(1) Name of workshop: Workshop on Computer Networks

(2) Title: ESP: An Efficient Storage Management Policy for Proxies

(3) Abstract

With the enormous growth of the WWW, the Internet has seen a large volume of traffic, which introduces severe congestion and worsens our surfing experience. Proxy cache is a common solution to this problem. It reduces the traffic on the Internet and the response time for Web accessing. However with the potential growth of the WWW, the proxy is overburdened and that in turn increases the response time even more.

In this paper we propose an efficient mechanism, ESP (Efficient Storage management policy for Proxies), to reduce the file operation and disk seek time in order to improve the performance of proxies. We implement ESP by modifying the source code of Squid. Squid with ESP shows a 100% performance improvement to the original Squid design.

.

(4) Name: Lichung Chiang (江立中) and Wen-Shyen E. Chen (陳文賢)

Current affiliation: Institute of Computer Science National Chung-Hsing University

Postal address: 台中市南區國光路 250 號

No. 250, Guoguang Rd., Nan Chiu, Taichung, Taiwan 402, R.O.C.

E-mail address: {encore, echen}@cs.nchu.edu.tw

Telephone number: 0936-288857, 04-22840497~901

Fax number: 04-22853869

(5) Contact author: Lichung Chiang (江立中)

Telephone number: 0936-288857

encore@cs.nchu.edu.tw

(6) Key words: proxy, cache, squid, performance

ESP: An Efficient Storage Management Policy for Proxies

Lichung Chiang and Wen-Shyen E. Chen
Institute of Computer Science
National Chung-Hsing University
Taichung, Taiwan
E-mail: {encore, echen}@cs.nchu.edu.tw

Abstract

With the enormous growth of the WWW, the Internet has seen a large volume of traffic, which introduces severe congestion and worsens our surfing experience. Proxy cache is a common solution to this problem. It reduces the traffic on the Internet and the response time for Web accessing. However with the potential growth of the WWW, the proxy is overburdened and that in turn increases the response time even more.

In this paper we propose an efficient mechanism, ESP (Efficient Storage management policy for Proxies), to reduce the file operation and disk seek time in order to improve the performance of proxies. We implement ESP by modifying the source code of Squid. Squid with ESP shows a 100% performance improvement to the original Squid design.

1 Introduction

With the enormous growth of the WWW, the Internet has seen a large volume of traffic which introduces severe congestion and worsens our surfing experience. Proxy cache is a common solution to this problem. It reduces the traffic on the Internet and the response time. However with the potential growth of the WWW, the proxy is overburdened and that in turn increases the response time even more.

Recent researches have indicated that disk I/O overhead is becoming an important bottleneck for the performance of proxies. Rousskov and Soloviev[1] observed that disk delay contribute about 30% toward total hit response time. Mogul [2] stated that their observations suggest the disk I/O overhead turns out to be even higher than the latency improvement from cache hit. Markatos [3] also pointed out that a file creation followed by a file deletion may easily take up to 50 milliseconds, even on modern hardware. Given that the median size of a cache object is 5 KB, and that for each object a proxy creates a file to store it and deletes another file to free space. A proxy can only store objects at a rate of 100Kbytes/sec, which is even lower than most Internet connections.

In order to eliminate the overhead of file creation and deletion, we modify the source code of Squid to implement our storage management policy, ESP, which stores all objects in a single file. However this means that we have to manage the space in the file by ourselves. Furthermore, writing data scattered all over the disk may cause additional movement of read/write head and consequently increases the seek time, so ESP stores a whole object together. To further reduce the additional movement of read/write head, ESP writes in a log-structured manner. A log-structured file system [4] writes sequentially to reduce the movement of read/write head while writing.

