Flash: An efficient and portable Web server *

Vivek S. Pai‡ Peter Druschel† Willy Zwaenepoel†
‡ *Department of Electrical and Computer Engineering*
† *Department of Computer Science*
Rice University

Abstract

This paper presents the design of a new Web server architecture called the asymmetric multi-process event-driven (AMPED) architecture, and evaluates the performance of an implementation of this architecture, the Flash Web server. The Flash Web server combines the high performance of single-process event-driven servers on cached workloads with the performance of multi-process and multi-threaded servers on disk-bound workloads. Furthermore, the Flash Web server is easily portable since it achieves these results using facilities available in all modern operating systems.

The performance of different Web server architectures is evaluated in the context of a single implementation in order to quantify the impact of a server's concurrency architecture on its performance. Furthermore, the performance of Flash is compared with two widely-used Web servers, Apache and Zeus. Results indicate that Flash can match or exceed the performance of existing Web servers by up to 50% across a wide range of real workloads. We also present results that show the contribution of various optimizations embedded in Flash.

1 Introduction

The performance of Web servers plays a key role in satisfying the needs of a large and growing community of Web users. Portable high-performance Web servers reduce the hardware cost of meeting a given service demand and provide the flexibility to change hardware platforms and operating systems based on cost, availability, or performance considerations.

Web servers rely on caching of frequently-requested Web content in main memory to achieve throughput rates of thousands of requests per second, despite the long latency of disk operations. Since the data set size of Web workloads typically exceed the capacity of a server's main memory, a high-performance Web server must be structured such that it can overlap the serving of requests for cached content with concurrent disk operations that fetch requested content not currently cached in main memory.

Web servers take different approaches to achieving this concurrency. Servers using a *single-process event-driven (SPED)* architecture can provide excellent performance for cached workloads, where most requested content can be kept in main memory. The Zeus server [32] and the original Harvest/Squid proxy caches employ the SPED architecture[1].

On workloads that exceed that capacity of the server cache, servers with *multi-process (MP)* or *multi-threaded (MT)* architectures usually perform best. Apache, a widely-used Web server, uses the MP architecture on UNIX operating systems and the MT architecture on the Microsoft Windows NT operating system.

This paper presents a new portable Web server architecture, called asymmetric multi-process event-driven (AMPED), and describes an implementation of this architecture, the Flash Web server. Flash nearly matches the performance of SPED servers on cached workloads while simultaneously matching or exceeding the performance of MP and MT servers on disk-intensive workloads. Moreover, Flash uses only standard APIs and is therefore easily portable.

Flash's AMPED architecture behaves like a single-process event-driven architecture when requested documents are cached and behaves similar to a multi-process or multi-threaded architecture when requests must be satisfied from disk. We qualitatively and quantitatively compare the AMPED architecture to the SPED, MP, and MT approaches in the context of a single server implementation. Finally, we experimentally compare the performance of Flash to that of Apache and Zeus on real workloads obtained from server logs, and on two operating systems.

The rest of this paper is structured as follows: Section 2 explains the basic processing steps required of all Web servers and provides the background for the following discussion. In Section 3, we discuss the asynchronous multi-process event-driven (AMPED), the

*To appear in Proc. of the 1999 Annual Usenix Technical Conference, Monterey, CA, June 1999.

[1] Zeus can be configured to use multiple SPED processes, particularly when running on multiprocessor systems

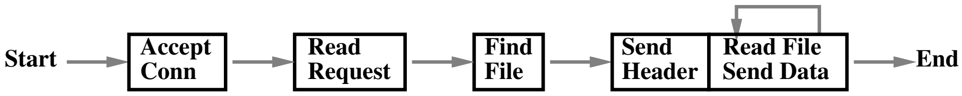

Figure 1: Simplified Request Processing Steps

single-process event-driven (SPED), the multi-process (MP), and the multi-threaded (MT) architectures. We then discuss the expected architecture-based performance characteristics in Section 4 before discussing the implementation of the Flash Web server in Section 5. Using real and synthetic workloads, we evaluate the performance of all four server architectures and the Apache and Zeus servers in Section 6.

2 Background

In this section, we briefly describe the basic processing steps performed by an HTTP (Web) server. HTTP clients use the TCP transport protocol to contact Web servers and request content. The client opens a TCP connection to the server, and transmits a HTTP request header that specifies the requested content.

Static content is stored on the server in the form of disk files. *Dynamic content* is generated upon request by auxiliary application programs running on the server. Once the server has obtained the requested content, it transmits a HTTP response header followed by the requested data, if applicable, on the client's TCP connection.

For clarity, the following discussion focuses on serving HTTP/1.0 requests for static content on a UNIX-like operating system. However, all of the Web server architectures discussed in this paper are fully capable of handling dynamically-generated content. Likewise, the basic steps described below are similar for HTTP/1.1 requests, and for other operating systems, like Windows NT.

The basic sequential steps for serving a request for static content are illustrated in Figure 1, and consist of the following:

Accept client connection - accept an incoming connection from a client by performing an `accept` operation on the server's `listen` socket. This creates a new socket associated with the client connection.

Read request - read the HTTP request header from the client connection's socket and parse the header for the requested URL and options.

