DESIGN ALTERNATIVES FOR SCALABLE WEB SERVER ACCELERATORS

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

Abstract—**We study design alternatives for, and describe implementations and performance of, a scalable and highly available Web server accelerator. The accelerator runs under an embedded operating system and improves Web server performance by caching data. 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. Based on measurement of implementations, we quantify the throughput ranges in which different designs are preferable. We also examine a combination of content based and TCP routing techniques. We examine optimizations, such as different communication and data delivery methods, replication of hot objects, and cache replacement policies that take into account the fact that there might be different bottlenecks in the system at different times; depending upon which resource is likely to become a bottleneck, a different cache replacement algorithm is applied.**

I. Introduction

The performance of Web servers is limited by several factors. In satisfying a request, the requested data is often copied several times across layers of software, for example 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 or Web server accelerators. 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 paper, we examine design alternatives for a scalable Web server accelerator which caches data on multiple processors for both improved performance and high availability. The system consists of a cluster of Web accelerator nodes. The cluster design alternatives are the focus in this paper. The Web accelerator node, on which our implementation is based, is described in [2] and runs under an embedded operating system and can serve up to 5000 pages/second on a uniprocessor functioning as a single cache node. (The single cache node 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.)

In our architecture, a load balancer directs Web requests to one of several Web accelerator nodes. The load balancer takes on two basic forms: (i) A TCP router [3], [9], which for performance reasons does not examine the contents of a Web request, and (ii) a content router [10], which routes requests based on the URL requested. The Web URL space is partitioned among the Web accelerator cache nodes; thus, each request for a URL has a corresponding partition owner at a Web accelerator node.

The TCP router directs requests to accelerator nodes based only on the load on these nodes, without examining the content of the request. (The TCP router does not complete the TCP connection; 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.) Using this approach, there is a high probability that a request for a cached object will initially be routed to a first 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 one of two methods. If the requested object is small, the first node obtains the object from the second cache using a UDP interface and then sends the object back to the client. If the object is large, better performance is obtained when the first cache hands off the request, along with the TCP connection, to the second cache, which we refer to as a TCP handoff. The second cache then responds directly to the client without going through the first cache.

The usage of the TCP router for a scalable Web server can be found in [3]. The study in [3] is about a scalable Web server composed of multiple Web server nodes and supported by traditional file sharing mechanisms, or replicated files. In contrast, this work 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 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.

In order to reduce the probability of the TCP router routing a request for a cached object to a wrong node, hot objects are

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

0-7803-6418-X/00/$10.00 © 2000 IEEE

replicated on multiple cache nodes. The replication policy is integrated with cache replacement and routing strategies. The replication and replacement algorithm for our system takes into account the fact that there might be different bottlenecks in the system at different times. The TCP router, cache nodes, Web server nodes, and the network are all potential bottlenecks. Depending on which set of resources is a bottleneck, a different combination of cache replacement and routing strategies is applied. Our cache replacement algorithm can be applied to other systems with multiple resources which could become bottlenecks and in which the cache replacement policy affects utilization of these resources.

The content router has the ability to examine a request in order to route the request to the proper cache node. This approach is known as content-based routing [10]. In order to perform content-based routing, the load balancer completes the TCP connection, and includes the URL in its routing decision. As we quantify later, compared to the TCP router described above, content-based routing reduces CPU usage on the cache nodes. However, it adds significant overhead to the front-end load balancer by completing the TCP connection 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.

A. Outline of the Paper

Section II describes the architectural alternatives examined in detail. Section III presents our cache replacement and replication algorithm. Section IV compares the performance of the different architectural alternatives examined. Related work is discussed in Section V. Finally, concluding remarks appear in Section VI.

II. Web Server Accelerator Design and System Flows

A. System Overview

As illustrated in Figure 1, our Web server accelerator consists of a load balancer that directs Web requests to a set of Web accelerator cache nodes; each cache node is referred to as a cache member. As outlined earlier, the load balancer could be a TCP router, which does not examine the contents of the requests, or a content router. The Web accelerator front-ends one or more Web server nodes. 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.

