the current implementation, we perform this task every one hundred accesses. We choose number of accesses, instead of absolute time, as the flush period because the memory size of the free list is related to the former, not the later.

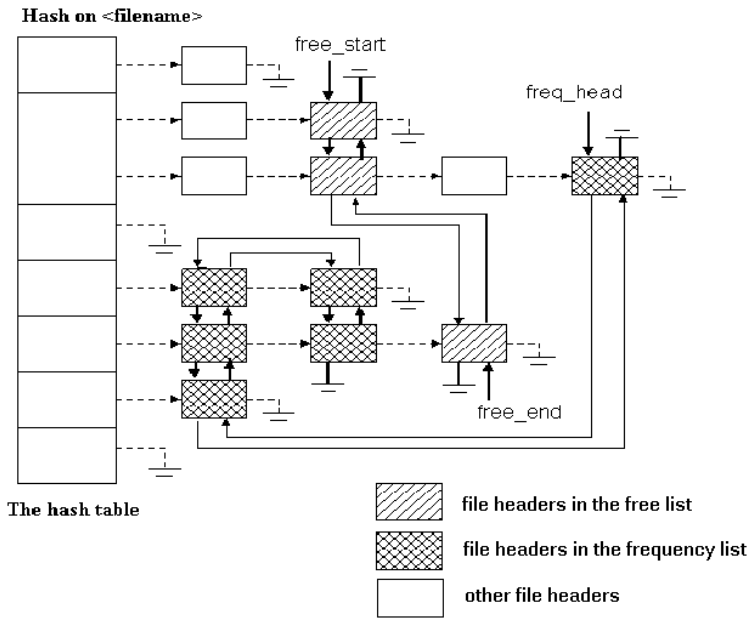


Fig. 5. Data structures: the hash table, the frequency list, and the free list.

4.2.4. The procedures of the memory cache

In this section we describe the procedures of our memory cache in detail. We focus on the initialization and the cache replacement procedures.

4.2.4.1. The initialization procedure. The first step of this procedure is to initialize the file headers of all the files that can be served by the web server. To achieve this, in our current implementation, we use a file (named *file_list*) to record all the names of those files. At initialization, the server reads the *file_list*, counts the number of files in it, and allocates an array of file headers for these files according to that number. The server then initializes all file headers by constructing their MIME headers. At last, the server constructs the hash table and inserts all the initialized file headers into hash chains.

4.2.4.2. The cache replacement procedure. The cache replacement procedure works as follows:

1. Suppose that we want to insert a file, say p, into the cache. We say that a cached file q can be "removed" by p if and only if the frequency of p is larger than that of q.
2. We check whether the total cache space of the smallest frequency cached files that can be removed by p is larger than or equal to the cache space of file p. If it is, the cache replacement procedure can proceed. Otherwise, the file p cannot be inserted into the cache and the procedure stops.
3. Move the least frequency cached files to the free list repeatedly until the accumulated cache space of these files are larger than or equal to the cache space needed by file p. Then, insert the file p into the cache. The data of p may come from two sources: the free list and the disk. The server first checks whether file p's data are in the free list by checking the *file_data* field. If it is (i.e. the *file_data* field is not null), the server then removes p's file header from the list and inserts it into the frequency list. Otherwise, the server loads the data from the disk into a memory buffer, sets the *file_data* pointing to it, and inserts the file header into the frequency list. Notice that, the size of the buffer is, as we describe earlier, one of the total size of the file's MIME header and its content, or MAX_FILE_SIZE, depending on which is smaller.

5. Experimental results

To evaluate the performance of our memory cache and cache management policy, we perform the following three experiments. First, we measure the performance and the throughput of our memory cache. Second, the efficiency of MIME header caching is evaluated. Third, we compare the performance of our cache management policy with others used in WWW. Below we first describe the experimental methodology, and then, the experimental results are shown.

