A Scalable and Highly Available Web Server Accelerator

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 describe the design, implementation and performance of a scalable and highly available Web server accelerator which runs under an embedded operating system and improves Web server performance by caching data. A single cache node implemented on a uniprocessor 200 MHz PowerPC 604 can serve up to 5000 pages/second, a figure which is an order of magnitude higher than that which would be achieved by a high-performance Web server running under a conventional operating system such as Unix or NT. 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 is not a bottleneck. Requests are routed to cache nodes using a combination of content-based routing and techniques which don't examine contents. Content-based routing reduces cache node CPU cycles but can make the front-end router a bottleneck. When content-based routing is not used, a request for a cached object may initially be sent to the wrong node. In order to reduce the likelihood of this occurring, hot objects are replicated on multiple cache nodes. The replication policy is integrated with cache replacement and load balancing. The cache replacement policy takes 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.

.

1. 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 obtain optimal performance, all data 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 describe a scalable Web server accelerator we have developed which caches data on multiple processors for both improved performance and high availability. Our accelerator runs under an embedded operating system and can serve up to 5000 pages/second with a uniprocessor 200 MHz PowerPC 604 functioning as a single cache node [2]. This throughput is an order of magnitude higher than that which would be achieved by a high-performance Web server running on similar hardware under a conventional operating system such as Unix or NT. The 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 router is not a bottleneck.

In our architecture, a TCP router directs requests to one of several processors functioning as caches. In the basic mode [3], which minimizes TCP router cycles, incoming requests are distributed to caches without examining their content. (In this mode, 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 node completes the connection, and directly returns the requested Web page to the client. Details of how this is accomplished are in [3, 9].) 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 the 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. The second cache then responds directly to the client without going through the first cache.

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 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.

In an extended mode, the TCP 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 TCP router completes the TCP connection, and includes the URL in its routing decision. As we quantify later, completing the TCP connection for content routing results in significant additional overhead as compared to the basic TCP router outlined above. While content-based routing reduces CPU usage on the cache nodes, it adds overhead to the TCP router which can result in the TCP router 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.

1.1 Outline of Paper

Section 2 describes the architecture of our system in detail. Section 3 presents our cache replacement and replication algorithm. Section 4 describes the performance of our system. Related work is discussed in Section 5. Finally, concluding remarks appear in Section 6.

2. Web Server Accelerator Design and System Flows

2.1 System Overview

As illustrated in Figure 2.1, our Web server accelerator consists of a TCP router that directs Web requests to a set of processors containing caches. The TCP router in our Web accelerator has the routing and load balancing capabilities of IBM's Network Dispatcher (ND)[9]. In addition, it can route some or all requests using content-based routing, a feature not present in Network Dispatcher. The Web accelerator front-ends one or more Web server nodes. Each processor containing a cache is known as a *cache member*. The set of of all cache members is known as a *cache array***.** From a scaleability 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, while also increasing the system throughput.

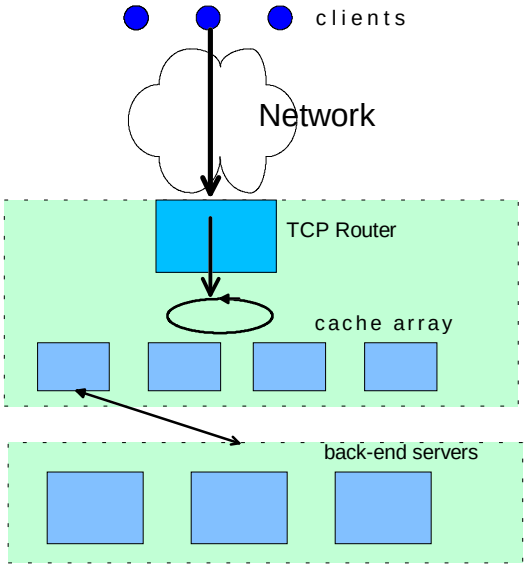


Figure 2.1: System Structure

The superior performance of each cache member 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.

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

The TCP router 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 TCP router without clients being aware of it. In Figure 2.1, the TCP router runs on a separate node. This design results in maximum throughput. It is also possible to run the TCP router on a cache member node; this is useful for cases where the TCP router is not the bottleneck, e.g. for two nodes.

The TCP router 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

The TCP router, in extended mode, has the ability to examine requests and perform content-based routing to route each request to the primary owner. While content-based routing reduces CPU usage on the cache array, it adds overhead to the TCP router which can result in the TCP router becoming a bottleneck. In order to minimize CPU utilization on the TCP router, content-based routing is not used all of the time.

The cache member which initially receives a request from the TCP router is known as the *first member*. If the TCP router sends a request to a first member which is not the primary owner of the requested object, the first member receiving the request determines the primary owner of the object. The first member sends the request to the primary owner, along with relevant information on the TCP connection (sequence numbers, ip addresses, tcp ports, etc.). The primary owner is responsible for fetching the requested object either from its cache or from a back-end server if the page is not cached. The primary owner returns the requested object using the following method:

1. If the requested object is less than a threshold size, then the primary owner returns the requested object to the first member which then returns the object to the client.
2. if the requested object is larger than the threshold size,
   (i) The first member hands off the TCP connection to the primary owner. A similar TCP handoff protocol is described in [10].
   (ii)The primary owner informs the TCP router that the request will now be handled by the primary owner (i.e., further requests in the corresponding TCP session with the client are to be sent to the primary owner);
   (iii) The primary owner returns the requested object directly to the client without going through the first member.