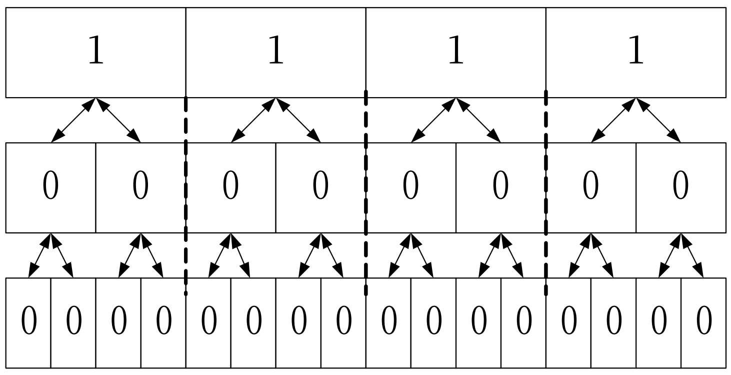
The rest of this paper is organized as follows. Section 2 provides a detailed introduction to ESP. In Section 3 we show the performance of ESP compared with original Squid. Section 4 gives our conclusions and future work.

2 ESP

As mentioned above ESP has the following features:

- It stores all objects in a single file
- It stores a whole object together
- It writes in a log-structured manner

To achieve these features, ESP maintains a pointer *pos* to indicate where the next object is to be stored and a variable *remain* to record the remaining size of the current contiguous free space. If *remain* is smaller than the object to be stored onto the disk, EPS searches for the next contiguous free space of which the size is larger than a certain threshold. In order to speedup the search of contiguous free space, ESP manages the space in the file with a data structure called multileveled-bitmap, which is a **variant** of buddy system [5]. Multileveled-bitmap can be regarded as a set of several buddy systems as Figure 1 shows.

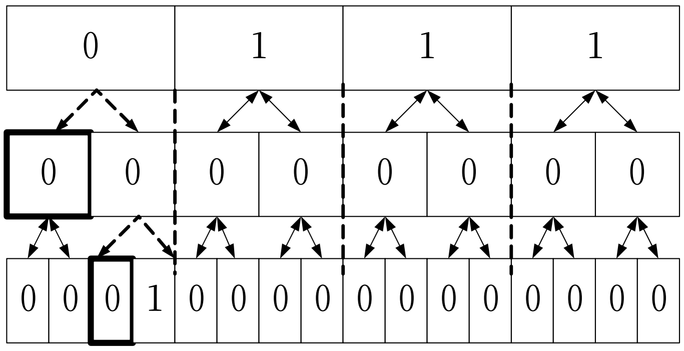


legend: 1 free space
0 used, split, or merged space

Figure 1: An example of the initial state of a 3 leveled-bitmap, it can be regarded as

several specialized buddy system, where each buddy system has only 3 levels.

While the concepts of splitting and coalescing are the same, the space allocation is very different. If the minimal allocation unit is 4KB (we called it a block in the rest of this paper), multileveled-bitmap allocates 12KB to a 9KB object and aligns it on a multiple of 4K (we can, logically, regard it as 3 4KB objects except for that they must be stored sequentially and contiguously) while buddy system allocates 16KB and aligns it on a multiple of 16KB which is the size of the space that the buddy system allocates to this object. Examples of ESP write are shown in Figures 2 and 3.

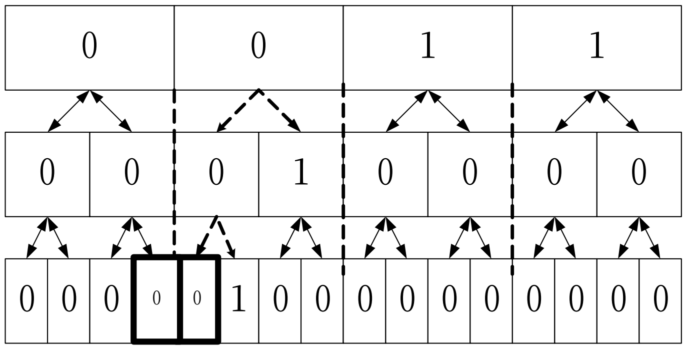


pos=3, *remain*=13

The dashed arrow lines are splits.

