2002 International Computer Symposium: Workshop on Computer Networks

Performance Issues in Squid

Abstract

This paper describes the architecture of Squid proxy cache. It discusses the improvements in Squid achieved by three principles: reducing the file management overheads by using a large file, increasing CPU utilization for Squid with asynchronous I/O, and using scalable event notification facility, *Kqueue*, to improve proxy server under heavy loads. We implemented our proxy server based on the Squid according to the above three principles. From test results, we improve 390 req/sec to the version of Squid 2.3 stable 4. We also have about 310 req/sec improvement compared with the version of Squid 2.4 stable 6, the newest stable version now.

Mao-yu, Jan P88009@csie.ntu.edu.tw

Yung-ching, Weng chin60@pchome.com.tw

Feipei, Lai (contact author) flai@cc.ee.ntu.edu.tw

Postal address: 1 Roosevelt Rd., Sec. 4, Taipei 106, TAIWAN

Tel: 02-33665001

Fax: 02-23637204

Keywords: Asynchronous I/O, Disk I/O, Internet, Proxy Server, Raw device, Squid.

1. Introduction

From some statistics, there are about 27 million web sites over the world [9] [11]. Since Internet is so developed, the traffic load of Internet becomes heavier than before.

In Internet world, the proxy server is a good choice for saving network bandwidth and reducing users' waiting time. From recent research, some results show that disk I/O overhead is an important factor of proxy performance. For example, Rousskov and Soloviev found that disk delay time contributes about 30% toward total hit response time [13]. Mogul said that they find the disk I/O overhead of caching is much higher than the latency improvement from cache hit in the web proxy at Digital Palo Alto firewall, so the server runs in non-caching mode for saving the disk I/O overhead [8].

Proxy server needs a large number of disk I/O accesses. If we can reduce the overhead of disk I/O, we may get large performance improvement of proxy server. We use two approaches to improve the performance of disk I/O. First, we reduce file management overheads. We try to store all objects in a large file, and then we can reduce system calls of file operations. Moreover, we implement the large file with raw device to increase performance. Second, we try to raise the usage of CPU by using asynchronous I/O.

Squid uses *select() and poll()* system calls to detect network I/O events. The performance is not good when load is heavy. So, we use the *Kqueue* [4] to increase the Squid performance under heavy load.

2. Background

Squid [12] springs from the Harvest cache project [2]. It was designed to use a single process to eliminate CERN's [5] overheads of process creation and termination. Rousskov and Soloviev analyzed the performance of the Squid and observed that disk delays contribute about 30% toward total hit response time. Markatos, Katevenis, Pnevmatikatos, and Flouris [7] studied the overheads associated with disk I/O for web proxies with their proxy simulator and proxy trace, and proposed some secondary storage management alternatives that improve performance.

Operating system researchers and vendors have investigated the impact of various kernel mechanisms on Internet server performance. Banga and Mogul [1] reported that their scalable versions of the *select()* system call and the descriptor allocation algorithm have led to an improvement of up to 58% in Web proxy throughput. Lemon implemented *Kqueue*: a generic and scalable event notification facility for FreeBSD.

3. Squid Architecture

The objective of a Web proxy is to save the backbone WAN bandwidth, and to reduce the request response time to the clients. The performance gain provided by

Web proxies is significant. Typically a Web proxy, as part of the network system, is installed in LANs and is directly attached to the gateway router connecting the LAN to the backbone WAN.

The Squid proxy is a public domain Web cache sprang from the Harvest cache project. It is commonly used for caching Web data. Figure 1 illustrates the architecture of the Squid. Functionally, the Squid was designed to be an event-driven server consisting of a single thread. The thread, event manager in figure 1, polls the network, disk, and timeout events by *select( )* or *poll( )* system call and then decides which events are ready to be proceeded. Once the ready events have been decided, the thread invokes the event handlers in protocol manager or storage manager corresponding to the ready events.

