Architecture of a Web Server Accelerator

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

June 29, 2001

Abstract

We describe the design, implementation and performance of a high-performance Web server accelerator which runs on an embedded operating system and improves Web server performance by caching data. It can serve Web data at rates 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 describe how our accelerator can be scaled to multiple processors to increase performance and availability. The basic design alternatives include a content router or a TCP router (without content routing) in front of a set of Web cache accelerator nodes, with the cache memory distributed across the accelerator nodes. Content-based routing reduces cache node CPU cycles but can make the front-end router a bottleneck. With the TCP router, a request for a cached object may initially be sent to the wrong cache node; this results in larger cache node CPU cycles, but can provide a higher aggregate throughput, because the TCP router becomes a bottleneck at a higher throughput than the content router. We quantify the throughput ranges in which different designs are preferable. We also examine a combination of content-based and TCP routing techniques. In addition, we present statistics from critical deployments of our accelerator for improving performance at highly accessed Sporting and Event Web sites hosted by IBM.

1 Introduction

There has been tremendous growth of the World Wide Web over the past several years in the number of users, sites, and data. To cope with such growth, Web servers often need to sustain high throughput levels. This performance requirement is further compounded by bursty access patterns which are common to many popular Web sites, resulting in high peak-to-average request rates [18]. In order to handle high request rates, it is often necessary to use multiple processors.

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

One technique for reducing the amount of hardware needed at a Web site and improving throughput is to place one or more high-performance caches in front of the Web servers known as Web server accelerators. In this paper, we address the problem of improving the performance of a Web server and present the architecture of a high-performance Web server accelerator we have built. It can be scaled to multiple processors in order to further increase throughput, main memory cache space, and availability.

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. Performance can be improved by caching data in a Web server accelerator which has significantly less overhead than a Web server.

Our accelerator runs under an embedded operating system and can serve Web data at rates 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. Our Web server accelerator has been used to improve performance at a number of highly accessed Web sites including the ones for the 2000 Olympic Games and 1999 Wimbledon tennis tournament. Cache hit rates of over 85% were achieved.

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 [17, 16, 3, 2].

Multiprocessor accelerators can further increase the performance. Our multiprocessor system architecture consists of a cluster of Web accelerator cache nodes and a front-end load balancer. From a scalability standpoint, the objective is to combine the individual cache space of each member of the cache array to scale the available space for caching, as well as to combine the individual throughput of each member of the cache array to scale the available throughput. Because of the high request rates our accelerator must sustain, all objects are cached in memory. The use of multiple cache members is thus a way to increase cache memory (main memory), while also increasing the system throughput.

Techniques for scaling Web servers are described in [9]. That study is about a scalable Web server composed of multiple Web server nodes and supported by traditional file sharing mechanisms, or replicated files. In contrast, our paper studies the scalability of high performance Web server accelerators, which are in front of a set of Web server nodes, and which are based on

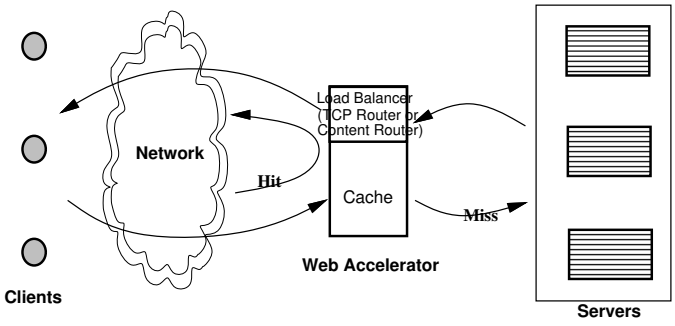


Figure 1: The accelerator resides in front of one or more Web servers and contains a cache and a load balancer which can function as either a content-based or TCP router. Cached objects are sent directly from the accelerator to clients.

main memory caching. Therefore, a focus of this work is to provide efficient ways of sharing main memory space across different cache nodes. This leads to significantly different design decisions.

The rest of the paper is organized as follows. Section 2 describes the basic architecture of our accelerator. Section 3 describes architectural alternatives for scalable accelerators. Section 4 presents performance measurements we have made of our accelerator and compares the performance of different architectural alternatives for scalable accelerators. Related work is discussed in Section 5. Finally, concluding remarks appear in Section 6.

2 Web Server Accelerator Design and Characteristics

2.1 System Description

As illustrated in Figure 1, the accelerator is placed in front of a set of Web server nodes. Multiple Web servers would be needed at a Web site which receives a high volume of requests. Uniprocessor accelerators consist of a load balancer and a cache. The load balancer presents a single IP address to clients regardless of the number of back-end servers and routes Web requests to the accelerator cache and back-end Web servers. The load balancer takes on two basic forms: (i) A TCP router [9, 14], which for performance reasons does not examine the contents of a Web request, and (ii) a content router [23], which routes requests based on the URL requested.

If the requested page is found in the cache, the page is returned to the client. Otherwise, the load balancer routes the request to a back-end Web server node using either round-robin or a method which takes server load into account. The use of TCP routing for sending cache misses to the server nodes results in better load balancing than using the round-robin Domain Name Server [9]. 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.

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, the load balancer may function as a TCP router. The TCP router does not complete a TCP connection corresponding to a request; rather it selects a node to handle the request, maintains this selection in a table, and sends the request to the selected node. The selected node completes the connection and directly returns the requested Web page to the client without going through the accelerator on the return path [9].

A benefit in performing content-based routing is that the accelerator can make intelligent decisions about where to route requests based on the URL [23]. 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 URLs 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 URLs. 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. Dynamic Web data change frequently and are typically created by server programs which execute when a request for the dynamic data is received. This process incurs significantly more overhead than serving static data; a request for static data is typically serviced by returning a file. 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 [17, 16, 3, 2]. 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 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. The 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 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.

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.

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 Scalable Web Server Accelerator Design and System Flows

In certain situations, it is desirable to scale a Web server accelerator to contain more than one processor. This may be desirable for several reasons:

- Multiple nodes provide more cache memory. Web server accelerators have to be extremely fast. In order to improve performance, the working set of Web objects should be cached in main memory instead of on disk. Multiple processors provide more main memory for caching data.
- Multiple processors provide higher throughputs than a single node.
- Multiple processors functioning as accelerators can offer high availability. If one accelerator processor fails, one or more other accelerator processors can continue to function.
- In some situations, it may be desirable to distribute an accelerator across multiple geographic locations.

In this section, we examine design alternatives for scalable Web server accelerators which cache data on multiple processors for both improved performance and high availability. The system consists of a cluster of uniprocessor Web accelerator nodes described in the previous section. The cluster design alternatives are the focus of this section.

In the scalable architecture, a load balancer directs Web requests to one of several Web accelerator nodes (Figure 2); each node is referred to as a *cache member*, and the set of all nodes is known as a *cache array*. The load balancer operates either as a TCP router or as a content router. The Web accelerator is placed in front of one or more Web server nodes. The Web URL space is hash partitioned among cache members such that one of the cache members is designated as the primary owner of each URL. If an object corresponding to a URL is cached in at least one cache member, the primary owner is guaranteed to contain a copy. A secondary owner of an object is a cache member other than the primary owner which contains the object. From a scalability standpoint, the objective is to combine the individual cache space of each member of the cache array to scale the available space for caching, as well as to combine the individual throughput of each member of the cache array to scale the available throughput.

Using a TCP router as the load balancer, there is a high probability that a request for a cached object will initially be routed to a cache node which is not an owner of the cached object. When this happens, the first node sends the request to a second cache node which is an owner of the object using different methods which will be described later in this section. In order to reduce the probability of the TCP router routing a request for a cached object to a wrong node, hot objects are replicated on multiple cache nodes. By contrast, the content router has the ability to route the request to the proper cache node. However, it adds significant overhead to the front-end load balancer and may result in the front end becoming a bottleneck. There are other situations as well where content-based routing cannot be assumed to always work or be available. In some architectures, objects may migrate between caches before the router is aware of the migration. This could result in a content-based router sending some requests for a cached object to a wrong cache node. In other situations, it may be desirable for a set of cache nodes to interoperate with a variety of routers both with and without the capability to route requests based on content. The set of cache nodes should still offer good performance for routers which cannot perform content-based routing.