Find file - check the server filesystem to see if the requested content file exists and the client has appropriate permissions. The file's size and last modification time are obtained for inclusion in the response header.

Send response header - transmit the HTTP response header on the client connection's socket.

Read file - read the file data (or part of it, for larger files) from the filesystem.

Send data - transmit the requested content (or part of it) on the client connection's socket. For larger files, the "Read file" and "Send data" steps are repeated until all of the requested content is transmitted.

All of these steps involve operations that can potentially block. Operations that read data or accept connections from a socket may block if the expected data has not yet arrived from the client. Operations that write to a socket may block if the TCP send buffers are full due to limited network capacity. Operations that test a file's validity (using `stat()`) or open the file (using `open()`) can block until any necessary disk accesses complete. Likewise, reading a file (using `read()`) or accessing data from a memory-mapped file region can block while data is read from disk.

Therefore, a high-performance Web server must interleave the sequential steps associated with the serving of multiple requests in order to overlap CPU processing with disk accesses and network communication. The server's *architecture* determines what strategy is used to achieve this interleaving. Different server architectures are described in Section 3.

In addition to its architecture, the performance of a Web server implementation is also influenced by various optimizations, such as caching. In Section 5, we discuss specific optimizations used in the Flash Web server.

3 Server Architectures

In this section, we describe our proposed asymmetric multi-process event-driven (AMPED) architecture, as well as the existing single-process event-driven (SPED), multi-process (MP), and multi-threaded (MT) architectures.

3.1 Multi-process

In the multi-process (MP) architecture, a process is assigned to execute the basic steps associated with serving a client request sequentially. The process performs all the steps related to one HTTP request before it accepts a new request. Since multiple processes are employed (typically 20-200), many HTTP requests can be served concurrently. Overlapping of disk activity, CPU processing and network connectivity occurs naturally, because the operating system switches to a runnable process whenever the currently active process blocks.

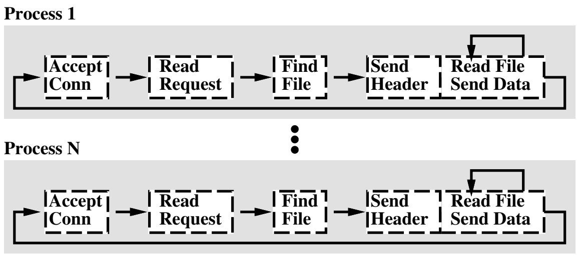

Figure 2: Multi-Process - In the MP model, each server process handles one request at a time. Processes execute the processing stages sequentially.

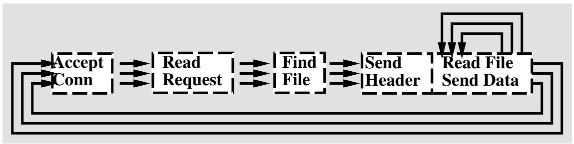

Figure 3: Multi-Threaded - The MT model uses a single address space with multiple concurrent threads of execution. Each thread handles a request.

Since each process has its own private address space, no synchronization is necessary to handle the processing of different HTTP requests[2]. However, it may be more difficult to perform optimizations in this architecture that rely on global information, such as a shared cache of valid URLs. Figure 2 illustrates the MP architecture.

3.2 Multi-threaded

Multi-threaded (MT) servers, depicted in Figure 3, employ multiple independent threads of control operating within a single shared address space. Each thread performs all the steps associated with one HTTP request before accepting a new request, similar to the MP model's use of a process.

The primary difference between the MP and the MT architecture, however, is that all threads can share global variables. The use of a single shared address space lends itself easily to optimizations that rely on shared state. However, the threads must use some form of synchronization to control access to the shared data.

The MT model requires that the operating system provides support for kernel threads. That is, when one thread blocks on an I/O operation, other runnable threads within the same address space must remain eligible for execution. Some operating systems (e.g., FreeBSD 2.2.6) provide only user-level thread libraries without kernel support. Such systems cannot effectively support MT servers.

[2]Synchronization is necessary inside the OS to accept incoming connections, since the accept queue is shared

3.3 Single-process event-driven

The single-process event-driven (SPED) architecture uses a single event-driven server process to perform concurrent processing of multiple HTTP requests. The server uses non-blocking systems calls to perform asynchronous I/O operations. An operation like the BSD UNIX `select` or the System V `poll` is used to check for I/O operations that have completed. Figure 4 depicts the SPED architecture.

A SPED server can be thought of as a state machine that performs one basic step associated with the serving of an HTTP request at a time, thus interleaving the processing steps associated with many HTTP requests. In each iteration, the server performs a `select` to check for completed I/O events (new connection arrivals, completed file operations, client sockets that have received data or have space in their send buffers.) When an I/O event is ready, it completes the corresponding basic step and initiates the next step associated with the HTTP request, if appropriate.

In principle, a SPED server is able to overlap the CPU, disk and network operations associated with the serving of many HTTP requests, in the context of a single process and a single thread of control. As a result, the overheads of context switching and thread synchronization in the MP and MT architectures are avoided. However, a problem associated with SPED servers is that many current operating systems do not provide suitable support for asynchronous disk operations.