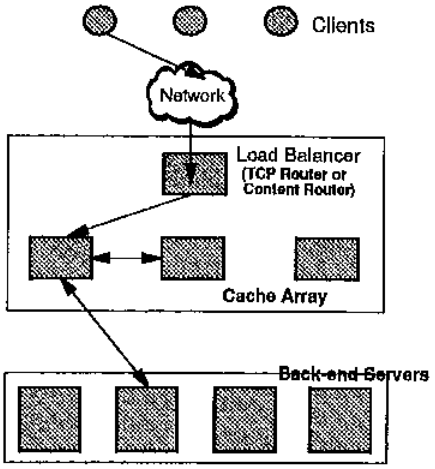

Fig. 1. System Structure

We mention in passing that the superior performance of individual cache members results largely from the embedded operating system and its highly optimized communications stack. Details can be found in [2]. 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.

The accelerator 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 API's 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

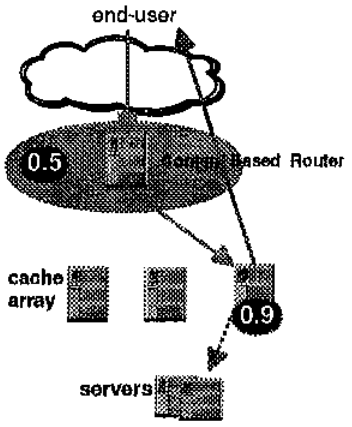

Fig. 2. Content-based Router as Load Balancer

is essential to cache dynamic pages to improve performance [5], [6]

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 1, 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. The 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.

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 requests to an owner. When the load balancer operates as a TCP router, it sends a request to a first member using a weighted round-robin policy; if the first member is not an owner of the requested object, the first member receiving the request determines an owner of the object. Then, the first member and the owner communicate to return the object to the client. This communication and delivery of data is done in multiple ways, *i.e.*, using HTTP, a UDP interface, or a TCP connection handoff.

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. In Section III, we also examine a combination of content-based and TCP routing techniques.

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 of our implementation 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.

B. Request Flows Through the System

B.1 Content Router as the Load Balancer

When the load balancer is a content-based router, the HTTP request is examined in order to route the request to an owner of a requested object. In order to examine a request, 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 [10]. Our measurement of CPU overheads shows that, in this scheme, the relative CPU cost incurred at the content router is 0.5 and that at the cache node is 0.9 (see Figure 2).

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 II-B.2.e 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 re-

quests 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. Section III describes how the proportion of requests routed by examining content can be varied to maximize throughput in conjunction with cache replacement and replication.

B.2 TCP Router as the Load Balancer

We say that a *cache member hit* occurs when the first member receiving a request from the TCP router is an owner (primary or secondary) of the requested object (Figure 3). Likewise, a *cache member miss* indicates the case when the first member is not an owner of the object (Figures 4, 5). 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;
   (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.

B.2.a 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 3). In both cases, requested objects are returned directly from a cache member to the client without going through the TCP router.

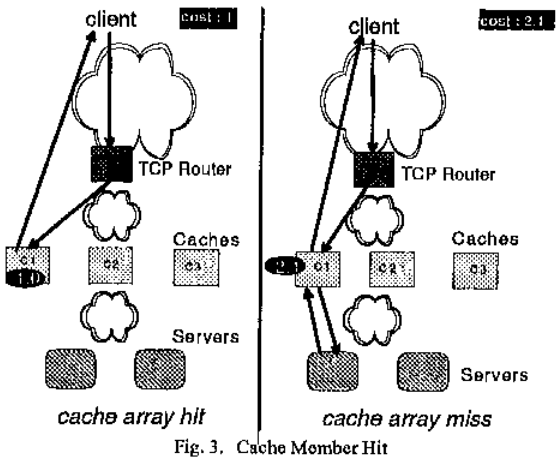

Fig. 3. Cache Member Hit

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.

B.2.b Cache member miss - HTTP interface.