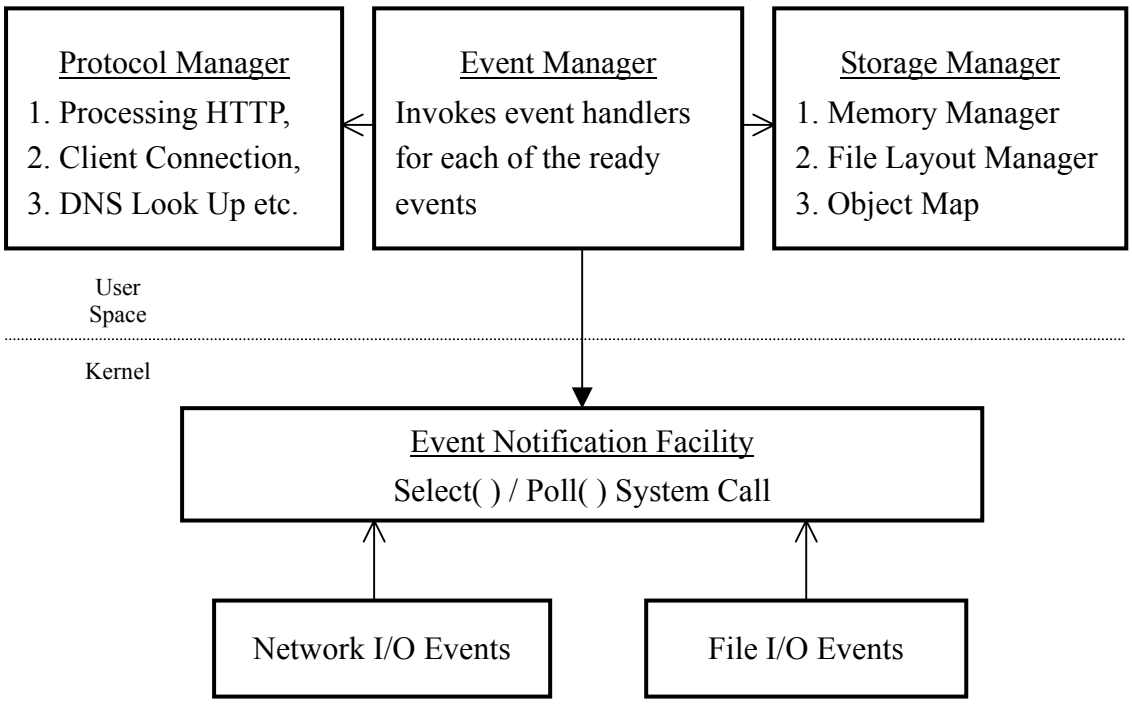

Figure 3.1 Squid block diagram.

To achieve high performance through parallelism, the thread simulates the finite-state machine. Each connection to the Squid must go through several states before the request from a connection is satisfied. The states of all connections are maintained into a large file descriptor table in the Squid.

Squid keeps meta-data about the cache contents in main memory. This enables the Squid to determine whether it can serve a given request from its cache without accessing the disk. Squid has a memory buffer used for storing intrinsic objects, for storing the most recently used objects, and for storing error responses which resulted from bad requests. This buffer is usually organized into a high-level main cache. To reduce DNS overheads, Squid implements its own DNS cache and uses a configurable number of “DNS server” which cooperates with the Squid to achieve non-blocking DNS requests.

Architecture of the Squid architecture has some interesting consequences [6]:

- A large number of file descriptors must be managed by a single process.
- Many operating system facilities must be replicated within the Squid.
- Storing the meta-data for each cached object in memory means that main memory utilization grows with the number of objects cached or the proxy cache size. Increasing the cache size requires increasing both disk and main memory.

4. Principles of Improving Squid Performance

In this paper, we use three approaches to improve the Squid's single node performance.

4.1 Reduce the file management overheads:

In the Squid, one object is saved within a unique file. Since there are so many objects that the Squid should handle, it needs many system calls to deal with the file operations (such as file open, file close). Markatos, Katevenis, Pnevmatikatos, and Flouris observed file management overheads incurred by "one file per objects". So, we try to store all objects in a large file cache. We can reduce system calls of file operations and then reduce file management overheads. In our implementation we only need to open file one time.