Since the load balancer presents a single IP address to clients regardless of the number of back-end cache members and servers, it is thus possible to add and remove cache members or servers behind the load balancer without clients being aware of it. In Figure 2, the load balancer runs on a separate node. This design results in maximum throughput since the load balancer is able to handle more requests. A load balancer can also be configured to run on a cache member node; this is useful for cases where the load balancer is not a bottleneck, such as when the cache array is composed of a small number of nodes.

The load balancer obtains availability as well as load information about each member of the cache array via its normal operations. This information is used to route requests to cache members.

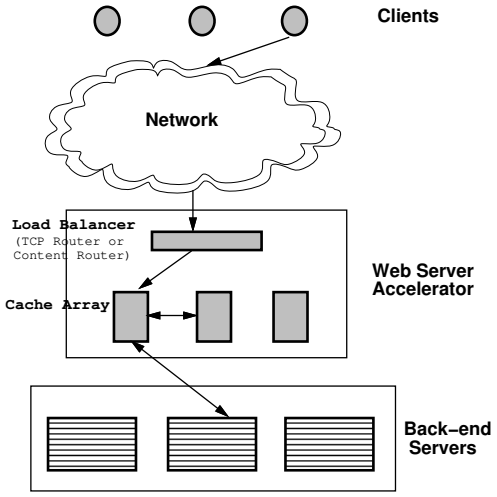


Figure 2: System Structure with a Scalable Web Server Accelerator

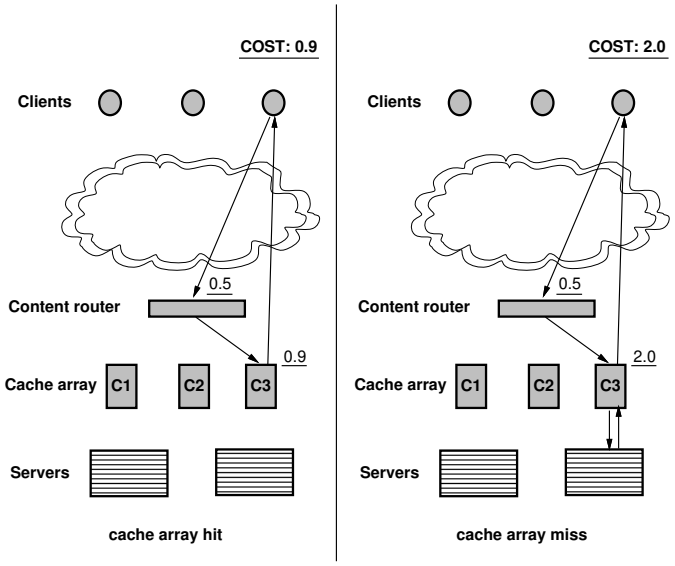


Figure 3: Content router as load balancer: content router uses handoff interface. Request results in cache array hit or cache array miss.

3.1 Request Flows Through the System

In the rest of the section, we detail and compare different configurations of the scalable accelerator. We compare the content router versus the TCP router as the load balancer. We then compare different communication and data delivery methods. Furthermore, we investigate the effect of object size on the different methods.

For the performance comparison of the different configurations, we have measured the number of CPU cycles required at each system component for different situations. The measurement is done on a scalable Web server accelerator which is composed of two cache member nodes and a front-end load balancer. These cache member nodes as well as the load balancer run on 200 MHz PowerPC processors. These nodes were directly inter-connected to each other using Token Ring. The measurement of our accelerator shows that the number of CPU cycles incurred at a cache node to serve an HTTP request for an object of 2 KB is about 31,500, and this number does not vary much for objects smaller than 2 KB. From now on, we use this number as the relative cost of 1 for the comparison of different configurations. After detailed study of different configurations, we summarize in Table 1 the relative CPU costs in each configuration for different situations.

3.1.1 Content Router as the Load Balancer

The cache member which initially receives a request from the load balancer is designated as the *first member*. When the load balancer is a content-based router, it can directly route a request to an owner of the requested object (*i.e.*, the first member is the owner). This is done by examining the HTTP request, and in order to examine it, the content router has to complete a connection with the client. After the content router has examined a request and selected an owner of the requested object, it uses one of two methods for sending the request to the owner.

The first approach is for the content router to hand off the connection to the owner regardless of the size of the requested object. The owner always responds directly to the client without going through the content router. A similar method for performing content-based routing is presented in [23]. Our measurement of CPU overheads shows that in this scheme, the relative CPU cost incurred is 0.5 at the content router and 0.9 at the cache node (see Figure 3).

In the second method, different interfaces are used between the content router and the selected cache node depending upon the size of the objects. If the requested object is small, the object is first returned from the owner to the content router via a UDP interface. It is then returned from the content router to the client. If the requested object is large, the owner performs a TCP handoff and responds directly to the client without going through the content router. More details of this adaptive method are described in Section 3.1.2 in the context of a TCP router-based accelerator.

Comparing the two approaches, the advantage to the latter is that it incurs very little overhead on the cache array for small objects. For objects up to 2 KB, the relative cost for the UDP interface is only 0.1 in cache array CPU cycles. The disadvantage is that significant overhead is incurred at the content router. For objects up to 2 KB, the relative cost for the UDP interface is 1.1 in content router CPU cycles. For large objects, both approaches use the handoff interface and hence have similar performance. In short, this approach incurs less total overhead for small objects. The overhead at the content router is higher while the overhead at cache members is lower.

While the approach using a content-based router as the load balancer reduces CPU cycles consumed by cache members, the disadvantage is that it consumes extra CPU cycles on the content router which can make the content router a bottleneck. A TCP router running on a 200 MHz PowerPC 604 can route 15 K requests per second (i.e., without doing content-based routing). A content-based router running on the same CPU can route 9.8 K requests per second using the handoff mechanism and 4 K requests per second using the UDP interface. If cache array CPUs are the bottleneck, content-based routing is a good approach. If, on the other hand, the load balancer is a bottleneck, content-based routing should not be used. If it is not clear whether the load balancer or the cache array will be the bottleneck, some requests can be routed by examining content while others can be routed without examining content.

3.1.2 TCP Router as the Load Balancer

When the load balancer operates as a TCP router, it sends a request to a first member using a weighted round-robin policy. We say that a *cache member hit* occurs when the first member receiving a request from the TCP router is an owner of the requested object (Figure 4). Likewise, a *cache member miss* indicates the case when the first member is not an owner of the object (Figures 5, 6).

If no replication is used, the probability of a cache member hit is roughly 1/n where n is the number of cache members in the cache array. The exact probability is dependent on the way objects are partitioned across the cache array, request traffic, and the load and availability of cache members. A cache member hit is distinct from a *cache array hit* which occurs when the cache array as a whole can satisfy a request (*i.e.,* at least one cache member has a copy of the requested object). Note that it is possible to have a cache member hit and a cache array miss. This would occur when the first member receiving a request from the TCP router is the primary owner of the requested object but the object is not cached. Conversely, it is possible to have a cache member miss and a cache array hit. This would occur when the first member receiving a request from the TCP router does not contain a cached copy of the requested object but another cache does.