When no replication is used, a cache member miss occurs roughly n-1 times out of n in a perfectly balanced system with n cache members. When this happens, the first member accepts the connection with the client, computes an owner of the requested object, and contacts the owner to get the requested object. In Figure 4, 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

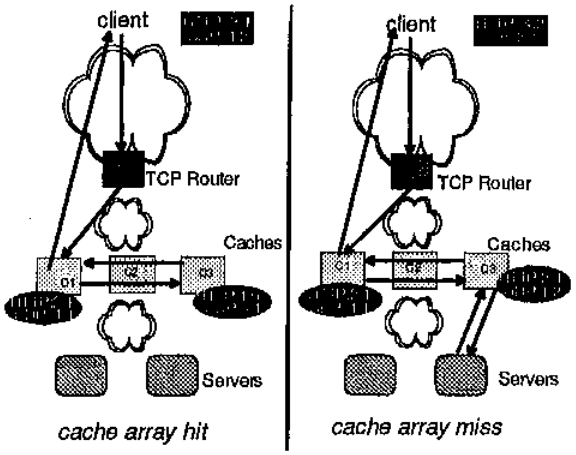

Fig. 4. Cache Member Miss: HTTP or UDP interface

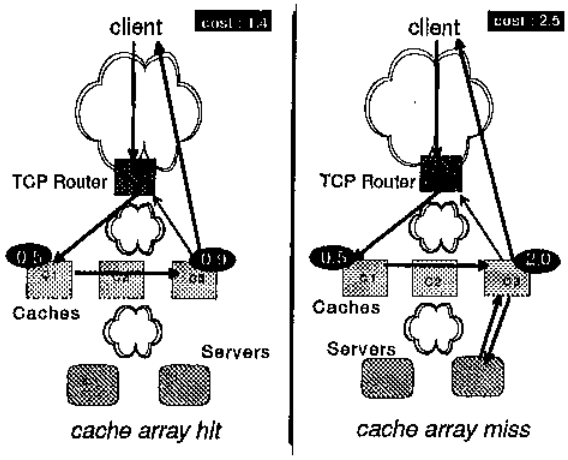

Fig. 5. Cache Member Miss: Handoff interface

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.

B.2.c 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 measurements show 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 4). 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.

B.2.d 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 5).

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.
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 [20]. 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 5). 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.

B.2.e 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 IV, 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).

C. 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) [12]. 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. URL's) 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. The method for choosing which objects to replicate for performance reasons is described in Section III.

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

III. Replication, Replacement, and Load Balancing

This section describes a new replication, replacement, and load balancing algorithm which is used by our Web accelerator to optimize performance. There are a number of key features of our algorithm which can be applied to other systems as well:

- The cache replacement policy affects cache array cycles, back-end server cycles, and network utilization. Different cache replacement strategies can be used to minimize

utilization of these three resources. Depending on which resources are actual or anticipated bottlenecks, different cache replacement policies can be used to achieve maximum throughput.

- The load balancer can employ different routing strategies to balance load across system resources. Routing methods are combined with cache replacement in order to achieve maximum throughput.
- Hot objects can be replicated in multiple cache members in order to reduce cache member misses and the accompanying cache array CPU cycles.

Our method uses a combination of cache replacement policies in which each replacement policy assigns a quantitative metric to objects which is correlated with the desirability of caching the object. Objects are ordered from most to least desirable on priority queues which require $O(log(n))$ instructions for performing insertions, deletions, and lookups, where n is the size of the priority queue [1]. Two metrics are maintained:

1. d_1 is a measure of desirability based on minimizing back-end server cycles.
2. d_2 is a measure of desirability based on minimizing cache array cycles.

Two priority queues are maintained, one for each metric. In order to handle cache replacement in a manner which minimizes back-end server cycles, the appropriate priority queue is examined to remove objects from the cache with the lowest d_1 values. In order to handle cache replacement in a manner which minimizes cache array cycles, the appropriate priority queue is examined to remove objects from the cache with the lowest d_2 values. In order to handle replacement in a manner which balances resource utilization to achieve maximum throughput, each priority queue would typically be selected a certain percentage of the time in order to identify cold objects to be replaced from the cache. It is possible to extend our method to handle other resources such as networks which could also become bottlenecks.

The desirability of caching an object based on minimizing back-end server cycles is the amount of back-end server CPU time saved by caching the object divided by the size of the object. The metric d_1 is an estimate of the number of hits per unit time if the object is cached multiplied by the back-end server processing time to fetch or generate the object divided by the size of the object.

The metric d_2 is calculated from empirical measurements of the overheads for different request flows through the system (see Section II-B). When content-based routing is not being used 100% of the time, it is sometimes desirable to replicate objects in order to reduce the number of cache member misses. Each copy of a replicated object has a d_2 value associated with it. The notation $d_2(i)$ represents the d_2 value for the ith copy of the object. For all $i > 1$, $d_2(i)$ results purely from the cache array cycles saved from decreased cache member misses as a result of adding the object to a new cache member after copies already exist in $i - 1$ cache members. Redundant copies do not reduce back-end server cycles. Therefore, $d_1(i) = 0$ for all $i > 1$.

