1. ***Workshop:*** Workshop on Computer Networks

2. ***Paper title:*** Improve Web Proxy Performance by Alleviating Disk I/O Overhead

3. ***Short abstract:*** In this paper, we first identify the performance bottleneck of Squid, and then propose an object management, called *UNIFIED*, which is a user-level technique for improving the performance of web proxy. In UNIFIED method, several techniques are used to improve the disk I/O performance. The proposed method had been implemented and embedded into Squid-2.3 without modifying the existing OS and file system. Experimental results show that, compared to Squid-2.4, our method can dramatically improve the proxy performance by reducing the overhead associated with disk I/O.

4. ***Authors:***

Yen-Jen Chang
Dept. of Computer Science and Information Engineering
National Taiwan University
Taipei, Taiwan
E-mail: d88017@csie.ntu.edu.tw
Tel.: 886-02-23625336-111

Feipei Lai
Dept. of Computer Science and Information Engineering &
Dept. of Electrical Engineering
National Taiwan University
Taipei, Taiwan
E-mail: flai@cc.ee.ntu.edu.tw
FAX: 886-02-2363-7204; Tel.: 886-02-2391-4116

5. ***Contact author:*** Yen-Jen Chang, d88017@csie.ntu.edu.tw

6. ***Keywords:*** Web proxy, Squid, Object management, disk I/O, Polygraph 2.5.4, Polymix-3

Improve Web Proxy Performance by Alleviating Disk I/O Overhead

*Yen-Jen Chang† and Feipei Lai†**

†Dept. of Computer Science and Information Engineering, NTU

**Dept. of Electrical Engineering, NTU*

Abstract

The dramatic increase of WWW traffic on the Internet has led to the wide use of web proxy. The web proxies are dedicated to caching and delivering web content. They can be used to improve security, save network bandwidth and reduce network latency. However, as the network bandwidth increased, the general-purpose file system is rapidly becoming the performance bottleneck of web proxies. In this paper, we first identify the performance bottleneck of Squid, and then propose an object management, called *UNIFIED*, which is a user-level technique for improving the performance of web proxy. In UNIFIED method, several techniques are used to improve the disk I/O performance. The proposed method had been implemented and embedded into Squid-2.3 without modifying the existing OS and file system. Instead of the traditional trace-driven simulation, we apply Polygraph 2.5.4 with Polymix-3 workload to evaluate our system realistically. To investigate how the proxy performance depends on the equipped disk, we offer two sets of test machines. One is equipped with one IDE disk and the other is equipped with five SCSI disks. Experimental results show that, in both tests, our method can dramatically improve the proxy performance by reducing the overhead associated with disk I/O.

1. Introduction

For reducing both network latency and traffic on the Internet, the web cache servers (proxies) are being increasingly used. The function of the web proxy is to serve the client’s requests by looking up the equipped cache that stores the previous web data, and only contacts the web servers in case of proxy miss. Unless the main memory becomes cheap enough, the web proxies always employ the disk to cache web data. Apart from the network latency, disk I/O is a major performance bottleneck of the web proxy. This conclusion can be found in lots of researches [1, 2, 3, 4, 5, 6, 7]. In some specific environment, Mogul et al. [8] even suggested to run the web proxy in non-caching mode because the disk I/O overhead is higher than the latency improvement obtained from the use of web proxy.

All public web proxies (CERN, Harvest and Squid) use the UNIX file system (UFS) for portability. Because the UFS is a general-purpose file system that is designed for the workstation workload and is not optimized for the workload of web proxy, the use of UFS would degrade the proxy performance. By contrast, some commercial web proxies are developed with a special operating system (or file system) that is optimized for disk I/O [9, 10, 11]. These vendors report their solutions can improve the proxy performance by many orders of magnitude. However, the major disadvantages of these commercial solutions are expensive and non-portable.

In this paper, we concentrate on the methods that can be implemented at user-level (application-level) without modifying the standard UFS. To alleviate the overhead associated with using UFS, we propose an object management, called *UNIFIED*, that stores all objects in a single file. Because all objects are stored in a single file, all *open*, *close* and *unlink* (*delete)* system calls can be eliminated completely. Besides, several techniques are used to improve the disk I/O performance in the proposed UNIFIED method. First, we develop a precise and dynamic space allocation algorithm that can satisfy all space requests for various sizes. It costs only *O(lg n)* time that is more efficient than the use of traditional linear search method (*O(n)*). Second, instead of *multi-read/write*, the *single-read/write* scheme is used to further reduce the number of read and write system calls. Third, we use *cluster write* to further improve write performance.

We had implemented the proposed UNIFIED method, and embedded it into Squid-2.3. Instead of traditional trace-driven simulation, we use Polygraph 2.5.4 with Polymix-3 workload, which is an industry-wide benchmark for proxy performance, to evaluate our system realistically. Compared to Squid-2.4 with DISKD scheme, the experimental results show that our method can dramatically improve the average response time and throughput of the web proxy by reducing the disk I/O overheads.

The rest of the paper is organized as follows. In Section 2, we describe the evaluation tools and test environment used in this paper. Section 3 identifies the important characteristics of Squid. Next, in Section 4 we detail the proposed UNIFIED object management, and provide the comparison between our method and previous work. Experimental results are given in Section 5, and Section 6 offers some conclusions.

