Optimizing Web Content Delivery using Web Server Accelerator

Dongjun Shin **Kern Koh**

School of Computer Science and Engineering
Seoul National University, Korea
Email: djshin@oslab.snu.ac.kr kernkoh@june.snu.ac.kr

Abstract

Traditional web servers generate and deliver web contents in the same context. While the generation part becomes more and more complicated due to the complexity of implementing various features, the delivery part remains relatively simple and can be efficiently processed. This asymmetry leads to difficult and challenging issues in optimizing the web server performance.

In this paper, we describe the design, implementation, and performance evaluation of a web server accelerator, called Tornader. It resides in front of a web server and improves server performance by efficiently delivering cached responses while leaving the role of content generation to the web server. Due to the scalable architecture design and various optimization techniques, Tornader can boost the performance of the most widely used Apache web server up to 150% under high load condition and shows scalable performance enhancement in multi-processor systems. Furthermore, it can be used on most modern operating systems, since it is entirely implemented as a user-level program using POSIX API.

Keywords: web server accelerator, web content delivery, design and implementation, performance evaluation

1 Introduction

The demand for feature-rich, powerful and economic web servers has become more intense due to the rapid deployment of web based information systems. Although there have been quite a few researches in this area, it is not easy to meet all of these requirements.

Probably the most challenging issue is to enhance the performance (or capacity) of web service while maintaining the required features. To enhance the performance of web service, we can upgrade hardware resources or change the web server software to a faster one. The hardware approach is uneconomical and it does not solve the scalability problem in the long run, because it does not resolve the basic inefficiencies of the software being used. The software approach, while the cost of upgrades may be low, sounds too aggressive to those who are already providing large-scale services, because the features provided by different servers are also different, and because changing a web server platform may result in the total re-design and re-engineering of the entire service structure.

An alternative is to use a so-called web server accelerator, which resides in front of a web server and acts as a reverse caching proxy to reduce the overhead of a web server. The accelerator approach is very attractive in that

- We can optimize the accelerator for efficient web object delivery. By separating the role of delivery from web server, the original features provided by old web server are maintained, or the web server developers can add more useful features without worrying the performance issues.
- We can increase the performance of the web server with minimal changes of the existing service environment. This greatly reduces the total cost of changes in service environment.

We have designed and implemented a portable web server accelerator, called Tornader, and evaluated its effectiveness.

2 Web Accelerator Design

2.1 System Overview

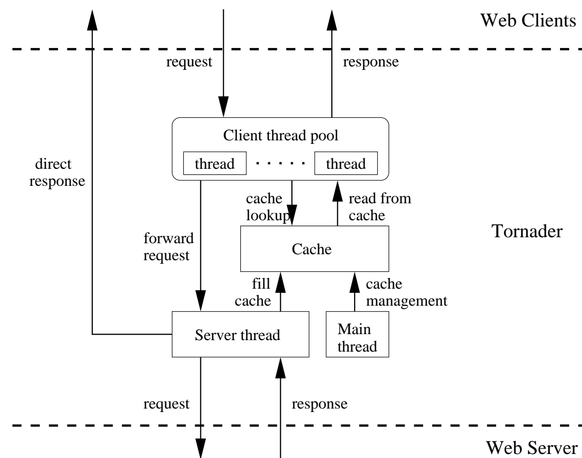


Figure 1: Overall architecture

Figure 1 shows the overall architecture of Tornader. Tornader is based on a design called asymmetric multi-thread event-driven (AMTED) architecture that consists of threads with heterogeneous roles: the client thread, the server thread, and the main thread.

The client thread handles front-end connections with web clients, receives requests, checks whether or not the requested object is cached, and send responses using cached object. If the requested object is not in cache, it passes client request to the server thread. Tornader can have a pool of client threads for scalable performance improvement in multi-processor systems.

The server thread handles back-end connections with the web server, forwards client requests to the web server, allocates cache entries, and fills the cache from the web server response. The client requests are passed back to the client thread with the completion of cache fill. If the response cannot be cached, it

Copyright ©2001, Australian Computer Society, Inc. This paper appeared at the Twenty-Fifth Australasian Computer Science Conference (ACSC2002), Melbourne, Australia. Conferences in Research and Practice in Information Technology, Vol. 4. Michael Oudshoorn, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

directly forwards the server response to the web client (dynamic response, no-cache pragma, etc).

