Design and Performance of a Web Server Accelerator

Eric Levy-Abegnoli[*], Arun Iyengar, Junehwa Song, and Daniel Dias
IBM Research
T. J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598

Abstract

We describe the design, implementation and performance of a Web server accelerator which runs on an embedded operating system and improves Web server performance by caching data. The accelerator resides in front of one or more Web servers. Our accelerator can serve up to 5000 pages/second from its cache on a 200 MHz PowerPC 604. This throughput is an order of magnitude higher than that which would be achieved by a high-performance Web server running on similar hardware under a conventional operating system such as Unix or NT. The superior performance of our system results in part from its highly optimized communications stack. In order to maximize hit rates and maintain updated caches, our accelerator provides an API which allows application programs to explicitly add, delete, and update cached data. The API allows our accelerator to cache dynamic as well as static data. We analyze the SPECweb96 benchmark, and show that the accelerator can provide high hit ratios and excellent performance for workloads similar to this benchmark.

1 Introduction

The performance of Web servers is limited by several factors. The underlying operating system on which a Web server runs may have performance problems which negatively affect the throughput of the Web server. In satisfying a request, the requested data are often copied several times across layers of software, such as between the file system and the application and again during transmission to the operating system kernel, and often again at the device driver level. Other overheads, such as operating system scheduler and interrupt processing, can add further inefficiencies. One technique for improving the performance of Web sites is to cache data at the site so that frequently requested pages are served from a cache which has significantly less overhead than a Web server. Such caches are known as *httpd accelerators* [5] or *Web server accelerators*. Httpd accelerators differ from proxy caches in that the primary purpose of an httpd accelerator is to speed up accesses to a local Web site whereas the primary purpose of a proxy cache is to speed up accesses to remote Web sites by storing data from the remote sites. It is possible for a cache to function as both a proxy cache and an httpd accelerator.

Our accelerator runs under an embedded operating system and can serve up to 5000 pages/second from its cache on a 200 MHz PowerPC 604. This throughput is an order of magnitude higher than that which would be achieved by a high-performance Web server running on similar hardware under a conventional operating system such as Unix or NT. The superior performance of our system results largely from the embedded operating system and its highly optimized communications stack. Buffer copying is kept to a minimum. In addition, the operating system does not support multithreading. The operating system is not targeted for implementing general-purpose software applications because of its limited functionality. However, it is well-suited to specialized network applications such as Web server acceleration because of its optimized support for communications.

In order to maximize hit rates and maintain updated caches, our accelerator provides an API which allows application programs to explicitly add, delete, and update cached data. Consequently, we allow dynamic Web pages to be cached as well as static ones, since applications can explicitly invalidate any page whenever it becomes obsolete. Caching of dynamic Web pages is essential for improving the performance of Web sites containing significant dynamic content [10, 9, 4, 3].

Httpd accelerators are contained in both the Harvest and Squid caches [5, 14]. Our httpd accelerator results in considerably better performance than the Harvest and Squid accelerators partly because our accelerator runs on an embedded operating system. Novell sells an httpd accelerator as part of its BorderManager product [11]. A key difference between our accelerator and previous ones is that applications can explicitly cache and invalidate data. In addition, we support caching of both static and dynamic Web pages, since applications can explicitly invalidate any page whenever it becomes obsolete. The Harvest, Squid, and Novell accelerators do not allow applications to explicitly cache and invalidate data. In addition, they don't allow dynamic pages to be cached.

1.1 Outline of Paper

Section 2 describes the architecture of our system. Section 3 presents performance measurements we have made of our accelerator. In Section 4 we present an analysis of the SPECweb96 benchmark, and analyze cache hit ratios and accelerator throughput for this benchmark. Finally, a summary and concluding remarks appear in Section 5.

2 Web Server Accelerator Design and Characteristics

2.1 System Description

As illustrated in Figure 1, the accelerator front-ends a set of Web server nodes. Multiple Web servers would be needed at a Web site which receives a high volume of requests. IBM's Network Dispatcher (ND) [6, 8] runs on the same node as the accelerator (although it could also run on a separate node). ND presents a single IP address to clients regardless of the number

[*]Author's current address: IBM France, LE-PLAN-DU-BOIS, 06610 La Gaude, FRANCE

of back-end servers and routes Web requests to the accelerator cache. If the requested page is found in the cache, the page is returned to the client. Otherwise, ND routes the request to a back-end Web server node using either round-robin or a method which takes server load into account. Use of ND for routing cache misses to the server nodes results in better load balancing than using the round-robin Domain Name Server [6]. Our accelerator can reduce the number of Web servers needed at a Web site since, as quantified later, a large fraction of the Web requests are handled by the accelerator cache.

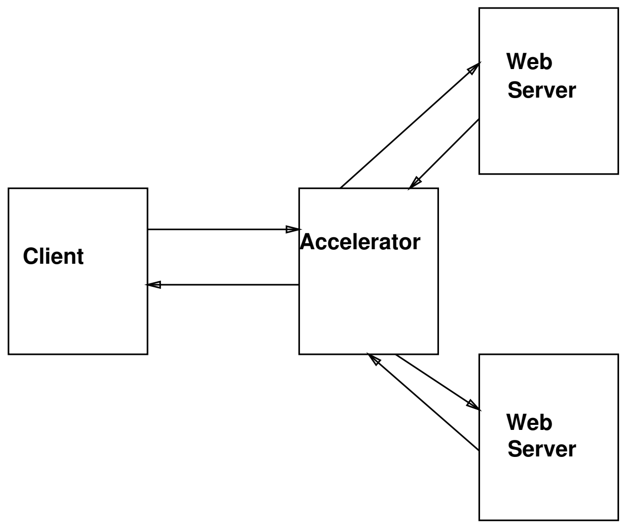


Figure 1: The accelerator front-ends one or more Web servers. Cached objects are sent directly from the accelerator to clients.

