The following paper was originally published in the

Proceedings of the 3rd USENIX Windows NT Symposium

Seattle, Washington, USA, July 12–13, 1999

HACC: AN ARCHITECTURE FOR CLUSTER-BASED WEB SERVERS

Xiaolan Zhang, Michael Barrientos, J. Bradley Chen, and Margo Seltzer

For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738
Email: office@usenix.org WWW: http://www.usenix.org

HACC: An Architecture for Cluster-Based Web Servers

Xiaolan Zhang, Michael Barrientos, J. Bradley Chen, Margo Seltzer
Division of Engineering and Applied Sciences, Harvard University

Abstract

This paper presents the design, implementation, and performance of the Harvard Array of Clustered Computers (HACC), a cluster-based design for scalable, cost-effective web servers. HACC is designed for *locality enhancement*. Requests that arrive at the cluster are distributed among the nodes so as to enhance the locality of reference that occurs on individual nodes in the cluster. By improving locality on individual cluster nodes, we can reduce their working set sizes and achieve superior performance for less cost than conventional approaches. We implemented HACC on Windows NT 4.0 and evaluated its performance for both static documents and workloads of dynamically generated documents adapted from logs of commercial web servers. Our performance results show that HACC's locality enhancement can improve performance by up to 121% for our stochastically generated static file case, by up to 40% for our trace-based static file case, and by up to 52% for our trace-based dynamic document case, compared to an IP-Sprayer approach to building cluster-based web servers.

1. Introduction

To handle the ever-increasing population of World Wide Web users, busy Web sites are frequently hosted on a cluster of computers. The load on these servers is further exacerbated by the trend towards *Web Application Servers (WAS)*, which generate documents on the fly, requiring a great deal of compute power on the servers. Clustered web servers are a natural solution to scaling the server for arbitrarily heavy loads. Researchers have studied clustered web server architectures extensively [DKM96, FGC97, KMR95, PAB98]. However, much of their research effort has been directed at support for static files. Anecdotal evidence suggests that the performance issues for WAS are more complex and intrinsically different from those of static file servers [CDW97]. WAS typically consist of a collection of distributed backend servers remotely connected to a web front-end. The performance of a WAS is usually limited either by the network delay between the front-end and the backend server, or by backend computing power [Ten99]. We call this "backend limited" as opposed to the conventional static file case where the server is usually "memory bound" or "disk bound", limited by the number of open sockets it can support. In this paper we present the Harvard Array of Clustered Computers (HACC), a cluster-based design to enhance performance for both static file service and WAS, with WAS as our target domain.

The conventional cluster-server approach puts a router or "IP-Sprayer" between the Internet and a cluster of web servers. The job of the router is to spread the load evenly over the nodes in the cluster. A number of commercial products [Cis96, Che97] employ this approach to distribute web site requests to a collection of machines, typically in a round-robin fashion while attempting to preserve affinity between users and server nodes. The purpose of the affinity is to give better behavior for web servers that maintain state about connected users. Increasing the aggregate performance of the cluster is simply a matter of adding more nodes, with scalability limited only by the capacity of the router.

This simple approach to clustering does a good job of addressing the scalability problem, but it is not a panacea. For example, a server node for a large or complicated web site might require a large amount of physical memory in order to handle requests efficiently. Each node added to the system will be responsible for the same document store, and so will require the same large physical memory. The result is that either the server nodes are expensive (they need a lot of memory), or they are slow (so you need many of them) or both. Overall this leads to an inefficient use of resources. Figure 1(a) shows a schematic representation of this situation. Notice that each node in the cluster is responsible for the same working set, namely the active elements of the entire document store.

HACC eliminates the inefficiencies in this system by two means: locality enhancement and dynamic load balancing. Rather than distributing requests in a round-robin fashion, HACC distributes requests so as to enhance the inherent locality of the request streams in the server cluster. We refer to this modified sprayer as a *Smart Router*, illustrated in Figure 1(b). Instead of being responsible for the entire working set, each node in the cluster is responsible for only a fraction of the document store. The size of each node's working set decreases each time a node is added to the cluster, resulting in more efficient use of resources at each

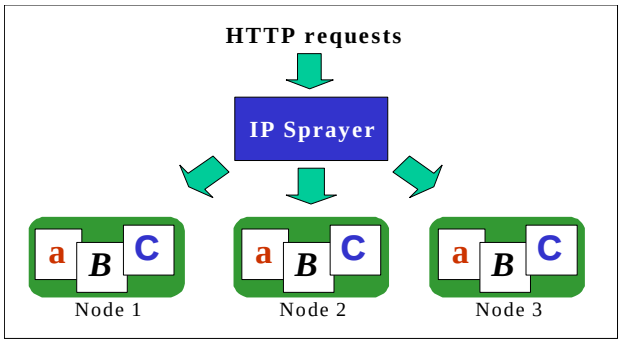

Figure 1(a). IP-Sprayer Architecture

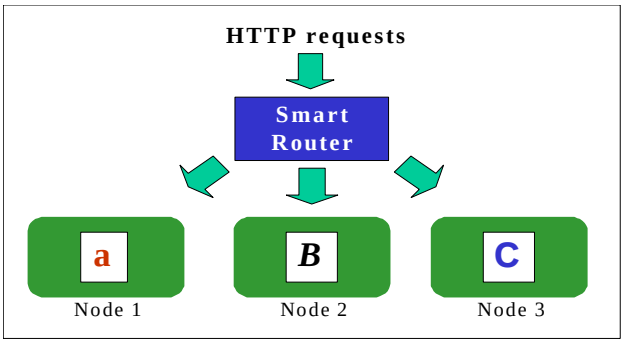

Figure 1(b). HACC Architecture

node. The Smart Router also uses an adaptive scheme to tune the load presented to each node in the cluster based on that node's capacity, so that each node is assigned a fair share of the load. For popular pages, say *Hot Site of the Day*, the Smart Router could direct requests for that page to multiple cluster nodes[1], preventing it from overloading any single node.