There are multiple methods for returning objects in the event of a cache member miss. An easy way is to use a separate HTTP connection between the first member and the owner, having the first member acting as an HTTP proxy. However, this method results in high overhead to the cache array. Alternatively, a UDP interface can be used. The UDP interface significantly reduces overhead in the system and is feasible in a cache cluster because the packet loss rate is minimal, especially when the cache nodes are in close proximity. Lastly, the first member can hand off the request to the owner along with the TCP connection. The owner then returns the data directly to the client which eliminates a hop along the return path. The different request flows through the system are thus summarized by the following:

1. Cache member hit, cache array hit.
2. Cache member hit, cache array miss.
3. Cache member miss, cache array hit,
   (a) page retrieved using HTTP;

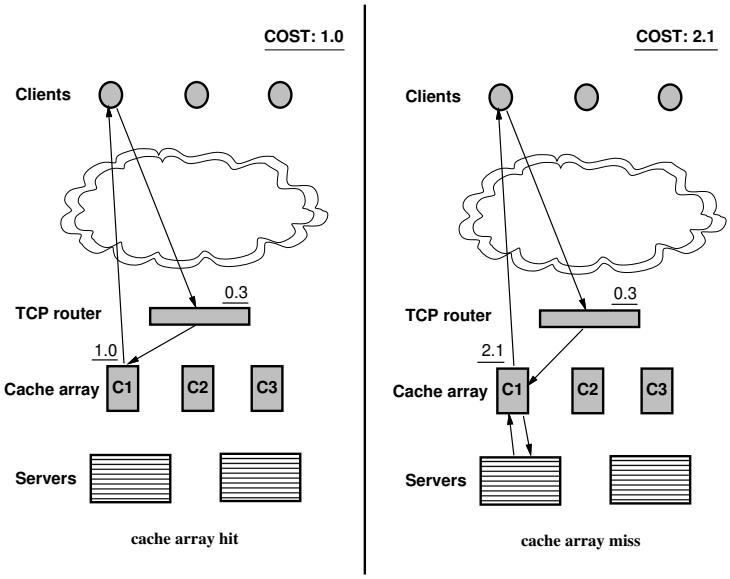


Figure 4: TCP router as load balancer: request results in cache member hit.

(b) page retrieved using UDP;

(c) page retrieved via a request handoff.

4. Cache member miss, cache array miss,

   (a) page retrieved using HTTP;

   (b) page retrieved using UDP;

   (c) page retrieved via a request handoff.

Cache member hit

Upon a cache member hit, if the first member has the requested object, it sends the object directly back to the client. Otherwise, the first member obtains the requested object from a back-end server and returns it to the client (Figure 4). In both cases, requested objects are returned directly from a cache member to the client without going through the TCP router.

The relative cost for the cache array CPU cycles consumed by a request for an object of up to 2 KB is 1 for a cache member hit and a cache array hit. This is the same as the cost for a cache hit in a uniprocessor accelerator. The cost for a cache member hit and a cache array miss is 2.1. This is further broken down into a cost of 1 per connection (two connections are used, one from cache member to client, and one from cache member to back-end server), plus about .1 to logically bind the two connections together inside the cache member.

Cache member miss - HTTP interface

When no replication is used, a cache member miss occurs roughly n-1 times out of n in a balanced system with n cache members. When this happens, the first member accepts the

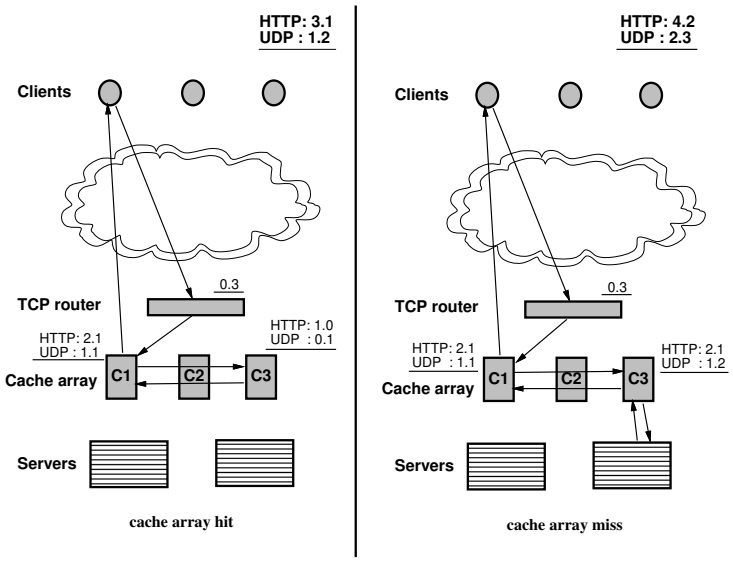


Figure 5: TCP router as load balancer: HTTP or UDP interface is used between the first member and the owner. Request results in cache member miss, followed by cache array hit or cache array miss

connection with the client, computes an owner of the requested object, and contacts the owner to get the requested object. In Figure 5, the first member communicates with the owner of the requested object via HTTP or a UDP interface.

We have measured the relative CPU cost (sum at both cache nodes) of a request for an object of up to 2 KB resulting in a cache member miss and a cache array hit to be 3.1 when the first member and the owner of the requested object communicate via HTTP. The TCP connections constitute the principal component of the overhead. The first member has two connections (one to the client and one to the owner of the requested object) while the owner of the requested object has one connection (to the first member). In addition, the overhead for binding the two connections in the first member is about 0.1.

As requested object sizes increase, the cache array CPU cost of serving an object for a cache member miss and a cache array hit increases three times faster than it would increase for a request resulting in a cache member hit and a cache array hit. The additional overhead results from the total number of times the object is sent or received by a cache member. In the case of a cache member hit, the object is sent only once (from the owner to the client) resulting in only one send/receive. In the case of a cache member miss, the object is sent twice (once by the owner and once by the first member) and received once (once by the first member) resulting in a total of 3 sends/receives.

The cache array CPU cost of a cache member miss (for objects up to 2 KB) resulting in a cache array miss is 4.2. This is because of the extra connection from the owner to the back-end server and another binding of two connections together in the owner. As the requested object size increases, the cache array CPU cost of serving an object upon a cache member miss and a

cache array miss increases twice as fast as it would increase for a request resulting in a cache member hit and a cache array miss. In the former case, the object is sent twice (once by the owner and once by the first member) and received twice (once by the owner and once by the first member). In the latter case, the object is sent once and received once.

Cache member miss - UDP interface

This is similar to the previous case, except that the interface between the first member and owner of the requested object is UDP, which has lower CPU overhead than HTTP. Our measurement shows that the cache array CPU cost of a cache member miss (for objects up to 2 KB) resulting in a cache array hit is only 1.2 using the UDP interface (Figure 5). This is further broken down into a cost of 1.1 at the first member and 0.1 at the owner node. UDP has lower overhead than HTTP largely because it avoids making a TCP connection. The cost of a cache member miss resulting in a cache array miss (for objects up to 2 KB) is 2.3, since an extra connection from the owner to a back-end server and an extra binding of two connections in the owner are needed.

While UDP is not as reliable as HTTP for communicating across the Internet, the unreliability of UDP is not a significant factor in our system because cache members communicate directly with each other on a private local network without going through any intermediate nodes. The packet loss rate is thus small. Any packets lost by UDP are handled by timeouts and garbage collection. While the corresponding Web request is lost, the probability of this occurring is low.

Cache member miss - Handoff interface

In this case, instead of the first member functioning as a proxy in order to obtain the requested object and return it to the client, the first member hands off the request, along with the TCP connection, to an owner of the requested object. The owner then sends the requested object directly back to the client without going through the first member (Figure 6).

The handoff is possible because the different entities in the system share an IP address. This virtual cluster address provides the framework so that an established connection with a client can be shared and dynamically moved to different entities even in the middle of an operation. In a sense, this handoff can be thought of as an extension of TCP routing in which a TCP router selects a node in the cluster and dispatches TCP connections to it. However, the implementation of the dynamic handoff of an already established and operating TCP connection is different and more complicated. First, it is an operation where three different entities, i.e., the TCP router, the first member, and the second member (*i.e.,* the owner node), should participate in a coordinated fashion. Second, the operation should occur transparently to clients. The major steps are as follows:

1. The second member node opens a TCP connection with the client. This connection is established transparently without the regular 3-way handshaking.

2. The second member duplicates exactly the same state of the TCP connection which was already established between the client and the first member.

3. The second member emulates the process of receiving the request which was initially sent to the first member from the client.