The accelerator examines each request to see if it can be satisfied from its cache. This requires the accelerator to terminate the connection with the client. Consequently, there is no way to forward a request directly to a server after a cache miss. Instead, the accelerator has to request the information from a server and send the information back to the client. Caching thus introduces some overhead in the event of a cache miss because the accelerator must now function as a proxy for the client. By contrast, when caching is turned off, responses to requests routed by ND are returned directly from servers to clients without going through ND on the return path [6].

A benefit in examining the request is that the accelerator can now perform content-based routing [15] in which it makes intelligent decisions about where to route requests based on the URL. For example, the accelerator could send all requests for static pages to one set of servers and all requests for dynamic pages to another set of servers. In other situations where the contents of the servers are not all identical, the accelerator could employ more sophisticated algorithms for routing requests based on the URL.

The cache operates in one or a combination of two modes: automatic mode and dynamic mode. In automatic mode, data are cached automatically after cache misses. The Webmaster sets cache policy parameters which determine which URL's are automatically cached. For example, different cache policy parameters determine whether static image files, static nonimage files, and dynamic pages are cached and what the default lifetimes are. HTTP headers included in the response by a server can be used to override the default behavior specified by cache policy parameters. These headers can be used to specify both whether the contents of the specific URL should be cached and what its lifetime should be.

In dynamic mode, the cache contents are explicitly controlled by application programs which execute either on the accelerator or a remote node. API functions allow application programs to cache, invalidate, query, and specify lifetimes for the contents of URL's. While dynamic mode complicates the application programmer's task, it is often required for optimal performance. Dynamic mode is particularly useful for prefetching hot objects into caches before they are requested and for invalidating objects whose lifetimes are not known at the time they are cached.

The presence of an API for explicitly invalidating cached objects often makes it feasible to cache dynamic Web pages. Web servers often consume several orders of magnitude more CPU time creating a dynamic page than a comparably sized static page. For Web sites containing significant dynamic content, it is essential to cache dynamic pages to improve performance [10, 9, 4, 3]. We are not aware of any httpd accelerator besides our own which allows dynamic pages to be cached.

All cached data must be stored in memory. Caching objects on disk would slow down the accelerator too much. Consequently, cache sizes are limited by memory sizes. Our accelerator uses the least recently used (LRU) algorithm for cache replacement.

2.2 Key Software Elements

The embedded operating system on which our cache runs contains a multi-layered collection of networking software which performs inter and intra network protocol packet forwarding over various hardware network interfaces. The operating system also provides process scheduling, timer services, buffer and memory management, and configuration and monitoring facilities.

The principal packet forwarding software corresponds to the first three layers of the OSI reference model. Minimal layer four transport and application functionality is also available which allows remote login services for management and monitoring. We extended and optimized layer four so that the accelerator could offer fast TCP applications such as the cache.

Key elements of the architecture which result in good performance include the following:

1. The device drivers fill in a packet queue on the system card memory with incoming packets. The system processor dequeues these packets at a high rate (it is not interrupted on packet arrival).

2. The accelerator performs its functions without performing task scheduling, task switches, or interrupts.

3. From the time a packet is queued by the network handler until the complete stack has processed it (up to the cache when applicable), no data copying takes place.

4. The queue elements that contain packets are sized so that no buffer linking is necessary. Any packet size that the accelerator receives can fit in one buffer. This saves the overhead of buffer linking (at the cost of wasted memory space, and the need to restart the system when network parameters such as MTU are changed).

These architectural features result in efficient IP forwarding. The combination of a lightweight operating system, a copyless path from input queue to output queue, a polling mechanism based on input/output queues with a tasker scanning these queues at high rates (no interrupt overhead) to feed a network handler, a single buffer data structure (no linked lists such as the mbufs used in general-purpose operating systems), and the absence of context switches significantly reduce the overhead of the system. Our system can route about 80,000 IP packets per second compared to 10,000 for a router running a general-purpose operating system on the same hardware.

2.3 TCP stack

In order to obtain optimal cache performance, it was necessary to modify the TCP stack on the initial embedded system which we started out with and modified to produce our accelerator. The TCP stack in the initial embedded system was initially designed solely for the purpose of providing remote login services for management and monitoring. Consequently, it contained a number of inefficiencies such as task scheduling and unnecessary data copying. The TCP stack was modified to use the same "scheduling logic" as the one applied to the IP path which was already optimized. This eliminated all task scheduling during TCP packet processing.

The system was also modified to use the same data structure for both TCP and IP, with the device driver I/O buffer in one contiguous piece so that no linkage of multiple buffers is necessary. The device drive I/O buffer is passed along from one layer to the next so that a new copy is not necessary.

Figure 2 shows the overall forwarding path for TCP. The path length for processing a TCP flow is far greater than what it is for IP. For example, processing an IP packet takes on average 200 instructions and 400 cycles on a PowerPC 604, while processing a TCP SYN packet can take up to 3000 instructions and 10,000 cycles. While one big constraint of the IP path is low latency, it is not possible to achieve this using TCP. In order to understand the consequences and verify that this issue is not critical for TCP, let us analyze the reasons for this constraint.

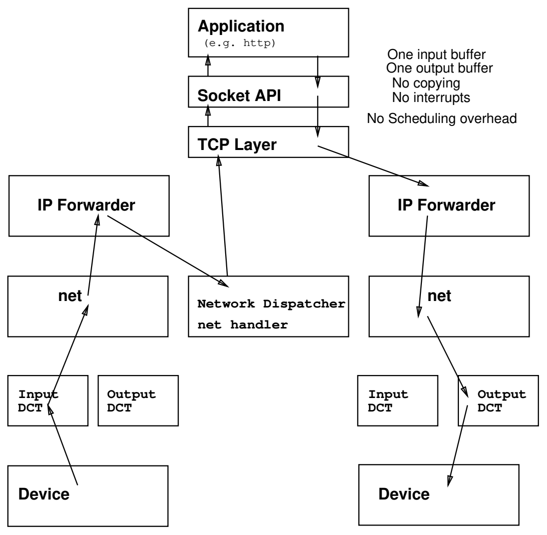


Figure 2:

IP is a connectionless datagram protocol. It does not contain flow control of any sort. Its only degree of freedom is to drop packets when they are arriving too fast. This is what happens when the forwarder is taking too long to process a packet; the input queue fills up (faster than the network handler can empty it), and finally, incoming packets get dropped. Unfortunately, when this happens, TCP running at each endpoint of the connection becomes fairly inefficient, trying to resend dropped packets, closing windows, etc., which results in poor overall connection throughput. This is why most routers tend to be able to run at media speed. That way, they can always process packets faster than they arrive (dropping a few packets is acceptable; problems arise when a significant number of packets are dropped).