The threshold size is chosen to optimize performance. For small objects, the first method of returning the object offers superior performance. For large objects, the second method of returning the object offers superior performance.

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

One problem with the basic scheme as outlined above is that in many cases, two cache members are required to satisfy requests which can result in overhead. In order to reduce the proportion of requests which require communication between two cache members, the hottest pages may be replicated on multiple cache members. Replication is important for high availability as well. If a hot object is stored in multiple caches, it will remain in cache even if the primary owner of the object goes down.

Each cache which contains a copy of a replicated object is an *owner* of the object. One node is designated as the primary owner and continues to be the primary owner of the object even if the object is not in the cache array. By contrast, a *secondary owner* of an object is only so designated while it contains a cached copy of the object. If an object is contained in the cache array, a copy of the object will be contained in its primary owner unless the primary owner is down or recently restarted. The replication and replacement algorithm we use is described in more detail in Section 3.

2.2 Request Flows Through the System:

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. 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 how 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. The first member and the primary owner can communicate via HTTP with the first member acting as an HTTP proxy. Alternatively, the two cache members can communicate via a UDP interface. The UDP interface offers superior performance to the HTTP interface, and is feasible in a cache cluster because the packet loss rate is minimal. Finally, the first member can hand off the request to the primary owner. As we shall see, the UDP interface offers superior performance for small objects while the handoff interface offers superior performance for large objects.

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,

   i. page retrieved using HTTP;
   ii. page retrieved using UDP;
   iii. page retrieved via a request handoff.
4. Cache member miss, cache array miss,
   i. page retrieved using HTTP;
   ii. page retrieved using UDP;
   iii. page retrieved via a request handoff.

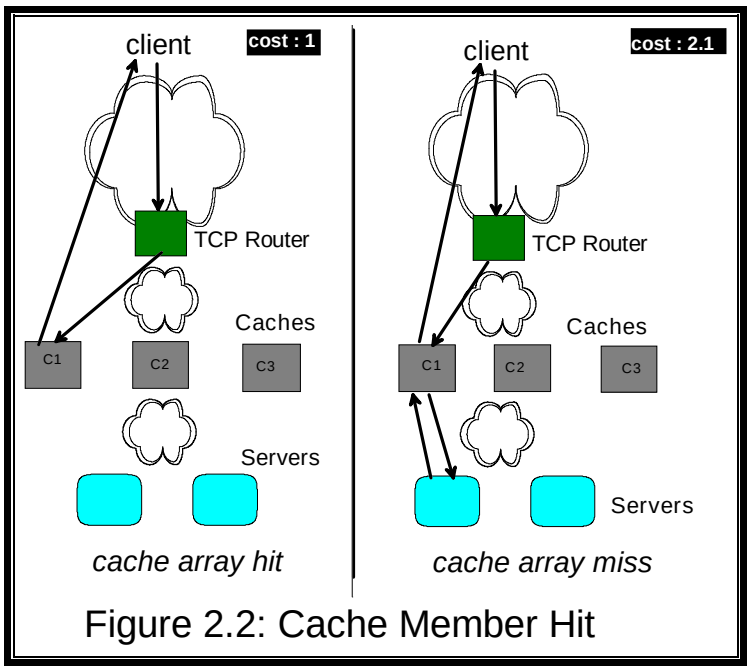


Figure 2.2: Cache Member Hit

2.2.1 Cache member hit

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. 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 2.2). In all cases, requested objects are returned directly from a cache member to the client without going through the TCP router (see [3] for details).

In order to compare the overheads for different flows through the system, we assign a relative cost of **1** for the cache array CPU cycles consumed by a request for an object of up to 2 Kbytes which results in a cache member hit and a cache array hit. (We have measured the relative CPU cost for objects less than 2 Kbytes to be almost the same, regardless of size.)

We have measured the relative CPU cost, at the cache nodes, of a request for an object of up to 2 Kbytes which results in a cache member hit and a cache array miss to be **2.1**. This can be 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.

2.2.2- Cache member miss - HTTP interface

A *cache member miss* occurs when the first member receiving a request from the TCP router is not an owner (primary or secondary) of the requested object (Figure 2.3). When this happens (n-1 times out of n in a perfectly balanced system with n cache members), 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 2.3, 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 Kbytes 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 resulting in 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 only sent once (from the owner to the client) resulting in only once 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.