In these operating systems, non-blocking `read` and `write` operations work as expected on network sockets and pipes, but may actually block when used on disk files. As a result, supposedly non-blocking `read` operations on files may still block the caller while disk I/O is in progress. Both operating systems used in our experiments exhibit this behavior (FreeBSD 2.2.6 and Solaris 2.6). To the best of our knowledge, the same is true for most versions of UNIX.

Many UNIX systems provide alternate APIs that implement true asynchronous disk I/O, but these APIs are generally not integrated with the `select` operation. This makes it difficult or impossible to simultaneously check for completion of network and disk I/O events in an efficient manner. Moreover, operations such as `open` and `stat` on file descriptors may still be blocking.

For these reasons, existing SPED servers do not use these special asynchronous disk interfaces. As a result, file `read` operations that do not hit in the file cache may cause the main server thread to block, causing some loss in concurrency and performance.

3.4 Asymmetric Multi-Process Event-Driven

The Asymmetric Multi-Process Event-Driven (AMPED) architecture, illustrated in Figure 5, combines

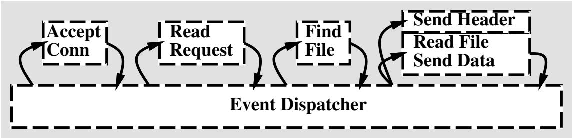

Figure 4: Single Process Event Driven - The SPED model uses a single process to perform all client processing and disk activity in an event-driven manner.

the event-driven approach of the SPED architecture with multiple *helper* processes (or threads) that handle blocking disk I/O operations. By default, the main event-driven process handles all processing steps associated with HTTP requests. When a disk operation is necessary (e.g., because a file is requested that is not likely to be in the main memory file cache), the main server process instructs a *helper* via an inter-process communication (IPC) channel (e.g., a pipe) to perform the potentially blocking operation. Once the operation completes, the helper returns a notification via IPC; the main server process learns of this event like any other I/O completion event via `select`.

The AMPED architecture strives to preserve the efficiency of the SPED architecture on operations other than disk reads, but avoids the performance problems suffered by SPED due to inappropriate support for asynchronous disk reads in many operating systems. AMPED achieves this using only support that is widely available in modern operating systems.

In a UNIX system, AMPED uses the standard non-blocking `read`, `write`, and `accept` system calls on sockets and pipes, and the `select` system call to test for I/O completion. The `mmap` operation is used to access data from the filesystem and the `mincore` operation is used to check if a file is in main memory.

Note that the helpers can be implemented either as kernel threads within the main server process or as separate processes. Even when helpers are implemented as separate processes, the use of `mmap` allows the helpers to initiate the reading of a file from disk without introducing additional data copying. In this case, both the main server process and the helper `mmap` a requested file. The helper touches all the pages in its memory mapping. Once finished, it notifies the main server process that it is now safe to transmit the file without the risk of blocking.

4 Design comparison

In this section, we present a qualitative comparison of the performance characteristics and possible optimizations in the various Web server architectures presented in the previous section.

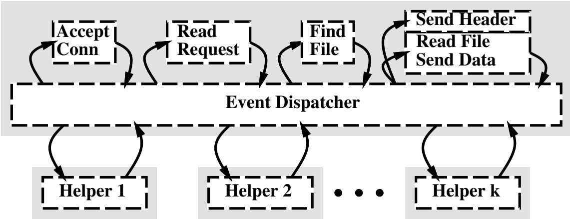

Figure 5: Asymmetric Multi-Process Event Driven - The AMPED model uses a single process for event-driven request processing, but has other helper processes to handle some disk operations.

4.1 Performance characteristics

Disk operations - The cost of handling disk activity varies between the architectures based on what, if any, circumstances cause all request processing to stop while a disk operation is in progress. In the MP and MT models, only the process or thread that causes the disk activity is blocked. In AMPED, the helper processes are used to perform the blocking disk actions, so while they are blocked, the server process is still available to handle other requests. The extra cost in the AMPED model is due to the inter-process communication between the server and the helpers. In SPED, one process handles all client interaction as well as disk activity, so all user-level processing stops whenever any request requires disk activity.

Memory effects - The server's memory consumption affects the space available for the filesystem cache. The SPED architecture has small memory requirements, since it has only one process and one stack. When compared to SPED, the MT model incurs some additional memory consumption and kernel resources, proportional to the number of threads employed (i.e., the maximal number of concurrently served HTTP requests). AMPED's helper processes cause additional overhead, but the helpers have small application-level memory demands and a helper is needed only per concurrent disk operation, not for each concurrently served HTTP request. The MP model incurs the cost of a separate process per concurrently served HTTP request, which has substantial memory and kernel overheads.

Disk utilization - The number of concurrent disk requests that a server can generate affects whether it can benefit from multiple disks and disk head scheduling. The MP/MT models can cause one disk request per process/thread, while the AMPED model can generate one request per helper. In contrast, since all user-level processing stops in the SPED architecture whenever it accesses the disk, it can only generate one disk request at a time. As a result, it cannot benefit from multiple disks or disk head scheduling.

4.2 Cost/Benefits of optimizations & features

The server architecture also impacts the feasibility and profitability of certain types of Web server optimizations and features. We compare the tradeoffs necessary in the various architectures from a qualitative standpoint.