With TCP, it becomes very difficult to run at media speed, and one would expect the phenomenon just described to result in an inefficient system. However, in the case where TCP is terminated inside the embedded system, there is a flow control that can regulate the flow of incoming packets provided by TCP itself. This flow control will naturally tend to close the window to slow down the data flow on a particular connection, with the effect that the source will send less traffic to the accelerator. In addition, because the application (such as the cache) is also sitting in the accelerator and engages in "query-response" exchanges, the source will wait for a response before sending new packets. This will further regulate the amount of data the accelerator receives. Combining these effects, the accelerator will receive pretty much what it can process.

3 Web Server Accelerator Performance

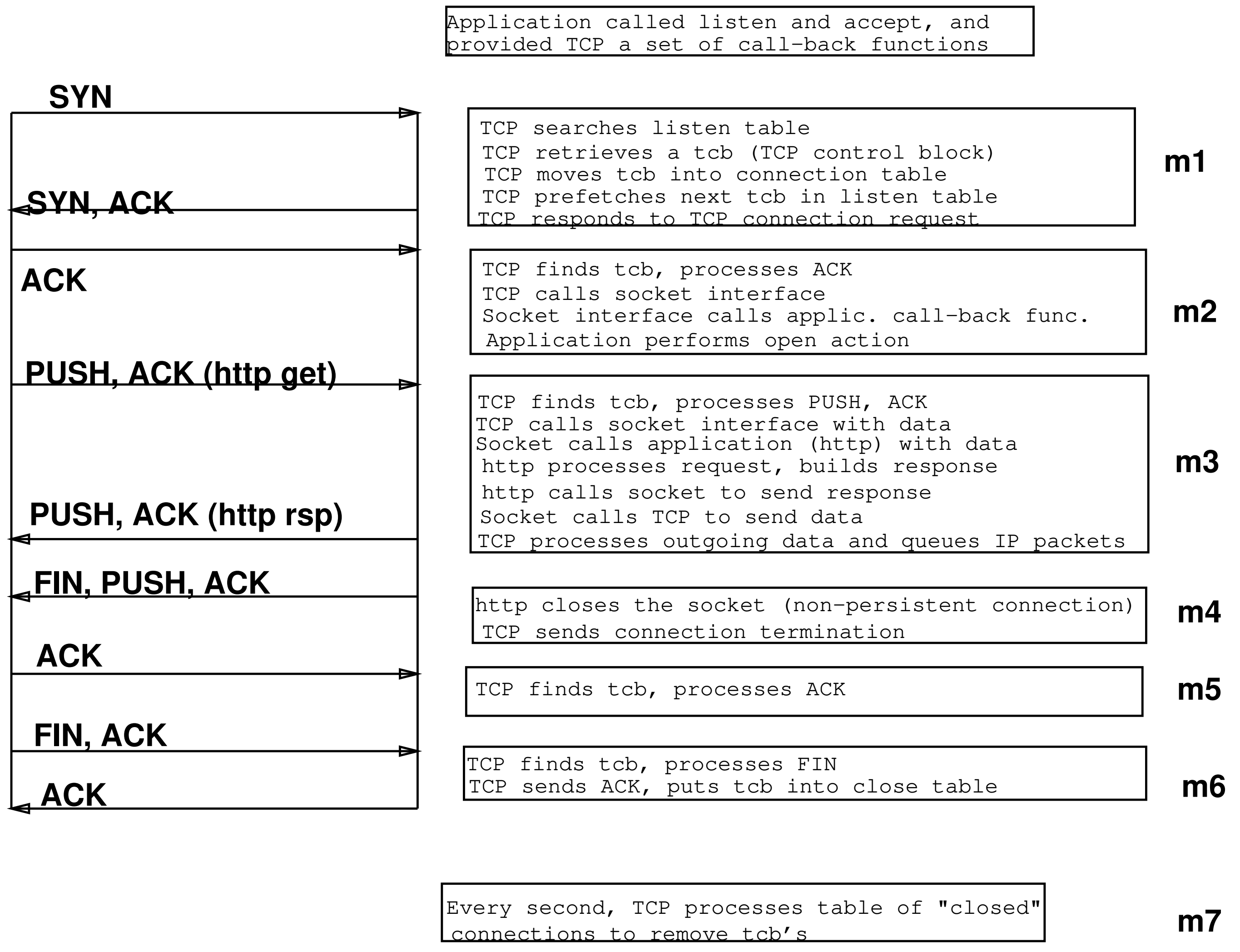
3.1 Path Length Measurements

Measuring path length is an accurate way to evaluate the capabilities of the system when certain conditions are present. Assuming two measurement points (one on the input, one on the output) can be precisely identified, and that every time the first point of measurement is hit for a given packet, the second point of measurement is also hit as a direct effect of the processing of this packet, the measure will give a value (in number of instructions and number of cycles) of the path length for processing this packet. Furthermore, when the processing that is intended to be measured does not involve any blocking mechanism, scheduling, timing etc., and consists only of a succession of function calls, this measure can help to evaluate the capabilities of the system, as well as compare several implementations. A reasonable number of samples must be used in order to obtain accurate average values, and the complex flows must be broken into elementary flows that satisfy the condition described above. Since the system under test is precisely and intentionally separated from any operating system involvement (no scheduling, no blocking, no timing, no copying), the problem was reduced to breaking the TCP/application flows into measurable elementary pieces, setting up the measurement points, and generating sufficient traffic to obtain enough samples (PowerPC registers give both the number of cycles and numbers of instructions, as well as details on cache hits and misses). The TCP flow is depicted in Figure 3.