The bold-lined rectangles are the space allocated to this objects.

Figure 2: The status after allocating 3 blocks from the status of Figure 1.



pos=5, *remain*=11

The dashed arrow lines are splits.

The bold-lined rectangles are the space allocated to this objects.

Figure 3: The status after allocating 2 blocks from the status of Figure 2. The allocation of the 2 blocks doesn't start on a multiple of 2 but start from the last write, *pos*.

When the object is released the bits that are represented with bold-lined rectangles are reset to 1 and coalescing is performed.

Shown in Figures 4, 5, and 6 are examples of a buddy system allocating space to objects. In Figure 6 we can see that buddy system allocates 4 blocks to the 3-block large object and aligns it on a multiple of 4, instead of appending it to the last write.

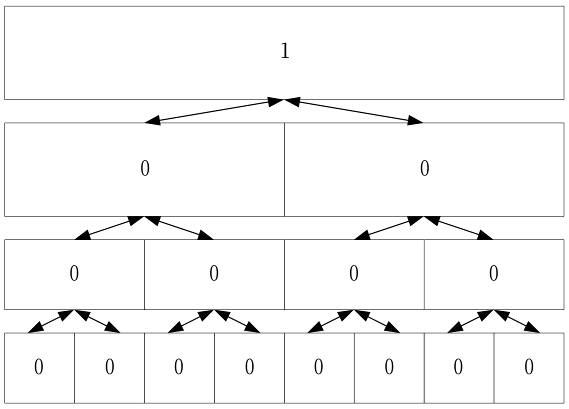
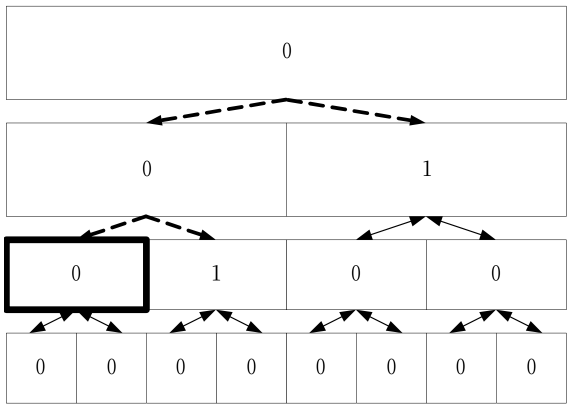


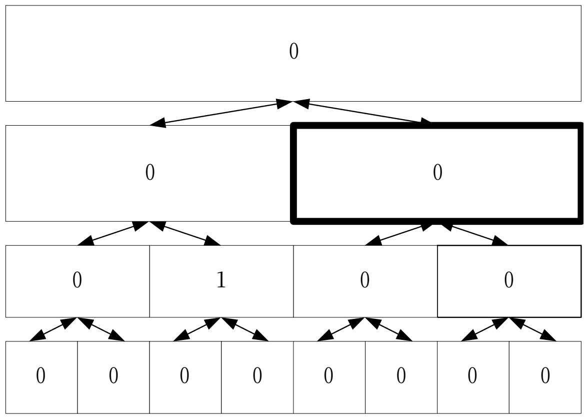
Figure 4: An example of the initial state of a buddy system, where the total space in this system is 8 block-large



The dashed arrow lines are splits.

The bold-lined rectangles are the space allocated to this objects.

Figure 5: The status after allocating 3 blocks from the status of Figure 4.



The bold-lined rectangles are the space allocated to this objects.

Figure 6: The status after allocating 3 blocks from the status of Figure 5.

If *remain* is smaller than the size of the new object, ESP searches the highest level for contiguous space that is larger than a certain threshold. Only searching the highest level speeds up this operation enormously at the cost of losing some free space in lower levels; however they are at most $2*(1+2+\ldots+2^{lvl-1})$ block-large; thus can be negligible.