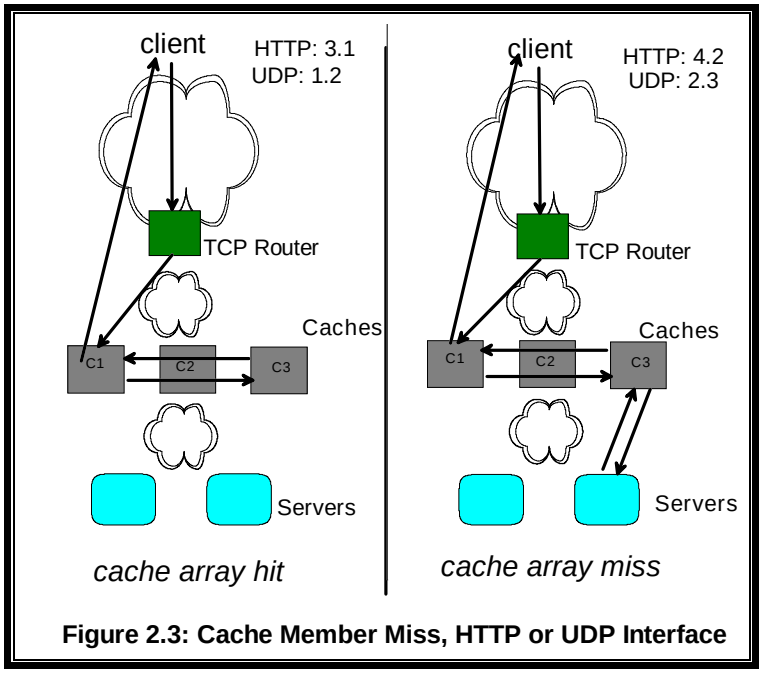


Figure 2.3: Cache Member Miss, HTTP or UDP Interface

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

2.2.3 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. The cache array CPU cost of a cache member miss (for objects up to 2 Kbytes) resulting in a cache array hit is only **1.2** using the UDP interface (Figure 2.2). UDP has less 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 Kbytes) 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. The extra overhead for serving pages larger than 2 Kbytes using the UDP interface is similar to that incurred by the HTTP interface.

While UDP is not as reliable as HTTP for communicating across the Internet, the unreliability of UDP was not a factor in our system because cache members were communicating directly with each other on a private network without going through any intermediate nodes.

2.2.4 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 which then sends the requested object directly back to the client without going through the first member. This process is possible because of several key features/mechanisms in the cache array :

1. The first member and the owner share the same IP address, which is the *cluster address.* Therefore, they can both accept requests for that cluster address and respond to the client.
2. A mechanism, which can be seen as a TCP kernel extension, is implemented to allow the transfer of the TCP connection from one cache member (one TCP stack) to another (a second TCP stack). The UDP protocol is used to transport information between the two cache members during that transfer, in order to minimize the transfer cost.
3. Along with this TCP transfer, another mechanism is implemented to transfer the HTTP request from one cache member to the other.

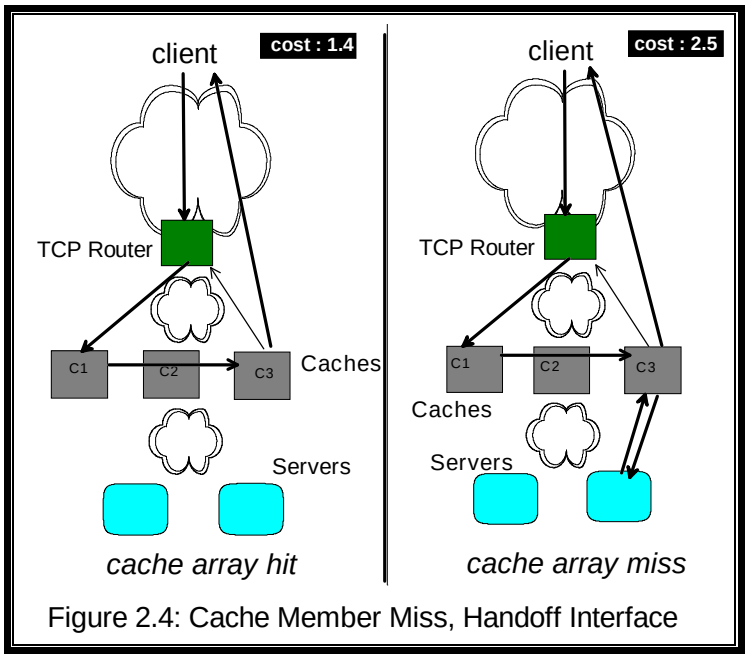


Figure 2.4: Cache Member Miss, Handoff Interface

4. A TCP "takeover" can take place because of the TCP router which front-ends the cache array. When the TCP/HTTP flow is transferred between the two cache members, it is also necessary to make sure that the incoming flows (from client to server) reach the new "owner" of the connection. In order to accomplish that, the owner requests the TCP router to update its entry for that connection, so that it will send all subsequent packets to the owner instead of the first member.

This mechanism is illustrated in Figure 2.4

The cache array CPU cost of a cache member miss (for objects up to 2 Kbytes) when the handoff interface is used is ***1.4*** for a cache array hit and ***2.5*** for a cache array miss. The additional overhead in the latter case 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 Kbytes 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 Kbytes for the handoff interface is similar to that which would be incurred by a cache member hit.

2.2.5 Cache member miss - Optimal Strategy

The UDP interface offers the best performance for small objects. The handoff interface offers the best performance for large objects. Therefore, the optimal strategy for handling cache member misses is to use the UDP interface for small objects and the handoff interface for large ones. 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 Kbytes and 4 Kbytes.

Our system implements the optimal strategy for cache member misses using 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 takeover followed by TCP and HTTP handoffs. There is no need for the first member to do anything during this process.
5. Asynchronously, the owner informs the first member to clean up connection information corresponding to the request.