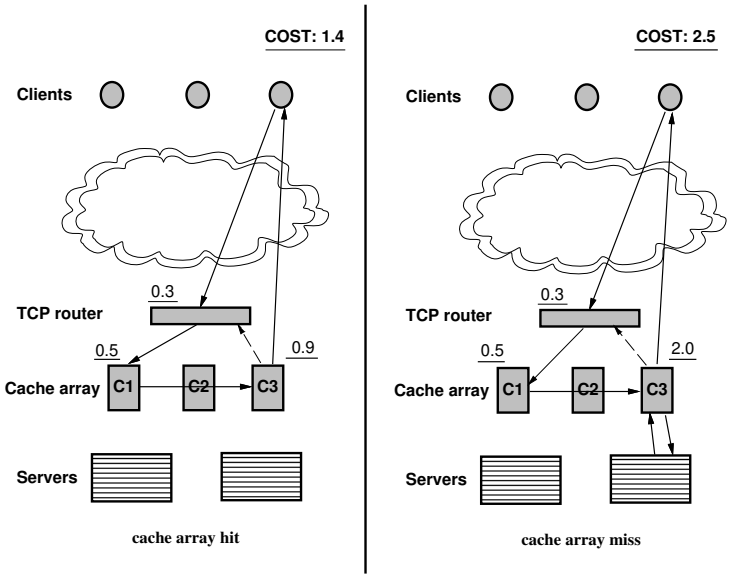


Figure 6: TCP router as load balancer: handoff interface is used between the first member and the owner. Request results in cache member miss, followed by cache array hit or cache array miss

4. The TCP router redirects any follow-up messages from the client to the second member.

5. The first member cleans up data structures related to the TCP connection with the client (e.g., socket and TCP control blocks).

To open a TCP connection and duplicate the connection state in the second member node, we copy parts of the TCP control block from the first member. This information is then sent to the second member along with the requests which were sent from the client. The connection set-up at the second member is done by taking steps similar to the TCP Passive Open [27]. Then, the information received from the first member is copied to the TCP control block of the new TCP connection.

The cache array CPU cost of a cache member miss (for objects up to 2 KB) when the handoff interface is used is 0.5 at the first member and 0.9 at the owner node, resulting in total of 1.4 for a cache array hit (Figure 6). For a cache array miss, an additional 1.1 is added to the owner node. This additional overhead results from an extra connection from the owner to the back-end server and an extra binding of two connections in the owner.

For objects of 2 KB or less, the performance of the handoff interface is superior to that of the HTTP interface but inferior to that of the UDP interface. For large objects, however, the performance of the handoff interface is superior to that of both the HTTP and UDP interfaces. This is because a system using the handoff interface eliminates the step of communicating requested objects between cache members. Consequently, the increase in cache array CPU cost resulting from object sizes over 2 KB for the handoff interface is similar to that which would be incurred by a cache member hit.

Load balancer	Node		cache array hit			cache array miss		
			HTTP	UDP	Handoff	HTTP	UDP	Handoff
TCP router	TCP router		0.3					
	member hit	cache array	1.0			2.1		
	member miss	first member	2.1	1.1	0.5	2.1	1.1	0.5
		owner	1.0	0.1	0.9	2.1	1.2	2.0
		cache array (total)	3.1	1.2	1.4	4.2	2.3	2.5
Content router	content router		2.1	1.1	0.5	2.1	1.1	0.5
	cache array		1	0.1	0.9	2.1	1.2	2.0

Table 1: Summary of relative CPU costs for objects of size up to 2KB in different accelerator configurations.

Cache member miss - Mixed Strategy

Among the options considered, the UDP interface offers the best performance for small objects, while the handoff interface offers the best performance for large objects. Therefore, a mixed strategy for handling cache member misses which uses the UDP interface for small objects and the handoff interface for large ones has better performance than the individual strategies. As we shall see in Section 4, the crossover point for our system when the UDP and handoff interfaces result in similar performance occurs when requested objects are between 3 KB and 4 KB.

To optimize performance, our system implements a mixed strategy for cache member misses as in the following steps:

1. The first member sends the request and TCP connection information to an owner of the requested object.
2. If the requested object is not cached, the owner obtains it from a back-end server (which may result in the object being cached).
3. If the object is small, the owner returns it to the first member which subsequently returns it to the client.
4. If the requested object is large, the owner performs a TCP handoff. There is no need for the first member to do anything during this process.
5. The owner returns the requested object directly to the client without going through the first member.
6. Asynchronously, the owner informs the first member to clean up connection information corresponding to the request.

In our system, this coordination is entirely driven by the owner. It has all the information needed to perform the TCP handoff. All the first member has to do is wait until the owner either sends back the requested object or informs it that it will clean up the connection information (off-line).

3.2 High Availability

Our system provides high availability via the load balancer and replication. The load balancer has the ability to detect when a cache member fails. When this happens, it informs the remaining live members of the failure and directs all requests to live members of the cache array.

Our system hashes objects across the cache array using an enhanced versions of CARP (Cache Array Routing Protocol) [21]. CARP is a hashing mechanism which allows a cache to be added or removed from a cache array without relocating more than a single cache's share of objects. When a new cache is added, only the objects assigned to the new cache are relocated. All other cached objects remain in their current cache. Similarly, removing a cache from the array will only cause objects in the removed cache to be relocated.

CARP calculates a hash not only for the keys referencing objects (e.g. URLs) but also for the address of each cache. It then combines key hash values with each address hash value using bitwise XOR (exclusive OR). The primary owner for an object is the one resulting in the highest combined hash score.

Whenever a cache member fails, no rehashing is necessary. The new primary owner for any object whose primary owner used to be the failed cache member is simply the live cache member resulting in the highest combined score. After a failed cache member is revived, objects for which the revived member is now the primary owner must be copied to the revived member. This can take place in the background while the accelerator continues to operate. While the revived member is warming up, an object for which the revived member is the primary owner might not yet be cached in the revived member but might be cached in the previous primary owner before the revival. In order to handle these situations, misses for such objects in the revived member will cause a lookup in the cache member which used to be the primary owner before the revival. This additional lookup is no longer necessary after all hot objects primarily owned by the revived member have been copied to the revived member.

In a TCP router-based configuration, replication of hot pages on multiple cache members not only improves the accelerator performance (by increasing the probability of cache member hits) but also reduces cache array miss rates after a cache member failure. This is because a copy of a hot page may still be in the cache array after an owner of the page fails. In order to store an object in n caches where $n > 1$, the object is stored in the n caches resulting in the highest combined hashing score for the objects.

It is possible to configure our system with a backup load balancer node to handle failure of a load balancer.

4 Performance

4.1 Breakdown of CPU Cycles

This section analyzes the CPU cycles consumed by the accelerator. Since the system under test is precisely and intentionally separated from any operating system involvement (no scheduling, no blocking, no timing, no copying), CPU overhead can be measured by 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 num-