Information gathering - Web servers use information about recent requests for accounting purposes and to improve performance, but the cost of gathering this information across all connections varies in the different models. In the MP model, some form of interprocess communication must be used to consolidate data. The MT model either requires maintaining per-thread statistics and periodic consolidation or fine-grained synchronization on global variables. The SPED and AMPED architectures simplify information gathering since all requests are processed in a centralized fashion, eliminating the need for synchronization or interprocess communications when using shared state.

Application-level Caching - Web servers can employ application-level caching to reduce computation by using memory to store previous results, such as response headers and file mappings for frequently requested content. However, the cache memory competes with the filesystem cache for physical memory, so this technique must be applied carefully. In the MP model, each process may have its own cache in order to reduce interprocess communication and synchronization. The multiple caches increase the number of compulsory misses and they lead to less efficient use of memory. The MT model uses a single cache, but the data accesses/updates must be coordinated through synchronization mechanisms to avoid race conditions. Both AMPED and SPED can use a single cache without synchronization.

Long-lived connections - Long-lived connections occur in Web servers due to clients with slow links (such as modems), or through persistent connections in HTTP 1.1. In both cases, some server-side resources are committed for the duration of the connection. The cost of long-lived connections on the server depends on the resource being occupied. In AMPED and SPED, this cost is a file descriptor, application-level connection information, and some kernel state for the connection. The MT and MP models add the overhead of an extra thread or process, respectively, for each connection.

5 Flash implementation

The Flash Web server is a high-performance implementation of the AMPED architecture that uses aggressive caching and other techniques to maximize its performance. In this section, we describe the implementation of the Flash Web server and some of the optimization techniques used.

5.1 Overview

The Flash Web server implements the AMPED architecture described in Section 3. It uses a single non-blocking server process assisted by helper processes. The server process is responsible for all interaction with clients and CGI applications [26], as well as control of the helper processes. The helper processes are responsible for performing all of the actions that may result in synchronous disk activity. Separate processes were chosen instead of kernel threads to implement the helpers, in order to ensure portability of Flash to operating systems that do not (yet) support kernel threads, such as FreeBSD 2.2.6.

The server is divided into modules that perform the various request processing steps mentioned in Section 2 and modules that handle various caching functions. Three types of caches are maintained: filename translations, response headers, and file mappings. These caches and their function are explained below.

The helper processes are responsible for performing pathname translations and for bringing disk blocks into memory. These processes are dynamically spawned by the server process and are kept in reserve when not active. Each process operates synchronously, waiting on the server for new requests and handling only one request at a time. To minimize interprocess communication, helpers only return a completion notification to the server, rather than sending any file content they may have loaded from disk.

5.2 Pathname Translation Caching

The pathname translation cache maintains a list of mappings between requested filenames (e.g., "/~bob") and actual files on disk (e.g., /home/users/bob/public_html/index.html). This cache allows Flash to avoid using the pathname translation helpers for every incoming request. It reduces the processing needed for pathname translations, and it reduces the number of translation helpers needed by the server. As a result, the memory spent on the cache can be recovered by the reduction in memory used by helper processes.

5.3 Response Header Caching

HTTP servers prepend file data with a response header containing information about the file and the server, and this information can be cached and reused when the same files are repeatedly requested. Since the response header is tied to the underlying file, this cache does not need its own invalidation mechanism. Instead, when the mapping cache detects that a cached file has changed, the corresponding response header is regenerated.

5.4 Mapped Files

Flash retains a cache of memory-mapped files to reduce the number of map/unmap operations necessary for request processing. Memory-mapped files provide a convenient mechanism to avoid extra data copying and double-buffering, but they require extra system calls to create and remove the mappings. Mappings for frequently-requested files can be kept and reused, but unused mappings can increase kernel bookkeeping and degrade performance.

The mapping cache operates on "chunks" of files and lazily unmaps them when too much data has been mapped. Small files occupy one chunk each, while large files are split into multiple chunks. Inactive chunks are maintained in an LRU free list, and are unmapped when this list grows too large. We use LRU to approximate the "clock" page replacement algorithm used in many operating systems, with the goal of mapping only what is likely to be in memory. All mapped file pages are tested for memory residency via `mincore()` before use.

5.5 Byte Position Alignment

The `writev()` system call allows applications to send multiple discontiguous memory regions in one operation. High-performance Web servers use it to send response headers followed by file data. However, its use can cause misaligned data copying within the operating system, degrading performance. The extra cost for misaligned data is proportional to the amount of data being copied.

The problem arises when the OS networking code copies the various memory regions specified in a `writev` operation into a contiguous kernel buffer. If the size of the HTTP response header stored in the first region has a length that is not a multiple of the machine's word size, then the copying of all subsequent regions is misaligned.

Flash avoids this problem by aligning all response headers on 32-byte boundaries and padding their lengths to be a multiple of 32 bytes. It adds characters to variable length fields in the HTTP response header (e.g., the server name) to do the padding. The choice of 32 bytes rather than word-alignment is to target systems with 32-byte cache lines, as some systems may be optimized for copying on cache boundaries.

5.6 Dynamic Content Generation

The Flash Web server handles the serving of dynamic data using mechanisms similar to those used in other Web servers. When a request arrives for a dynamic document, the server forwards the request to the corresponding auxiliary (CGI-bin) application process that generates the content via a pipe. If a process does not currently exist, the server creates (e.g., forks) it.