The remainder of this paper is organized as follows: Section 2 discusses related work. Section 3 describes the implementation of HACC on Windows NT 4.0. Section 4 presents experimental results. Section 5 discusses future research directions and Section 6 concludes.

2. Related Work

A number of commercial IP-Sprayer products are currently available. One of the better-known products is the Cisco LocalDirector [Cis96]. In addition to performance and scalability, Cisco's product aims to provide failure recovery by detecting long response times that might signal an overloaded or malfunctioning server.

Microsoft's Windows NT Load Balancing Service (WLBS) [WLBS] represents a distributed version of the IP-Sprayer. WLBS functions as a filter between the NIC driver and the TCP/IP protocol stack. It maps incoming requests to the cluster node based on the source IP address and port number and only passes packets that are destined for the local node to the upper network layer. The cluster nodes exchange messages periodically to maintain a coherent view of the mapping and the cluster status. Since the mapping algorithm is fully distributed, this approach removes the single point of failure of the IP-Sprayer. However, as with other IP-Sprayer systems, it does not take advantage of content locality of web requests.

IBM researchers [AS92, DKM96] proposed a variation of the IP-Sprayer approach, called the *TCP router* approach. The TCP router is just like a traditional IP-Sprayer except that the reply sent back by the cluster node bypasses the TCP router and goes directly to the client making the request. This approach requires modifications to the kernel code of each server node in the cluster.

Fox *et al*. [FGC97] provide an interesting overview of the design space for cluster systems, and describe TranSend and HotBot, two cluster-based Internet services implemented at UC Berkeley and Inktomi Corporation. There are a number of important differences between HACC and the Berkeley/Inktomi services. First, HACC is designed to operate with off-the-shelf web server software. TranSend and HotBot are substantially or completely custom web-server software. Second, load balancing in TranSend and HotBot appears to be statically configured, in contrast to the dynamic load balancing in HACC.

The LARD system, developed jointly by Rice and IBM [PAB98], incorporates similar ideas to HACC. In a LARD configuration, the front-end directs incoming requests to back-end cluster nodes based on the content of the requests, and it assumes the role of load balancing. Simulation results and measurements on a prototype implementation show substantial performance enhancement over conventional cluster approaches. However, their work appears to focus entirely on the static file case.

The basic idea of HACC also bears some resemblance to that of affinity-based scheduling schemes for shared-memory multiprocessor systems [TG89, VZ91, SL93], which try to schedule a task on a processor where relevant data already resides.

[1] We assume that the document store is read-only. Mutable document stores introduce a number of interesting issues. Although we believe these issues can be handled in our design, we have focused on the read-only case for this paper.

3. Implementation

The main challenge in realizing the potential of the HACC design is building the Smart Router, and within the Smart Router, designing the adaptive algorithms that direct request streams at the cluster nodes based on the locality properties and capacity of the node. Additionally, the Smart Router must be robust and efficient enough to handle a large number of cluster nodes without becoming the bottleneck in the system. Another challenge is creating a suitable request stream for evaluating HACC performance; therefore, after describing the Smart Router, we also include a brief discussion of DBench, the benchmark used to evaluate our system.

3.1. Smart Router Implementation

The Smart Router implementation is partitioned into two layers, the High Smart Router (HSR) and the Low Smart Router (LSR). As the names suggest, the LSR corresponds to the low-level, nuts-and-bolts kernel-resident part of the system[2], whereas the HSR implements the high-level, user-mode "brains" of the system. This partitioning encourages a separation of mechanism and policy, with mechanism implemented in the LSR and policy in the HSR.

The LSR

The LSR encapsulates the networking functionality required by our design. It is responsible for TCP/IP connection setup and termination, for forwarding requests to cluster nodes, and for forwarding result documents back to clients. Apart from these functional requirements, the main requirement of the LSR is performance — it is on the critical path of every request handled by the HACC cluster and will generally determine the degree of scalability within the cluster.

Our in-kernel LSR is implemented as a Windows NT 4.0 device driver that attaches to the top of the TCP transport driver. The upper edges of all NT transport drivers have an abstract interface known as the Transport Driver Interface (TDI). This layer of abstraction allows us to implement the LSR on top of the TCP/IP transport layer without any modification to the Windows NT networking subsystem. Installing the LSR is just like installing an ordinary device driver. This simplifies the design of the LSR since it does not need to handle any protocol-related issues. Note that this new layered driver is needed only on the Smart Router. Server nodes in the cluster run entirely off-the-shelf software and do not need any new or modified network drivers.