The main thread periodically replaces cache entries that are expired, manages time related information, and gathers statistical data.

The advantages of using AMTED architecture are as follows.

- Low overhead: In a multi-process model web server, each process is dedicated to a single connection at a time, but in an event-driven model, a process (or thread) can simultaneously handle thousands of connections, which results in reduced overhead for resource usage and for OS-level synchronizations (i.e. context switch, lock).
- SMP friendly: Systems with multiple processors are nowadays both common and economical. By increasing the number of client threads, Tornader can fully utilize the processing power of a multi-processor system.
- Separate optimization: Each thread has a different role and can be individually optimized for its own purpose. For example, the client thread is optimized to deliver responses over instable and slow connections, while the server thread is optimized to handle relatively stable and fast connections.
- Resource sharing: The use of thread makes it easy to share resources among threads without the overhead of costly IPC operations. This is really important for the sharing of web object caches, because all of the threads share it and cache-related operations are frequent.

2.2 Web Object Cache

The cache plays two important roles in Tornader: one is to separate the object delivery mechanism from web server operation, and the other is to shorten the length of the critical-path.

In traditional web servers, the following operations occur for each HTTP request.

Step 1: accept connection
Step 2: read socket and parse request
Step 3: translate URI into file pathname
Step 4: open file
Step 5: read file and send response to socket
Step 6: close file and socket

In brief, the operations fall into three categories: preparation (step 2, 3, 4), generation (step 5), and delivery (step 5). By caching the web server responses, the preparation and the generation parts can be skipped. This contributes both to the shortening of the critical-path and to the separation of object delivery.

Tornader maintains a small cache in the main memory area. There are several considerations for using memory-only cache: First, the web access pattern is heavy-tailed and the working-set size of hot objects is small enough to fit into a small main memory cache. Second, the cost of cache miss is small, because it doesn't take much time to retrieve the missed object from a nearby web server. And last, if we use a disk system for the cache, every cache operation will introduce disk I/O's, which will disturb the operation of the web server when Tornader and the web server are running on the same machine.

The cache pool is structured as a hash table that is keyed by the (host, URI) pair. It consists of two cache pools: the stable and waiting cache pools. The stable cache pool contains objects that are fully fetched from the web server. This cache, which is accessed by all threads, contains either meta-data only or meta-data with content. The waiting cache pool, which is exclusively used by the server thread, contains partially fetched objects. If the fetch is completed, it is moved into the stable cache pool. The division of the cache pool leads to simple implementation and reduced lock usage.

We used a MIME type based expiration for the cache replacement policy. This policy prefers objects of a possibly invariant type (i.e. image, application binary) and gives longer expiration values to them. When the replacement candidates are objects with the same MIME type, the smaller one is preferred, so that the cache can contain more objects. The expiration time is fully configurable, which allows us to fine-tune the accelerator for specific service environments, such as image service or file download service. The list in hash bucket is ordered by expiration time so that we can short-cut the list lookup operation and shorten the *"lock-holding-time"* during list traversal. For example, when the main thread replaces expired objects, it just traverses the list while the expiration time of the candidate object is equal to or earlier than the current time.

2.3 Portability and Adaptability

We designed our accelerator as a user-level program and implemented using POSIX API only — no platform specific features or kernel extensions are used. Tornader can run on any system where *pthread* is supported. Currently, it has been successfully compiled and run on Linux and Solaris.

As Tornader operates as a reverse proxy, every web server that complies to the HTTP standard can be accelerated. It can run on the same machine on which the web server runs, or on a different machine dedicated to reverse-proxy usage.

3 Optimization Techniques

3.1 Event Handling

The *poll()* system call is used rather than *select()* for event handling, because *select()* has an upper limit on the number of file descriptors that can be handled at once and the limit is platform-dependent.

Unfortunately, the event-driven architecture performs poorly when handling mixed connections with large speed gaps (Banga, Mogul & Druschel 1999). We try to avoid this situation by differing the role of each thread: the client thread handles slow external (or front-end) connections, and the server thread handles fast internal (or back-end) connections. Also, connections that are inactive for more than the *idle-threshold* are removed from the poll list, so that the overhead of *poll()* can be minimized. These optimizations reduce the chance of each thread entering unnecessary busy-loop for event checking.