2. Methodology

Most of researchers use trace-driven simulation to evaluate the web proxy and make performance claims regarding their methods. However, such claims were essentially meaningless due to the lack of widely available tools and standard workloads. They only concentrate attention on some specific component in which they are interested, but not the overall performance of web proxy. They ignore the overhead of running web proxy in the real-world, for example, the interactions of web proxy and operating system, the request load on client side, network delay and network performance, etc.

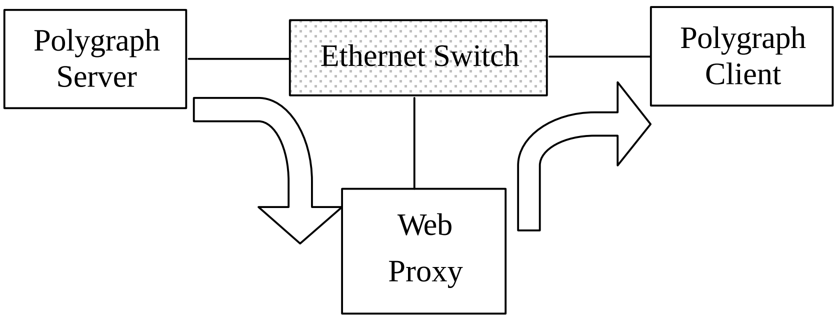


Figure 1: Test environment. (The arrows show the direction of data flow.)

Instead of trace-driven simulation, in this paper, we use *Web Polygraph* [12] to evaluate the proxy performance. It is a high performance tool for generating web traffic and measuring proxy performance. In Web Polygraph, there are many features absent from the traditional trace-driven simulation method. As shown in Fig. 1, our test environment consists of two polygraph machines (Polygraph Server and Polygraph Client), a web proxy, and a network to tie them together. Polygraph Server generates HTTP responses for the requests issued from Polygraph Client in case of proxy miss. Depending on the configuration, Polygraph Client can emulate the end-user surfing the web with a browser, generate hundreds of HTTP requests per second and maintain thousands of concurrent connections for hours. The detailed information of Web Polygraph can be found in [12].

3. File Management used in Squid

Because Squid [13] is the most popular web proxy and has been studied widely, we destine it to be our target proxy. As well known, the major feature of storage architecture used in Squid is that each URL (object) is stored on a separate file. Squid has many features designed to improve I/O performance. Unlike traditional servers, Squid handles all requests in a single, non-blocking, I/O-driven process. Over the years, the poor performance of UFS has been shown a vital bottleneck of Squid [13]. On average, Squid-2.3 only serves about 30-50 requests per second due to the use of UFS.

3.1 Analysis of Squid

Fig. 2 shows the number of UFS operations executed in Squid. The total request numbers have increased with the request rates. They are around 4.3 million, 6.0 million, 7.8 million and 9.5 million in the order of 50 req/sec, 100 req/sec, 150 req/sec and 200 req/sec. We observed the proportion of *open/close* and *unlink (delete)* to the total requests number are roughly 76% and 40%. Besides, *write* and *read* take the more notable fraction of UFS operations executed in Squid. Especially *write* operation dominates the traffic sent to the disk subsystem.

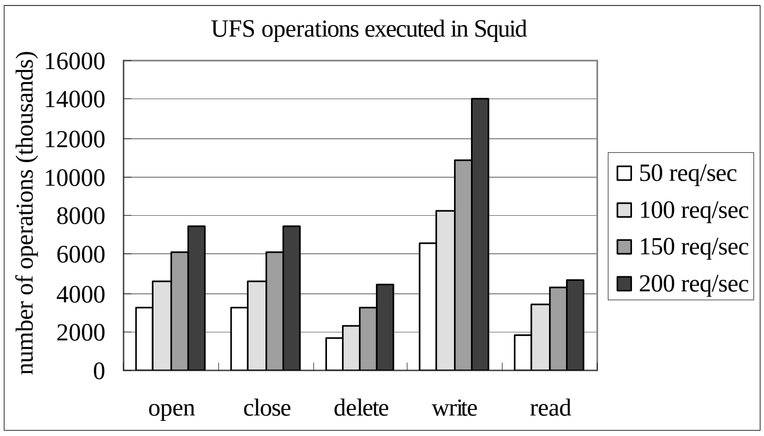


Figure 2: UFS operations executed in Squid.

- ***Multiple Read/Write (Multi-R/W) Scheme***

By tracking the Squid source code, we observe that Squid uses multiple *write* operations to perform caching one object on disk. The default size is 8192 bytes for each *write* operation. The advantage of *multi-write* scheme is that Squid does not require to allocate additional memory to store the coming object before it is written to the disk, but such *multi-write* scheme would result in more *write* operations for caching the objects. Similarly, Squid uses *multi-read* scheme to perform reading the cached object and reply it to client. The default size is 4096 bytes for each *read* operation.

From the request flow, no matter whether the request is a hit or not, at least one disk operation (read or write) would be executed. The standard UFS disk operations used in Squid are synchronous that means the control is not returned to Squid process until I/O completion. Therefore, Squid spends a lot of time on disk operations. This is the reason why Squid has poor performance.