5.1. Experimental methodology

5.1.1. The environment

The experimental environment is shown in Fig. 6. The modified web server runs on top of the server host whose OS is FreeBSD Release 2.2.5 and has 80 Mbytes physical memory. On the client host, a popular web performance benchmark, WebStone [20], is used for the performance measurement of our web server. It is configured to

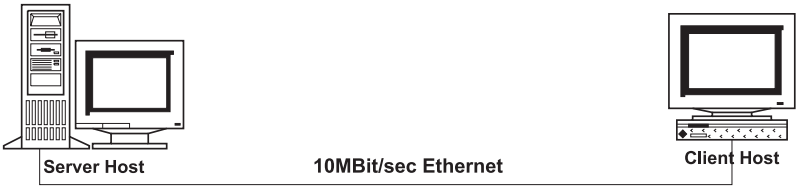


Fig. 6. The experimental environment.

fork 16 child processes that access the WWW documents on the server host simultaneously. The client and server hosts are isolated from other machines and connected directly with a 10 M bits/s Ethernet.

5.1.2. The workload

The workload in our experiments was gathered from the access log of the web site: www.bvi.-com.tw, which has 3022 files and the size is 281 Mbytes in total. We collected the access log that started at 7 July 1997 and stopped at 30 July 1997. During this period, there were 34 372 accesses to the server. We refer the workload as *current workload* since it represents the current WWW access pattern. Table 1 shows the distribution of access frequency according to the file sizes in the current workload.

5.2. Performance of the memory cache

In this section we present the performance of our memory cache in terms of byte hit ratio (BHR) and throughput. Fig. 7 shows the BHR which is computed using the following formula:

$$\text{BHR} = \frac{\text{number of bytes cached}}{\text{number of bytes requested}} \times 100\%.$$

Fig. 8 shows the throughput. In these experiments, we set the threshold, MAX_FILE_SIZE, as one-forth of the cache size. From these figures, we can see that with a 64 Mbytes cache, the BHR is about 70% and the throughput improvement is about 32.7%.

Table 1
The file sizes and their corresponding access frequencies in the current workload

File size (bytes)	Access frequency (times)	Access percentage (%)
0–1 K	3748	10.9
1–10 K	16 367	47.6
10–100 K	13 419	39.1
100 K–1 M	443	1.3
1–10 M	208	0.6
10 M	187	0.5

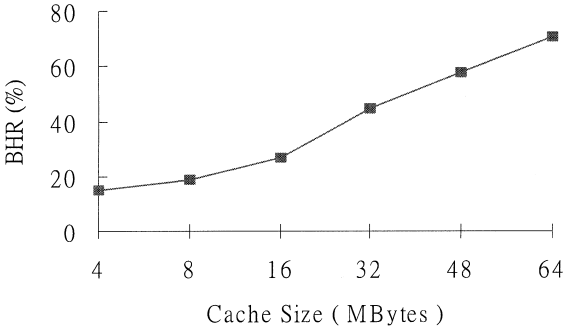


Fig. 7. The BHR of the memory cache.

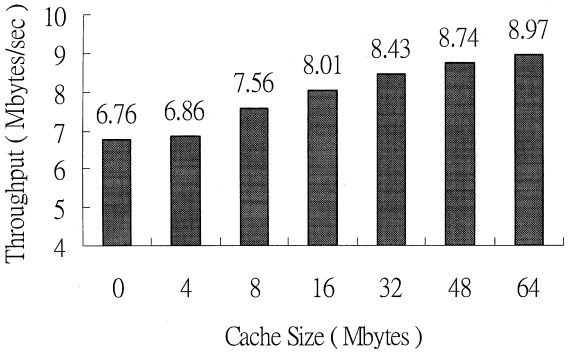


Fig. 8. Throughput of the memory cache.

5.3. Efficiency of MIME header caching

To evaluate the efficiency of MIME header caching, we compare the MIME header processing time under two conditions: with and without MIME header caching. Table 2 shows the results. Without MIME header caching, the MIME header processing time is used to *construct* the MIME header and send it to the client. With MIME header caching, the time is used to *lookup* the MIME header and send it to the client. From the table we can see that, MIME header caching reduces 44.6% of the MIME header processing time.