3.2 Resource Sharing

In a multi-threaded architecture, resource sharing is easy because all threads share resources like memory address space and open files. However, sharing all resources is not a good idea for the following reasons: all accesses to shared resources must be synchronized and this incurs a performance penalty in a multi-processor environment because of the processor's cache consistency mechanism and the lack of cache affinity.

We avoid these problems using the *"data partitioning"* approach, which partitions shared data structures across all threads. For example, each thread has a *num_conn* variable, which represents the number of connections it handles. If we want to know the

total number of connections, it can be calculated by summing up the *num_conn* variables of all threads. We group these thread-partitioned data structures to make it cache-aligned and local to each thread to prevent them from spreading across multiple processors' caches.

3.3 Lock Optimization

The lock is needed whenever each thread accesses globally shared resources. As mentioned in 3.2, we partition shared resources as much as possible to reduce the usage of locks, but there are three resources that must be globally shared: cache, inter-thread communication, and access logging.

For the cache, we reduce lock contention by using bucket-level fine-grained locks and reader-writer locks — client threads acquire reader lock while server thread and main thread acquire writer lock for access to the bucket-hashed stable cache pool. Under normal circumstances, the cache hit-ratio is high and client threads are more active than server thread. As multiple client threads can simultaneously access the cache with the reader lock, the performance penalty of lock contention is greatly reduced.

Inter-thread communication is achieved via pipes and access logging is achieved with the single non-buffered *write()* system call. Each thread can access these resources without locks, because the kernel guarantees the atomicity of these operations.

3.4 Efficient Web Server Utilization

Studies have shown that a multi-process (MP) web server, such as Apache, performs poorly when the number of concurrent connections exceeds a certain threshold (Hu, Nanda & Yang 1999). Tornader can limit the maximum number of back-end connections to a configurable value, and all the client requests that are passed to the server thread are multiplexed into these limited back-end connections to make web server keep efficiently utilized.

This optimization has a great effect when there are many clients connecting via slow links or when the client keeps the connection idly open during the *think time*. These kinds of inactivities result in underutilization of the web server, because in an MP model, a web server process is dedicated to a client during the session lifetime. With Tornader, all the in-activities are efficiently taken care of by client threads, and the web server only needs to communicate with the accelerator at high-speed.

4 Experimental Methodology

4.1 Hardware and Software

Three machines are used in this experiment: one for the client and two for the server. Each machine is equipped with 1Gbps NIC to avoid the possible performance bottleneck in the network bandwidth. During the experiment, we directly connected the client machine to the server machine with optical fibre to reduce the round trip delay. The configuration of each machine is shown in Table 1.

Apache 1.3.20 was used for the base-line of the performance comparison. As mentioned in 3.4, the MP architecture of the Apache web server allocates one process for each client connection. The default configuration of Apache limits the total number of concurrent connections (or processes) to 256, so we increased it to 4096 to make it handle much more concurrent connections than with default setting. Also, we tuned the maximum number of open files per process to 4096 in both the client and the server machines, as this limits the maximum number of concurrent connections in the workload generator and Tornader. We set the number of client threads in Tornader as one in Server1, and two in Server2 to reflect the number of CPUs in the server machine.

During the Tornader test, Apache was used for content generation. Both Tornader and Apache run on the same machine.

4.2 Workload Generation

We used httperf (Mosberger & Jin 1998) for workload generation. We configured it to repeatedly retrieve several files. The file sizes are selected from 1KByte to 8KByte because these are the most frequently requested sizes of web objects (Barford & Crovella 1998). The connection keep-alive feature of HTTP/1.1 is used and there are 10 subsequent requests in a single HTTP session. We set the timeout to 5 seconds, so that if there is no activity in connection for more than 5 seconds, the httperf aborts that connection. This prevents the workload generator from consuming all TCP port addresses.

The simplicity of this workload allows the servers to perform at their highest capacity, since the requested files are cached either in the object cache (Tornader) or in the virtual memory area (Apache). Thus, in Apache, we can safely assume that the overhead of content generation is greatly reduced, because the content generation is as simple as reading a file without disk I/O. With these observations, we can compare the best achievable performance of Tornader and Apache. Also, we can evaluate the effect of optimizing the content delivery even when the overhead of content generation is minimal.