2.2.6 Content-Based Routing

The TCP router also has the ability to perform content-based routing in which a request is examined in order to route the request to the owner of a requested object and avoid cache member misses. In order to examine a request, the TCP router has to complete a connection with the client. After the TCP router has examined a request and selected an owner of the requested object, it uses one of two methods for sending the request to an owner. In the first method, a strategy similar to the optimal one described in Section 2.2.5 is applied. If the requested object is small, the object is first returned from the owner to the TCP router via a UDP interface. It is then returned from the TCP router to the client. If the requested object is large, the owner performs a TCP takeover followed by TCP and HTTP handoffs and responds directly to the client without going through the TCP router.

The advantage to this method is that it incurs very little overhead on the cache array for small objects. For objects up to 2 Kbytes, 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 TCP router. For objects up to 2 Kbytes, the relative cost for the UDP interface is ***1.1*** in TCP router CPU cycles.

The second approach is for the TCP 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 TCP router. A similar method for performing content-based routing is presented in [10]. In Section 2.2.4, we described that the relative CPU cost for the handoff interface is ***1.4*** . We have measured that this cost is divided into ***0.5*** at the TCP Router, and ***0.9*** at the owner of the requested object. This approach incurs more total overhead for small objects. The overhead at the TCP router is lower while the overhead at cache members is higher. For

large objects, both content-based routing approaches use the handoff interface and hence have similar performance.

While the content-based routing approach reduces CPU cycles consumed by cache members, the disadvantage is that it consumes extra CPU cycles on the TCP router which can make the TCP router a bottleneck. A TCP router running on a 200 MHz PowerPC 604 can route 15 K requests without doing content-based routing, 9 K requests using the handoff mechanism for content-based routing, and 4 K requests using the UDP interface for content-based routing. If cache array CPUs are the bottleneck, content-based routing is a good approach. If, on the other hand, the TCP router is a bottleneck, content-based routing should not be used. If it is not clear whether the TCP router 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 3 describes how the proportion of requests routed by examining content can be varied to maximize throughput in conjunction with cache replacement and replication.

2.3 High Availability

The TCP router has the ability to detect when a cache member fails. When this happens, the TCP router informs the remaining live members of the cache array of the failure and directs all requests to live members of the cache array. Replication of hot pages on multiple cache members minimizes higher cache array miss rates resulting from a cache member failure.

Our system hashes objects across the cache array using an enhanced versions of CARP which provides replication for both high availability and improved performance. CARP (Cache Array Routing Protocol) [12] 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. The primary owner for an object is the one resulting in the highest combined score.

It may be desirable to store copies of an object in two different cache members so that a copy will still be available if one of the cache members fails. It may also be desirable to store copies of an object in two or more cache members in order to improve performance by increasing the probability of cache member hits. 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 score for the objects. The method for choosing which objects to replicate for performance reasons is described in Section 3.

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.

It is possible to configure our system with a backup TCP router node to handle failure of a TCP router.

3. Replication, Replacement, and Load Balancing

3.1 - A General Strategy for Handling Multiple Potential System Bottlenecks

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 TCP router 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 basic approach for cache replacement can be applied to other systems where the cache replacement policy affects multiple resources in the system. It may not be possible to come up with a monolithic replacement policy which minimizes utilization of all system resources. Instead, one cache replacement policy might minimize utilization of a resource *a* but not resource *b.* Another cache replacement policy might minimize utilization of resource *b* but not *a*. In this situation, the following approach should be used:

1. Use a combination of cache replacement policies, each designed to minimize utilization of a critical resource which could become a system bottleneck.
2. The frequencies for using the different cache replacement policies should be based on the expected mix which would maximize system throughput.
3. In response to a critical resource becoming a bottleneck, increase the frequency with which the cache replacement policy designed to minimize utilization of the critical resource is applied.

This basic approach can be applied to a variety of cache replacement algorithms described in the literature [13] and not just to the specific ones we describe in the next subsection

.

3.1 - Applying the General Strategy to our System

One method for using a combination of cache replacement policies would be to manage a certain fraction of the cache space using one policy and other parts of the cache using other policies. In our method which is slightly different, 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. *d1* is a measure of desirability based on minimizing back-end server cycles.
2. *d2* 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 in order to remove objects from the cache with the lowest *d1* values. In order to handle cache replacement in a manner which minimizes cache array cycles, the appropriate priority queue is examined in order to remove objects from the cache with the lowest *d2* 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 *d1* 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 *d2* is calculated from empirical measurements of the overheads for different request flows through the system (see Section 2.2). 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 *d2* value associated with it. The notation *d2(i)* represents the *d2* value for the *ith* copy of the object. For all $i>1$, *d2(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 never reduce back-end server cycles. Therefore, $d1(i)=0$ for all $i>1$.

The TCP router 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 TCP router CPU cycles and possibly turning the TCP router into a bottleneck. The TCP router 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. We will assume for the remainder of this section that a request can be handled by any back-end server.

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

1. The frequencies with which *d1* and *d2* are used for cache replacement.
2. The percentage of requests which are sent via content-based routing to a cache member. This section assumes that a single method is used for content-based routing. Section 2.2.6 describes two methods for content-based routing, one which minimizes TCP router cycles for small objects and the other which minimizes cache array cycles for small objects. A straightforward extension of our load balancing algorithm would be to use a combination of these two content-based routing approaches.
3. The percentage of requests which are sent from the TCP router 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. If the TCP router becomes the sole bottleneck of the system, the percentage of requests sent to the cache array via content-based routing is decreased. If the TCP

router is still a bottleneck without any content-based routing, nothing further can be done to improve system throughput.