5.4. Comparison with other cache management policies

In this section we compare the performance of our cache management policy with others used in WWW. All these policies use LFU as their cache

Table 2
The performance improvement of MIME header caching

	MIME header processing time	Performance improvement
Without MIME header caching	387.687 μs	44.6%
With MIME header caching	387.687 μs	

Table 3
Descriptions of the three WWW cache policies

Policy	Description
Total	A file can be cached totally as long as its size is smaller than the cache size
Small	A file can be cached only if its size is smaller than a predefined threshold
Adaptive	This is our cache management policy. A file can be completely cached if its size is smaller than a threshold (i.e. the size of a chunk). Otherwise, only a chunk of it can be cached

replacement strategy, and they are described in Table 3.

We first compare the BHR of the three policies under the current workload. The performance result is shown in Fig. 9. From this figure, we can see that there are no obvious differences among these policies. This is because the three policies differ only when the files processed are large, but the current WWW traffic tends to access small files.

Due to the improvement of network speed, multimedia objects such as audio and video files, will become more and more popular. Since these files are large, the increasing popularity of them will affect the current WWW traffic and may result in performance differences among the three cache management policies. In order to see how the changing of WWW traffic impacts the performance of the three cache management policies, we choose the largest three files in the current workload, increase their access frequencies, and measure the BHRs (we refer to this workload as the *modified workload*). The result is shown in Fig. 10. From this figure, it is clear that our cache management policy noticeably outperforms others in BHR by 15–20%.

Furthermore, because we use memory cache and the memory size is limited, it is very possible that a multimedia file is larger than the total cache size. Thus, to investigate the performance of the three policies under this condition, we set the cache size as 32 Mbytes which is smaller than each of the three largest files and measure the BHRs. Fig. 11 shows the experimental result. From this Figure,

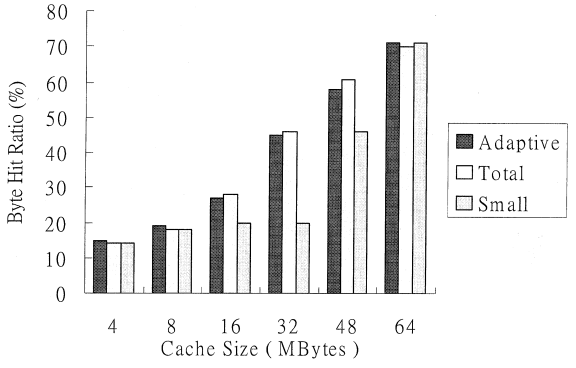


Fig. 9. BHR comparison of the three cache policies (current workload).

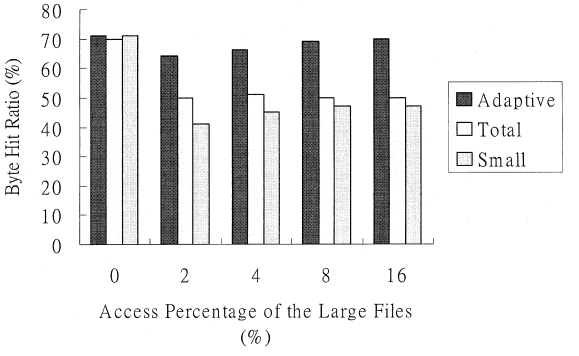


Fig. 10. BHR comparison of the three cache policies (modified workload).

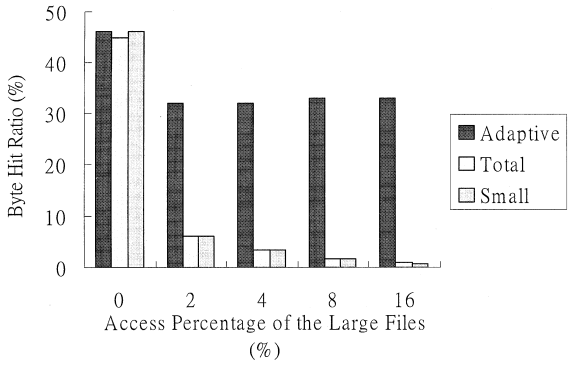


Fig. 11. BHR comparison of the three cache policies (when the total cache size is smaller than a requested file).

we can see that our policy is significantly better than others since their BHRs are less than 2% when the access frequency of the three largest files is larger than 8% while the BHR of our policy remains on 33%. Obviously, our policy outperforms others in this experiment. The reason is that, other policies never caches a file larger than the cache size even though the file is the most popular one. However, our policy does cache a portion of it and therefore results in a better BHR.