4.3 httperf Statistics

httperf provides various useful statistical summaries about the benchmark result. We used some of them to evaluate the performance of tested server.

4.3.1 Connection Time

The *"connection time"* means the duration from the time a client sends a TCP connection request to the time the connection is established (or the time taken for the 3-way handshake). The connection time represents not only the efficiency of handling incoming connection requests at server TCP stack, but also how fast the server software can accept established connections. This is because the accept queue and the SYN queue are organized as serially chained buffers. If the web server doesn't accept the established connections fast enough, the following may occur.

1. The accept queue overflows
2. The overflow propagates downward to the SYN queue
3. Server can't ACK newly arrived connection requests
4. The connection time gets longer.

In our experiment, only the architectural efficiency of the server software makes a difference, because we used the same operating system. Thus, we can compare the difference in architecture between our implementation and Apache using the connection time.

Table 1: System configuration

	Client	Server1	Server2
CPU	1 Pentium-3 667MHz	1 Pentium-3 667MHz	2 Pentium-3 667MHz
Main Memory	512MByte	1GByte	1GByte
Network Interface	1Gbps Ethernet	1Gbps Ethernet	1Gbps Ethernet
Operating System	Linux 2.4.2	Linux 2.4.2	Linux 2.4.2

4.3.2 Call Processing Time

The *"call processing time"* — or processing time, in short — means the duration from the time a client sends an HTTP request to the time it begins to receive the HTTP response. Because the client sends an HTTP request as soon as the connection is established, the call-processing-time can be calculated as follows (Shin, Won & Koh 2000).

$$T_{cp} = T_{rt} + T_{accept} + T_{rp} \quad (1)$$

where T_{cp} : call processing time
T_{rt} : round-trip time
T_{ac} : accept delay
T_{rp} : request processing time.

In this equation, T_{rt} can be ignored, because the client machine and the server machine are directly connected.

The use of caches influences the request processing time, but not the accept delay. We can exclude the accept delay from processing time by understanding the HTTP/1.1 keep-alive mechanism.

For the first request in a HTTP/1.1 session, the server must *accept()* the connection and perform request processing. For the rest of the requests, *accept()* is not needed, because the client reuses open connection.

We modified the httperf to separate the summaries for the first request and for the rest from average processing time. Let *accept time* denote the processing time for the first request and *internal processing time* denote the processing time for the rest. Then, the accept time represents the efficiency of accepting established connection, which is largely affected by the server architecture, and the internal processing time represents the preparation overhead, which can be skipped by using cache.

4.3.3 Transmission Time

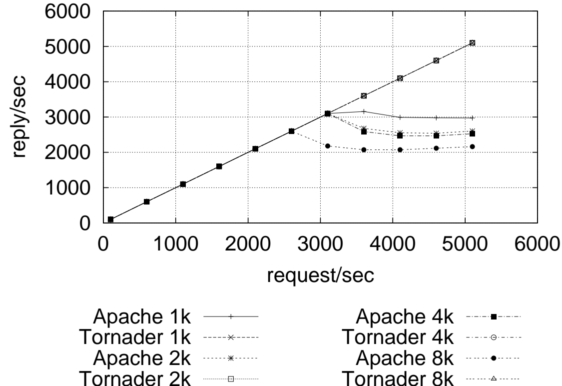
The *"transmission time"* represents the time taken for retrieving the HTTP response. Most traditional web servers, like Apache, intermix the content generation with delivery, while our accelerator only deals with the delivery during the transmission period. Comparison of the transmission time will give us insight on how much performance enhancement can be achieved by separating the delivery operation even when the overhead of content generation is negligible.

5 Benchmark Results

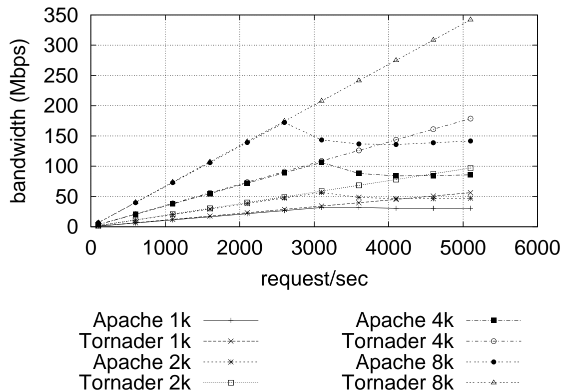
5.1 Throughput

Figure 2 shows the throughput of Server1 (single CPU). Figure 2(a) plots the server's reply rate against the request rate, and Figure 2(b) show the transfer rate.