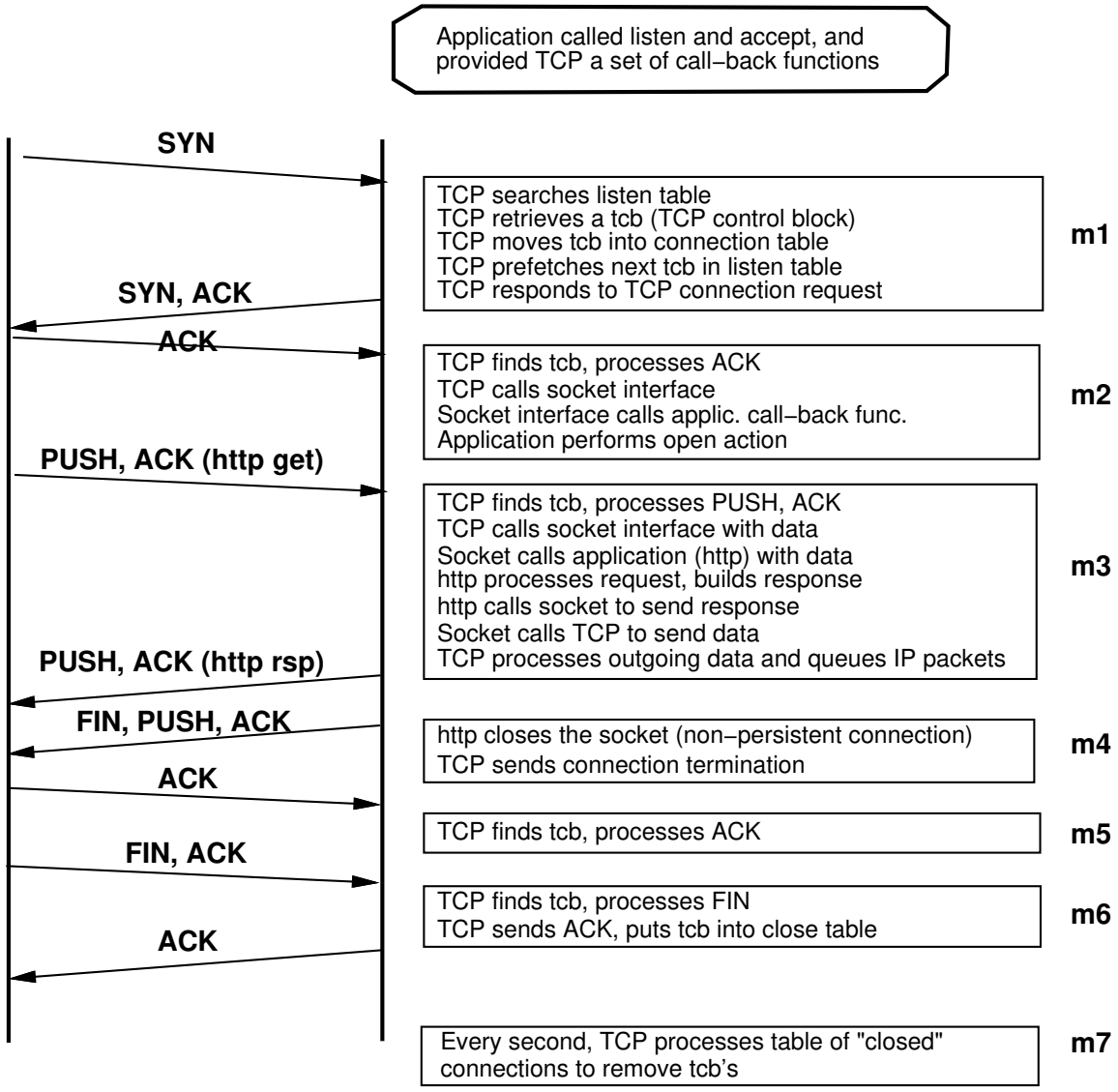


Figure 7: TCP flow in a uniprocessor Web server accelerator. Blocks m_1 through m_7 depict operations performed by the CPU. The numbers of CPU cycles spent for each block are shown in Table 2.

ber of cycles and the numbers of instructions, as well as details on cache hits and misses). The TCP flow is depicted in Figure 7.

Request traffic was generated by WebStone. WebStone is a benchmark which measures the number of requests per second a Web server can handle. It is done by simulating one or more clients and seeing how many requests can be satisfied during the duration of the test [22].

Table 2 shows the measurements we obtained for the components of the TCP flow depicted in Figure 7. Component $m3$ is the only one which varies significantly with data size.

For a 200 MHz PowerPC 604 processor, the theoretical capability would be 6000 requests per second for an 8 Kbyte page. In practice, several factors 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.

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
		3,608	8,330	128 bytes
		3,707	8,460	256 bytes
		3,990	7,280	1K bytes
		4,310	9,600	2K bytes
		4,608	8,740	4K bytes
		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 2: Measurements for the components of the TCP flow depicted in Figure 7. The flows m1 through m7 correspond to the boxes in the figure. Component m3 varies with data size.

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 de-queueing 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

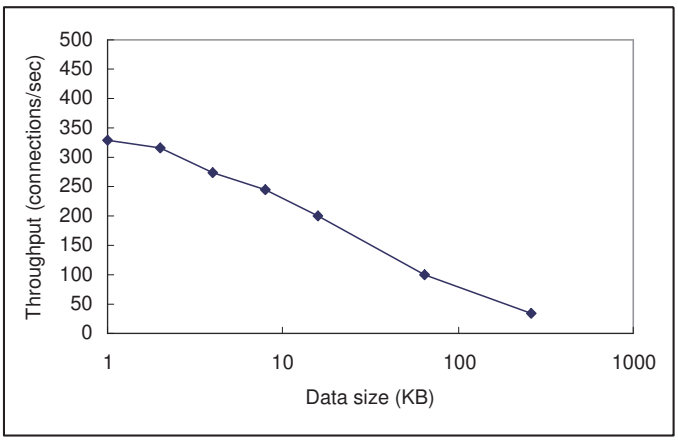


Figure 8: Web Server Throughput: measurement of an Apache Web server running on an AIX system with a 200 MHz Power PC 604e (data from [12, 13]) using the WebStone benchmark [22]. By contrast, our accelerator running on similar hardware achieves an order of magnitude higher throughput.

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 than 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.

4.2 Uniprocessor Web Server Accelerator Throughput

The system used to measure the Web accelerator throughput is illustrated in Figure 9. It consisted of two Web accelerators and two SP2 frames containing a total of 16 nodes. The accelerators ran on 200 MHz powerPC processors and were connected to each other via four 16 Mbit/s token rings. The first accelerator was the accelerator under test and the 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 SP2 frames were connected to the accelerator under test through four 16 Mbit/s token rings. The SP2 nodes issued requests to the accelerator by running WebStone. 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. Each accelerator had 512 Mbytes of main memory. The performance numbers in this section and

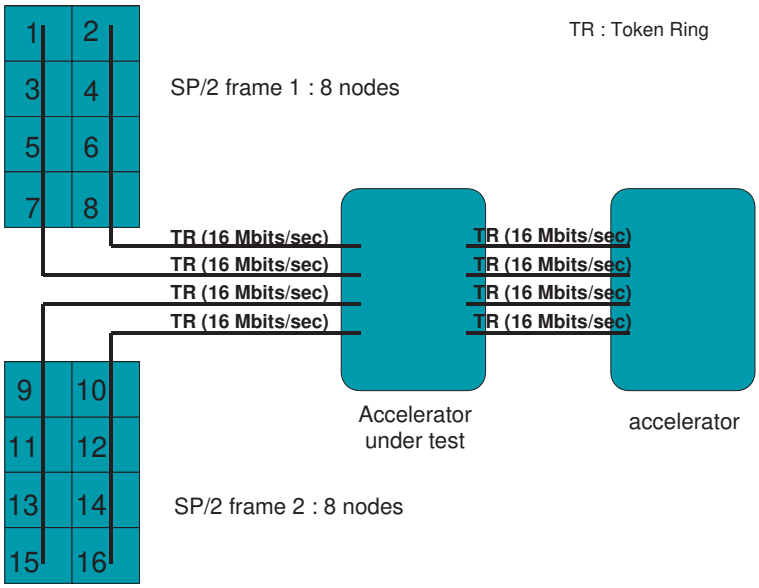


Figure 9: The system used to test our accelerator's performance: 16 SP/2 nodes were used for clients, each node generating 20 to 40 clients. Totally, there were 320 to 640 clients. Also, an extra accelerator was used for 5 to 100 additional clients.

Section 4.3 are for prototypes we have implemented and are not for any IBM products.

Figure 10 shows the number of cache hits per second a uniprocessor 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 11.

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 [22] 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 performance difference between our accelerator and a conventional Web server can be seen by comparing Figures 10 and 11 to Figure 8.