The load balancer can employ two techniques to prevent the cache array from becoming a bottleneck. It can route some or all requests using content-based routing to reduce or eliminate cache member misses. This has the drawback of increasing load balancer CPU cycles and possibly turning the load balancer into a bottleneck. The load balancer can also route a certain percentage of requests directly to back-end servers without going through the cache array. This has the drawback of increasing cycles consumed by back-end servers.

The parameters which the system can modify in order to improve throughput thus include the following:

1. The frequencies with which d_1 and d_2 are used for cache replacement.
2. The percentage of requests which are sent via content-based routing to a cache member.
3. The percentage of requests which are sent from the load balancer directly to back-end servers, bypassing the cache array.

These parameters are selected at initialization time to provide maximum throughput based on the expected workload. They are then varied as necessary to increase throughput when one or more resources become bottlenecks.

IV. Performance

A. Methodology

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 WebStone benchmark [11]. 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. Therefore, we first measured the performance of a single processor accelerator. A Web server accelerator containing a single cache member can serve around 5000 cache hits per second and around 2000 cache misses per second for objects up to 2 KB. We measured the CPU overheads for the various cases described in Section II 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 the CPU overhead for TCP handoffs and other cases described in Section II for both slow and fast accelerators.

B. Results and Discussion

We first show how the system scales when we increase the number of cache nodes in the cache array. Figure 6 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

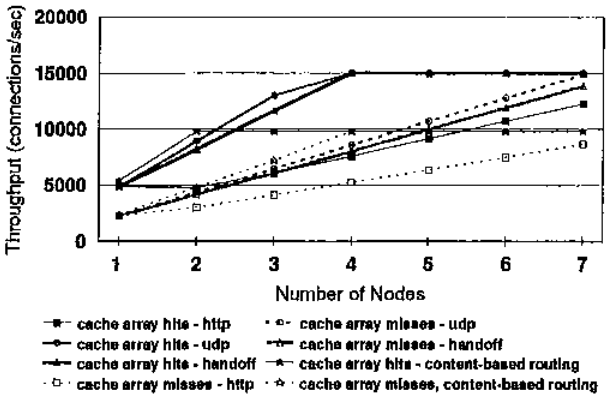

Fig. 6. Throughput versus number of cache nodes, up to 2 KB objects

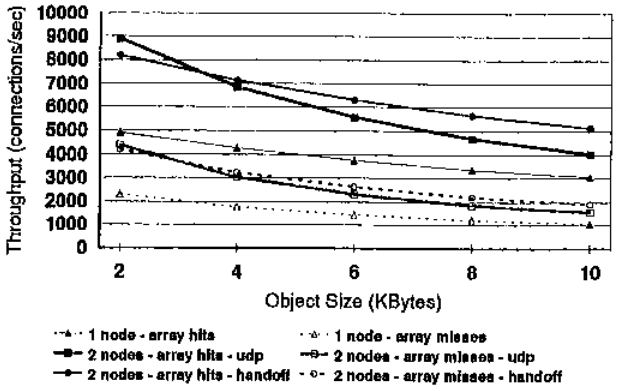

Fig. 8. Impact of object size, 2 K to 10 KB objects. This graph expands a portion of the graph in Figure 7.