When the request rate exceeds a certain threshold, the throughput of Apache is degraded due to the system overhead (scheduling and aggressive memory footprint) and the synchronization overhead (lock, mutex, etc). The throughput of Tornader increases almost linearly regardless of the size of the requested object. Under high load (5100 request/sec), the reply rate and the transfer rate of Tornader is almost 250% of those of Apache.



(a) Reply rate



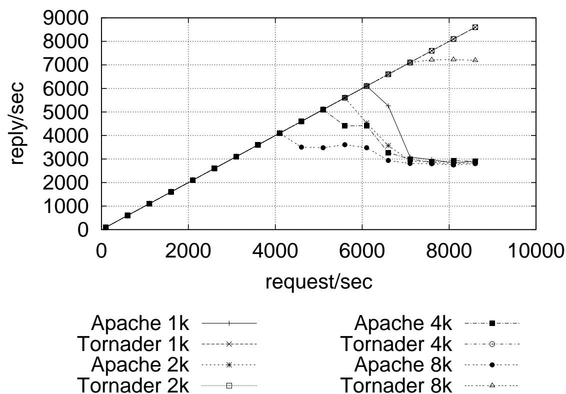
(b) Transfer rate

Figure 2: Server1 throughput

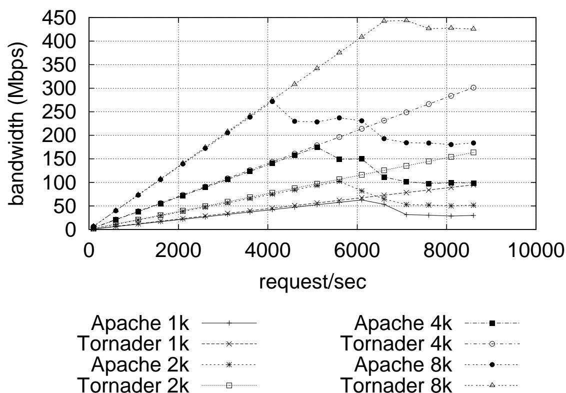
On a single CPU server, Tornader behaves almost like a single-process event-driven server and operates with less overhead than Apache. As mentioned in (Welsh, Gribble, Brewer & Culler 2000), the lightweight event-driven architecture of Tornader gives it better performance and renders it more stable.

5.2 SMP scalability

Figure 3 shows the result for Server2 (dual CPU). The additional CPU power of Server2 benefits both Apache and Tornader, because both MP and AMTED architecture can fully utilize the processing power of multiple CPUs. The overall behaviors of both servers are almost the same as the single CPU case except that the performance degradation of Apache is more noticeable. This is due to the fact that, as the load level increases, the number of processes of Apache also increases and the contention for shared resources (listen socket and locks) becomes in-