Usually there are two methods in UNIX-like system to implement a large file: one by UNIX File System, the other by raw device. A raw device, also known as a raw partition, is a disk partition that is not mounted and not written to via the UNIX file system, but is accessed via a character-special device driver. It is up to the application how the data are written since there is no file system to do this on the application's behalf. Raw devices are not like regular file management systems. The information they store cannot be identified or accessed by users. There can be a performance benefit from using raw devices, since a write to a raw device bypasses

the UNIX buffer cache, the data is transferred directly to the disk. Another benefit of raw devices is that no file system overhead is incurred in terms of I-node allocation and maintenance or free block allocation and maintenance.

The major disadvantage of using a raw device file is that the maximum file size is fixed by the size of the partition. If the partition becomes full, the raw device file must be moved to a larger partition. In the worst case, the disk must be partitioned in order to create a larger partition.

They are almost the same for coding on raw device and on the UNIX file system except for the file open. Figure 4.1 shows two examples for file open in raw device and UNIX file system.

```
Raw device:
#define CacheFile "/dev/ad0s2“
cache_fd = open(CacheFile, O_RDWR);
```

```
Unix File System:
#define CacheFile "/cache/Cache“
cache_fd = open(CacheFile, O_RDWR);
```

Figure 4.1 The usage of file open on raw device and on the UNIX file system

The first field of *open* system call is the position and name of the file you want to open in the UNIX file system. But it is the position and device name in raw device. In this example of raw disk, “ad0s2” means the second partition of the first IDE hard

disk, and device names are always in “/dev”. These are fixed when you select a partition. In the other hand, when you select the UNIX file system to handle your file operation, you can decide the position and name of the file you want to open. In the example of UNIX file system, “/cache/Cache” just means the name of the file is “Cache”, and the position is in “/cache/”.

To get speedier, we modify the Squid to use raw device as a large file cache. Some important points need to be considered when we use the raw device:

- The first 10KB must be reserved for the header.
- Overlapping partitions must be manually avoided.
- It should be verified that backup procedures include raw devices.
- Always take a complete backup before enabling use of a raw device.

4.2 Squeeze more CPU out to the Squid:

In recent research, asynchronous I/O outperforms synchronous I/O over 35% [3]. Because the Squid just uses a single thread to manage all events, if it uses synchronous I/O *read* or *write*, that makes it poor performance. So it is better for the Squid to use asynchronous I/O rather than synchronous I/O. Some UNIX systems (such as FreeBSD) provide asynchronous I/O mechanisms, which allow processes to perform I/O without blocking. Using asynchronous I/O can improve I/O performance. We modify the Squid to run with parallel reads and parallel writes by asynchronous

I/O. That thereby reduces I/O waiting time. The Squid with asynchronous I/O maximizes I/O throughput by parallelizing reads and writes.

4.3 Use scalable kernels:

In Gaurav and Jeffrey study [1], they investigated scalable kernel performance for Internet servers under realistic loads and found their scalable version of *select( )* improving the performance of Web servers. *Kqueue* is a new mechanism on FreeBSD that allows the application to register its interest in a specific event, and then efficiently collect the notification of the event later. The primary function of *Kqueue* is to create a system that would be efficient and scalable to a large number of descriptors. The secondary function is to make the system flexible. We use the primary function to increase the Squid performance under heavy load. The Squid uses *select( )*or *poll( )* to register events and poll the ready events. In this paper, we use *Kqueue* to replace *select( )* and *poll( )* in the Squid. Table 4.1 summarizes our modification.

The first column of the table is activities that the event-driven server should take, the second one is description of variables and function calls, and the third one is the code locations we need to modify.

5. Benchmark

5.1 Web polygraph

Table 4.1: The summary of modifying the Squid to use *Kqueue*

Activity	Variables and Function Calls	Code Location
Create Kqueue	1. int kq : queue for all events 2. kqueue( ) : creates a new Kqueue.	1. comm._select.c 2. comm._select.c : comm._select_init( )
Register events	1. kevent[ ] : This data structure would be passed to Kqueue. 2. ke_change( ) : This is an addition to the Squid original code.	1.comm.c: commSetSelect( ) 2.comm.c: ke_change( )
Poll Kqueues	1. kevent[ ] : This data structure would be passed to Kqueue 2. kevent( ): Polling Kqueue to detect the ready events.	1. comm._select.c : comm._select( ) 2.comm_select.c : comm._select( )
Handle the ready events	1. Invoke handlers corresponding the ready events	1. comm._select.c : comm._select( )

Polygraph [10] is a set of programs that simulate Web clients and servers. Polygraph can be configured to send HTTP requests through a proxy. The scheme is shown as Figure 5.1. The benchmarking results can be used for tuning proxy performance, evaluation of caching solutions, and for many other interesting activities. There are two main processes “polysrv” and “polyclt” to simulate server and client behavior in Internet.