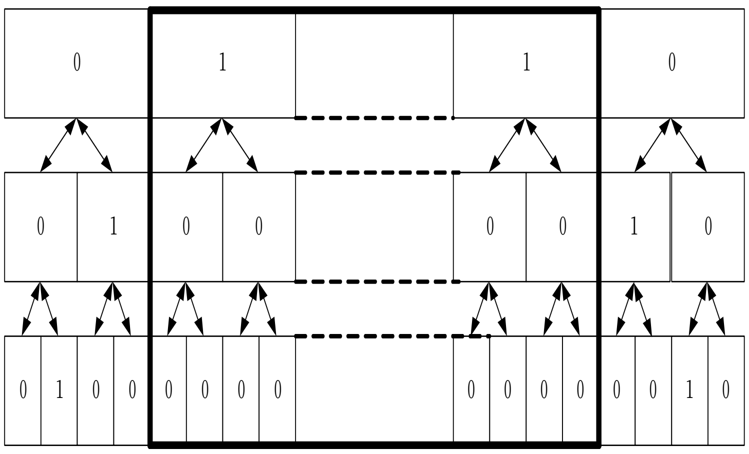


Figure 7: The area marked by the bold line is the contiguous free space found for oncoming writes.

The traditional Squid doesn't remove objects until the used space is higher than some low watermark and manages to keep the used size between the low watermark and high watermark. However this may lead to a condition that the used size is lower than the low watermark and ESP can't find free space by just searching the highest level. As a result, ESP cannot allocate space to objects nor can the replacement policy delete any objects; thus no new oncoming objects can be stored. In order to ease this condition, we modify the replacement policy to be more aggressive as follows

- While the used space is lower than low watermark evict 10+evict_more objects per second
- While the used space is between low and high watermark evict 200 objects per second
- While the used space is higher than high watermark evict 200 objects per 0.5 second

where *evict_more* is a variable to make the replacement policy even more aggressive if the contiguous free space we last found isn't large enough. It is set as follows

$$evict\_more = \left\{ \begin{array}{ll} 15 & \textit{the contiguous space found is smaller than the threshold} \\ 10 & \textit{the contiguous space found is smaller than 2 times the} \\ & \textit{threshold and larger than the threshold} \\ 5 & \textit{the contiguous space found is smaller than 3 times the} \\ & \textit{threshold and larger than 2 times the threshold} \end{array} \right\}$$

3 Simulations

3.1 Simulation Environment

We benchmark our implementation of ESP with web polygraph [6,7]. Web polygraph has the architecture shown in Figure 8. It uses several pairs of computers to simulate web servers and clients. Each server runs a *polysrv* to simulate several web servers which respond to HTTP requests while each client runs a *polyclt* to simulate several web clients (referred to as robots in the rest of this paper) which generates HTTP requests. Table 1 shows the objects distribution generated by web polygraph.

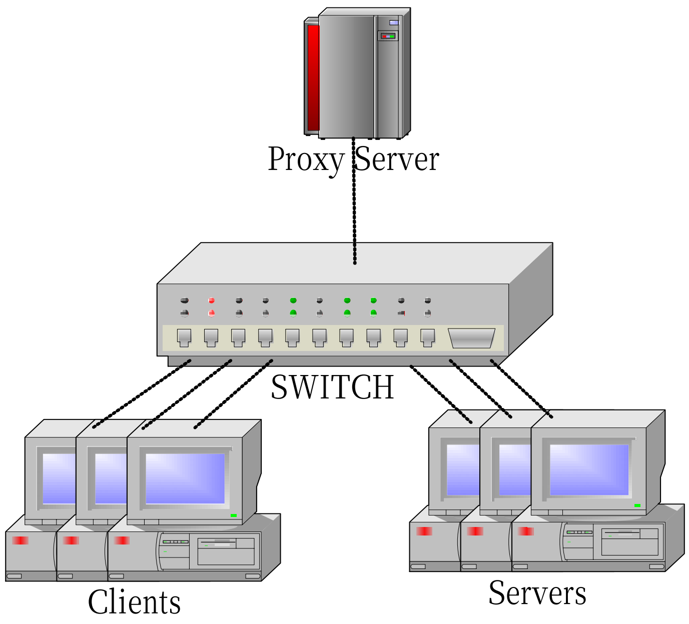


Figure 8: The architecture of web polygraph.