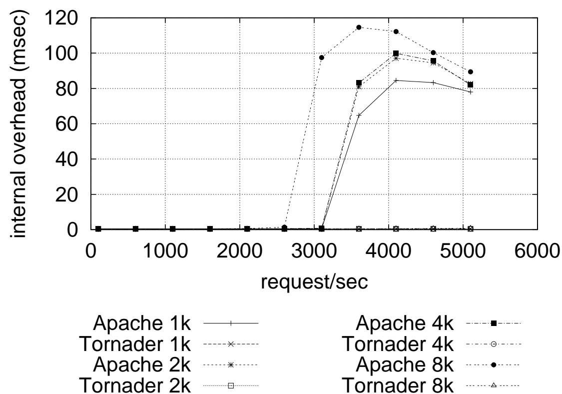


(a) Reply rate

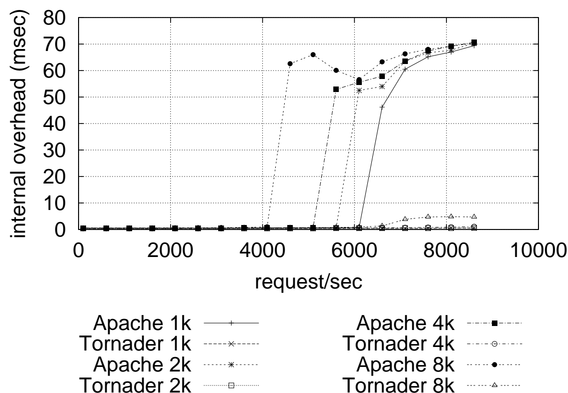


(b) Transfer rate

Figure 3: Server2 throughput



(a) Server1



(b) Server2

Figure 4: Internal processing time

tensified. As a result, the SMP system behaves almost like a single CPU system, because accesses to those resources have to be serialized.

Under heavy load, Tornader boosts the performance of Apache by more than 150%. We confirm that Tornader still shows scalable performance enhancement in a multi-processor system until the network is saturated. Note that the maximum bandwidth of our experimental environment is limited to 500Mbps (measured by *netperf*) and Tornader nearly approaches this limit when the requested object size is 8KByte.

5.3 Effectiveness of the Cache

Figure 4 shows the internal processing time of Server1 and Server2. As we have mentioned earlier, the internal processing time represents the overhead in the preparation phase, which are skipped in Tornader. As shown in Figure 4, the internal processing time of Tornader remains almost the same regardless of the load, because the preparation phase is replaced by simple hash lookup. In contrast, the internal processing time of Apache increases sharply after a certain point. This is mostly due to the increased overhead in system calls for file I/O during the preparation step. As shown in Figure 4(b), the additional CPU power of Server2 lessens this overhead, because each CPU separately processes the system calls.

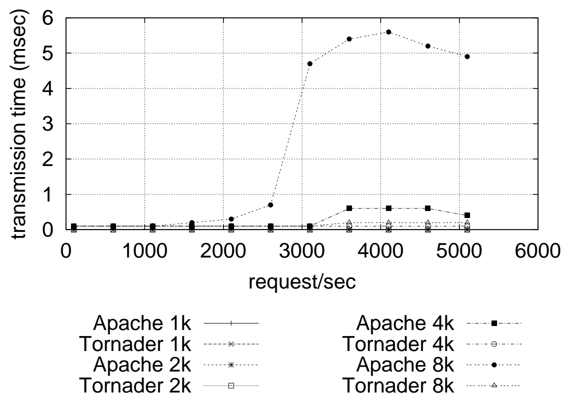
Figure 5 show the transmission time. Although the absolute length of transmission time is not long, there is a noticeable difference between Tornader and Apache. One may argue that this is quite natural because Tornader uses a cache for optimization. However, considering that the requested object is cached either in the object cache (Tornader) or in the virtual memory (Apache), the separation of content delivery is really effective and the effect is more eminent as the load becomes heavier. In Apache, the transmission time suddenly increases at a hyper-linear rate when the load exceeds a certain point due to the cumulative effect of the content generation overhead. The addition of CPU only delays the moment of sudden increase, but it does not contribute to the decrease in overall transmission time.

We carefully conjecture that there will be much more differences when the size of the requested object is large or the complexity of generated contents increases, because as the number of I/O operations for content generation increases, its overhead will increase at an exponential rate under high load.

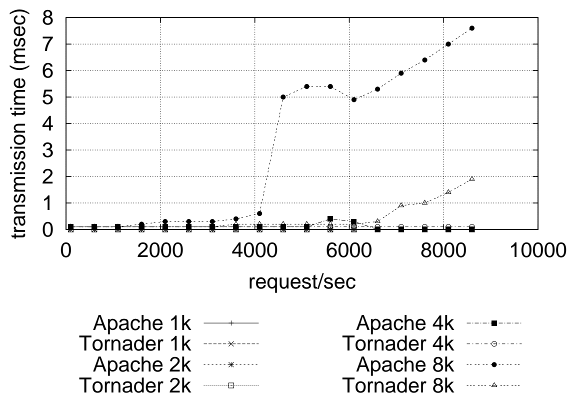
From these results, we can conclude that the use of web object caches to shorten the length of the critical-path and to separate the content delivery really contributes to the decrease of response time.