The resulting data is transmitted by the server just like static content, except that the data is read from a descriptor associated with the CGI process' pipe, rather than a file. The server process allows the CGI application process to be persistent, amortizing the cost of creating the application over multiple requests. This is similar to the FastCGI [27] interface and it provides similar benefits. Since the CGI applications run in separate processes from the server, they can block for disk activity or other reasons and perform arbitrarily long computations without affecting the server.

5.7 Memory Residency Testing

Flash uses the `mincore()` system call, which is available in most modern UNIX systems, to determine if mapped file pages are memory resident. In operating systems that don't support this operation but provide the `mlock()` system call to lock memory pages (e.g., Compaq's Tru64 UNIX, formerly Digital Unix), Flash could use the latter to control its file cache management, eliminating the need for memory residency testing.

Should no suitable operations be available in a given operating system to control the file cache or test for memory residency, it may be possible to use a feedback-based heuristic to minimize blocking on disk I/O. Here, Flash could run the clock algorithm to predict which cached file pages are memory resident. The prediction can adapt to changes in the amount of memory available to the file cache by using continuous feedback from performance counters that keep track of page faults and/or associated disk accesses.

6 Performance Evaluation

In this section, we present experimental results that compare the performance of the different Web server architectures presented in Section 3 on real workloads. Furthermore, we present comparative performance results for Flash and two state-of-the-art Web servers, Apache [1] and Zeus [32], on synthetic and real workloads. Finally, we present results that quantify the performance impact of the various performance optimizations included in Flash.

To enable a meaningful comparison of different architectures by eliminating variations stemming from implementation differences, the same Flash code base is used to build four servers, based on the AMPED (Flash), MT (Flash-MT), MP (Flash-MP), and SPED (Flash-SPED) architectures. These four servers represent all the architectures discussed in this paper, and they were developed by replacing Flash's event/helper dispatch mechanism with the suitable counterparts in the other architectures. In all other respects, however, they are identical to the standard, AMPED-based version of Flash and use the same techniques and optimizations.

In addition, we compare these servers with two widely-used production Web servers, Zeus v1.30 (a high-performance server using the SPED architecture), and Apache v1.3.1 (based on the MP architecture), to provide points of reference.

In our tests, the Flash-MP and Apache servers use 32 server processes and Flash-MT uses 64 threads. Zeus was configured as a single process for the experiments using synthetic workloads, and in a two-process configuration advised by Zeus for the real workload tests. Since the SPED-based Zeus can block on disk I/O, using multiple server processes can yield some performance improvements even on a uniprocessor platform, since it allows the overlapping of computation and disk I/O.

Both Flash-MT and Flash use a memory-mapped file cache with a 128 MB limit and a pathname cache limit of 6000 entries. Each Flash-MP process has a mapped file cache limit of 4 MB and a pathname cache of 200 entries. Note that the caches in an MP server have to be configured smaller, since they are replicated in each process.

The experiments were performed with the servers running on two different operating systems, Solaris 2.6 and FreeBSD 2.2.6. All tests use the same server hardware, based on a 333 MHz Pentium II CPU with 128 MB of memory and multiple 100 Mbit/s Ethernet interfaces. A switched Fast Ethernet connects the server machine to the client machines that generate the workload. Our client software is an event-driven program that simulates multiple HTTP clients [3]. Each simulated HTTP client makes HTTP requests as fast as the server can handle them.

6.1 Synthetic Workload

In the first experiment, a set of clients repeatedly request the same file, where the file size is varied in each test. The simplicity of the workload in this test allows the servers to perform at their highest capacity, since the requested file is cached in the server's main memory. The results are shown in Figures 6 (Solaris) and 7 (FreeBSD). The left-hand side graphs plot the servers' total output bandwidth against the requested file size. The connection rate for small files is shown separately on the right.

Results indicate that the choice of architecture has little impact on a server's performance on a trivial, cached workload. In addition, the Flash variants compare favorably to Zeus, affirming the absolute performance of the Flash-based implementation. The Apache server achieves significantly lower performance on both operating systems and over the entire range of file sizes, most likely the result of the more aggressive optimizations employed in the Flash versions and presumably also in Zeus.

Flash-SPED slightly outperforms Flash because the AMPED model tests the memory-residency of files before sending them. Slight lags in the performance of Flash-MT and Flash-MP are likely due to the extra kernel overhead (context switching, etc.) in these architectures. Zeus' anomalous behavior on FreeBSD for file sizes between 10 and 100 KB appears to stem from the byte alignment problem mentioned in Section 5.5.

All servers enjoy substantially higher performance when run under FreeBSD as opposed to Solaris. The relative performance of the servers is not strongly affected by the operating system.

6.2 Trace-based experiments

While the single-file test can indicate a server's maximum performance on a cached workload, it gives little indication of its performance on real workloads. In the next experiment, the servers are subjected to a more realistic load. We generate a client request stream by replaying access logs from existing Web servers.