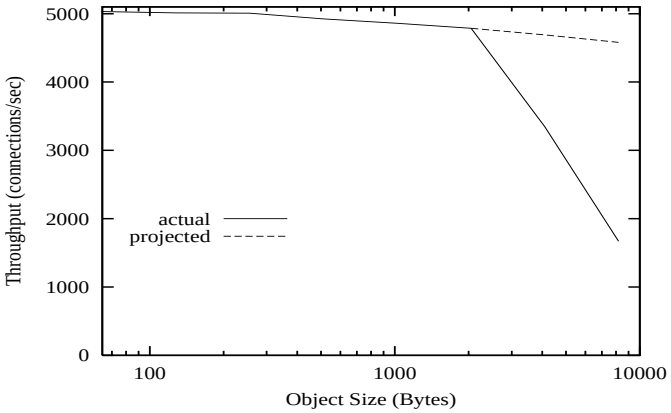


Figure 10: The number of cache hits per second a uniprocessor accelerator can sustain and the projected number which would be expected if the network were not a bottleneck.

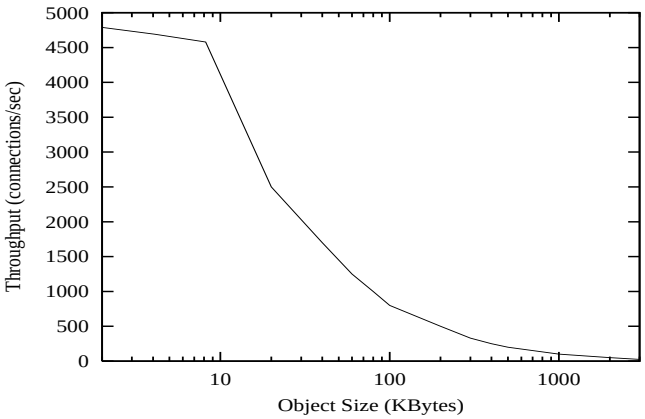


Figure 11: The projected number of cache hits per second a uniprocessor accelerator can sustain for large pages.

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. The presence of dynamic Web pages can hurt performance significantly. We have encountered several commercial Web sites where a single request for a dynamic page typically consumes several seconds of CPU time. However, 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 12. Each curve represents a back-end server configuration with a different capacity. For example, the curve marked *WST* (Web server throughput)*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.

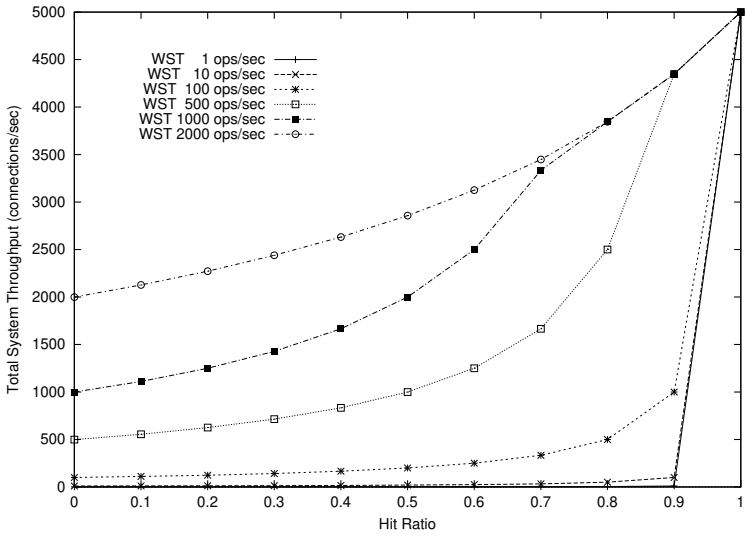
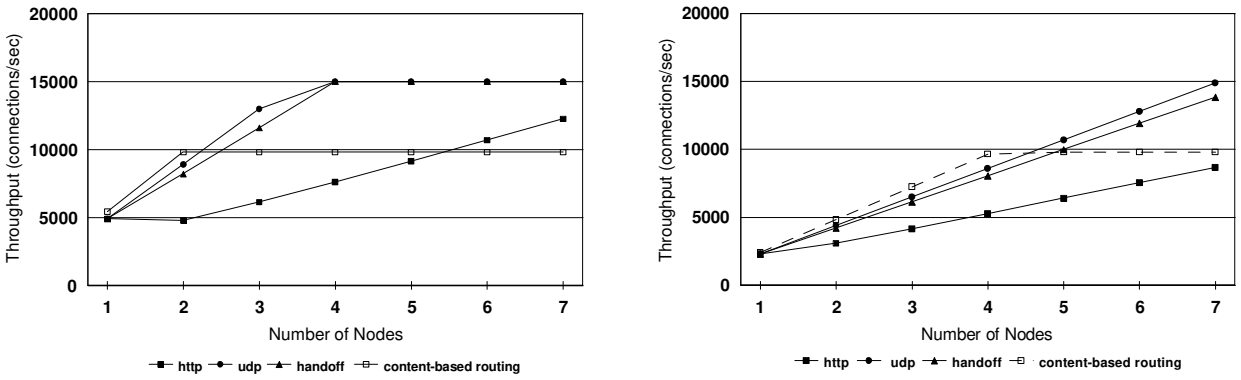


Figure 12: The overall performance of a system utilizing a uniprocessor accelerator. (WST: Web server throughput)

4.3 Scalable Accelerator Performance

We have built a scalable Web server accelerator where each cache member runs on a 200 MHz PowerPC processor. The system used to measure the Web accelerator throughput consisted of two SP2 frames containing a total of 16 nodes which were connected through local area networks to the TCP router. Some of the SP2 nodes issued requests to the cache array by running the



(a) Cache array hit

(b) Cache array miss

Figure 13: Throughput versus number of cache nodes, up to 2 KB objects

WebStone benchmark [22]. Other nodes were used as back-end servers to handle cache array misses.

Our test configuration did not have the capacity to drive more than two cache member nodes. We measured the CPU overheads for the various cases described in Section 3.1 for two cache array nodes. Then, we constructed a separate slow accelerator consisting of multiple cache members and measured the performance on it for multiple cache array nodes. (Each cache member of this slow accelerator runs on a Motorola 68040 processor.) We project the performance of fast accelerators containing multiple cache members from that of slow ones, and that of a single node fast accelerator. We validated our projections by comparing measurements of the CPU overhead for TCP handoffs and other cases described in Section 3.1 for both slow and fast accelerators.

We first show how the system scales when we increase the number of cache nodes in the cache array. Figure 13 shows the results for the number of requests served by the cache array for objects up to 2 KB. In this figure, the number of nodes excludes the load balancer. The curves flatten out when the load balancer becomes the bottleneck. For the TCP-router based approaches, the UDP interface scales the best for both cache array hits and cache array misses. The HTTP interface has significant overhead. Cache array hits in a multi-node system incur more overhead on average using HTTP than cache array misses using other interfaces. For the case of cache array hits, the HTTP interface with two nodes results in slightly lower throughput than using a single node. The UDP interface for two nodes only results in higher throughputs for cache array hits compared with a single node for objects up to about 40 KB (Figure 14). However, no replication was used in these runs. By replicating hot objects, the overhead for cache array hits using all three interfaces can be reduced. In addition, a 2-node system using any of the three interfaces results in higher throughputs than a 1-node system for cache array misses for small objects.

Figures 14 and 15 show the system performance when the sizes of the requested data objects

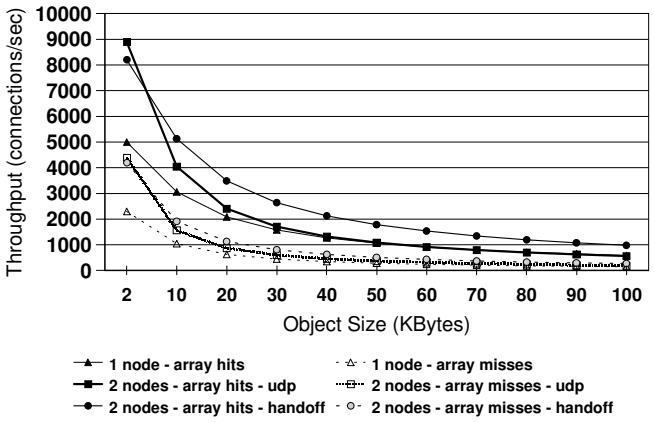


Figure 14: Impact of object size, 2 to 100 KB objects

are increased. The measurement was made in the cache array with 2 cache nodes (faster accelerators). As mentioned earlier, the UDP interface has the best performance with small size objects. However, the relative performance of the handoff interface improves with increasing object size. This is because the advantage of eliminating one hop from the data return path becomes greater as the data size gets larger. The graph shows that the cross-over point between the two cases is when the object size is between 3 and 4 KB.