If the cache array is the sole bottleneck in the system, the system attempts to do one or more of the following:

1. Increase the percentage of objects which are removed from the cache array based on *d2* values.
2. Replicate cached objects in order to decrease cache member misses. The object to be replicated is the one with the highest *d2(1)* value not contained in all caches. Replication can continue until caching a new copy of an object would force out a set of one or more objects whose sum of *d2* values weighted by sizes is greater than the *d2* value for the new copy weighted by its size.
3. Increase the fraction of requests to the cache array which are routed based on content.
4. Increase the fraction of requests which are sent from the TCP router directly to a back-end server.

If the back-end servers are the sole bottleneck in the system, one or more of the following is attempted:

1. Increase the percentage of objects which are removed from the cache array based on *d1* values.
2. Decrease the percentage of requests which are sent from the TCP router directly to back-end servers.

If all objects are already being replaced based on *d1* values, no requests are being sent directly to back-end servers, and the back-end servers are still a bottleneck, nothing further can be done to improve system throughput.

If the TCP router, cache array, and back-end servers are all operating at 100% capacity, there is little that can be done to improve performance.

If only the cache array and back-end servers are operating at 100% capacity, not much can be done to improve performance unless the TCP router is routing less than 100% of requests to the cache array using content-based routing. If the TCP router is sending some requests without examining their contents, it increases the fraction of requests which are sent to the cache array via content-based routing. This should free up some CPU cycles on the cache array. The system then applies the previously described method for situations where the back-end servers are the sole bottleneck.

If only the TCP router and cache array are operating at 100% capacity, nothing can be done if the TCP router is not performing any content-based routing. If the TCP router is performing content-based routing, it frees up CPU cycles on the cache array by increasing the percentage of requests which are routed directly to back-end server nodes. It then frees up cycles on itself by reducing the percentage of requests which are sent to the cache array via content-based routing.

If only the TCP router and back-end servers are operating at 100% capacity, nothing can be done if the TCP router is not performing any content-based routing. If the TCP router is performing content-based routing, it reduces the fraction of requests which are routed based on content. This should result in the back-end servers being the only bottleneck in the system. The system then applies the previously described method for handling this situation.

4. Performance
4.1 - Methodology

We have built two types of scalable Web server accelerators. *Fast* accelerators are constructed from cache members which are 200 MHz PowerPC 604 processors. *Slow* accelerators are constructed from Motorola 68040 processors. A Web server accelerator containing a single fast cache member can serve around 5000 cache hits per second and around 2000 cache misses per second for objects up to 2 Kbytes. In the systems we have tested, all processors within a single cache array have the same processing power; we have not tested systems which mix fast and slow cache members within the same cache array.

We have obtained performance measurements for slow accelerators consisting of multiple cache members. Our test configuration doesn't have the network bandwidth to accurately determine the performance of fast accelerators containing multiple cache members. Instead, we project the performance of fast accelerators containing multiple cache members from the performance of slow accelerators and the performance of a fast accelerator containing a single cache member.

The system used to measure the Web accelerator throughput consisted of two SP2 frames containing a total of 16 nodes which were connected through four 16 Mbps token rings to the TCP router (Figure 4.1.1). Some of the SP2 nodes issued requests to the cache array by running the WebStone benchmark [11]. Also, some other nodes were used as back-end servers to handle cache array misses.