5.4 Architecture Difference

Figure 6 shows the accept overhead. The accept overhead is calculated by subtracting the average internal processing time from average accept time. For Apache, a non-negligible amount of time is spent in the accept queue, because all of the web server processes contend for shared resources and the number

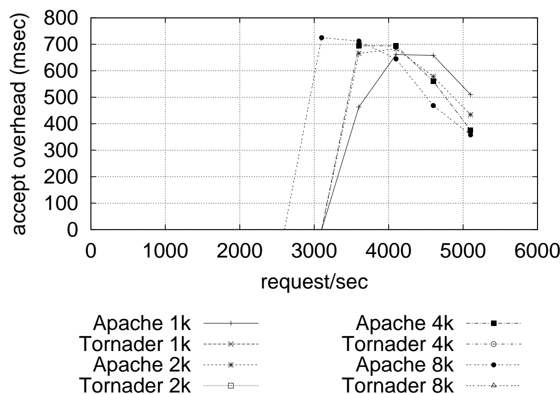


(a) Server1

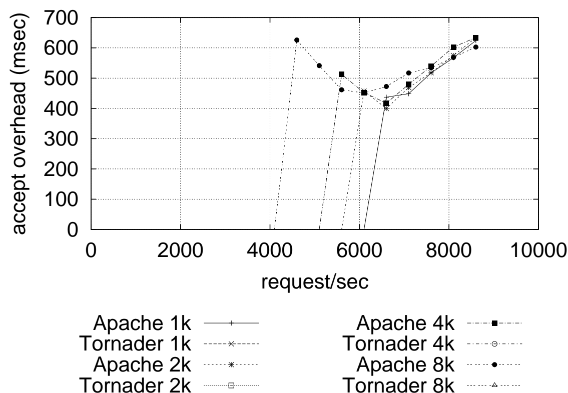


(b) Server2

Figure 5: Transmission time



(a) Server1



(b) Server2

Figure 6: Accept overhead

of processes increases with the request rate. However, the accept overhead of Tornader remains almost the same because it uses fixed number of threads for accepting incoming connection requests. Note that the accept overhead does not continue to increase at a hyper-linear rate, because we configured the timeout of httperf to 5 seconds.

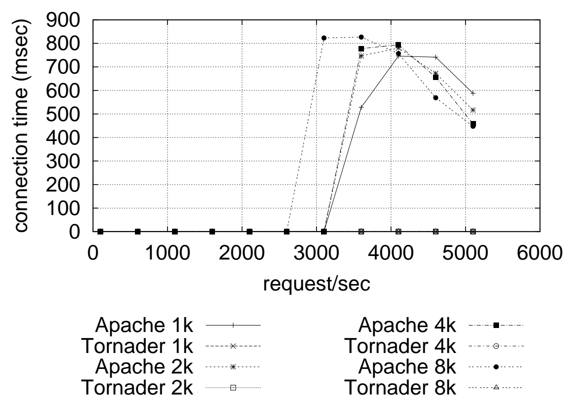
Figure 7 shows the connection time. As the request rate increases, the connection time of Apache suddenly goes up because of the SYN queue overflow. When the SYN queue overflows, the client should retry the connection request with exponentially increasing intervals and this results in longer connection time. As we have mentioned in 4.3.1, the overflow of the accept queue propagates downward to the SYN queue, and this makes the pattern of the connection time graph look similar to the accept overhead graph. The length of connection time is slightly larger than the accept overhead, because it includes the effect of the overflow at both the accept queue and the SYN queue.

From these results, we can conclude that the lightweight architecture of Tornader really contributes to overall performance enhancement, and the overhead introduced by the MP architecture of Apache is not of much benefit, although more CPU power is available.