Request traffic was generated by WebStone which is a benchmark which measures the number of requests per second which a Web server can handle by simulating one or more clients and seeing how many requests can be satisfied during the duration of the test [13].

Table 3.1 shows the measurements we obtained for the components of the TCP flow depicted in Figure 3. Component $m3$ is the only one which varies significantly with data size. In order to have accurate measurements, the MSS/MTU were set to big enough values so that 8K packets were not fragmented by IP or TCP.

For a 200 MHz PowerPC 604 processor, the theoretical capability would be 200,000,000 / 32,823 for an 8 Kbyte page which is 6093 requests per second. In practice, several factors



Application called listen and accept, and provided TCP a set of call-back functions
SYN
SYN, ACK
ACK
PUSH, ACK (http get)
PUSH, ACK (http rsp)
FIN, PUSH, ACK
ACK
FIN, ACK
ACK
TCP searches listen table
TCP retrieves a tcb (TCP control block)
TCP moves tcb into connection table
TCP prefetches next tcb in listen table
TCP responds to TCP connection request
m1
TCP finds tcb, processes ACK
TCP calls socket interface
Socket interface calls applic. call-back func.
Application performs open action
m2
TCP finds tcb, processes PUSH, ACK
TCP calls socket interface with data
Socket calls application (http) with data
http processes request, builds response
http calls socket to send response
Socket calls TCP to send data
TCP processes outgoing data and queues IP packets
m3
http closes the socket (non-persistent connection)
TCP sends connection termination
m4
TCP finds tcb, processes ACK
m5
TCP finds tcb, processes FIN
TCP sends ACK, puts tcb into close table
m6
Every second, TCP processes table of "closed" connections to remove tcb's
m7

Figure 3:

Flow	Description	Instructions	Cycles	Size
m1	connection request from client	2,778	8,589	N/A
m2	end of connection setup	2,770	5,409	N/A
m3	http request received and served	3,448	8,221	64 bytes
m3	“	3,608	8,330	128 bytes
m3	“	3,707	8,460	256 bytes
m3	“	3,990	7,280	1K bytes
m3	“	4,310	9,600	2K bytes
m3	“	4,608	8,740	4K bytes
m3	“	4,730	10,990	8K bytes
m4	server initiates connection end	2,041	2,933	N/A
m5	client acknowledgement	678	1,163	N/A
m6	client terminates connection	1,545	2,349	N/A
m7	server deletes connection record	1,330	1,390	N/A
m1, m7	complete request	15,812	32,823	8 Kbytes

Table 1: The flows m1-m7 correspond to the boxes in Figure 3.

degrade this number, as will be seen in the subsequent results in terms of the number of requests per second measured. First, as the number of connection records in the accelerator increases, so does the time to retrieve a connection control block, an operation performed an average of six times per connection. In order to reduce this dependency, a large hash table (256,000 buckets) and an efficient hashing function (98% of the buckets occupied with 256,000 connections) were used. Consequently, even with as many as 150,000 connection records at any given time of the test, few collisions occurred.

A second degradation due to the number of concurrent connection records is the "connection record cleanup latency". Every second, one out of every 30 connections are examined for possible deletion. With 150,000 connection records, 5000 will potentially go through the $m7$ flow, resulting in 5000 * 1390 or about seven million cycles. During that time (which is the worst case latency of the system), many packets will arrive and be dropped because the system processor is not dequeueing the input queue. Dropping packets has a negative effect on the overall throughput of the system. In order to reduce this problem, the frequency with which the *wait_close* connections were examined as candidates for dropping was reduced. In addition, timer management was improved so that fewer connections had to be scanned.

Finally, because of the latency of the http request and response ($> 10,000$ cycles), when the number of packets received was high (corresponding to high request rates), a significant number of packets (but less than one percent) were dropped by the device because the input queue was full. As mentioned earlier, dropping packets has an effect on overall throughput which is greater that just the percentage of dropped packets. This factor also contributed to reducing the maximum throughput of the system from the theoretical maximum. Despite all of these factors, the measured capacity of the system was within about 80% of the theoretical limit as we show in the next section.

3.2 Web Server Accelerator Throughput

The system used to measure the Web accelerator throughput is illustrated in Figure 4. It consisted of two SP2 frames containing a total of 16 nodes which were connected through four 16 Mbit/s token rings to the accelerator. The SP2 nodes issued requests to the accelerator by running WebStone. A second accelerator functioned both as a Web server for handling cache misses as well as a client in order to issue additional requests to the first accelerator's cache. The two accelerators were connected via four 16 Mbit/s token rings. The total I/O bandwidth to the accelerator under test was thus 128 Mbit/s, half from the SP2 frames and half from the other accelerator. Each SP2 node typically ran about 20-40 WebStone clients at a time. The second accelerator ran up to 100 WebStone clients at a time.

Figure 5 shows the number of cache hits per second the accelerator can sustain as a function of requested page size. For pages smaller than 2 Kbytes, the accelerator was the bottleneck. For pages larger than 2 Kbytes, the network was the bottleneck. Since the network becomes the bottleneck for requested pages greater than 2 Kbytes, it is useful to estimate the throughput attainable for larger sizes assuming a higher bandwidth network and the path lengths presented previously. If we assume that the maximum segment size is 2 Kbytes and that the network is not the bottleneck, the path length for sending any additional 2 Kbytes is on the order of 3000 cycles. Each 2 Kbyte delta involves one or two additional packets, some minimum TCP processing, but no socket, cache processing, or data copying. For instance, a request for 20 Kbytes will require another 30,000 cycles, doubling the path length and reducing the throughput by a factor of two. The resulting projections are shown in Figure 6.