5.2 Workload

The workload file is polymix-3.pg. There are some important features. We will mention these as the following.

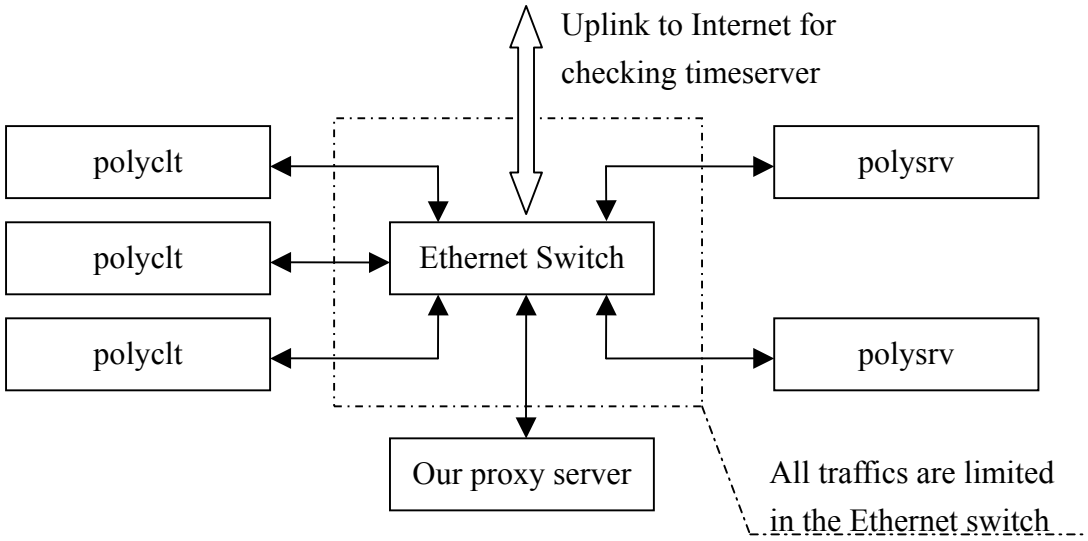

Figure 5.1: Polygraph working structure

5.2.1 Phase schedule

Table 5.1 describes all the important phases in a PolyMix-3 test. Not counting the fill phase, the test takes about 12 hours. Filling the cache usually takes an additional 12-24 hours, depending on the product.

5.2.2 Content types

PolyMix-3 defines a mixture of content types. Each content type has the following properties: popularity, content size distribution, cachability percentage, life-cycle parameters, and file name extensions distribution. The approximate parameters for the first four properties are given in Table 5.2.

5.2.3 Simulated robots and servers

A single Polygraph client machine supports many simulated *robots*. A robot can emulate various types of Web clients, from a human surfer to a busy peer cache. All

Table 5.1: Phase schedule

Phase Name	Duration	Activity
framp	30 min	The load is increased from zero to the peak fill rate.
fill	variable	The cache is filled twice, and the working set size is frozen.
fexit	30 min	The load is decreased to 10% of the peak fill rate. At the same time, recurrence is increased from 5% DHR to its maximum level.
inc1	30 min	The load is increased during the first hour to reach its peak level.
top1	4 hours	The period of peak ``daily'' load.
dec1	30 min	The load steadily goes down, reaching a period of low load.
idle	30 min	The ``idle'' period with load level around 10% of the peak request rate.
inc2	30 min	The load is increased to reach its peak level again.
top2	4 hours	The second period of peak ``daily'' load.
dec2	30 min	The load steadily goes down to zero.