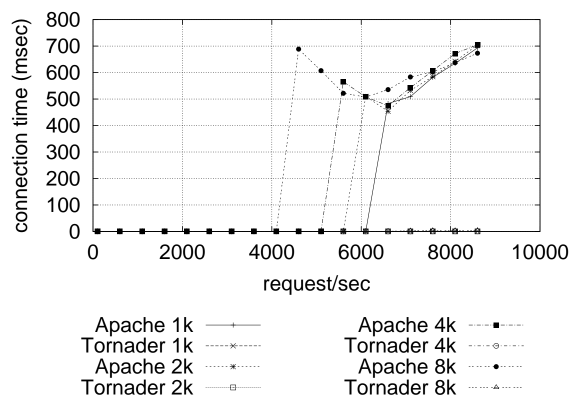
6 Related Work

There are some commercial and non-commercial web server accelerators available — CacheFlow's Server Accelerator (Cacheflow 1998), IBM's Web Server Accelerator (Levy-Abegnoli, Iyengar, Song & Dias 1999), Squid, etc. Most commercial implementations are tightly integrated with their specific hardware or OS platform, which makes them hard to deploy on various platforms. We can use Squid, which is also a portable implementation, as a web server accelerator (Wessel 2001), but it is mostly geared to general (forward) proxy use and does not show scalable performance in multi-processor environments.

The architecture design of the Flash web server (Pai, Druschel & Zwaenepoel 1999) influenced our design to a degree. The asymmetric multi-process event-driven (AMPED) model of the Flash server also uses multiple processes with different roles. On a cached workload, it behaves like a single-process event-driven server, while on a disk-bound workload, it behaves like a multi-process server. However, for cached workloads, Flash does not fully utilize the processing power of the multi-processor system, while ours does. The performance for cached workloads is very important, because the hit-ratio of a typical web server is rather high (Levy-Abegnoli et al. 1999). Tornader, in essence, has nothing to do with disk-related operations (except for access logging), but for disk-bound workload, it can limit the number of back-end connections for efficient operation of the accelerated web server, which contributes to the reduced overhead in disk-related operations of a web server.



(a) Server1



(b) Server2

Figure 7: Connection overhead

7 Conclusion

This paper presents a portable high-performance web server accelerator, called Tornader. It is based on a light-weight high-performance AMTED architecture and uses a cache for separating the role of content delivery from the web server. Also, we applied several optimization techniques for additional performance gains.

Benchmark results show that Tornader boosts the performance of the Apache web server up to 150% in both single CPU and dual CPU cases. It also shows scalable performance in a multi-processor system. The analysis of the benchmark results verifies that this is due to the efficient architecture of Tornader which is optimized for content delivery.

Although our experiment does not precisely reflect real-world situations, the results of our work reveal that there is a good chance of success in using a web server accelerator for optimizing content delivery to enhance the performance of a web server without sacrificing the required features.

8 Acknowledgement

This work was supported by Brain Korea 21 project and conducted jointly with Clunix, Inc., Seoul, Korea.

References

Banga, G., Mogul, J. C. & Druschel, P. (1999), A scalable and explicit event delivery mechanism for unix, *in* 'Proceedings of the 1999 USENIX Annual Technical Conference', Monterrey, California.

Barford, P. & Crovella, M. (1998), Generating representative web workloads for network and server performance evaluation, *in* 'Proceedings of the ACM SIGMETRICS '98', Madison, Wisconsin.

Cacheflow (1998), *High-performance web caching white paper*, World Wide Web, http://www.cacheflow.com/technology/wp/.

Hu, Y., Nanda, A. & Yang, Q. (1999), Measurement, analysis and performance improvement of apache web server, *in* 'IEEE International Performance, Computing, and Communications Conference', Phoenix, Arizona.

Levy-Abegnoli, E., Iyengar, A., Song, J. & Dias, D. (1999), Design and performance of a web server accelerator, *in* 'In Proceedings of the IEEE Infocom '99 Conference', New York, New York.

Mosberger, D. & Jin, T. (1998), httperf - a tool for measuring web server performance, *in* 'Workshop on Internet Server Performance', Madison, Wisconsin.

Pai, V., Druschel, P. & Zwaenepoel, W. (1999), Flash: An efficient and portable web server, *in* 'In Proceedings of the 1999 USENIX Annual Technical Conference', Monterrey, California.

Shin, D., Won, Y. & Koh, K. (2000), Measurement and analysis of web server response time, *in* 'In Proceedings of the 6th Asia-Pacific Conference on Communications', Seoul, Korea.

Welsh, M., Gribble, S. D., Brewer, E. A. & Culler, D. (2000), A design framework for highly concurrent systems, Technical report, UC Berkeley CS.

Wessel, D. (2001), *Squid Frequently Asked Questions*, World Wide Web, http://www.squid-cache.org/Doc/FAQ/.