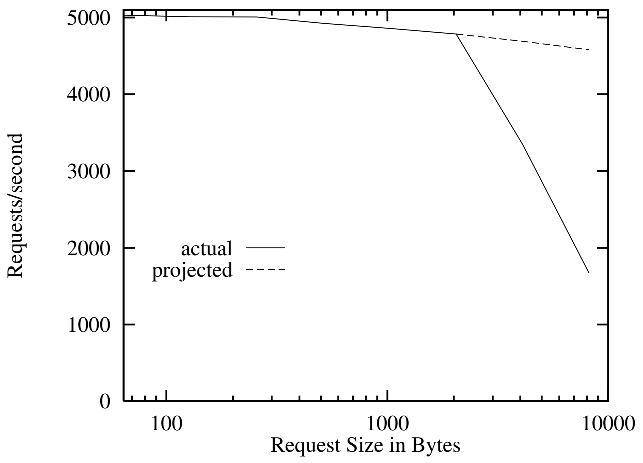


Figure 5: The number of cache hits per second our system can sustain and the projected number which would be expected if the network were not a bottleneck.

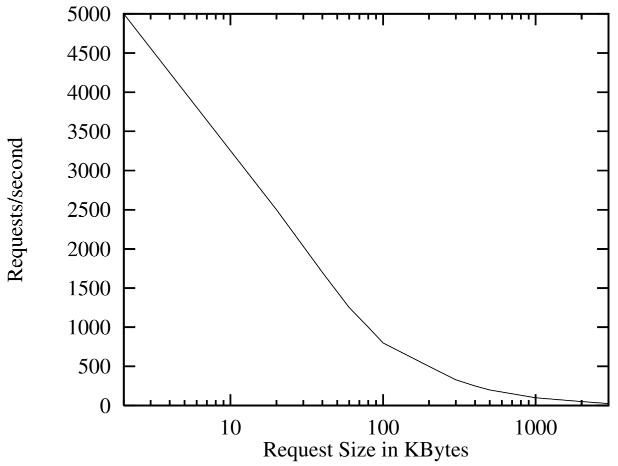


Figure 6: The projected number of cache hits per second our accelerator can sustain for large pages.

A cache miss for a page of 8 Kbytes or less consumes around 100 Kcycles. In the event of a cache miss, the accelerator must request the information from a back-end server before sending it back to the client. Requesting the information from a server requires considerably more instructions than fetching the object from cache. If the miss rate is 100%, the accelerator can serve about 2000 pages per second before its CPU is 100% utilized.

Our own measurements as well as published performance reports on Web servers [13] indicate that Web servers running under Unix or NT on hardware of similar capacity to that of our accelerator can serve a maximum of several hundred pages per second, an order of magnitude less than the rate achieved by our accelerator. The throughput of Web servers decreases with

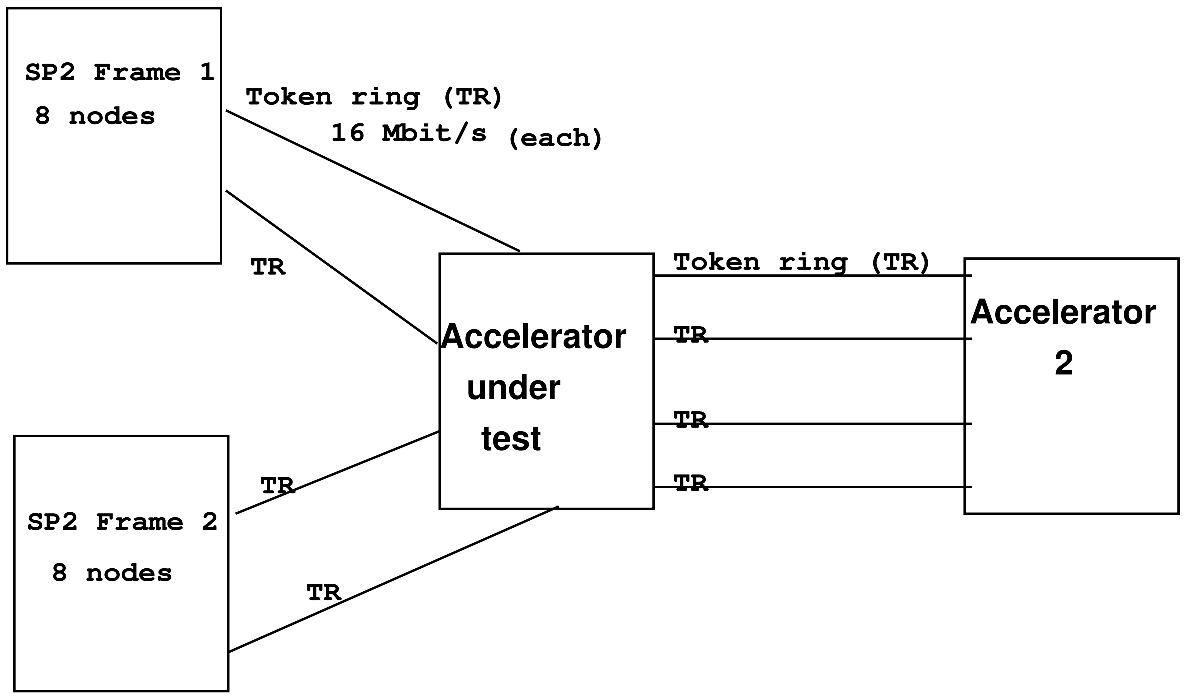


Figure 4: The system used to test our accelerator's performance.

increasing page size. In addition, the presence of dynamic Web pages can hurt performance even more. We have encountered several commercial Web sites where a single request for a dynamic page typically consumes several seconds of CPU time.