6. Conclusions and future works

6.1. Conclusions

Due to the large retrieval latency of WWW documents, many researches pay attention to the reduction of the latency by either reducing network delay or improving the throughput of WWW servers. In this paper, we use a memory cache to improve the throughput of a web server. By caching the most frequently accessed WWW documents in the server's address space, lots of WWW requests can be served without touching the file system on the server host and hence improves the server's throughput. We implement the memory cache by modifying an existing WWW server, *Thttpd*. From the experimental results, we have shown that the throughput improvement of our memory cache can achieve 32.7%

In addition to the implementation of the memory cache, we also propose a new web cache management policy named the adaptive-level policy, which takes the file size into account when caching a file. The policy has been proved, by our experiments, to be suitable for current web traffic. Besides, with the increasing popularity of multimedia files, our policy will outperform other policies that are currently used in WWW.

It should be noted that, although we chose a tiny web server (i.e. Thttpd) for our implementation, we believe that similar results will also apply to a practical server such as an Apache HTTP server. This is because that Thttpd differs with other popular servers only in that the latter put more efforts on managing and customizing themselves. For example, they provide APIs for users to implement modules to extend the original servers. This makes their servers more manageable and flexible. However, it has little influence on the throughput of servers.

6.2. Future works

In this paper, we add a memory cache in a web server's address space. This makes the operating system's (specifically, the file system's) buffer cache becomes a second-level cache. We will investigate the performance influence of this two-level caching in the future.

Moreover, Padmanabhan and Mogul [15] show that accesses to WWW documents are predictable. Therefore, prefetching documents into web server's address space may further improve the throughput. In the future, we will evaluate the performance benefit of prefetching and try to design an effective prefetching mechanism.

Although we cache web documents in a web server's memory space, accesses to cached data may sometimes involve disk reads, which will result in performance degradation. This is because most operating systems support virtual memory. All the memory held by a web server may be paged out to the disks at any time. In our experiments, we did not consider the impact of the virtual memory. Clearly, different page replacement policies will have different impacts on our web server. In the future, we will identify them on our web server.

The web server is currently a user-level process. It requires many user–kernel data copies when serving

requests and hence degrades the performance. In the future, we will try to move the web server into the kernel so as to reduce all user–kernel copies.

References

[1] M. Abrams, C.R. Standridge, G. Abdulla, S. Williams, E.A. Fox, Caching proxies: limitations and potentials, in: Proceedings of the Fourth International World Wide Web Conference, Boston, December 1995.

[2] M.F. Arlitt, C.L. Williamson, Web sever workload characterization: the search for invariants, in: Proceedings of the SIGMETRICS, Philadelphia, PA, April 1996.

[3] H. Braun, K. Claffy, Web traffic characterization: an assessment of the impact of caching documents from NCSA's web server, in: Proceedings of the Second World Wide Web Conference, Chicago, October 1994.

[4] A. Bestavros, R.L. Carter, M.E. Crovella, C.R. Cunha, A. Heddaya, S.A. Mirdad, Application-level document caching in the Internet, in: Proceedings of the Second International Workshop on Services in Distributed and Networked Environments (SDNE'95), Whistler, Canada, June 1995.

[5] C.M. Bowman, P.B. Danzig, D.R. Hardy, U. Manber, M.F. Schwartz, The Harvest information discovery and access system, in: Proceedings of the Second World Wide Web Conference, Chicago, October 1994.