The LSR listens on the well-known web server port for a connection request. When a connection request is received, TCP passes a buffer to the LSR containing the HTTP request. The URL from the request is extracted and copied to the HSR[3]. The LSR enqueues all data from this incoming request and waits for the HSR to indicate which cluster node should handle the request. When the HSR identifies the node, the LSR establishes a connection with it and forwards the queued data (including the URL) over this connection. The LSR continues to ferry data between the client and the cluster node serving the request until either side closes the connection.

This design has some important consequences. As we are maintaining all open TCP connections in the Smart Router, a criticism is that the router immediately becomes the bottleneck, requiring as much networking resources as all the cluster nodes combined. However, we find that in our target WAS domain, managing network state is not the system bottleneck. Each dynamic request requires a significant amount of computation on the server, such that the networking system is not stressed handling incoming packets (cf. sections 4.4 and 4.5).

The HSR

The job of the HSR is to monitor the state of the document store, the nodes in the cluster, and properties of the documents passing through the LSR. It then must use this information to make decisions about how to distribute requests over HACC cluster nodes. To date we have implemented two decision distribution algorithms, one modeling a tree-based name space such as would be appropriate for static file service and one for the document store used by Lotus Domino.

To support the tree-based namespace, the HSR maintains a tree that models the structure of the document store. Leaves in the tree represent documents and nodes represent directories. As the HSR processes requests, it annotates the tree with information about the document store to be applied in load balancing. This

[2] An earlier version of our system used a user-level LSR, similar to proxy server implementations. However, the overhead of repeated crossings of the kernel/user boundary became a significant bottleneck.

[3] The Windows NT 4.0 version of the TCP/IP TDI layer does not support zero copy when data is passed up from the TDI layer to the LSR. We expect this feature to be supported in the future versions of NT, which should produce a significant improvement in LSR performance.

information could include node assignment, document sizes, request latency for a given document, and, in general, sufficient information to make an intelligent decision about which node in the cluster should handle the next document request. In our prototype implementation, load balancing is performed on a per cluster node base. Therefore, only node assignment information is recorded.

When a request for a particular file is received for the first time, the HSR adds nodes representing the file and any newly reached directories to its model of the document store, initializing the file's node with its server assignment. In our current HACC prototype, incoming new documents are assigned to the least loaded server node. After the first request for a document, subsequent requests will go to the same server, improving locality of reference.

However, the tree-structured name space only works for the case when the structure of the document store is hierarchical. Some WAS platforms, such as Lotus Domino, a web-server product from IBM Lotus, embed keys or request parameters into URLs, requiring further semantic analysis of the URL in order to model the structure of the document store. A Domino URL, for example, is composed of three fields: host name, Notes object, and action: *http://host_name/Notes_-object?action*. The host name is the name of the web site. The Notes object field identifies a Notes object within a database, typically a database view followed by a document that belongs to the view. The action field denotes the Lotus command to be activated on the Notes object. Typical actions are "OpenDocument", "OpenNavigator" and "OpenView". Actions can have parameters separated by the "&" character. The HSR extension for Domino incorporates the same tree-structured name space model for Notes objects and enhances it with the "action" information. The model only tracks down the hierarchy to the database view level (as opposed to individual document level). The HSR decides where to forward the request within the cluster based on the Notes object and the requested action.

3.2. Dynamic Load Balancing

Dynamic load balancing is implemented using Windows NT's Performance Data Helper (PDH) interface [PDH98]. The PDH interface allows one to collect a machine's performance statistics remotely, thus relieving us from the burden of implementing a monitoring agent on each cluster node. The only monitoring agent needed is the one on the Smart Router. Another advantage of PDH is that it allows web application developers to add application-specific performance objects and counters that can be retrieved the same way as system performance counters using a set of well-defined APIs.

When the Smart Router starts, it spawns a performance-monitoring thread that collects performance data from each cluster node at a fixed interval. The performance data is used for load balancing in two ways by the HSR. First, a least loaded node is identified and new (unseen) requests are assigned to the least loaded node. Second, when a node becomes overloaded (i.e., its load exceeds that of the least loaded node by a certain amount), the HSR tries to offload a portion of the documents for which the overloaded node is responsible to the least loaded node.

The monitoring thread collects each node's load statistics, such as CPU utilization, disk activity, paging activity, number of outstanding web requests in the queue, etc., and combines these performance metrics into a single load indicator using a weighted average. In our prototype implementation, we use two performance metrics: CPU utilization and bytes transferred to/from disk per second. The load is calculated using the following formula:

$$load = weight_{CPU} \times load_{CPU} + weight_{disk} \times load_{disk}.$$

For the static file workloads, $weight_{CPU}$ is set to zero and $weight_{disk}$ to one, since disk activity is the dominant factor of a server's load. For the Domino workload, both $weight_{CPU}$ and $weight_{disk}$ are set to ½, since a server node should be balanced between the complexity of the tasks it handles and the working set size[4]. In spite of its simplicity, this formula works well for our test cases.

It is an interesting research question to determine how to combine a set of performance statistics into a single metric that reflects the cluster node's real load in the context of the particular web application. This is an area of further research and beyond the scope of this paper.

3.3. DBench

DBench [CDW97] is designed specifically for evaluating WAS performance. The D in DBench is for *dynamic*, emphasizing the key difference between DBench and existing benchmarks — its ability to test WAS performance. DBench is based on a simulator that models the activity of multiple individual users accessing a Web site. DBench supports requests for WAS by replaying the sequence of document requests generated by an actual user and by dynamically

[4] Statistics of disk activity are normalized to the same scale as that of CPU load.