- ***DISKD***

Compared to Squid-2.3 that handles all disk I/O operations in a single process, in Squid-2.4 or later, *DISKD* was introduced to improve disk I/O performance. The basic idea of DISKD is that each cache directory has its own *diskd* child process. One *diskd* process performs all disk I/O operations for one cache directory. Of course, the performance of DISKD would be further enhanced if the cache directories were located on different disks. At the third IRCache Bake-Off, Squid-2.4 with DISKD got 160 req/sec [14].

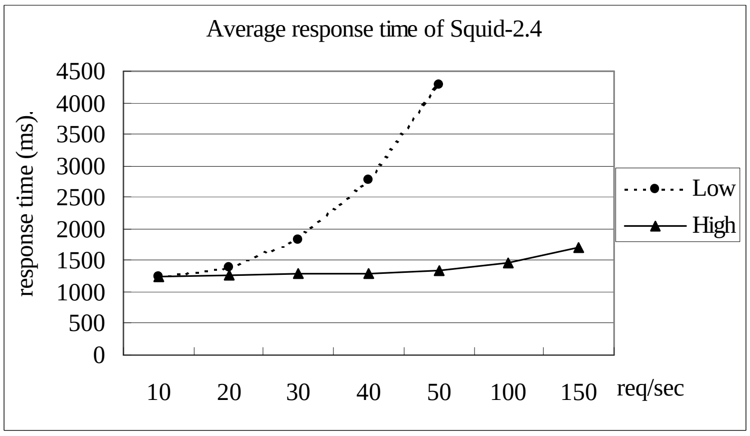


Figure 3: Average response time of Squid-2.4.

Fig. 3 shows the average response time of Squid-2.4. In this experiment, Squid-2.4 can afford to deliver 50-60 req/sec in low-end test (equipped with one IDE disk), and 150-200 req/sec in high-end test (equipped with five SCSI disks). Due to the proxy workload characteristics are different from the traditional UNIX workload, for web proxy, UFS has a number of features that are not necessary, or could be simplified. Even Squid-2.4 was equipped with the competent disk in the high-end test; it is much less than the performance of other commercial products, e.g., 500 req/sec for IBM-220-2 and 780 req/sec for iMimic-1300 [14].

4. UNIFIED Object Management

As indicated previously, the disk I/O operations have been shown to be the major performance bottleneck of a web proxy. To alleviate the disk I/O overhead, we propose an object management called *UNIFIED* that stores all objects in a single file, which we refer to as the UNIFIED file. This idea is very simple and straightforward, but it is hard to be implemented due to the difficulty in space allocation. Despite the apparent performance gain from storing all objects in a single file, however, the hidden cost of this approach is potentially more than its benefits due to the space management.

4.1 Bit Vector

The UNIFIED file used in our method can be regard as a largely and continuously logical space. First, this continuously logical space must be partitioned into fixed-sized blocks called *chunks*. We use *bit vector* (or *bit map*) to indicate whether the chunks are available (free) or not. Each chunk is represented by one bit. If the chunk is available, the bit is “0”; otherwise, the bit is “1”. Note that, for fast manipulation, the entire bit vector should be kept in main memory.

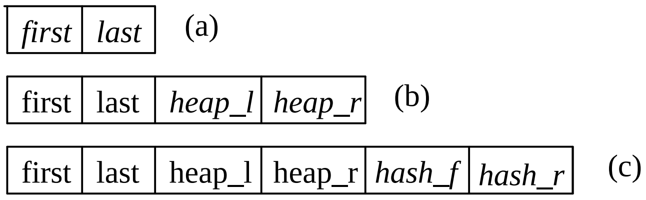

first | last (a)
first | last | heap_l | heap_r (b)
first | last | heap_l | heap_r | hash_f | hash_r (c)

Figure 4: The node structure for a free piece.

There must be a lot of free pieces in the UNIFIED file after repeated removing and storing the objects. As shown in Fig. 4(a), one node is used to track one free piece, in which *first* and *last* fields are used to locate the first free chunk number and the last free chunk number respectively. The size of a free piece is *last-first+1*.

4.2 Continuous Allocation & Single Read/Write (Single-R/W) Scheme

In our method, the first problem is how to allocate space to those objects such that the UNIFIED file is utilized effectively and the objects can be accessed quickly. There are three methods used in space allocation: contiguous, linked, and indexed. Except for the contiguous allocation, the other two would scatter the cached object all over the UNIFIED file, and then slowdown the object access. Thus, we decide to use the contiguous allocation in our method

From Section 3, we know that the *multi-R/W* scheme used in Squid is beneficial for the machines equipped with less memory, but it degrades the proxy performance due to many system calls. By contrast, we use *single read/write* (*single-R/W*) scheme which is straightforward and more efficient due to the use of continuous allocation. The *write* operation is executed once to write the entire object data to the disk when the proxy finishes reading the reply from the origin server. Similarly, before submitting the cached data to the request client, the *read* operation is executed once to read the entire object data from the disk in the case of proxy hit. Thus, the number of both *write* and *read* operations could be reduced dramatically. The disadvantage of the *single-R/W* scheme is that the more memory would be consumed in storing the object data temporarily, but these allocated memory would be released after using. It is reasonable to use appropriate memory to improve the disk I/O performance as well as the proxy performance.