The API our accelerator provides which allows an application program to explicitly control the contents of the cache makes it feasible to cache dynamic data in many situations. Furthermore, our accelerator serves dynamic data at the same high rate at which it serves static data. Consequently, our cache can often speed up the rate at which dynamic data is served by several orders of magnitude compared with a single order of magnitude for static pages.

The overall performance of a system deploying our cache is summarized in Figure 7. Each curve represents a back-end server configuration with a different capacity. For example, the curve marked *WST 1000 ops/sec* corresponds to a system which can handle 1000 cache misses per second. In order to obtain a back-end server configuration of this capacity, it may be necessary to place multiple servers behind the accelerator. For Web sites which generate significant dynamic content, it is not uncommon to have server throughputs of well below 100 requests per second.

4 Caching and SPECweb96

In this section, we analytically determine the cache hit rates which one would expect to see when using the industry-popular standard benchmark system, SPECweb96. Our results are applicable to other systems deploying Web caching in addition to our accelerator. We also use this analysis to determine the SPECweb96 throughput our accelerator can sustain.

SPECweb96 estimates system performance by measuring the average server response times while the requested through-

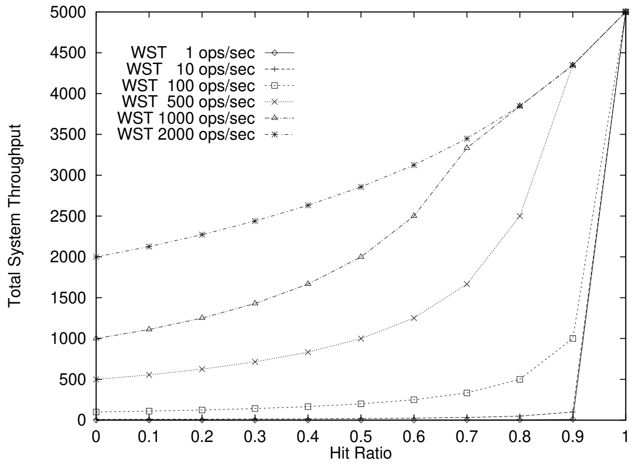


Figure 7: The overall performance of a system utilizing our accelerator.

put levels vary [16]. However, for a system such as ours, the cache hit ratio is also critical to understanding performance. One important aspect to note is that, in SPECweb96, the workload data size (and hence the required amount of disk space) varies with the levels of the requested throughput. This means that the cache hit ratio also changes along with the throughput level. In the following subsection, we analyze the workload of SPECweb96 and see its effect on the cache hit ratio. As a conclusion, we will see that, even with a small cache, high hit ratios can be achieved under the SPECweb96 workload.

4.1 SPECweb96 Workload

The workload in SPECweb96 is structured at three different levels, i.e., directories, classes, and files. At the top level of the SPECweb96 data set are the different directories. Each directory includes 36 data files of which the total size is about 5 MB. The number of directories is a function of the expected throughput, and is given by:

$$10 * \sqrt{\frac{T_p}{5}}$$

where T_p is the expected throughput.

From this equation, the total size of the data set is scaled with the expected throughput level of a server. This is to reflect the expectation for a high-end server compared to that for a lower-end server. A high-end server will be expected to serve a larger number of files as well as to serve at higher request rates. Also, the number of directories grows sublinearly (the total data size doubles as the expected throughput quadruples), accounting for the possible overlap of files among different requests. Once the number of directories are fixed, the accesses to the different directories are uniformly distributed.

Within each directory, there are four different classes: class 0 through class 3. These different classes represent the bins for data files of different size ranges. Also, different classes are assigned different access frequencies. Class 0 includes files of size less than 1KB and accounts for 35% of all requests. Class 1 includes files between 1 KB and 10 KB and accounts for 50% of the requests. Similarly, class 2 includes files between 10 KB and 100 KB with 14% of the requests, and finally class 3 includes files between 100 KB and 1 MB with 1% of requests. This class structure reflects the relationships, based on observation and analysis of logs from several servers, between the file sizes and access frequencies.

Within each class, there are 9 files of discrete sizes with a uniform step. For example, class 1 includes 9 different files of sizes 1 KB, 2 KB, and so on, up to 9 KB. The accesses within each class are not evenly distributed; they are allocated using a Poisson distribution centered around the midpoint within each class. The resulting access pattern mimics the behavior where some files are more popular than the rest, and some files are rarely requested

The data files in SPECweb96 can be grouped into 36 different bins of the same size by separating each data file within each class. Each bin includes all files of the same size from all directories. Then, each bin has the access probability as shown in Table 2. Note that in the specification of the SPECweb96 workload, it is not clearly stated what the parameter for the Poisson Distribution used for the access for each file in a class is. Therefore, we assume the most likely case which is $alpha = 4.0$.

4.2 Hit Ratio Analysis

Given a cache size and expected target throughput, the average hit ratio can be estimated by applying the model in [1].

	Class 0	Class 1	Class 2	Class 3
Size 0	0.0066	0.0094	0.0026	0.0002
Size 1	0.0262	0.0374	0.0105	0.0007
Size 2	0.0524	0.0749	0.0210	0.0015
Size 3	0.0699	0.0998	0.0279	0.0020
Size 4	0.0699	0.0998	0.0279	0.0020
Size 5	0.0559	0.0799	0.0224	0.0016
Size 6	0.0373	0.0532	0.0149	0.0011
Size 7	0.0213	0.0304	0.0085	0.0006
Size 8	0.0106	0.0152	0.0043	0.0003

Table 2: Each cell represents the access probability for a different file size.