Workload	Description	Server software	Web site size (MB)	Average file size (KB)	Number of files
Static baseline	Simulated clients request static files. Files and scripts are stochastically generated.	Microsoft IIS 4.0	100	280	367
Static FAS	Simulated clients request static files. Based on www.fas.harvard.edu.	Microsoft IIS 4.0	80 (subset)	14	5715
Domino	Simulated clients make requests to Lotus Domino Server. Based on lotus.domino.com.	Lotus Domino 4.6	129 (database)	N/A	N/A
ASP	Simulated clients make ASP as well as static file requests to the Server. Based on www.thecrimson.com.	Microsoft IIS 4.0	54 (static) 7 (database)	12	4495 (including asp files)

Table 1. Workload Descriptions

controlling the number of simulated web users. Internally, users are modeled using a collection of user profiles. For example, the Domino-based DBench test used about 500 profiles, each of which describes the pattern of references that occurred for a real Domino user. The profiles are created by analyzing the access log for a representative web site and extracting the sequence of requests made by each individual that accessed the system. The profiles include the URLs requested by each user as well as timing information that specifies the user pause time between each client request. Each concurrent user in DBench is modeled by replaying the sequence of requests as recorded in a user profile.

DBench reports its results using two primary metrics: *Concurrent Users*, which is to help site managers make capacity planning decisions and *Aggregate Throughput*. DBench also reports statistics such as A*verage Request Latency* and *Number of Requests Completed* for each sub-epoch (10 seconds). We use a subset of these statistics to compare performance of HACC with other approaches. DBench measures server performance by gradually increasing the number of concurrent simulated users until one of three conditions occurs:

- the server begins to generate request failures,
- the average time to establish a connection with the server exceeds 3 seconds (1 second for the static file case), or
- the maximum time to establish a connection with the server exceeds 5 seconds (2 seconds for the static file case).

DBench terminates when the number of concurrent simulated users is stable for 100 seconds.

4. Experimental Results

4.1. Methodology

In this section we present experimental results to compare the performance of a HACC cluster to that of a cluster implemented with an IP-Sprayer. For the HACC cluster, the Smart Router distributes requests using the scheme described in Section 3. For the IP-Sprayer, we replace the HSR portion of the Smart Router with a simple round robin request distribution scheme. Readers might notice that this is not a true implementation of an IP-Sprayer, since IP-Sprayers distribute packets at the IP layer. Consequently our IP-Sprayer implementation will have inferior performance compared to commercial implementations. However, as we will describe in section 4.2, for our target domain of WAS, the overheads incurred by the Smart Router module are minimal compared to the request latency and do not affect the significance of the improvements obtained with the Smart Router.

We used four DBench workloads for this evaluation, a stochastically generated static file test, a static file test based on Harvard's *FAS* (Faculty of Arts and Sciences) web site, a WAS test based on Lotus Domino, and another WAS test based on Microsoft ASP [Wei99]. Details are given in Table 1. For the static-FAS workload, due to the excessive number of user profiles, we randomly select a subset of user profiles and use them to drive DBench. Our cluster consists of four Hewlett Packard Netserver E40 uniprocessors, each with a 200MHz PentiumPro processor with a 256KB 2nd level cache. Three of them are used as cluster nodes and one is used as the Smart Router. To test the capability of HACC to work with uneven cluster node capacities, we intentionally configured the cluster nodes with different memory sizes. Two of the cluster nodes and the Smart Router have 64MB of main memory, and the third cluster node has 32MB. All systems run Windows NT 4.0 updated with Service Pack 4. For the cluster interconnect, we use 100Mb/s switched Ethernet, with a Hewlett Packard AdvanceStack Switch 800T. Unless otherwise specified, all results reported in this paper are the average of three runs preceded by a warm-up run.

We are aware that Domino is not the most popular web server product on the Windows NT platform and our workloads are relatively small. Obtaining large WAS workloads is extremely difficult, since performance evaluation for WAS requires the original contents of the web site (as opposed to the static file case where the document store can usually be regenerated from the web log), and the large web sites in which we are interested are unwilling to give us the contents due to privacy issues. However, as the workload does not fit in the capacity of a single cluster node, we believe that the techniques demonstrated in this paper are readily applicable to larger workloads.

Section 4.2 presents measured overhead of the Smart Router and analyzes the potential overhead of an IP-Sprayer implementation. Section 4.3, 4.4 and 4.5 compare the performance of HACC and an IP-Sprayer using the two static files workloads, the Domino workload and the ASP workload, respectively.

4.2. Overhead of the Smart Router

To quantify the overhead of the Smart Router, we measured the latency of static documents for documents ranging from 512 bytes to 2MB in size under two situations: (1) send the request directly to the web server (Direct); (2) send the request via the Smart Router (Via Smart Router). To isolate the cost of the decision process in the Smart Router from the cost of the extra network hop, we also measured the latency of using a Windows-NT based IP router, omitting the decision that must be made by the Smart Router. Figure 2 depicts the three different scenarios[5].