Figure 8 shows the throughput in Mb/sec achieved with various Web servers on two different workloads. The "CS trace" was obtained from the logs of Rice University's Computer Science departmental Web server. The "Owlnet trace" reflects traces obtained from a Rice Web server that provides personal Web pages for approximately 4500 students and staff members. The results were obtained with the Web servers running on Solaris.

The results show that Flash with its AMPED architecture achieves the highest throughput on both workloads. Apache achieves the lowest performance. The comparison with Flash-MP shows that this is only in part the result of its MP architecture, and mostly due to its lack of aggressive optimizations like those used in Flash.

The Owlnet trace has a smaller dataset size than the CS trace, and it therefore achieves better cache locality in the server. As a result, Flash-SPED's relative performance is much better on this trace, while MP performs well on the more disk-intensive CS trace. Even though the Owlnet trace has high locality, its average transfer size is smaller than the CS trace, resulting in roughly comparable bandwidth numbers.

A second experiment evaluates server performance under realistic workloads with a range of dataset sizes (and therefore working set sizes). To generate an input stream with a given dataset size, we use the access logs from Rice's ECE departmental Web server and truncate them as appropriate to achieve a given dataset size. The clients then replay this truncated log as a loop to generate requests. In both experiments, two client machines with 32 clients each are used to generate the workload.

Figures 9 (BSD) and 10 (Solaris) shows the performance, measured as the total output bandwidth, of the various servers under real workload and various dataset sizes. We report output bandwidth instead of request/sec in this experiment, because truncating the logs at different points to vary the dataset size also changes the size

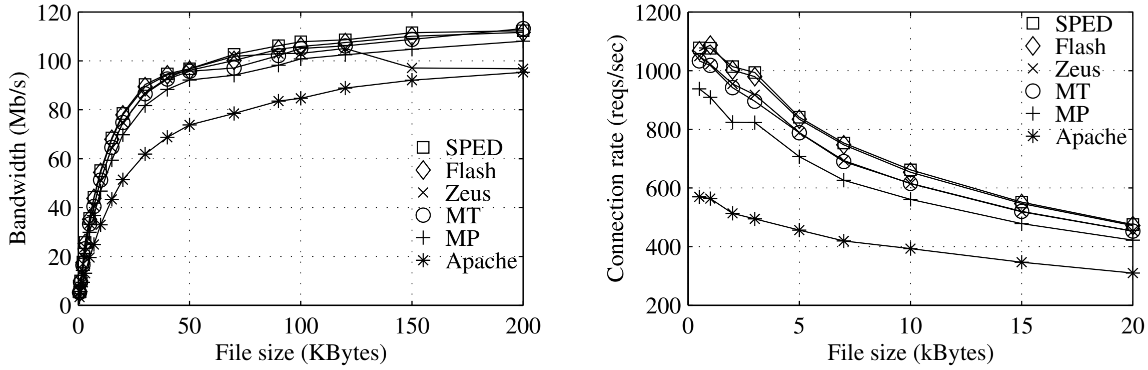

Figure 6: Solaris single file test — On this trivial test, server architecture seems to have little impact on performance. The aggressive optimizations in Flash and Zeus cause them to outperform Apache.

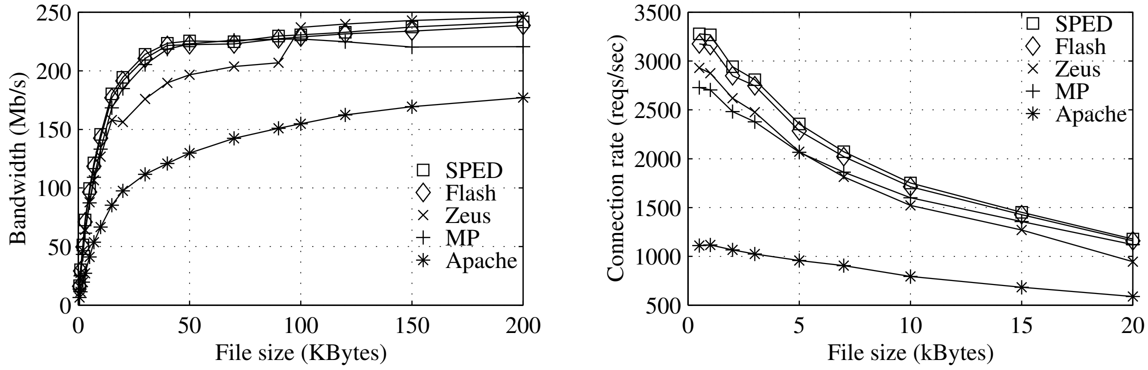

Figure 7: FreeBSD single file test — The higher network performance of FreeBSD magnifies the difference between Apache and the rest when compared to Solaris. The shape of the Zeus curve between 10 kBytes and 100 kBytes is likely due to the byte alignment problem mentioned in Section 5.5.

distribution of requested content. This causes fluctuations in the throughput in requests/sec, but the output bandwidth is less sensitive to this effect.

The performance of all the servers declines as the dataset size increases, and there is a significant drop at the point when the working set size (which is related to the dataset size) exceeds the server's effective main memory cache size. Beyond this point, the servers are essentially disk bound. Several observation can be made based on these results:

- Flash is very competitive with Flash-SPED on cached workloads, and at the same time exceeds or meets the performance of the MP servers on disk-bound workloads. This confirms that Flash with its AMPED architecture is able to combine the best of other architectures across a wide range of workloads. This goal was central to the design of the AMPED architecture.