Table 5.2: Content types

Type	Portion	Reply Size	Cachability	Expiration
Image	65.0%	*exp*(4.5KB)	80%	*logn*(30day, 7day)
HTML	15.0%	*exp*(8.5KB)	90%	*logn*(7day, 1day)
Download	0.5%	*logn*(300KB,300KB)	95%	*logn*(0.5year, 30day)
Other	19.5%	*logn*(25KB,10KB)	72%	*unif*(1day, 1year)

robots in PolyMix-3 are configured identically, except that each has its own IP address. It limits the number of robots (and hence IP aliases) to 1000 per client machine. A PolyMix-3 robot requests objects using a Poisson-like stream, except for embedded objects (images on HTML pages) that are requested simulating cache-less browser behavior. A limit on the number of simultaneously open connections is also supported, and may affect the request stream. PolyMix-3 servers are configured identically, except that each has its own IP address.

5.3 Experimental platform

Our proxy server is based on the Squid-2.3 stable 4 and we use FreeBSD 4.1 version as the operation system. Table 5.3 summarizes the hardware and software of experimental platform.

Table 5.3: Experimental equipment

	Proxy server	**Server machine**	**Client machine**
Hardware	CPU:PIII 650*2 Memory: 1G MB HDD: SCSI 9G*6	CPU:PIII 800*1 Memory: 256 MB HDD: IDE 30G*1	CPU:PIII 800*1 Memory: 256 MB HDD: IDE 30G*1
Software	OS: FreeBSD 4.1 Our proxy server	OS: FreeBSD 3.4.TMF Polygraph-2.5.4 Polymix-3.pg	OS: FreeBSD 3.4.TMF Polygraph-2.5.4 Polymix-3.pg

6. Results

Before we discuss our results, we first define some terminologies:

Throughput: The number of requests that proxy server replies to client in one second.

Response time: The time it spends from the client side requesting to proxy server to receiving a response from proxy server. Its acceptable value is under 3 seconds.

Peak request rate: The maximum rate that client side offers. The unit is requests per second.

Miss time: The response time when request is a miss request.

Hit time: The response time when request is a hit request.

Hit ratio: (The amount of hit requests) / (the amount of total requests)

Errors: The percentage of errors happened during the experiment.

Duration: Test duration.

Phases: The phases we extract data to make this report. We usually use the data of Top 1 and Top 2 to be our test source.

6.1. Test result of traditional Squid

6.1.1. The Squid 2.3 stable 4

This version is what we are based on. The test result is as Table 6.1.

Table 6.1: Test result of the Squid 2.3 stable 4

Throughput:	60.02	rep/sec
Response time:	1809.19	msec
- misses:	3426.94	mese
- hits:	393.26	msec
Hit Ratio:	56.14	%
Errors:	0.00	%
Duration:	8.00	hour
Phases:	top1 & top2	

6.1.2 The Squid 2.4 stable 6

This version is the newest stable version now. So, we take it to be a comparison with our proxy server. The test result is as Table 6.2.

Table 6.2: Test result of the Squid 2.4 stable 6

Throughput:	139.96	rep/sec
Response time:	1481.91	msec
- misses:	2847.03	mese
- hits:	172.05	msec
Hit Ratio:	53.71	%
Errors:	0.00	%
Duration:	8.00	hour
Phases:	top1 & top2	

6.2 The test results of our proxy server

6.2.1 Version 1 of our proxy server

Version 1, we call *expv1,* mainly includes the usage of one large file with raw device and asynchronous I/O. The best peak request is 400 req/sec. The result is as

Table 6.3:

Table 6.3: Test result of expv1

Throughput:	399.95	rep/sec
Response time:	2631.29	msec
- misses:	4205.21	mese
- hits:	1039.02	msec
Hit Ratio:	52.91	%
Errors:	0.00	%
Duration:	8.00	hour
Phases:	top1 & top2	

6.2.2 Version 2 of our proxy server

Version 2, we call expv2, is the version of expv1 with *Kqueue*. In expv2, we use *Kqueue( )* system call, rather than *select( )*, to detect disk, network, and timeout events. The result is as Table 6.4.

Table 6.4: Test result of expv2

Throughput:	449.8	rep/sec
Response time:	2630.19	msec
- misses:	4720.21	mese
- hits:	988.02	msec
Hit Ratio:	52.11	%
Errors:	0.00	%
Duration:	8.00	hour
Phases:	top1 & top2	

6.2.3 Comparison of test results

Finally, we arrange our test results into Table 6.5 to see what we have improved. In table 6.5, symbols (1), (2), (3), and (4) are Squid 2.3 stable 4, Squid 2.4 stable 6, expv1 and expv2, respectively.