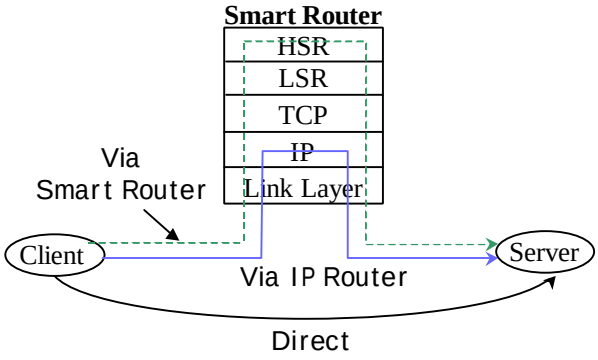

Figure 2. Overhead Measurement under Three Different Situations

[5] To isolate monitoring overhead from that of the request-handling overhead of the Smart Router, overhead was measured with load monitoring disabled. Since load monitoring occurs at infrequent intervals (once per 200 ms), we do not believe that it would affect our results.

The overhead is defined to be the difference between the "Direct" case latency and the latencies of the other two cases. Figure 3 shows the results. The difference of latencies between the "Smart Router" case and the "IP Router" case is the actual cost incurred by the additional TCP connection handling and the decision process in the Smart Router.

To better understand the sources of latency, we split the request latency into two types: fixed overhead and per-byte overhead. Per-byte (or per-packet) overhead includes the cost of going up/down the IP stack. For the Smart Router, there is also an additional per-byte overhead for copying the data between the two TCP/IP connections. This component will be smaller in a zero-copy implementation of the Smart Router. The fixed overhead includes the overhead of the decision process of where to route the packets, plus, for the Smart Router, the time for TCP connection setup and tear-down. We can express the overhead using the following equation:

$$overhead = overhead_{per-byte} \times number\ of\ bytes + overhead_{fixed}$$

or simply $y=ax+b$. Figure 3 shows the measured request handling overhead for our test cases. It also displays the linear equation fit of the data points that gives the fixed and per-byte overhead. The fixed delay induced by the Smart Router is about 1.4 milliseconds. We consider this overhead tolerable in our WAS target domain. More than half of the per-byte overhead of the Smart Router is due to the IP stack. The per-kilobyte overhead of the IP Router is 0.0132 ms/KB, or about 55% of the per-kilobyte overhead of the Smart Router (0.0242 ms/KB).

In summary, for small files, the fixed overhead of the Smart Router dominates. For large files, the Smart Router is about 10-15% slower than an IP Router. It can be argued that a commercial IP Router, such as Cisco's, would probably perform much better than the NT router, however a similar argument can be made about a commercial implementation of the Smart Router. There are many Smart Router optimizations that currently remain unexplored, due to time constraints.

Notice that this test demonstrates the worst case overhead because the time required to service a static file request at the server is minimal (assuming the file is in cache). For CPU-intensive workloads, such as those supported by Domino, serving the request takes more time at the server. At the same time, the result data sent back to the client is relatively small. Thus the overhead for the additional copying is not significant. These two facts together dramatically reduce the actual overhead

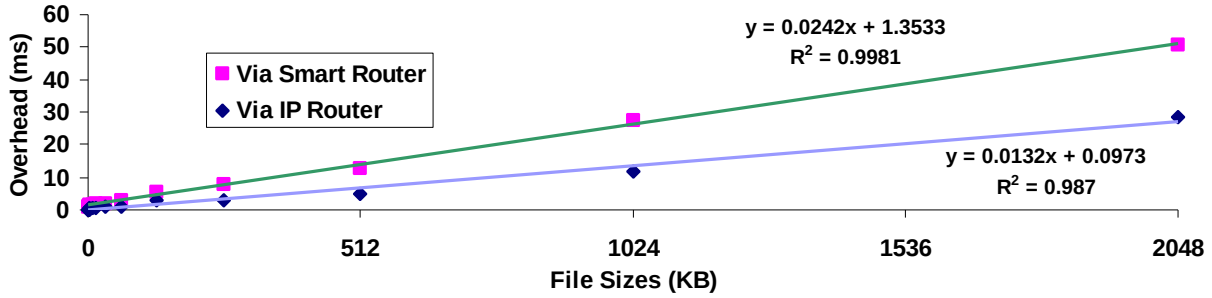

Figure 3. Overhead of the Smart Router for the Static File Case.
As the X-axis is indexed in kilobytes, *a* represents per-kilobyte cost.

as a percentage of the total request service time. For example, in our test benchmark, the average file size for Domino is less than 8KB and the average service time is about 200ms. This results in an overhead of less than 1.6/200=0.8% for the Smart Router. In a real Internet WAN environment, web request latencies are typically dominated by network latencies. This would further reduce the percentage overhead of the Smart Router. Therefore, we conclude that the overhead of the Smart Router is insignificant for CPU intensive workloads or network-latency dominated workloads.

4.3. Static File Results

We use two static file workloads for our HACC evaluation (see Table 1). The first serves as a proof of the basic concept that if the working set of a document store doesn't fit in a single machine's main memory, and the requests can be distributed to preserve locality, then HACC should provide performances superior to that of a conventional IP-Sprayer. For this baseline test case, we use a stochastically generated workload with uniform request distribution to drive DBench. The distribution of the file sizes follows the long tail distribution, i.e., about 40% of the files have sizes between 5KB and 8KB and the rest of the files are scattered between 8KB and 1.4MB. A second static file workload, derived from real logs of the Harvard's FAS web server, is to evaluate how well HACC performs in practice. Figures 4 and 5 give the results for these two test cases.