4.3 Cluster Write

One difficulty in contiguous allocation is finding a sufficient space for a new coming object. This procedure can be reduced to the general *dynamic storage-allocation* problem, which is how to satisfy a request of size *n* from a set of free space. Instead of writing a new coming object in arbitrary free piece, we select the largest free piece to accommodate these new coming objects, i.e., worst-fit strategy. By continually appending objects until the largest free piece is exhausted, we can cluster the *write* operations and then improve the write performance.

Our cluster write scheme always performs the longest sequential write operations due to the use of worst-fit strategy. The concept of cluster write is inspired from the log-structured file systems [15]. Although the *cluster write* used in logical file space makes no effect to layout data on disk, its effect on improving write performance had been certified in [5]. The worst-fit is easy to be implemented with the maximum-heap that is a priority queue to track the element with the largest key. The size of free piece can be used as key to build the maximum-heap. The node structure shown in Fig. 4(a) should be augmented with two additional fields to adapt for heap structure. Fig. 4(b) shows the modified node. Field *heap_l* and *heap_r* point to left and right subtrees respectively.

4.4 Precise Space Management

When an object was deleted, we must merge it with the neighbor free chunks to track each free piece precisely. For example, in Fig. 5, the neighbor chunks of object *x* are free. We can merge these free pieces from chunk number 5 to 15 to form a larger contiguous free piece after object *x* was removed.

● Merging

By testing the bit vector, it is easy to determine whether a chunk is free or not. The main difficulty in merging is how to determine the size of neighbor free piece. The straightforward method is sequential and exhausted testing from the first free chunk which is next to the removed object to the first used chunk. The time complexity is *O(n)*. This approach is efficient in case of free pieces with small size, but it is inadequate for those free pieces with extremely large size.

Here, we introduce another efficient merging method, called *front-rear hashing*, in which two hash tables are used: one is *front hash table* and the other is *rear hash table*. Each free piece must be hashed into the front hash table with the first free chunk number, and hashed into the rear hash table with the last free chunk number. The main idea of front-rear hashing method is that we can retrieve a free piece by looking up the front hash table (with the first free chunk number) or by looking up the rear hash table (with the last free chunk number). The node structure shown in Fig. 4(b) must be augmented with two additional fields to store the linked list information used in both hash tables, as shown in Fig. 4(c). In Fig. 5, we use an example to illustrate the merging flow and describe it as follows:

1. The object *x* was removed from the web proxy, and then the free piece (8, 11) was released. The bit vector corresponding to this object *x* has to be reset to "0". This step costs only *O(1)* time.
2. ***Left merging:*** First, the chunk next to the first chunk of object *x* (i.e., chunk 7) must be tested. If chunk 7 is in use, we can skip this step. Otherwise, chunk 7 must be the last chunk of some free piece, and then we look up the rear hash table with the chunk number 7. In this example, we can find the free piece (5, 7) in the rear hash table. Note that, to keep the data consistency between these two hash tables, if the free piece (5, 7) was removed from the rear hash table, its corresponding node hashed with chunk number 5 in the front hash table must be removed at the same time. Finally, after removing the free piece (5, 7) from the maximum-heap, the free piece (8, 11) can be merged with (5, 7) to form a larger new free piece (5, 11). In this step, under the use of adequate hash function, the expected time to search for a node in the hash table

is *O(1)*, and the time to remove a node from the maximum-heap is *O(lg n)*. Thus, the total cost is roughly *2×O(1)+O(lg n)=O(lg n)* in the left merging.

3. ***Right merging:*** After left merging, the chunk next to the last chunk of object *x* (i.e., chunk 12 in this example) should be tested, and then the following operation is the same as the left merging. The only difference is that the front hash table should be looked up with the chunk number 12 in the right merging. In this example, we can find the free piece (12, 15) in the front hash table, and merge it with the new one obtained in left merging to form a new free piece (5, 15). The time cost is the same as that in Step 2.

4. After left and right merging, maybe we can receive a larger free piece than the removed object. We have to hash this new free piece to both hash tables with the first chunk number and the last chunk number respectively, and insert it into the maximum-heap. This step also costs *2×O(1)+ O(lg n)=O(lg n)* time, in which *O(1)* is for hashing operation and *O(lg n)* is for heap insertion operation.

In summary, the running time of the proposed object management is roughly *O(lg n)* time complexity. It is more efficient than the use of sequential test in merging flow (i.e., *O(n)*).