Table 6.5: Summary of test results

	(1)	(2)	(3)	(4)
Peak request rate (req/sec)	60	140	400	450
Compare to (1) (req/sec)	-	+80	+340	+390
Compare to (2) (req/sec)	-	-	+260	+310

6.2.4 Result discussion

Now, we want to discuss some characteristics of proxy server behavior from our test results. Table 6.6 is the test results of our proxy server, and it is under the best configuration.

Table 6.6: Test result of our proxy server

peak request rate (req/sec)	response time (msec)	miss time (msec)	hit time (msec)	hit ratio (%)
200	1398.02	2941.2	58.08	56.36
220	1410.83	2950.64	70.12	56.28
240	1423.91	2960.64	79.14	56.18
260	1446.20	2981.7	90.22	55.90
280	1467.12	3001.92	103.91	55.77
300	1494.39	3025.85	121.88	55.52
320	1525.41	3052.63	141.93	55.20
340	1569.85	3092.8	172.65	54.89
360	1619.5	3140.61	206.12	54.53
380	1746.96	3232.94	277.7	53.75
400	1818.80	3338.05	365.06	52.91
420	2230.57	3659.79	645.54	52.52
450	2630.19	4720.21	1039.02	52.11

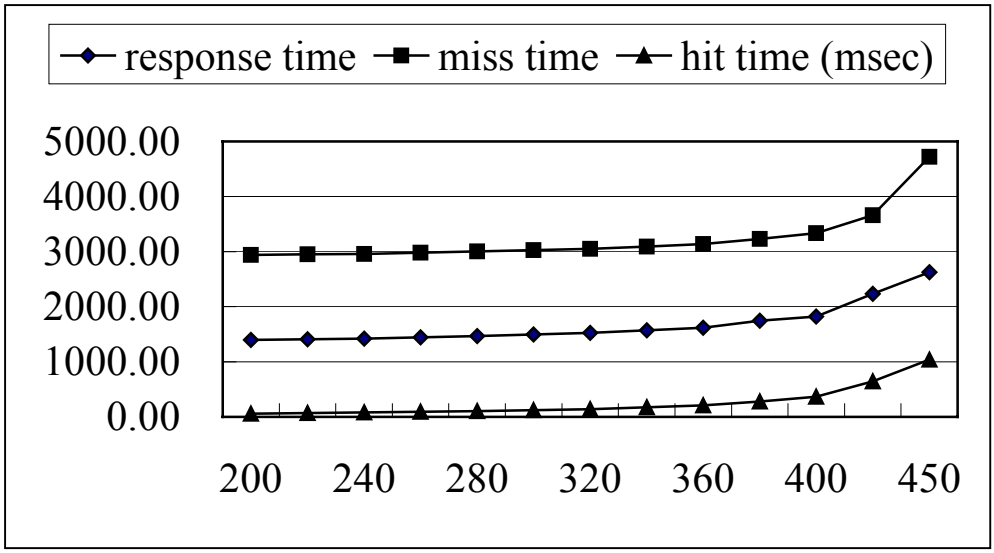

Figure 6.1: Peak request rate versus response time, miss time, and hit time

We find the response time, miss time, and hit time are all positively proportional to peak request rate. When the peak request rate is higher, the load of proxy server is heavier. So, the response time, miss time, and hit time get higher when the peak request rate gets higher.