Figures 4 (a) and (b) show the throughput and the number of concurrent users of the IP-Sprayer and the HACC Cluster for our baseline test. Figure 4 (c) shows the average request latency. The HACC cluster gives consistently better performance than the IP-Sprayer cluster. Each node in the HACC cluster is only responsible for a portion of the file set, as opposed to the entire file set for the IP-Sprayer organization. This leads to a 121% improvement in throughput for HACC. The number of concurrent users also increases by 113%, indicating that the HACC cluster is able to support about 113% more users than the IP-Sprayer architecture. Request latencies also decrease dramatically because of the better locality achieved in the HACC cluster. In the case of HACC, CPU utilization for the Smart Router is about 80%, indicating that the Smart Router is close to saturation as the throughput approaches network limit.

As shown in Figure 5, similar improvements, though smaller in magnitude, are attained for the FAS test case. HACC achieves 40% higher throughput and supports 36% more users. It reduces request latency by 37%. In this case the CPU utilization at the Smart Router is only about 20%. The reason for the less significant improvement compared to the baseline case is that the request distribution for the FAS document store is skewed (as opposed to the uniform distribution in the

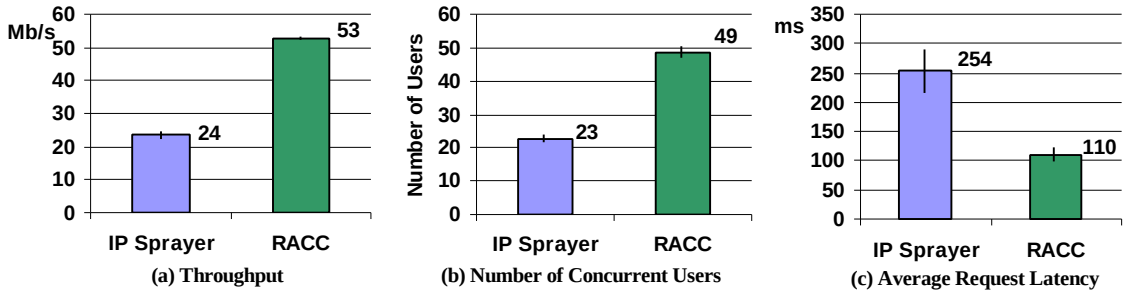

Figure 4. HACC vs. IP-Sprayer Performance for the Baseline Static File Test.
The disk activity for HACC is about 200KB - 300KB per second, and about 700KB - 1MB per second for the IP-Sprayer. The CPU utilization for HACC is about 20% vs. about 10% for IP-Sprayer.

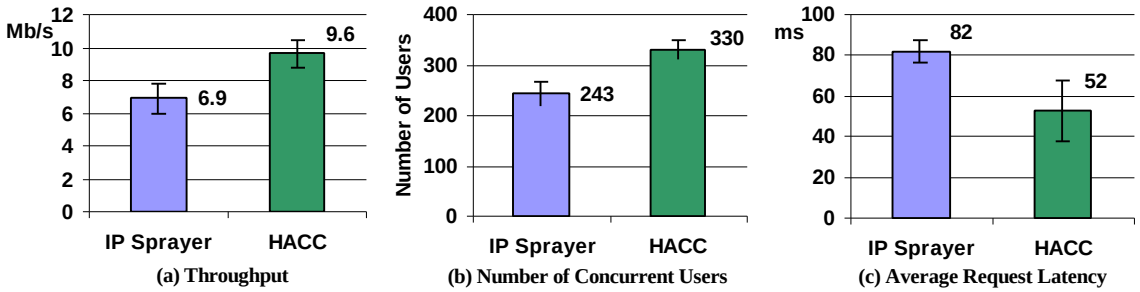

Figure 5. HACC vs. IP-Sprayer Performance for the FAS Static File Test.
The disk activity for HACC is about 200KB - 300KB per second, and about 200KB per second for the IP-Sprayer. The CPU utilization for HACC is about 20% vs. about 10% - 15% for IP-Sprayer.

baseline case). As a result frequently requested files stay in the cache most of the time, even in the IP-Sprayer case, limiting the performance gains attainable by the Smart Router.

Our static file test cases demonstrate that the HACC design can provide substantially improved performance over an IP-Sprayer-based cluster for static file servers.

4.4. Domino Results

The Domino workload is based on the web log and actual document store from domino.lotus.com, a corporate web site for Lotus Notes. While the underlying database is only about 130MB, there is sufficient activity in the web log to keep the server busy. The average request size is about 6KB. Although the log doesn't contain request latency information, our local experiments show that a typical OpenDocument request takes about 200ms and an OpenView request takes between 0.5 to 2.0 seconds when the system is not loaded. The average time gap between subsequent requests is 29.8 seconds, of which a substantial fraction is user "think" time and WAN network latency. For this experiment, the IP-Sprayer is enhanced with a load balancing scheme that always forwards a request to the least loaded node.