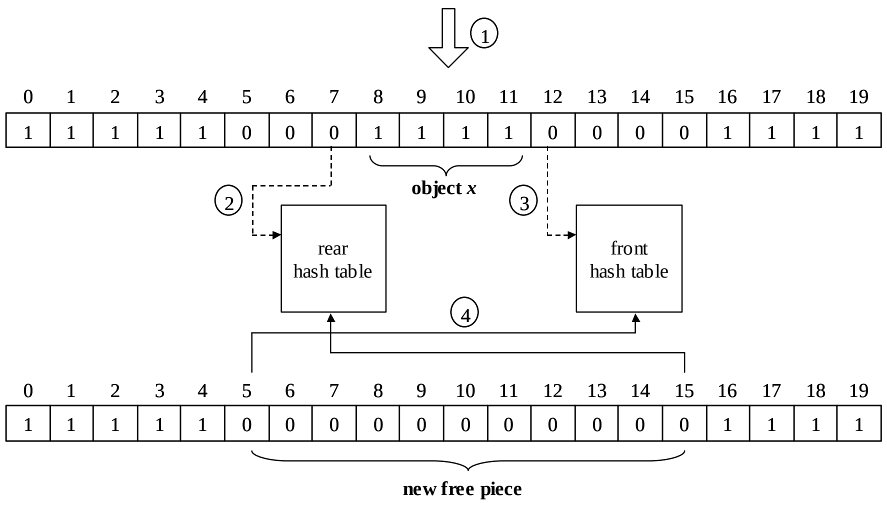


Figure 5: *Front-rear hashing* method. Dashed-line represents the search for a node in hash tables, and solid-line represents the insertion into hash tables.

4.5 Recovery

To make our system robust, we must append sufficient information to object metadata. For example, if a piece *(s, t)* was allocated to the object *x*, the chunk numbers *s* and *t* must be appended to the metadata of object *x*, and then write object and metadata to disk. In case of crash, after the system reboots, the bit vector used in our method can be recovered by scanning the entire UNIFIED file. At the same time, the maximum-heap, and both hash tables are also rebuilt. Clearly, there is no overhead to prevent system crash in our method.

4.6 Comparison

Unlike *buddy* system, which suffers from internal fragmentation, our proposed front-rear hashing can manage space dynamically and precisely with a competitive performance *O(lg n)*. Without internal fragmentation, the entire space can be utilized effectively in our method.

The idea behind our method is very simple, and the similar object-packing techniques have been proposed in [3, 4, 5, 7]. Unlike these object-packing techniques, which only pack a part of objects into one file, our method packs all objects into one file with a dynamic and precise space management. Thus, all *open/close* and *delete* system calls can be eliminated completely. In addition, the use of both *single R/W* scheme and *cluster write* further improve the proxy performance by reducing the number of *read* and *write* operations and optimizing the *write* operations, they are absent from the previous work.

5. Experimental Results

We had implemented the proposed UNIFIED object management and embedded it into Squid-2.3. For verifying the efficiency of our method, we destine the competitive Squid-2.4, which performance is much better than Squid-2.3, as the comparison target.

To measure how the proxy performance depends on the equipped hard disk, we offer two suits of test machines. One is low-end test and the other is high-end test. (1) In low-end test, the web proxy is tested on the x86 PC with a 600 MHz Pentium III processor running FreeBSD 4.1. The proxy machine is equipped with 256MB memory and a 20GB Seagate 7,200 RPM Ultra ATA/100 IDE (ST320414A) disk. The sizes of memory cache and disk cache are configured as 100MB and 10GB respectively. (2) In high-end test, the web proxy is tested on the x86 PC with two 600 MHz Pentium III processors running FreeBSD 4.1. The proxy machine is equipped with 1GB memory and five 9GB IBM 10,000 RPM Ultra2 SCSI disks. The sizes of memory cache and disk cache are configured as 300MB and 10GB respectively. In our system, these five disks are configured in RAID0 mode, and in Squid-2.4, we configure that each cache directory is 2GB size on each disk.

In the following discussions, we use the number of disk I/O operations, average hit time, average miss time, average response time and peak throughput as the criteria to compare Squid-2.4 to our system (i.e., Squid-2.3 embedded with our proposed UNIFIED object management). In this paper, we concentrate on the performance of disk I/O. We do not modify the replacement strategy, and we use the same LRU replacement as Squid. Thus, both Squid-2.4 and our system have the same hit ratio, so we do not consider the hit ratio here.

5.1 Disk I/O operations

Table 1 shows the number of *open*, *close* and *delete* operations executed in Squid for various configurations of request rate. We measure the affordable request load of Squid-2.4 tested in low-end and high-end test are 50~60 req/sec and 170~180 req/sec respectively. In our system, because all objects are stored in a single file, all *open*, *close* and *delete* system calls can be eliminated completely.

Table 1: The number of *open, close* and *delete* operations executed in Squid. The affordable request load of Squid-2.4 in low-end test and high-end test are 50~60 req/sec and 170~180 req/sec, as shown in gray row.

(a) Low-end test.

	open	close	delete
10 req/sec	1,830,968	1,830,967	1,002,896
20 req/sec	2,108,715	2,108,715	1,182,368
30 req/sec	2,400,413	2,400,413	1,204,749
40 req/sec	2,894,680	2,894,680	1,409,723
50 req/sec	3,378,287	3,378,287	1,659,489
100 req/sec	4,726,937	4,726,937	2,206,019
120 req/sec	5,224,545	5,224,545	2,698,164

(b) High-end test.

	open	close	delete
50 req/sec	3,220,714	3,220,714	1,691,382
100 req/sec	4,627,888	4,627,888	2,281,725
150 req/sec	6,080,883	6,080,883	3,235,120
170 req/sec	6,683,492	6,683,492	3,697,432
200 req/sec	7,477,177	7,477,177	4,404,863
250 req/sec	8,800,143	8,800,143	5,718,264

Table 2 shows how *read* and *write* operations can be reduced in our system. The key observation is that with the use of *single-read/write* scheme, our system further reduces roughly 50% of *read* operations and 63% of *write* operations for various request rates in both tests. As expected, from the data shown in Table 1 and Table 2, the proposed UNIFIED object management can dramatically reduce the number of disk I/O operations.