Content type	ratio (%)	mean size (bytes)	distribution
image	65	4604.28	exponential
HTML	15	8697.42	exponential
download	0.5	307218.10	log normal
other	19.50	25631.78	log normal

Table 1: The object distribution the web polygraph generates.

Hardware:

- Server: ACER 7100, CPU: P-III 1G with 256MB RAM, 30G IDE HD
- Client: ACER 7100, CPU: P-III 1G with 256MB RAM, 30G IDE HD
- Proxy Server: ACER 7100, CPU: P-III 1G with 512MB RAM, 2*30G IDE HD

Software:

- Server: RedHat 7.2, polygraph 2.7.6
- Client: RedHat 7.2, polygraph 2.7.6
- Proxy Server: RedHat 7.2, Squid-2.4.STABLE6, ext2 fs

Metrics: There are three metrics

- **Delay and response time:** On receiving a request web polygraph waits for a *think_time* which is randomly generated with a mean time 2.5s and standard deviation 1s. Web polygraph uses *think_time* to simulate the delay to fetch an object from a real web server that may be far away from the proxy server. The proxy server doesn’t think if it has the object the client requests so it replies immediately. We can regard the average response time as the quality of service that the proxy server can provide the larger the response time is the worse the quality is.
- **Hit ratio:** when a robot generates a request, it inserts a unique transaction-id in the HTTP header. After the requested server receives this request it inserts the mutant version of this transaction-id in the header of the response. As the robot receives the response, it checks if the transaction-id is the mutant version of the current transaction-id. If so, a miss is counted;if it is a mutant of some other transaction-id, a hit is counted. Web polygraph simulates an ideal proxy, which has infinite space to store all cacheable objects to calculate the ideal hit ratio for this simulation. Generally speaking, the higher the hit ratio, the lower the response time.
- **Throughput:** the number of replies per second a proxy server can sustain. With the same quality, that is the average response time, the higher throughput indicates the better performance.