Figure 6 shows the performance for the Domino workload. In Figure 6(a) we used *Peak Requests/Minute*, the number of requests completed per minute during the peak period when the number of concurrent users is stable, instead of the *Throughput* metric, since we believe that for the dynamic case, the number of requests a server can handle is a more relevant measure of performance. The HACC cluster delivers over 52% more requests per minute than the IP-Sprayer and supports 46% more concurrent users for this test. Additionally, the average request latency is much smaller for HACC. These results demonstrate that locality enhancement in HACC, even for a mainly CPU-intensive workload, improves performance substantially when the total working set size doesn't fit in a single node's main memory. During all the experiments, the CPU utilization on the Smart Router is only about 1%. Memory usage is also minimal. The cluster nodes show a CPU utilization of 20% - 50% and disk transfers of 400KB - 900KB during the peak period. These data show that for our Domino workload, backend servers are the bottleneck, not the front-end network.

Readers are advised against comparing the absolute numbers of the Domino results with those of the static file cases directly, as these workloads have drastically different characteristics. For example, in the baseline static file case, the scripts used to drive DBench are stochastically generated, and requests from the same

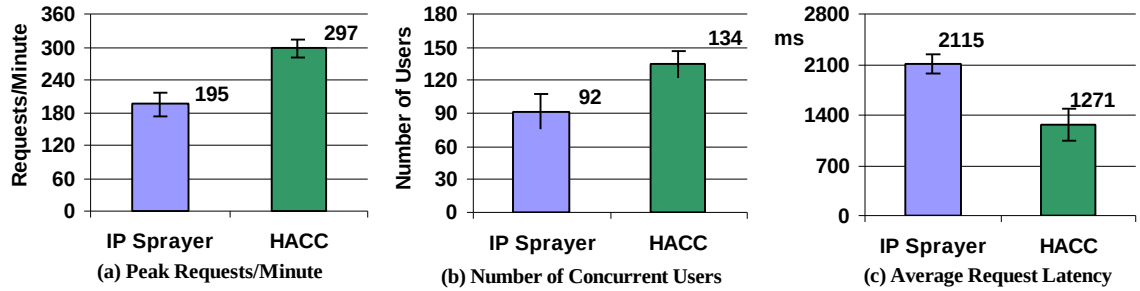

Figure 6. HACC vs. IP-Sprayer Performance for the Domino Test.
The cluster nodes show a CPU utilization of 20% - 50% and disk transfers of 400KB - 900KB during the peak period for both cases.

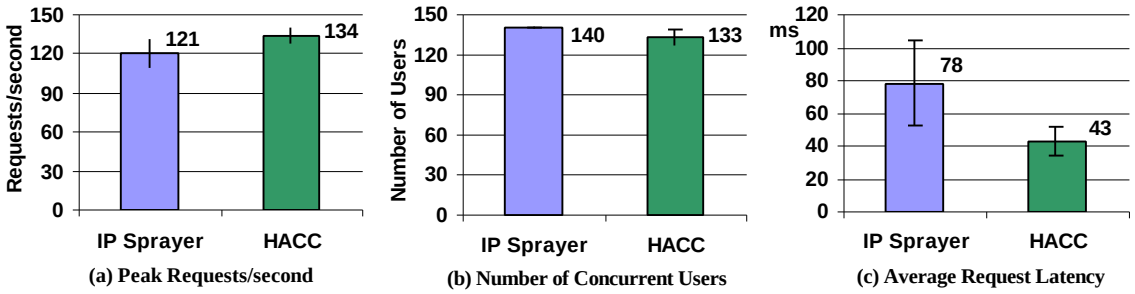

Figure 7. HACC vs. IP-Sprayer Performance for the ASP Test.
For the IP-Sprayer, the CPU utilization of the three cluster nodes is around 50-60%. For the Smart Router, the CPU utilization is about 20% for the node that handles static file requests, and about 40-50% for the other two nodes.

simulated user are 2 seconds apart. In the Domino case, the scripts model actual user behavior and incorporate user "think" time and network latency. The gap between successive requests from the same simulated user is thus significantly larger for the Domino case, resulting in a larger number of concurrent users, even though it takes a much longer time to process a Domino request than a static file request.

4.5. ASP Results

The ASP workload is based on the web log and actual document store from www.thecrimson.com, the online version of *The Crimson* newspaper (see Table 1). Analysis of the web logs reveals that most requests are for static files, and only about 5% of requests are ASP requests. Although only a tiny fraction of the logs are ASP requests, our measurements show that even this low volume of ASP traffic overloads the CPU, long before the network becomes a bottleneck. This confirms our assertion that it is common for WAS environments to be "backend limited."

Since the active working set fits in a cluster node with 64MB of memory, the Smart Router's locality-based request distribution scheme will not offer much advantage over an IP-Sprayer approach. However, we demonstrate here another way of utilizing the Smart Router's content-based routing. For a heavily loaded web server, a CPU-intensive dynamic web request (such as an ASP request) will delay many static file requests, resulting in longer response times for static files. Thus, if we separate ASP requests from static file requests and send them to different cluster servers, we should be able to reduce response time significantly.

We implemented this simple content-based routing scheme and evaluated its performance against an IP-Sprayer approach. The Smart Router forwards static file requests to one of the cluster nodes with 64MB of memory and ASP requests to the other cluster nodes. The results are shown in Figure 7. As expected, the Smart Router approach reduces the average request time by 45%. The IP-Sprayer is able to support slightly more concurrent users because of the perfect load balancing between the three cluster nodes.