- The slight performance difference between Flash and Flash-SPED on the cached workloads reflects the overhead of checking for cache residency of requested content in Flash. Since the data is already in memory, this test causes unnecessary overhead on cached workloads.

- The SPED architecture performs well for cached workloads but its performance deteriorates quickly as disk activity increases. This confirms our earlier reasoning about the performance tradeoffs associated with this architecture. The same behavior can be seen in the SPED-based Zeus' performance, although its absolute performance falls short of the various Flash-derived servers.

- The performance of Flash MP server falls significantly short of that achieved with the other architectures on cached workloads. This is likely the result of the smaller user-level caches used in Flash-MP as compared to the other Flash versions.

- The choice of an operating system has a significant impact on Web server performance. Perfor-

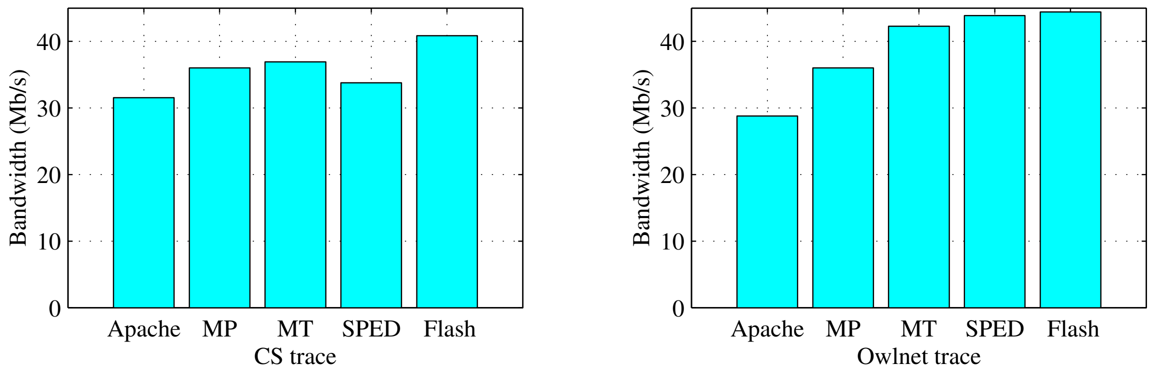

Figure 8: Performance on Rice Server Traces/Solaris

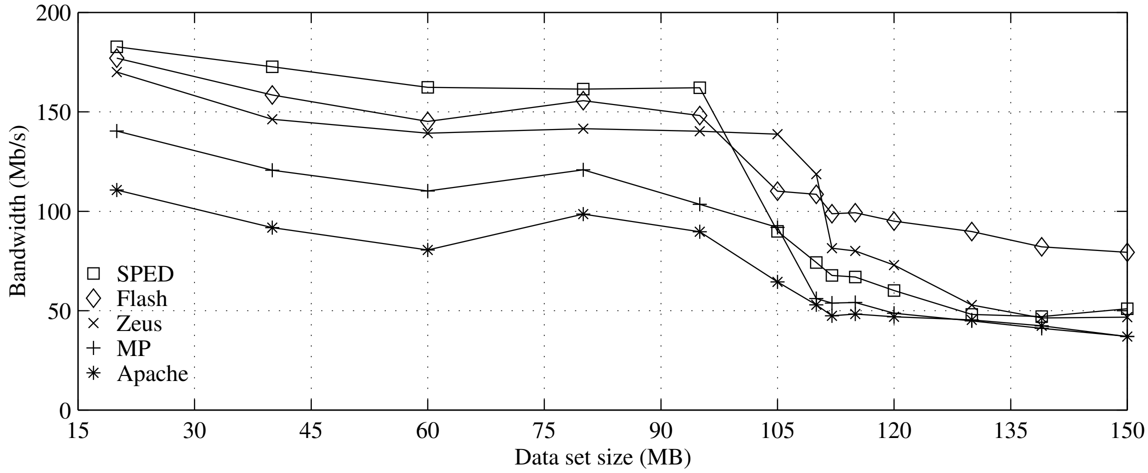

Figure 9: FreeBSD Real Workload - The SPED architecture is ideally suited for cached workloads, and when the working set fits in cache, Flash mimics Flash-SPED. However, Flash-SPED's performance drops drastically when operating on disk-bound workloads.

mance results obtained on Solaris are up to 50% lower than those obtained on FreeBSD. The operating system also has some impact on the relative performance of the various Web servers and architectures, but the trends are less clear.

- Flash achieves higher throughput on disk-bound workloads because it can be more memory-efficient and causes less context switching than MP servers. Flash only needs enough helper processes to keep the disk busy, rather than needing a process per connection. Additionally, the helper processes require little application-level memory. The combination of fewer total processes and small helper processes reduces memory consumption, leaving extra memory for the filesystem cache.

- The performance of Zeus on FreeBSD appears to drop only after the data set exceeds 100 MB, while the other servers drop earlier. We believe this phenomenon is related to Zeus's request-handling, which appears to give priority to requests for small documents. Under full load, this tends to starve requests for large documents and thus causes the server to process a somewhat smaller effective working set. The overall lower performance under Solaris appears to mask this effect on that OS.