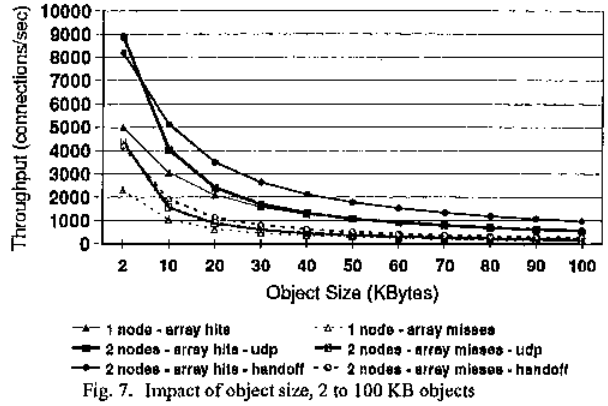

Fig. 7. Impact of object size, 2 to 100 KB objects

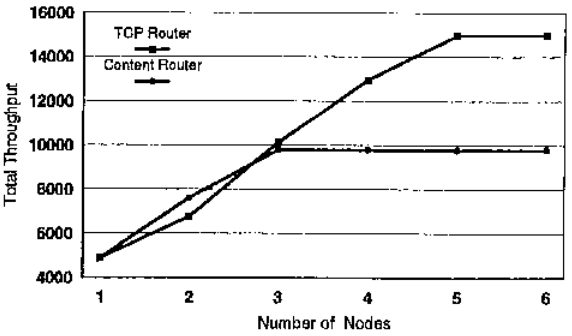

Fig. 9. Comparison of TCP and content-based routers.

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 7). 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 7 and 8 show the system performance when the sizes of the requested data 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 9 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.

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.

V. Related Work

Uniprocessor accelerators are described in [2], [4], [8]. Novell's BorderManager [7] is an example of a uniprocessor accelerator product currently on the market. Several Web proxy caches are available such as Inktomi's Traffic Server [14], [19], Network Appliance's NetCache [15], the CacheFlow 2000 [16], and IBM's Web Traffic Express [17]. 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 [3], [9]. Content-based routing is discussed by Pai et. al. in [10]. 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.

Considerable work has been done on cache replacement algorithms for Web caches mostly in the context of proxy caching [13]. However, we are not aware of previous cache replacement strategies similar to ours which combine several individual cache replacement algorithms designed to minimize different system resources. In response to a system resource becoming a bottleneck, the cache replacement algorithm is modified to reduce consumption of the bottleneck resource. Our cache replacement strategy is integrated with replication and routing techniques for load balancing.

VI. Conclusions

We have presented design alternatives for a scalable and highly available Web server accelerator, and have quantified the efficiency and scaling achieved by implementations of the schemes. The accelerator improves Web server performance by caching data and runs under an embedded operating system. 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.

Our system includes a load balancer sending requests to multiple processors collectively known as a cache array. The load balancer takes on 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. When this approach is used, a request for a cached object is often sent to the wrong node. If the requested object is small, the first node receiving the request obtains the requested object from a node containing a cached copy and then sends the object back to the client. If the requested object is large, better performance is obtained when the first node hands off the request, along with the TCP connection, to a node containing a cached copy of the requested object. The node containing the cached copy of the requested object then sends the object directly to the client without going through the first node or the load balancer.

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. The replication policy is integrated with cache replacement and routing policies. The replication, replacement, and routing algorithm for our system takes into account the fact that there might be different bottlenecks in the system at different times such as the TCP router, cache nodes, back-end server nodes, or network. Depending on which set of resources is a bottleneck, a different combination of cache replacement and routing policies is applied.

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.

As described herein, the Web cache accelerator scales until the load balancer becomes the bottleneck. As described in [3], it is possible to use more than one load balancer, using a domain name server (DNS) to perform coarse grained load balancing among the load balancers, and providing fine grained balancing using the load balancers.

Our architecture can be applied to other systems of multiple cache nodes on the Web and not just to scalable Web server accelerators. In addition, our cache replacement method can be applied to other systems with multiple resources which could become bottlenecks and in which the cache replacement policy affects utilization of these resources.

References

[1] A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[2] 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.

[3] 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.

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

[5] 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.

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

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

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

[9] 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, 1998.

[10] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum, *Locality-Aware Request Distribution in Cluster-based Network Servers*. In Proceedings of ASPLOS-VIII, 1998.

[11] Mindcraft, *Mindcraft - WebStone Benchmark Information*. http://www.mindcraft.com/webstone/.

[12] A. Luotonen, *Web Proxy Servers*, Prentice Hall, 1997.

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

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

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

[16] CacheFlow Inc., *CacheFlow*, http://www.cacheflow.com/Default.html

[17] IBM, *Web Traffic Express*, http://www.software.ibm.com/webservers/wte

[18] Cisco Systems Inc., *Local Director*, http://www.cisco.com/warp/public/751/lodir/index.shtml

[19] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gautbier, *Cluster-Based Scalable Network Services*, Proc. 1997 Symposium on Operating Systems Principles, St-Malo, France, Oct. 1997

[20] W. Richard Stevens, *TCP/IP Illustrated*, Vol 1, Addison-Wesley, 1997