Table 2: The number of *read* and *write* operations executed in Squid-2.4 and our system.

(a) Low-end test.

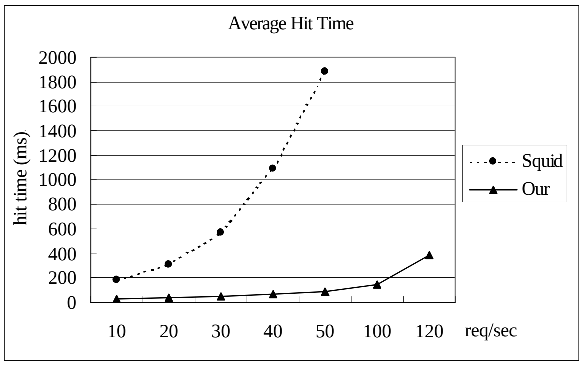
	read		write	
	Squid-2.4	Our	Squid-2.4	Our
10 req/sec	534,704	255,763	4,318,921	1,575,083
20 req/sec	976,325	468,838	4,588,152	1,691,836
30 req/sec	1,208,030	618,340	4,877,634	1,781,951
40 req/sec	1,588,713	808,123	5,281,763	1,987,364
50 req/sec	1,931,362	996,848	6,520,977	2,381,317
100 req/sec	3,454,337	1,796,287	8,027,956	2,930,528
120 req/sec	3,806,434	1,906,340	9,781,763	3,573,648

(b) High-end test.

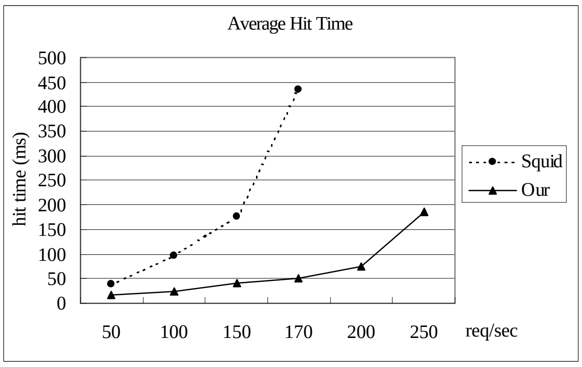
	read		write	
	Squid-2.4	Our	Squid-2.4	Our
50 req/sec	1,844,889	809,374	6,608,094	2,411,219
100 req/sec	3,379,529	1,622,303	8,228,432	3,005,464
150 req/sec	4,305,185	2,124,848	10,814,209	3,955,914
170 req/sec	4,514,545	2,193,743	12,871,436	4,638,126
200 req/sec	4,705,627	2,350,592	14,005,715	5,126,464
250 req/sec	5,683,479	2,748,622	19,635,487	7,256,498

5.2 Average Hit Time

The hit time measured by Polygraph client is the elapsed time from the request submission to the receipt of reply data, where this request is a hit in web proxy. Because the *read* operation is on the critical path in the case of cache hit, this criterion can be used to characterize the *read* performance of web proxy. Figure 6 shows the average hit time of Squid-2.4 and our system in both test environments. From this figure, we summarize the most important aspects.



(a) Low-end test.



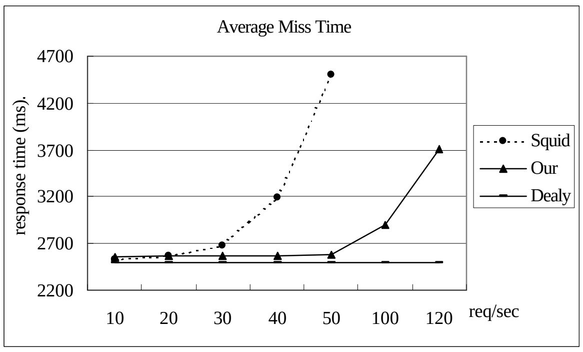
(b) High-end test.

Figure 6: The average hit time of Squid-2.4 and our system in both tests.

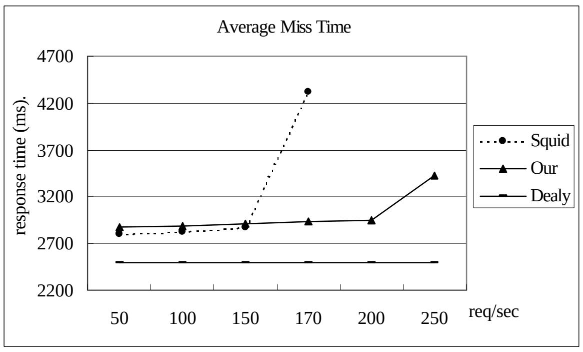
First, the average hit time measured in low-end test is much higher than that measured in high-end test. This is because the performance of disk used in low-end test (one IDE disk) is much poorer than the performance of disk used in high-end test (five SCSI disks). Second, in all cases depicted in this figure, the hit time goes up with the request rate. Once the web proxy gets busy, disk I/O can become the bottleneck. The hit time increased sharply when the request load approaches the delivering limit of web proxy. If the hit time is too high such that the entire test cannot be finished, that implies the web proxy cannot afford to deliver such request load. Third, because the *single-read* scheme shortens the elapsed time by reading data from the allocated buffer instead of the disk, our proposed object management can improve average hit time dramatically. The speedup of average hit time is defined as $speedup=HT_{Squid}/HT_{Our}$, where HT is hit time. In the low-end test, the speedup is from 6 to 22 times, and in the high-end test, the speedup is from 2 to 9 times. The key observation is that the proposed method is effective in both tests, low-end test especially.