We assume that LRU is used for cache replacement. We also assume that all data at the Web site can be cached. Let N be the number of requests. First, the average hit ratio $H_{i,j}(N)$ for a bin $b_{i,j}$ is estimated as

$$H_{i,j}(N) \approx 1 - (1 - \frac{1}{d})^{P_{i,j}N}$$

where d is the number of directories in the SPECweb96 data set and $P_{i,j}$ is the access probability for each bin as shown in Table 2. Then, the cache size required for the bin $b_{i,j}$ is estimated as

$$B_{i,j}(N) \approx S_{i,j} H_{i,j}(N)$$

where $S_{i,j}$ is the size of $b_{i,j}$. Now, the overall cache hit ratio is estimated as

$$H(N) \approx \sum_{i,j} P_{i,j} H_{i,j}(N).$$

This approximation applies under the condition that the number of requests is such that the total required cache size does not exceed the available cache size B. That is,

$$\sum_{i,j} B_{i,j}(N) \leq B.$$

We have analyzed the expected hit ratios when we vary the throughput and cache size. The results are shown in Figure 8 and are applicable to any system utilizing Web caching, not just to our accelerator. The analysis is only meaningful up to the maximum throughput level which a system can sustain. The vast majority of systems in deployment today would be saturated at significantly less than 5000 operations/second. The figures show that high hit ratios can be achieved with caches larger than or equal to 256 MBytes, in the throughput level less than 5000 operations per second. The hit ratio drops with smaller cache sizes. However, with a 128 MB cache, the hit ratio is still about 90% for the throughput level of 2000 operations per second.

The byte hit rates (i.e. fraction of bytes served from the cache) are shown in Figure 9.

The high hit ratio was achieved due to the sharp skewness of the file sizes and their access frequencies. That is, the cache space was mostly occupied by a large number of small files which were frequently accessed. However, thus far, the cache space was allocated for all file bins according to their file sizes

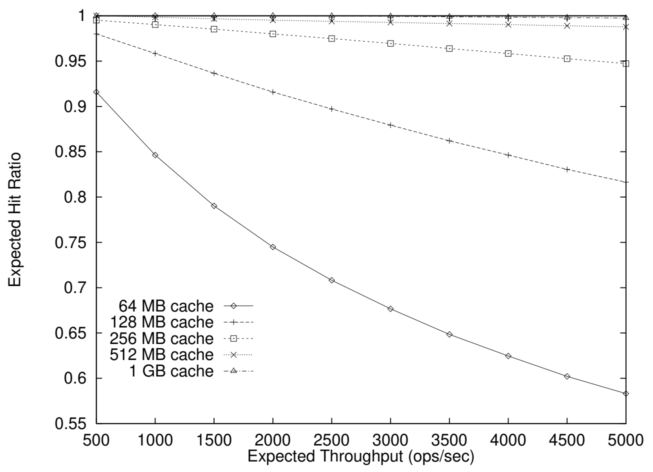


Figure 8: Hit rates as a function of expected throughput on SPECweb96 using LRU. This graph is applicable to other Web caches in addition to our accelerator.

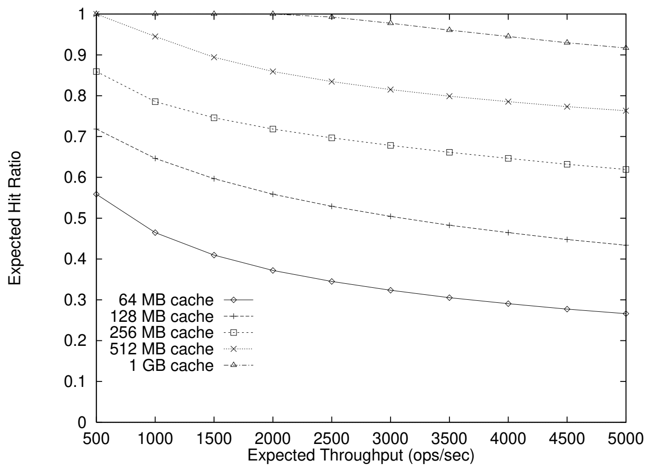


Figure 9: Byte hit rates as a function of expected throughput on SPECweb96 using LRU. This graph is applicable to other Web caches in addition to our accelerator.

and access frequencies. This implies that higher hit ratios may be possible by restricting the cache only to files smaller than a certain size. In Figure 10, we show the estimated cache hit ratios under the condition that the maximum cacheable file size is 500 MB.

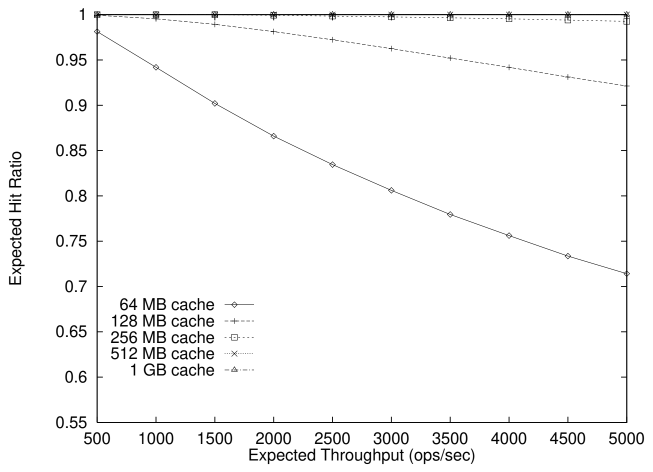


Figure 10: Hit rates as a function of expected throughput on SPECweb96 using LRU when files 500 Kbytes or larger are not cached.

Comparing Figure 10 with Figure 8, noticeable improvements of hit ratios have been achieved, especially for relatively small caches. When the cache size is 64 MB, the hit ratio went up to around 0.98 from around 0.91 at the throughput level of 500 operations per second and up to around 0.72 from around 0.58 at the throughput level of 5000 operations per second.

4.3 Accelerator Performance on SPECweb96

Finally, we put together the throughput measurements of Section 3 with the hit ratio estimates from Section 4.2 and estimate the accelerator's throughput for different cache sizes. The results are summarized in Figure 11. The curves in Figure 11 corresponding to different cache sizes are the same as those in Figure 8. The line intersecting the curves represents the throughput the accelerator can sustain for workloads similar to those of SPECweb96 before the accelerator's CPU becomes 100% utilized.