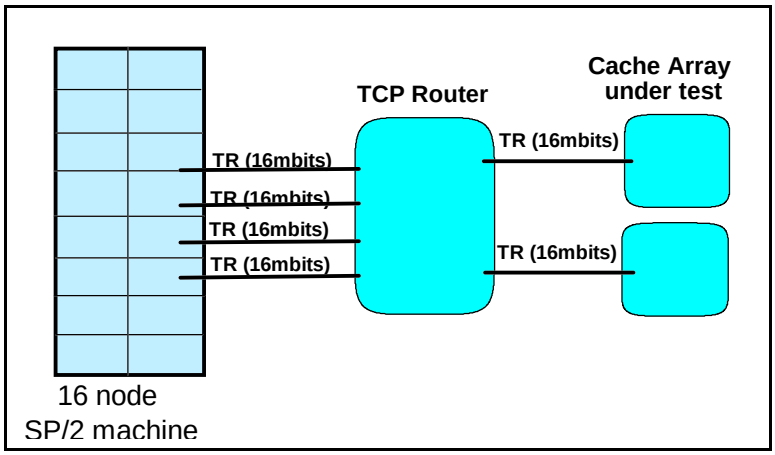
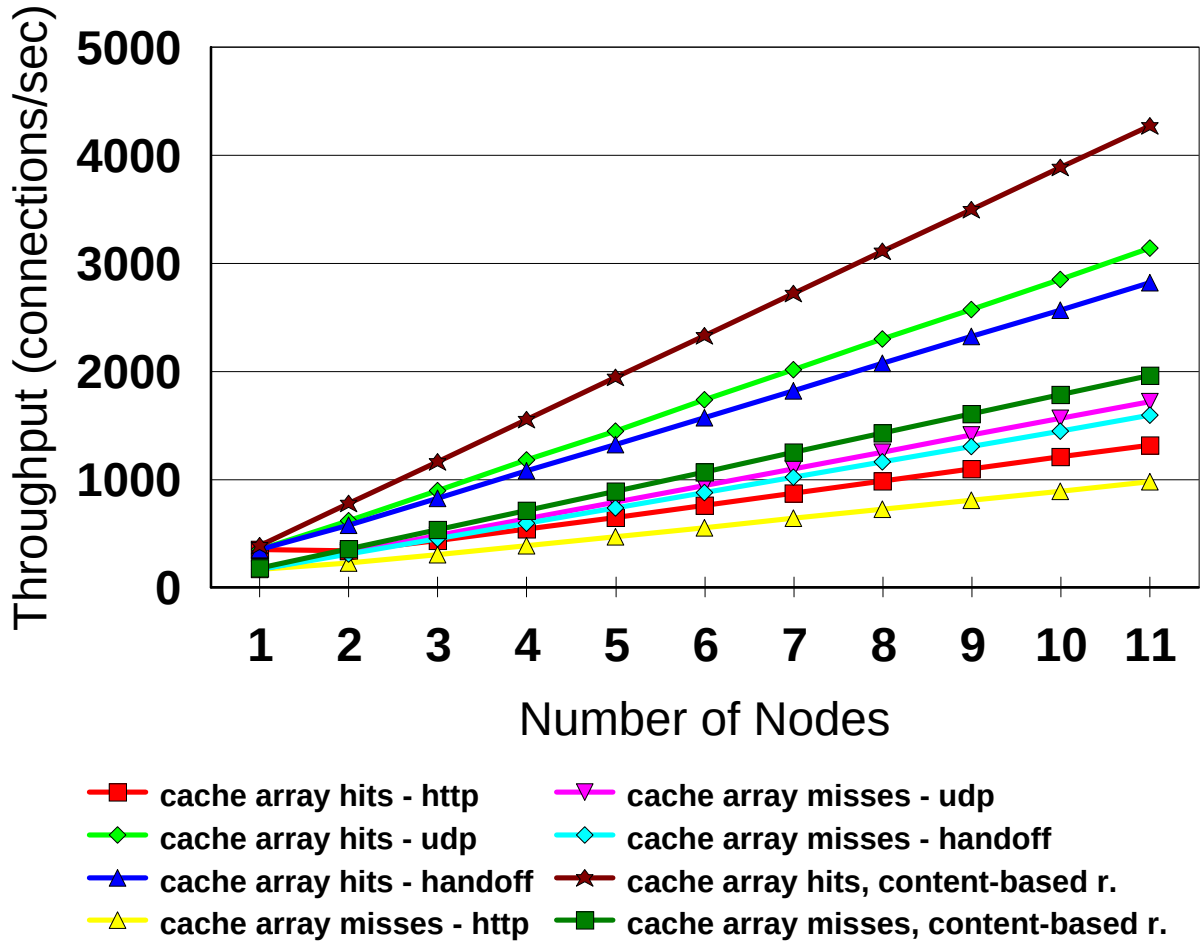
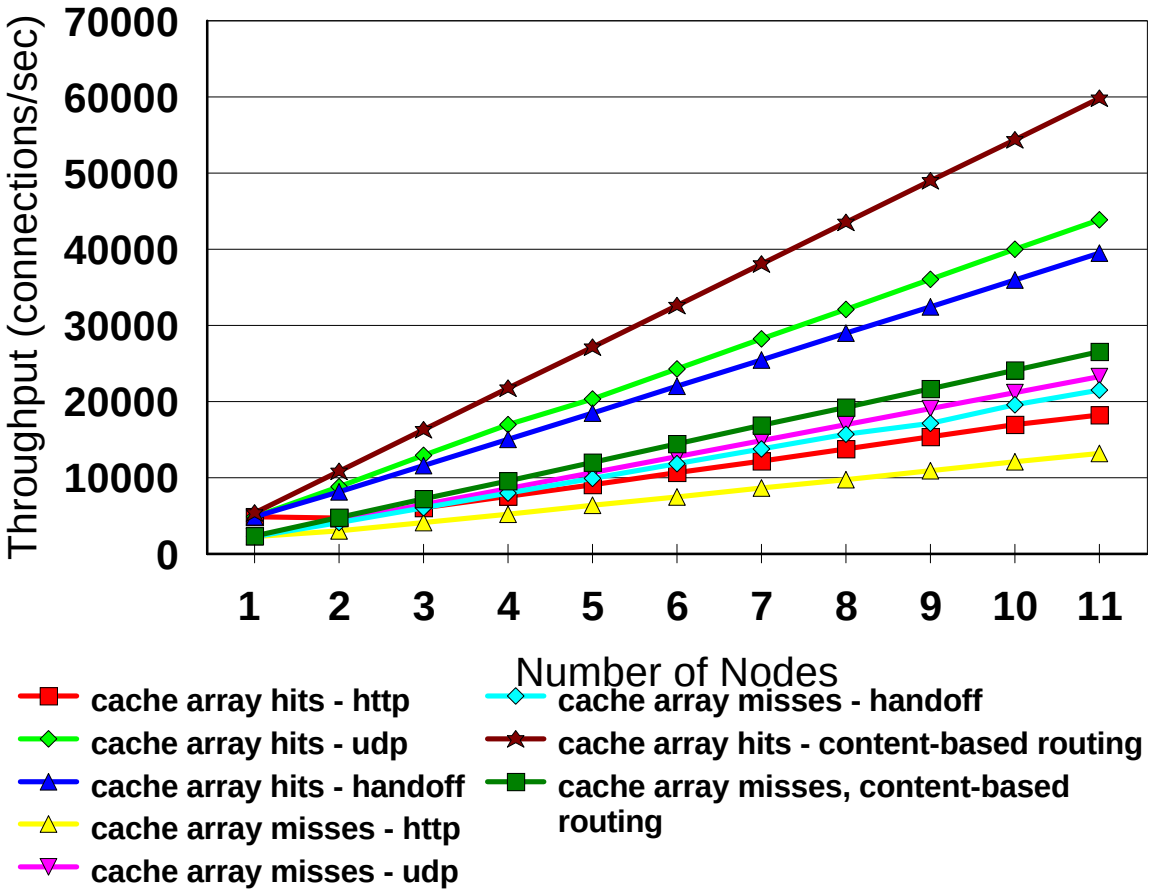