Figure 16 compares the maximum achievable throughputs when using the TCP router versus the content-based router as the load balancer while the number of nodes in the cache array varies. In the figure, the number of cache nodes includes the load balancer (unlike the previous figures). The figure shows that with a small number of cache nodes in an accelerator, a content-based router results in a higher throughput whereas for a higher-end system, the TCP router results in a higher throughput. When the cache array is composed of two or three nodes, the front-end load balancer also works partly as a cache node.

Maximum throughput is limited to 15,000 connections per second due to the front-end router. In order to get higher throughputs, it is possible to use multiple scalable accelerators and route requests to the accelerators using domain name servers (DNS) [9].

In all of the graphs in this section, the cache array did not perform any replication of hot objects on multiple cache members. When the TCP router is used, performance can be improved further by replication of hot data. When the load balancer is a potential bottleneck, routing requests without examining content while replicating hot objects to reduce cache member misses is preferable to content-based routing.

4.4 Experience in Real Deployments

The Web server accelerator has been used at a number of highly accessed Web sites including those for the 2000 Olympic Games, the US Open Tennis Tournament, Masters Golf Tournament, and the Wimbledon Open Tennis Tournament. We first report on our experience at the Wimbledon Open Tennis Tournament which was held from June 23 to July 2, 1999. The Web

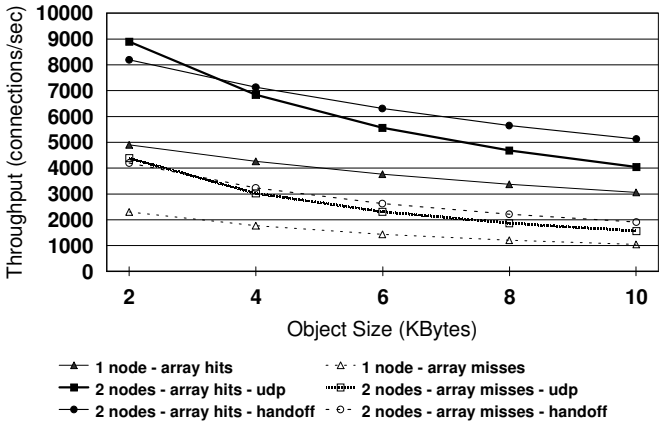


Figure 15: Impact of object size, 2 K to 10 KB objects. This graph expands a portion of the graph in Figure 14.

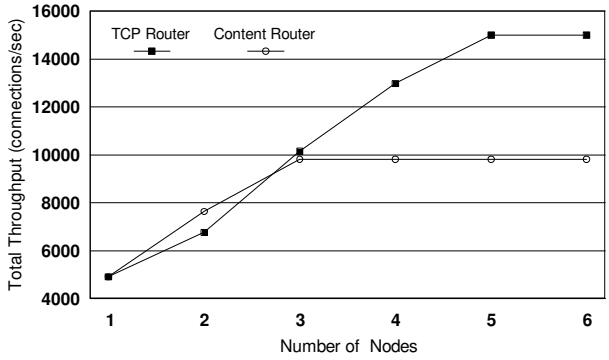


Figure 16: Comparison of TCP and content-based routers.

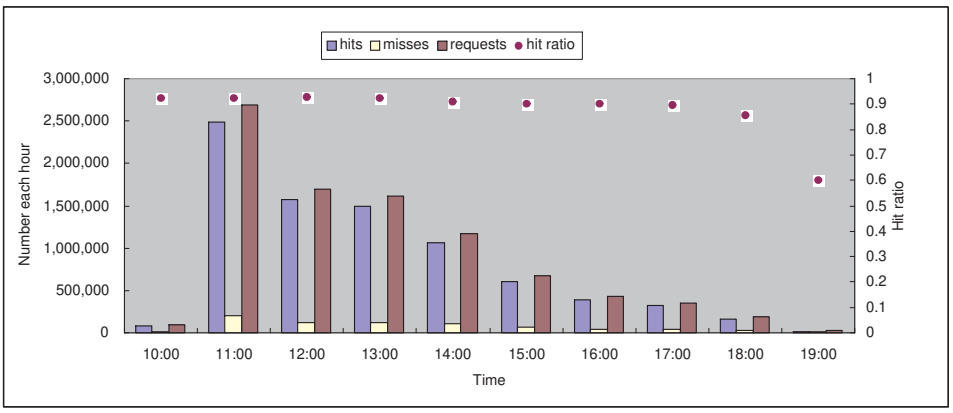


Figure 17: Hits, misses (array hits and array misses), and hit ratios on the Web accelerators at the Wimbledon Tennis site for ten hours on July 2, 1999.

site was distributed over three locations: Schaumburg, Illinois; Columbus, Ohio; and Bethesda, Maryland. Fifty Web servers and four Web server accelerators were used for the site. Initially, two accelerators were used, and two were added later. There was a separate TCP router as a front-end load balancer and thus the accelerators functioned only as caches. The traffic to accelerators was controlled by the front-end load balancer with a relative weight of four compared to the regular Web servers. (The load balancer dispatched the requests in a weighted round robin fashion considering the relative capacities of respective nodes. The capacity of the accelerators was assumed to be four times that of the regular Web servers.) During the event, there were a total of about one billion requests over two weeks. Processing by a single accelerator peaked at 66,095 requests per minute around 10:30 a.m. at the Bethesda site.

Figure 17 shows the number of requests, hits, and misses processed by the Web server accelerators for ten hours on July 2 along with the corresponding hit ratios. The number of total requests directed to the four accelerators was 8,932,303. The total number of requests to the site for the 10 hours was about 38 million. Out of them, 8,190,429 were cache hits and the rest were cache misses, resulting in a 92% cache hit ratio.

Our Web server accelerator was also a critical component for the 2000 Olympic Games Web site. The Web site deployed 71 front-end accelerator nodes distributed geographically across seven sites. All requests to the Web site initially went to a front-end accelerator node (By contrast, at the Wimbledon site, only a fraction of the requests were directed to an accelerator node). The hit rates across all front-end accelerator nodes was 87%. For the most active time period from September 15 through October 1, the site received 5,544,552,719 total requests of which 4,816,070,515 were cache hits. The traffic to the front-end accelerators and hit rates are shown in Figure 18.

Each Web server accelerator had 512 Mbytes of memory. This memory was used for both cache storage and connection management. The amount of usable cache space depended on the traffic. When traffic to an accelerator was light, most of the 512 Mbytes could be used for the

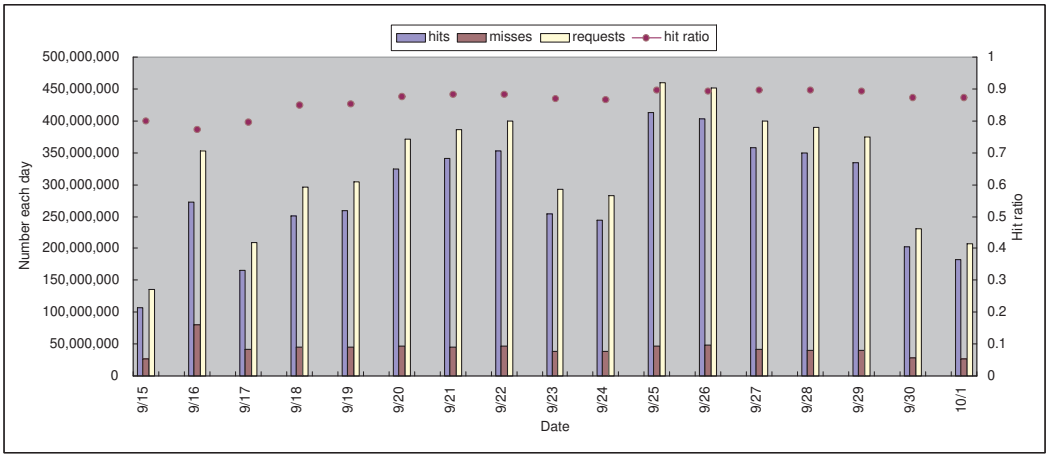


Figure 18: Hits, misses (array hits and array misses), and hit ratios on the Web accelerators at the 2000 Olympic Games Web site.

cache. When traffic to the accelerator was high, less caching space was available.