5.3 Average Miss Time

The miss time measured by Polygraph client is the elapsed time from the request submission to the receipt of reply data, where this request is a miss in web proxy. To simulate real-world conditions, PolyMix-3 introduces an artificial delay on the server side. The server delays are normally distributed with a 2.5 sec mean and 1 sec deviation [12]. These delays play a crucial role in creating a reasonable number of concurrent "sessions" in the cache. Because the *write* operation is on the critical path in the case of cache miss, this criterion can be used to characterize the *write* performance of web proxy.



(a) Low-end test.



(b) High-end test.

Figure 7: The average miss time of Squid-2.4 and our system. The default network delay is 2.5 sec.

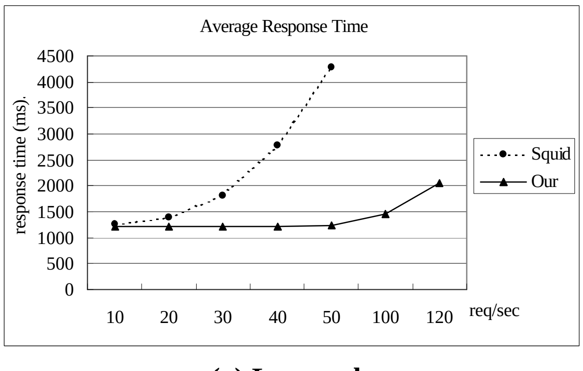
The impact of request load on the average miss time is shown in Figure 7. An interesting variation in the degree of miss time reduction is observed among the data in this figure. The average miss time of our system is not better than that of Squid-2.4 in case of proxy with light load (e.g., 10~20 req/sec in Fig. 7(a), 50~150 req/sec in Fig. 7(b)). This is because traditional UNIX systems use the *block buffer cache* to minimize disk I/O operations by caching recently accessed disk blocks in memory. To eliminate most *write* operations and also to reorder the *writes* in a way that optimizes disk performance, the UNIX buffer cache is primarily *write-back* (or *write-behind*). The *write* operation is finished when OS copies the write data from user buffer to block buffer cache, and then these data would be written to the disk at a later time. This write-back scheme is similar to the copy function used in our *single-write* scheme, in which the data retrieved from the server would be stored in the allocated buffer temporally. Thus, when the proxy is in light load, the use of *single-write* scheme does not result in the kind of time reduction that is realized for the *single-read* scheme used in case of cache hit. The effect of *single-write* scheme in reducing *write* system calls would be pronounced when the proxy is in heavy load. This phenomenon can be observed from 30~50 req/sec in Fig. 7(a) and 150~170 req/sec in Fig. 7(b).

5.4 Average Response Time

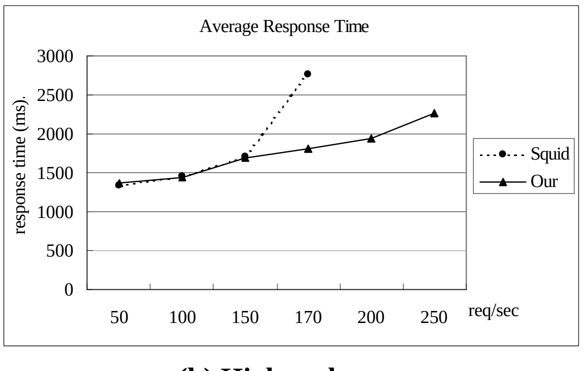
The average response time measured by Polygraph is defined as follows:

Average response time=(average hit time×HR)+(average miss time×MR),

where *HR* is the hit ratio and *MR* is the miss ratio. Figure 8 shows the average response time of both Squid-2.4 and our system. We observe the improvement of response time depends on the request load. In case of light request load, our proposed UNIFIED object management does not exhibit the effect on improving the response time. By contrast, in case of heavy request load, our method is beneficial to alleviate the disk I/O overhead. The experimental results show that the affordable request load of Squid-2.4 are about 50~60 req/sec and 170~180 req/sec in low-end and high-end test respectively. In low-end test, our system results in roughly 3.5 times speedup of response time under 50 req/sec affordable request load. Similarly, in high-end test, the speedup of response time is about 1.5 times under 170 req/sec affordable request load.



(a) Low-end test.



(b) High-end test.

Figure 8: The average response time of Squid-2.4 and our system.