The maximum SPECweb96 throughput which the accelerator can sustain is obtained from the intersection of the line with the appropriate curve. For example, if the accelerator contains a 256 MByte cache, the maximum SPECweb96 throughput which the accelerator can sustain would be around 4000 operations/second at which point the hit ratio would be around 0.95. Note that these throughput projections do not include all aspects of the SPECweb96 benchmark, such as logging, which will cause some degradation in performance over this projection.

In order for the entire system to sustain the throughput enabled by the accelerator cache, the back-end servers must have an aggregate throughput of

$$S_{tp} = A_{tp} * (1 - h)$$

where A_{tp} is the accelerator throughput and h is the cache hit rate. In the previous example, the back-end servers must have an aggregate throughput of 200 requests/second in order

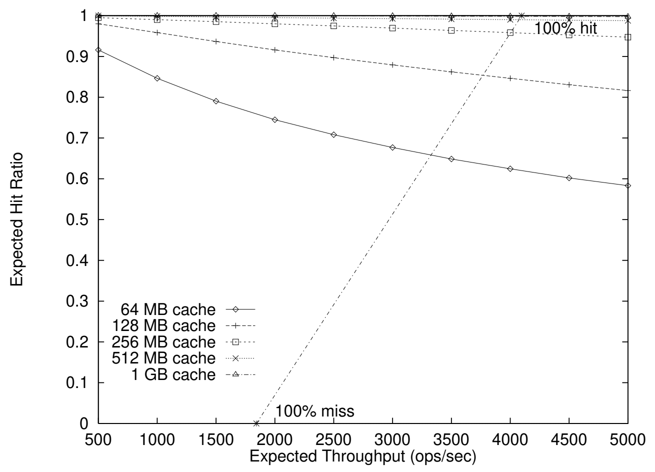


Figure 11: The maximum SPECweb96 throughput which our accelerator can sustain is obtained from the intersection of the line with the appropriate curve.

to obtain a SPECweb96 throughput of 4000 operations/second for the entire system.

As we mentioned previously, the accelerator must cache all objects in memory. Although this limits the size of the cache, we have just shown that memory sizes available today are sufficient for achieving high hit rates for workloads similar to SPECweb96. The space requirements for accelerator caches are generally lower than those for proxy caches because accelerators cache data from a single Web site. By contrast, proxy caches store data from many Web sites [7, 2, 12]. A future area of research for us is determining whether it is possible to cache objects on disk without incurring too much overhead when memory overflows.

5 Summary and Conclusion

In this paper, we presented the design, key issues in the implementation, and the performance of a Web server accelerator. Our accelerator improves Web server performance by caching data and runs under an embedded operating system. The accelerator can serve up to 5000 pages per second. By contrast, a Web server running under a general-purpose operating system on similar hardware can serve a maximum of several hundred pages a second. Our accelerator provides an API which allows application programs to explicitly cache, invalidate, and modify cached data. This API can be used to cache dynamic as well as static data.

We described how the accelerator's OSI layer four was extended and optimized to support TCP applications such as the cache. We also analyzed the SPECweb96 benchmark in order to determine hit ratios as a function of cache size for the SPECweb96 workload. We used this analysis to determine the SPECweb96 performance of our accelerator.

Several extensions to the accelerator we have described here are ongoing. For example, we have prototyped a scalable accelerator cache which stores data across several processors functioning as a single logical cache. This results in more cache memory as well as higher throughputs. Our scalable accelerator cache can continue to function if some but not all of the cache nodes fail.

References

[1] A. Bhide, A. Dan, and D. Dias. A Simple Analysis of the LRU Buffer Replacement Policy. In *Proceedings of the 1993 Data Engineering Conference*, February 1993.

[2] P. Cao and S. Irani. Cost-Aware WWW Proxy Caching Algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[3] J. Challenger, P. Dantzig, and A. Iyengar. A Scalable and Highly Available System for Serving Dynamic Data at Frequently Accessed Web Sites. In *Proceedings of ACM/IEEE SC98*, November 1998.

[4] J. Challenger, A. Iyengar, and P. Dantzig. A Scalable System for Consistently Caching Dynamic Web Data. In *Proceedings of IEEE INFOCOM'99*, March 1999.

[5] A. Chankhunthod et al. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 USENIX Technical Conference*, pages 153–163, January 1996.

[6] D. Dias, W. Kish, R. Mukherjee, and R. Tewari. A Scalable and Highly Available Web Server. In *Proceedings of the 1996 IEEE Computer Conference (COMPCON)*, February 1996.

[7] B. M. Duska, D. Marwood, and M. J. Feeley. The Measured Access Characteristics of World-Wide-Web Client Proxy Caches. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[8] G. Hunt, G. Goldszmidt, R. King, and R. Mukherjee. Network Dispatcher: A Connection Router for Scalable Internet Services. In *Proceedings of the 7th International World Wide Web Conference*, April 1998.

[9] A. Iyengar and J. Challenger. Improving Web Server Performance by Caching Dynamic Data. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[10] A. Iyengar, E. MacNair, and T. Nguyen. An Analysis of Web Server Performance. In *Proceedings of GLOBECOM '97*, November 1997.

[11] R. Lee. A Quick Guide to Web Server Acceleration. http://www.novell.com/bordermanager/accel.html.

[12] S. Manley and M. Seltzer. Web Facts and Fantasy. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[13] Mindcraft, Inc. WebStone Benchmark Information. http://www.mindcraft.com/webstone/.

[14] National Laboratory for Applied Network Research (NLANR). Squid internet object cache. http://squid.nlanr.net/Squid/.

[15] V. Pai et al. Locality-Aware Request Distribution in Cluster-based Network Services. In *Proceedings of ASPLOS-VIII*, October 1998.

[16] System Performance Evaluation Cooperative (SPEC). SPECweb96 Benchmark. http://www.specbench.org/osg/web96/.