3.2 Simulation Results

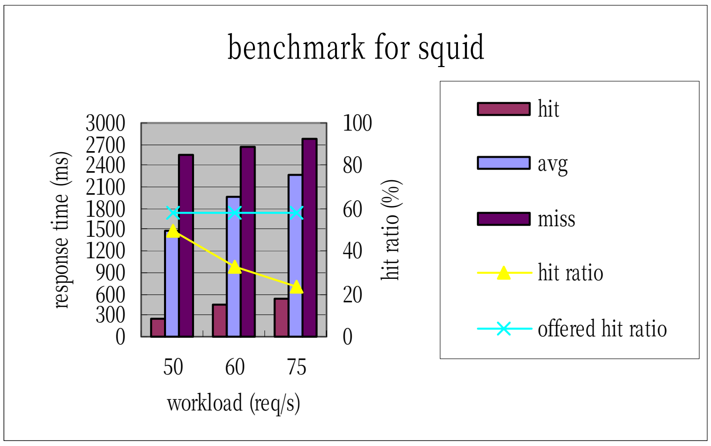


Figure 9: Benchmark for Squid with different workloads

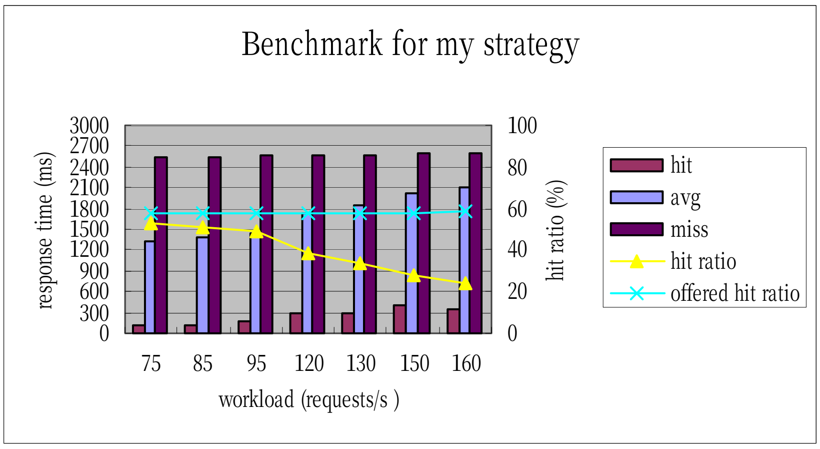


Figure 10: Benchmark for ESP with different workloads

As is seen in Figures 9 and 10, as the workload increases, the hit ratio drops and the average response time increases.

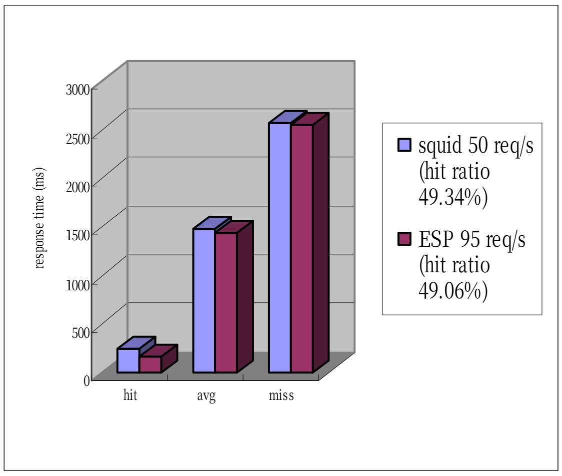


Figure 11: Comparison of ESP and Squid with average response time close to 1.5s.

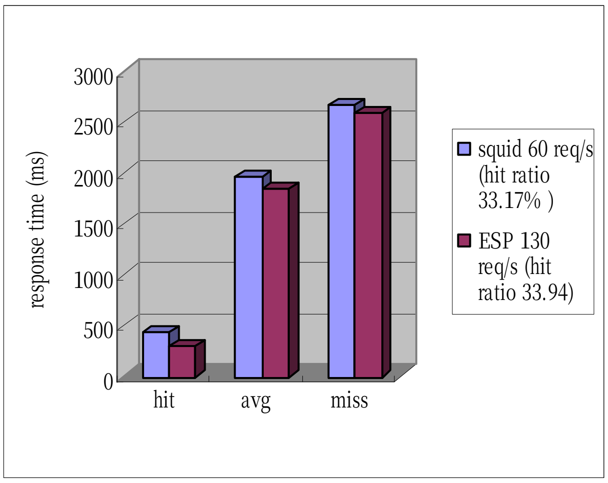


Figure 12: Comparison of ESP and Squid with the average time close to 2s.

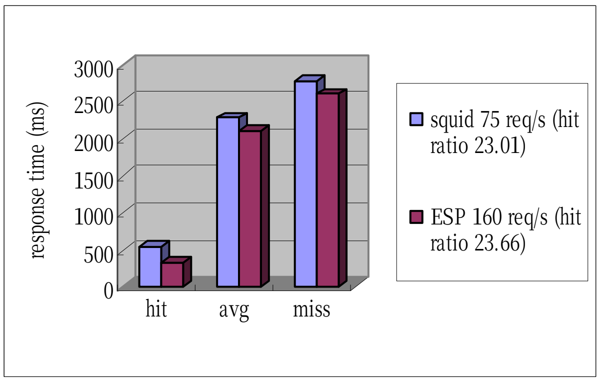


Figure 13: Comparison of ESP and Squid with average response time larger than 2s.