Figure 4.1.1: Experimental Setting

4.2 - Results

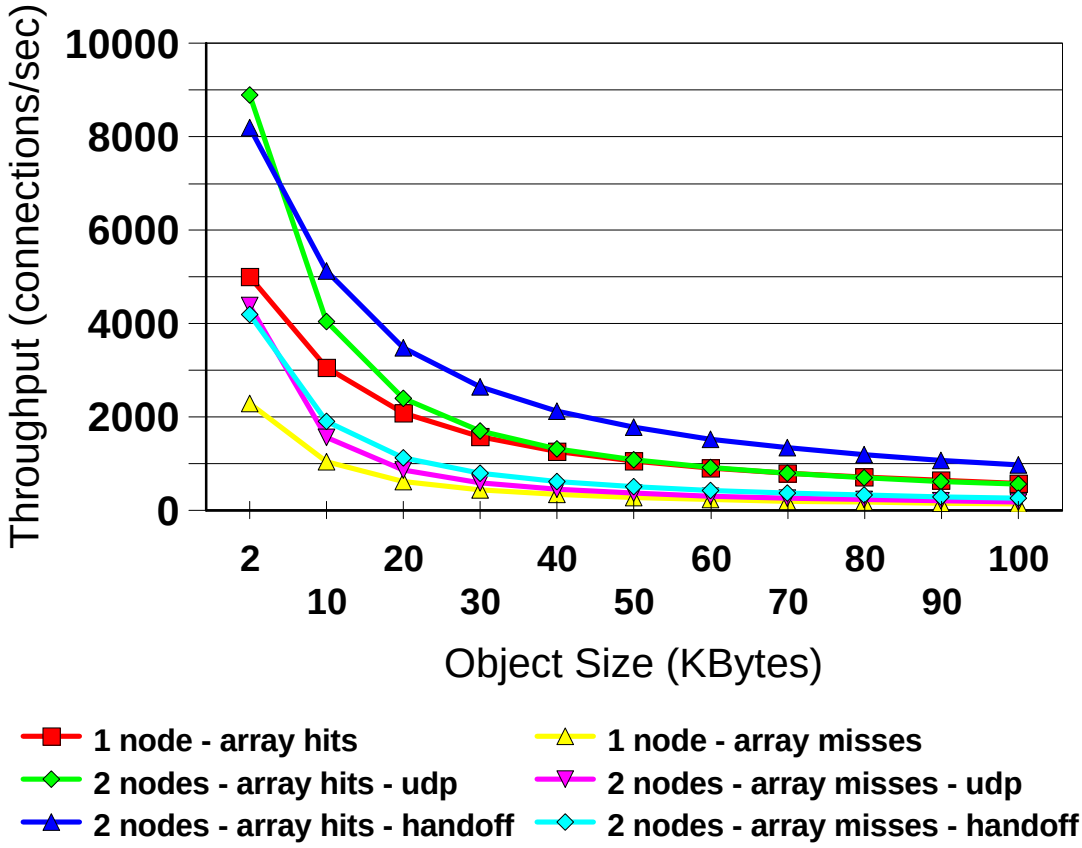


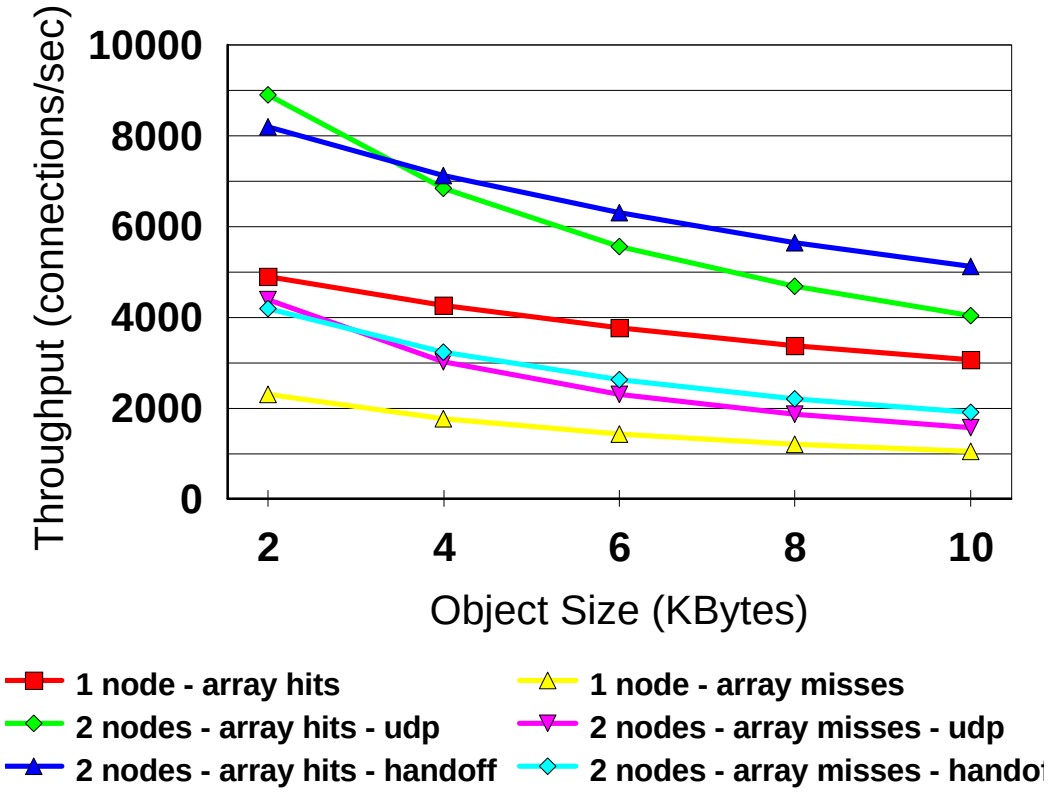
Graph 4.2.1: slow accelerator, up to 2 Kbyte objects



Graph 4.2.2: fast accelerator, up to 2 Kbyte objects

Graph 4.2.3 and 4.2.4: Fast accelerator, 2K to 100K objects





4.3 - Discussion

In Graphs 4.2.1 and 4.2.2, content-based routing results in the highest throughputs because cache member misses are avoided. The curves for content-based routing are for the implementation using the handoff interface. The drawback to content-based routing is that it adds overhead to the TCP router which can therefore become a bottleneck. A TCP router running on a 200 MHz PowerPC 604 can route 15 K requests without doing content-based routing, 9 K requests using the handoff mechanism for content-based routing, and 4 K requests using the UDP interface for content-based routing.

In all of the graphs in Section 4.2, the cache array did not perform any replication of hot objects on multiple cache members. Using replication, it should be possible to improve performance when content-based routing is not used. When the TCP router is a potential bottleneck, routing requests without examining content while replicating hot objects to reduce cache member misses is probably preferable to content-based routing.

Graphs 4.2.1 and 4.2.2 also show the superior performance of the UDP interface over the HTTP interface for intercache communication. The graphs even indicate that for cache array hits, use of 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 Kbytes (Graph 4.2.3). However, use of a 2-node system with the HTTP interface will still often outperform a single node. This is partly because miss rates for the 2-node system are usually less than that for a 1-node system since cache memory is doubled. In addition, a 2-node system using any of the three interfaces for handling cache member misses results in higher throughputs than a 1-node system for cache array misses for small objects.

For smaller objects, the UDP interface is superior to the handoff interface as illustrated by Graphs 4.2.1 and 4.2.2. For large objects, Graphs 4.2.3 and 4.2.4 show that the handoff interface is superior. The crossover point occurs when objects are between 3 and 4 Kbytes.

5. Related Work

Uniprocessor Web server accelerators are described in [2,4,7,8]. The design and performance of a uniprocessor version of our accelerator are described in detail in [2]. None of these previous papers explore how to scale Web server accelerators in order to achieve increased cache memory, request throughput, and availability.