The Web server accelerators had sufficient throughput to deliver responses quickly at all times at both the 1999 Wimbledon and 2000 Olympic Games Web sites.

5 Related Work

Web server accelerators are contained in both the Harvest and Squid caches [5, 10]. 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 [19]. Microsoft's Scalable Web Cache (SWC) [8] and kHTTPd for Linux (http://www.fenrus.demon.nl/) are Web server accelerators which are implemented as kernel-mode caches on the serving platform. Such accelerators require special operating system support on servers. By contrast, our accelerator can be used in conjunction with any server platform and allows a single accelerator to be associated with multiple servers.

A key differentiating feature of our accelerator from others is that we allow dynamic pages to be cached in addition to static ones. This is possible because we provide an API for an application to explicitly control what is cached. None of the other accelerators we are aware of provide the scalability features which we provide wherein multiple processors are used to increase the throughput, memory, and availability of the accelerator.

Several Web proxy caches are available such as Inktomi's Traffic Server [6, 11], Network Appliance's NetCache [1], the CacheFlow 2000 [15], and IBM's Web Traffic Express [7]. As far as we know, none of these products provide scalability using our approach in which all objects are cached in main memory and multiple processors are used to scale the size of main memory.

There have been a few papers describing enabling technologies which are utilized by our accelerator. The TCP router used to route requests to caches is analyzed in [9, 14]. Content-

based routing is discussed by Pai et. al. in [23]. A key difference in our work is that we analyze the overhead for doing content-based routing and present alternative methods for routing requests when the overhead for performing content-based routing is likely to make the router become a bottleneck.

6 Conclusions

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. Our accelerator has been deployed at several highly accessed Web sites for improving performance. Hit rates of over 85% were achieved at the 2000 Olympic Games and 1999 Wimbledon Open Tennis Tournament Web sites. 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. We described how the accelerator's OSI layer four was extended and optimized to support TCP applications such as the cache. 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 have also presented a multiprocessor accelerator as a scalable and highly available solution. The memory of our accelerator scales linearly with the number of cache nodes, and the throughput scales almost linearly with the number of cache nodes as long as the front-end load balancer is not a bottleneck. We have compared design alternatives for the scalable accelerators and have quantified the efficiency and scaling achieved by the schemes.

The multiprocessor accelerator includes a load balancer sending requests to multiple processors collectively known as a cache array. The load balancer takes on one or a combination of two forms: a content router, in which requests are sent to specific nodes of the cache array based on the URL requested; and a TCP router, where the request is routed without regard to the requested URL. While content-based routing reduces CPU usage on cache nodes, it adds overhead to the load balancer, which can result in the load balancer becoming a bottleneck. Greater throughputs can often be achieved when some or all requests are routed without their content being examined.

A back-up load balancer can be integrated into our system in order to handle load balancer failure. Our Web accelerator can also continue to function if some but not all of the processors comprising the cache array fail. Replication of hot objects minimizes decreased performance resulting from a cache node failure.

There are a number of extensions to our Web server accelerator which we are currently working on. One such extension is to apply similar ideas to improve the performance of Web proxy caches [25, 24]. Since miss rates to proxy caches are often 50% or higher, performance can be adversely affected by the time to service cache misses. Our proxy accelerator architecture reduces cache miss overheads by using an accelerator to offload requests to a proxy cache which are likely to be misses.

A second area we are exploring is techniques which allow personalized pages to be cached. Personalized Web pages contain content specific to users; a personalized page cannot be shared by a large pool of clients, so conventional caching techniques are not very effective. Our approach breaks up Web pages into fragments [4] and represents personalized information via fragments.

A cache stores nonpersonalized fragments. When a complete Web page is needed, it is constructed from nonpersonalized cached fragments and personalized fragments which are typically not cached.

A third area we are working on is scalable techniques for achieving cache consistency when an accelerator is not tightly coupled with a server. An accelerator could be at a remote point in the network. Since expiration times are often insufficient for achieving strong cache consistency, servers must have the ability to invalidate content in remote accelerator caches. The cache consistency techniques need to be scalable to accommodate large numbers of accelerator caches and cached objects [28].

Acknowledgements

Several people have contributed to the accelerators deployed at the IBM Sporting and Event Web sites discussed in this paper including Gwen Adams, Paul Dantzig, Cameron Ferstat, Dave Johnson, and Burt Silverman.

References

[1] Network Appliance. Netcache: Highly Scalable Network and Web Caching Solutions. http://www.netapp.com/products/internet.html.

[2] 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.

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

[4] J. Challenger, A. Iyengar, K. Witting, C. Ferstat, and P. Reed. A Publishing System for Efficiently Creating Dynamic Web Content. In *Proceedings of IEEE INFOCOM 2000*, March 2000.

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

[6] Inktomi Corp. Traffic Server. http://www.inktomi.com/products/traffic/technology.html.

[7] IBM Corporation. IBM Net.Commerce. http://www.software.ibm.com/commerce/net.commerce/.

[8] Microsoft Corporation. Installation and Performance Tuning of Microsoft Scalable Web Cache (SWC 2.0). http://www.microsoft.com/technet/iis/swc2.asp.

[9] 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.

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

[11] A. Fox, S. Gribble, Y. Chawatbe, E. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *Proceedings of the Symposium on Operating Systems Principles*, October 1997.

[12] Y. Hu, A. Nanda, and Q. Yang. Measurement, Analysis and Performance Improvement of the Apache Web Server. Technical Report 1097-0001, Dept. of Electrical and Computer Engineering, University of Rhode Island, October 1997.

[13] Y. Hu, A. Nanda, and Q. Yang. Measurement, Analysis and Performance Improvement of the Apache Web Server. In *Proceedings of the 18th IEEE International Performance, Computing, and Communications Conference (IPCCC'99)*, February 1999.

[14] 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.

[15] CacheFlow Inc. CacheFlow. http://www.cacheflow.com/Default.html.

[16] 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.

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

[18] A. Iyengar, M. Squillante, and L. Zhang. Analysis and characterization of large-scale Web server access patterns and performance. *World Wide Web*, 2(1,2):85–100, June 1999.

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

[20] E. Levy, A. Iyengar, J. Song, and D. Dias. Design and Performance of a Web Server Accelerator. In *Proceedings of IEEE INFOCOM'99*, March 1999.

[21] A. Luotonen and K. Altis. World Wide Web proxies. *Computer Networks and ISDN Systems*, 27:147–154, 1994.

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

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

[24] D. Rosu, A. Iyengar, and D. Dias. Web Proxy Acceleration. *To appear in Cluster Computing.*

[25] D. Rosu, A. Iyengar, and D. Dias. Hint-based Acceleration of Web Proxy Caches. In *Proceedings of the 19th IEEE International Performance, Computing, and Communications Conference (IPCCC' 2000)*, February 2000.

[26] J. Song, A. Iyengar, E. Levy, and D. Dias. Design Alternatives for a Scalable Web Accelerator. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software*, April 2000.

[27] W. Stevens. *TCP/IP Illustrated*, volume 1. Addison-Wesley, 1997.

[28] J. Yin, L. Alvisi, M. Dahlin, and A. Iyengar. Engineering Server-Driven Consistency for Dynamic Web Services. In *Tenth International World Wide Web Conference Proceedings (WWW10)*, pages 45–57, May 2001.