This simple experiment demonstrates the flexibility of the Smart Router's content-based routing scheme. Even for a small web site whose working set fits in main memory, the Smart Router can help improve performance.

5. Discussion

There are a number of possible directions for extending and completing the functionality of the system. Mutable document stores present an interesting challenge for HACC. One solution is to offload the consistency responsibilities to backend database servers by using a "three-tier" architecture, with the web servers as a front end for a standard relational database. Use of a HACC cluster as the middle tier of such a three-tier system makes scalable computing available for content access.

A more fundamental issue is scalability of the Smart Router. We conducted a simple scalability test which suggested that our prototype Smart Router can handle between 400 to 500 requests of size 8KB per second. Though not an impressive number for static file web servers, for the Domino Lotus case, this means that it can support around 100 cluster nodes. Therefore, we believe that for web sites that contain a non-trivial portion of dynamically generated documents, the Smart Router will be able to scale.

Furthermore, there are many ways to improve our un-optimized prototype Smart Router implementation. One approach is to implement a TCP connection handoff protocol [PAB98] such that after the Smart Router determines to which node to distribute the request, the TCP connection is handed off to that particular node, which then sends the reply directly to the client, bypassing the Smart Router.

The "Keep Alive" feature of HTTP poses some potential problems. If "Keep Alive" is enabled, the browser is allowed to reuse the TCP connection for subsequent requests, which are not intercepted by the Smart Router. This would interfere with the Smart Router's load balancing decision. However, major web server vendors recommend that the use of "Keep Alive" be limited to prevent a client from hogging the server resources [Apache]. Therefore we expect that it will not affect the effectiveness of Smart Router in a significant way.

6. Summary

In this paper we have presented the HACC architecture and experiments that explore how locality enhancement in HACC improves web-server performance. Experiments with an actual implementation of the HACC cluster on Windows NT show that HACC is able to support more than twice the number of concurrent users in the baseline static file test, 36% more in the FAS static file test, and 46% more in the Domino test than alternative schemes for creating cluster-based web servers. We conclude that the HACC design can be effective for both the static file case and the dynamic case, but in practice, expect it to be most beneficial in the dynamic case where the additional overhead of the Smart Router is tolerable.

7. Acknowledgements

We thank our shepherd Susan Owicki and our anonymous reviewers for their valuable comments. We are grateful to Mark Day at Lotus who answered many of our questions about Lotus Domino. We especially thank Skylar Byrd and Dawn Lee from Harvard Crimson for their generous help in providing the logs and content databases. This research is sponsored in part by Microsoft.

References

[Apache] "Apache Keep-Alive support." This document can be obtained from http://www.apache.org/docs-1.2/keepalive.html.

[AS92] C. Attanasio, and S. Smith, "A Virtual Multiprocessor Implemented by an Encapsulated Cluster of Loosely Coupled Computers." *IBM Research Report RC18442*, 1992.

[CDW97] J. B. Chen, A. Wharton, and M. Day, "Benchmarking the Next Generation of Internet Servers." This document can be obtained from http://www.notes.net/today.nsf by a full text search on "DBench" on the archives of Iris Today.

[Che97] Check Point Software Technologies Inc., "ConnectControl: Advanced Server Load Balancing." Software product. Additional information on this software is available from, http://www.checkpoint.com/products/-floodgate-1/cc.html.

[Cis96] Cisco Systems Inc., "How to Cost-Effectively Scale Web Servers." Packet Magazine, Third Quarter 1996. See http://www.cisco.com/warp/public/784/5.html.

[DKM96] D. Dias, W. Kish, R. Mukherjee, and R. Tewari, "A Scalable and Highly Available Server." In *COMPCON 1996*, IEEE-CS Press, Santa Clara, CA, February 1996, pp. 85-92.

[FGC97] A. Fox, S. Gribble, Y. Chawathe, E. Brewer, and P. Gauthier, "Cluster-Based Scalable Network Services." In *Proceedings of the 16th Symposium on Operating Systems Principles*, ACM, Saint-Malo, France, October 1997, pp. 78-91.

[KMR95] T. Kwan, R. McGrath, and D. Reed, "NCSA's World Wide Web Server: Design and Performance." In *IEEE Computer*, 28(11):68-74, November 1995.

[PAB98] 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 the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, San Jose, CA, October 1998.

[PDH98] "Performance Data Helper", Microsoft Developer Network Platform SDK, Microsoft, July 1998.

[SL93] M. Squillante and E. Lazowska, "Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling." In *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131-143, February 1993.

[Ten99] Jessie Tenenbaum, Microsoft, personal communication.

[TG89] A. Tucker and A. Gupta, "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors." In *Proceedings of the 12th Symposium on Operating Systems Principles*, ACM, Litchfield Park, AR, December 1989, pp. 159-166.

[VZ91] R. Vaswani and J. Zahorjan, "The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors." In *Proceedings of the 13th Symposium on Operating Systems Principles*, ACM, Pacific Grove, CA, October 1991, pp. 26-40.

[WLBS] Microsoft Windows NT Load Balancing Service. This document can be obtained from http://www.microsoft.com/ntserver/NTServerEnterprise/exec/feature/WLBS/.

[Wei99] A. K. Weissinger, *ASP In a Nutshell*. O'Reilly, Sebastopol, CA, 1999.