Pai et. al. have done considerable work in the area of content-based routing [10]. They considered cluster-based network servers in which a front-end directs incoming requests to one of a number of back-end nodes, to choose which back-end will handle a request. In the experiments that they performed, their locality-aware request distribution strategy resulted in considerably better performance than weighted round-robin. They also presented a TCP handoff protocol which is similar to the one we use in our system. We view our work as being complementary to that of Pai et. al. We present strategies for handling requests when content-based routing is not used for all requests either because of the overhead this incurs at the router or other limitations of the system. We are not aware of any paper which presents and analyzes our algorithm for routing requests without examining content. We also presented a new method for content-based routing in which requests for small objects are handled using a UDP interface instead of a handoff interface.

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.

6. Conclusions

We have presented the architecture of a scalable and highly available Web server accelerator. Our accelerator improves Web server performance by caching data and runs under an embedded operating system. Our accelerator can serve up to 5000 pages per second with a uniprocessor 200 MHz PowerPC 604 functioning as a single cache node. 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. 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 router is not a bottleneck.

Our system includes a TCP router sending requests to multiple processors collectively known as a cache array. While content-based routing can be used to reduce CPU usage on cache nodes, it adds overhead to the TCP router which can result in the TCP router 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 via a UDP interface 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.

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 TCP router can be integrated into our system in order to handle router 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.

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. To appear 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. To appear in Proceedings of INFOCOM'99, March 1999.

[7] R. Lee. 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 Algorthms. In Proceedings of the USENIX Symposium on Internet Technologies and Systems, 1997.