5.5 Peak Throughput

The throughput of web proxy is defined as the number of requests serviced per second. Table 3 shows the impact of UNIFIED method on peak throughput. As expected, in conventional Squid, throughput goes up with the efficiency of the equipped disks. Even though the DISKD was used to improve disk I/O performance, Squid-2.4 only affords to deliver 50~60 req/sec and 170~180 req/sec in low-end and high-end test respectively. Compared to Squid-2.4, our system can afford to deliver 120~130 req/sec and 250~300 req/sec in low-end and high-end test respectively. An increasing of 127% and 57% in peak throughput for two tests can be achieved by using the proposed UNIFIED object management.

Table 3: The peak throughput of both Squid-2.4 and our system in two tests.

	low-end test	high-end test
Squid-2.4	**50~60**	**170~180**
Our	**120~130**	**250~300**

6. Conclusions

As the tremendous growth of *WWW* has significantly contributed to the network traffic on the Internet, the demand for a web proxy with high performance is increasing. In this paper, we destine Squid as our target web proxy and identify the performance bottleneck by analyzing characteristics of workload obtained from the test of Squid. We then propose an object management, called *UNIFIED*, that stores all objects in a single file. For portability, the proposed method can be implemented at user-level (application-level) without modifying the standard UNIX system and UFS.

Except all *open*, *close* and *unlink (delete)* system calls can be eliminated completely, the UNIFIED object management employs several techniques to boost proxy performance. The *single-read/write* scheme and *cluster write* are introduced to further reduce the number of read/write operations and improve the write performance. We also develop a dynamic space management, named *front-rear hashing*, which can manage space dynamically and precisely with a competitive performance *O(lg n)*. Not only used in web proxy, we may apply the front-rear hashing to other appliances that need space management.

We compare our method to previous work and indicate the difference among these methods. The proposed method was implemented and embedded into Squid-2.3. Instead of traditional trace-driven simulation, we use an industry-wide benchmark tool, i.e., Polygraph 2.5.4 with Polymix-3 workload, to evaluate our system realistically. We offer two suits of test machines. One is low-end test, in which the web proxy is equipped with one IDE disk, and the other is high-end test, in which the web proxy is equipped with five SCSI disks. The experimental results show that our proposed UNIFIED object management can reduce roughly 50% of *read* operations and 63% of *write* operations for various request rate while all *open*, *close* and *delete* operations are eliminated completely. In low-end test, our system results in roughly 3.5 times speedup of response time under 50 req/sec affordable request load and an increasing of 127% in peak throughput. Similarly, in high-end test, the speedup of response time is about 1.5 times under 170 req/sec affordable request load, and an increasing of 57% in peak throughput would be achieved. Because the proxy performance is severely constrained by disk efficiency, our method is more beneficial for the web proxy equipped with the inefficient disk.

References

[1] J. Almeida and Pei Cao, "Measuring Proxy Performance with the Wisconsin Proxy Benchmark," TR 1373, Computer Science Department, University of Wisconsin-Madison, April 13, 1998.

[2] A. Rousskov and V. Soloviev, "A Performance Study of the Squid Proxy on HTTP/1.0," World-Wide Web Journal, Special Edition on WWW Characterization and Performance Evaluation, 1999.

[3] S. L. Fritchie, "The Cyclic News Filesystem: Getting INN To Do More With Less," in Proc. of the 1997 Systems Administration Conference, 1997, pp. 99-111.

[4] C. Maltzahn, K. J. Richardson and D. Grunwald, "Reducing the Disk I/O of Web Proxy Server Caches," In Proc. of the 1999 USENIX Annual Technical Conference, June 1999, pp. 225-238.

[5] E. P. Markatos and M. G. Katevenis, "Secondary Storage Management for Web Proxies," In Proc. of the 2nd USENIX Symp. on Internet Technologies and Systems, Oct. 1999, pp. 93-114.

[6] E. Shriver, E. Gabber and L. Huang, "Storage Management for Web Proxies," In Proc. of the 2001 USENIX Annual Technical Conference, June 2001, pp. 203-216.

[7] K. C. Chinen, E. Kawai, Y. Kadobayashi and S. Yamaguchi, "User Level Techniques for Improvement of Disk I/O in WWW Caching," in Proc. of International Conf. on System, Man and Cybernetics, vol. 5, 2001, pp. 3033.

[8] J. C. Mogul, "Speedier Squid: A Case Study of an Internet Server Performance Problem," The USENIX Association Magazine, 24(1), 1999, pp. 50-58.

[9] P. Danzig, "NetCache Architecture and Deployment," Network Appliance TR-3029, Santa Clara, California, 1998.

[10] CacheFlow, Inc. High-performance Web Caching White Paper.

[11] IMimic Networking, Inc., "The iMimic DataReactor Architectural Overview," http://www.imimic.com /documents/WP-ArchitecturalOverview.pdf

[12] Polyteam, "Polygraph User Manual," http://polygraph.ircache.net/UserManual/

[13] D. Wessels, "Squid Web Proxy Cache," http://www.squid-cache.org/

[14] The Third Cache-Off Official Report, http://www.measurement-factory.com /results/public/cacheoff/N03 /report.by-alph.html

[15] M. Seltzer, M. K. McKusick, K. Bostic and C. Staelin, "An Implementation of a Log-Structured File System for UNIX," in Proc. of the 1995 Winter USENIX Technical Conference, January 1995.