[6] P.M.E. De Bra, R.D.J. Post, Information retrieval in the world-wide web: making client-based searching feasible, in: Proceedings of the First World Wide Web Conference, Geneva, Switzerland, May 1994.

[7] C.R. Cunha, A. Bestavros, M.E. Crovella, Characteristics of WWW client-based traces, Technical Report BU-CS-95-010, Computer Science Department, Boston University, July 1995.

[8] The CERN Proxy Cache, available at http://www.w3.org/pub/WWW/Daemon/User/Config/Caching.html.

[9] J. Gwertzman, M. Seltzer, The case for geographical push-caching, in: Proceedings of the HOTOS Conference, 1994.

[10] T.T. Kwan, R.E. McGrath, D.A. Reed, NCSA's world wide web server: design and performance, IEEE Computer 28 (11) (1995).

[11] T.T. Kwan, R.E. McGrath, D.A. Reed, User access patterns to NCSA's world wide web server, Technical Report UIUCDCS-R-95-1934, Department of Computer Science, University of Illinois, February 1995.

[12] E.P. Markatos, Main memory caching of web documents, in: Proceedings of the Fifth International World Wide Web Conference, May 1996.

[13] J.C. Mogul, Improving HTTP latency, in: Electronic Proceedings of the Second World Wide Web Conference: Mosaic and the Web, Chicago, IL, October 1994.

[14] J.C. Mogul, The case for persistent-connection HTTP, in: Proceedings of the ACM SIGCOMM'95 Conference on Communications, Architectures, and Protocols, Boston, August 1995.

[15] V.N. Padmanabhan, J.C. Mogul, Using predictive prefetching to improve world wide web latency, Computer Communication Review 26 (3) (1996).

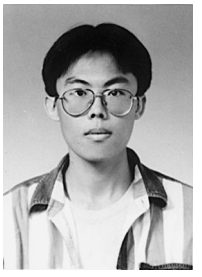
[16] S.E. Spero, Analysis of HTTP performance problems, available at http://metalab.unc.edu/mdma-release/http-prob.html, July 1994.

[17] Squid internet object cache, available at http://www.squid-cache.org/.

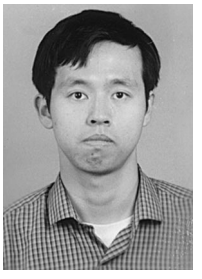
[18] Thttpd software, available at http://www.acme.com/software/thttpd/.

[19] S. Williams, M. Abrams, C.R. Standridge, G. Abdulla, E.A. Fox, Removal policies in network caches for world-wide web documents, in: Proceedings of the ACM SIGCOMM'96 Conference, Stanford University, August 1996.

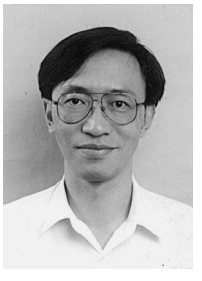
[20] WebStone benchmark, available at http://www.mindcraft.com/benchmarks/ webstone/.



Da-Wei Chang is a Ph.D. student in Computer and Information Science Department, National Chiao Tung University, Taiwan, ROC. He received his B.S. (1995) degree and Master (1997) degree both from the Computer and Information Science Department, National Chiao Tung University, Taiwan, ROC. His main interests are in operating systems, embedded systems, world-wide web, and Java.



Hao-Ren Ke was born on 29 June 1967 in Taipei, Taiwan, ROC. He received the B.S. degree in 1989 and his Ph.D. degree in 1993, both in Computer and Information Science, from National Chiao Tung University. Now he is an associate professor in Library, National Chiao Tung University, Taiwan, ROC. His main interests are in digital library, virtual reality, and knowledge discovery.



Ruei-Chuan Chang was born on 30 January 1958 in Keelung, Taiwan, ROC. He received his B.S. degree (1979), M.S. degree (1981), and Ph.D. degree (1984), all in Computer Engineering from National Chiao Tung University. Now he is a professor in Computer and Information Science Department, National Chiao Tung University, Taiwan, ROC. His main interests are in operating systems, wireless communication, embedded systems, world-wide web, and Java.