From Figures 11,12, and 13, it is obvious that ESP can sustain a throughput that is twice the throughput original Squid can, while providing less response time or the response time close to what Squid provides.

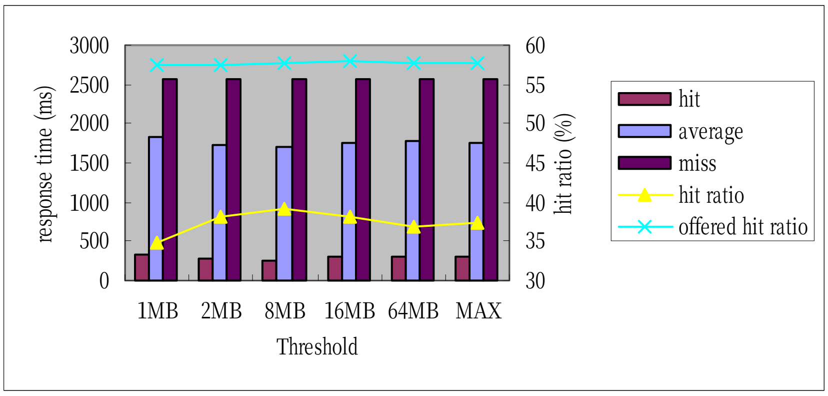


Figure 14: Comparison of ESP with different threshold

Figure 14 shows a comparison of ESP with different threshold with lower threshold like 1MB. We may lose the locality of objects from the same site, and therefore store objects from the same site all over the disk and increase the seek time. With higher threshold, we can ease this condition. However, with a threshold that is too high may cause the read/write head to move further (ignoring nearby contiguous free space) and result in higher seek time. High seek time reduces the throughput of disk and therefore reduces the hit ratio.

4 Conclusions and Future Work

In this paper we propose an efficient mechanism ESP (Efficient Storage management policy for Proxies) to reduce the file operation and disk seek time in order to improve the performance of Squid. ESP shows a 100% performance improvement to the original Squid. When comparing the performance of ESP with different thresholds, an appropriate threshold does affect the hit ratio of ESP up to 10%.

In this paper we use a modified version of LRU as ESP's replacement policy. However, LRU replaces objects according to the time an object was last referenced and doesn't help in making contiguous space for ESP. After running for a long time, there might be a lot of very small slice of space scattered all over the disk which may cause the inefficiency of ESP. Developing a dedicated replacement policy which takes objects' locality on disk into consideration would help increase the size of contiguous space.

Markatos [3] stated that the aggregate write operations outnumber read ones; it is obvious that most write operations are not necessary. If we can reduce the useless write operations then we can also improve the performance of the proxy servers.

Utilizing raw device can eliminate the file system overhead and further improve the performance of proxies. We hope to investigate this aspect more in our future research.

References

[1] A. Rousskov and V. Soloviev, "On Performance of Caching Proxies," In *Proc. of the 1998 ACM SIGMETRICS Conference*, 1998

[2] J. C. Mogul, "Speedier Squid: A Case Study of an Internet Server Performance Problem, " *The USENIX Association Magazine*, Vol. 24, No. 1, pp.50- 58, 1999.

[3] E. P. Markatos and M. G. H. Katevenis, "Secondary Storage Management for Web Proxies," *The 2nd Usenix Symposium on Internet Technologies and System*, Boulder, Colorado, USA, Oct. 11-14, 1999.

[4] T. Blackwell, J. Harris, and M. Seltzer, “Heuristic Cleaning Algorithms for Log-Structured File Systems,” In *Proceedings of the 1995 Usenix Technical Conference*, January 1995.

[5] K. C. Knowlton, “A Fast Storage Allocator,” *Communications of the ACM*, Vol. 8, No. 10, pp.623-625, October 1965.

[6] http://www.web-polygraph.org

[7] http://polygraph.ircache.net/doc/papers/paper01.ps.gz