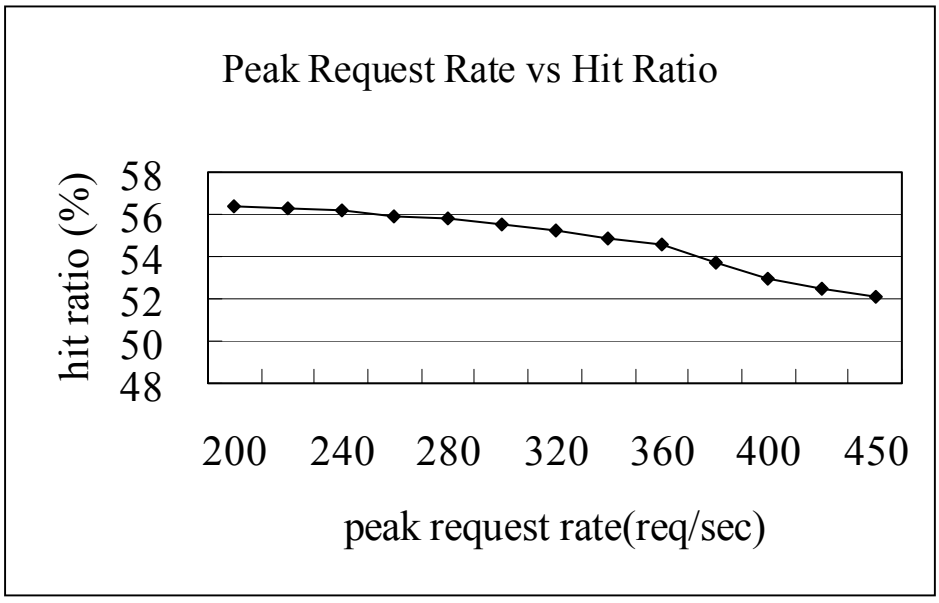

Figure 6.2: Peak request rate versus hit ratio

From Figure 6.2, we find that the hit ratio is negatively proportional to peak request rate. When the peak request rate is getting higher, the working set size is getting larger. But the cache disk is still the same. Finally, the hit ratio will become lower. If we want to get a good hit ratio, we must have enough cache space to store objects. So, when the capability of proxy server gets higher, the space of cache disk must get larger for maintaining a good level of hit ratio.

7. Conclusion

We used three approaches to improve the Squid's single node performance. From our test results, we find the disk I/O is an important factor of overall proxy server performance. The version 1 of our proxy server increases about 340 req/sec compared with the Squid 2.3 stable 4. It even increases about 260 req/sec compared with Squid 2.4 stable 6. It enhances overall proxy server performance by improving disk I/O. The test results of expv2, implemented with *Kqueue( ),* shows that *Kqueue( )* contributes the improvements of up to 8% request rates under heavy load.

8. Reference

1. G. Banga, and J. C. Mogul. Scalable Kernel Performance for Internet Servers under Realistic loads. In Proceeding of the USENIX Annual Technical Conference, New Orleans, Louisiana, June 1998.

2. Anawat Chankunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz,

and Kurt J. Worrell. A hierarchical Internet Object Cache. In 1996 USENIX Technique Conference, San Diego, CA, January 1996.

3. Richard Clark, “Building Better Applications via Asynchronous I/O,” http://www.mactech.com/articles/mactech/Vol.12/12.12/AsynchronousIO/

4. Kqueue. http://people.freebsd.org/~jlemon/.

5. Ari Luotonen, Henrik Frystyk Nielsen, and Tim Berners-Lee. CERN http3.0A. http://www.w3.org/pub/WWW/Daemon/, July 1996.

6. C. Maltzahn, K. J. Richardson, and D. Grunwald. Performance Issues of Enterprise Level Web Proxies. In Proceeding of the ACM SIGMETRICS ’97 Conference, Seattle, WA, June 1997.

7. Evangelos P. Markatos, Manolis G. H. Katevenis, Dionisis Pnevmatikatos, and Michail Flouris. Secondary Storage Management for Web Proxies. In Second USENIX Symposium on Internet Technologies and Systems, 1999.

8. Jeffrey C. Mogul. Speedier Squid: A Case Study of an Internet Server Performance Problem. ;login: The USENIX Association Magazine, 24(1):50-58, 1999.

9. Matthew Gray, “Internet Statistics Growth and Usage of the Web and Internet,” <http://www.mit.edu/people/mkgray/net/>.

10. Measurement Factory INC., http://www.measurement-factory.com

11. Netcraft Web Site Finder, <http://www.netcraft.com>.

12. Squid. http://squid.nlanr.net/Squid/

13. A. Rousskov and V. Soloviev. On Performance of Caching Proxies. In Proc. Of the 1998 ACM SIGMETRICS Conference, 1998.