- As explained above, Zeus uses a two-process configuration in this experiment, as advised by the vendor. It should be noted that this gives Zeus a slight advantage over the single-process Flash-SPED, since one process can continue to serve requests while the other is blocked on disk I/O.

Results for the Flash-MT servers could not be provided for FreeBSD 2.2.6, because that system lacks support for kernel threads.

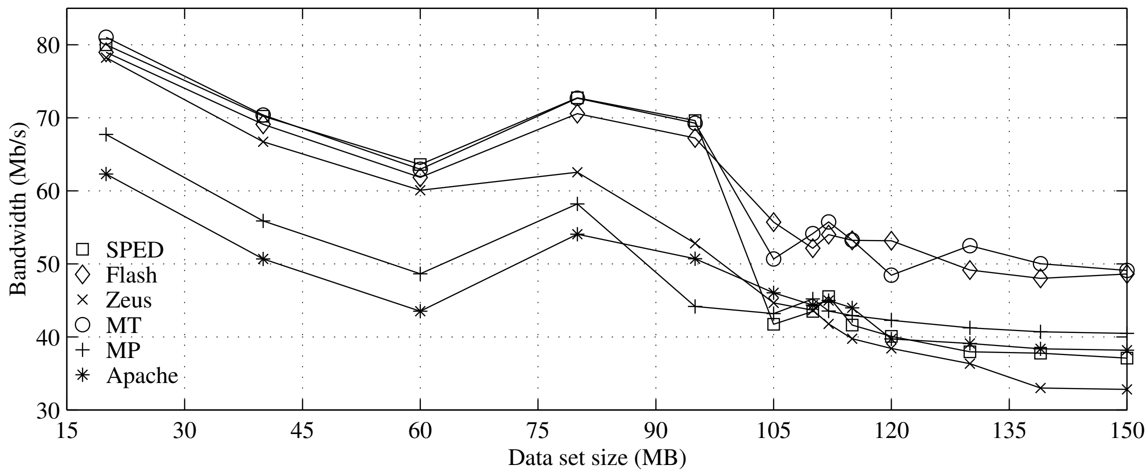

Figure 10: Solaris Real Workload - The Flash-MT server has comparable performance to Flash for both in-core and disk-bound workloads. This result was achieved by carefully minimizing lock contention, adding complexity to the code. Without this effort, the disk-bound results otherwise resembled Flash-SPED.

6.3 Flash Performance Breakdown

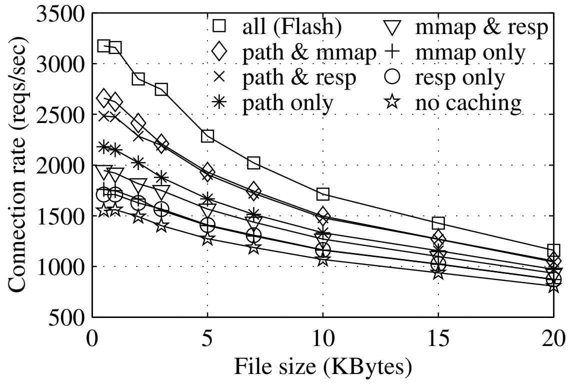

Figure 11: Flash Performance Breakdown - Without optimizations, Flash's small-file performance would drop in half. The eight lines show the effect of various combinations of the caching optimizations.

The next experiment focuses on the Flash server and measures the contribution of its various optimizations on the achieved throughput. The configuration is identical to the single file test on FreeBSD, where clients repeatedly request a cached document of a given size. Figure 11 shows the throughput obtained by various versions of Flash with all combinations of the three main optimizations (pathname translation caching, mapped file caching, and response header caching).

The results show that each of the optimizations has a significant impact on server throughput for cached content, with pathname translation caching providing the largest benefit. Since each of the optimization avoids a per-request cost, the impact is strongest on requests for small documents.

6.4 Performance under WAN conditions

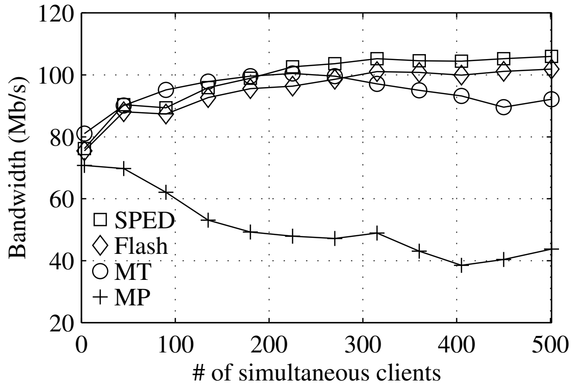

Figure 12: Adding clients - The low per-client overheads of the MT, SPED and AMPED models cause stable performance when adding clients. Multiple application-level caches and per-process overheads cause the MP model's performance to drop.

Web server benchmarking in a LAN environment fails to evaluate an important aspect of real Web workloads, namely that fact that clients contact the server through a wide-area network. The limited bandwidth and packet losses of a WAN increase the average HTTP connection duration, when compared to LAN environment. As a result, at a given throughput in requests/second, a real server handles a significantly larger number of concurrent connections than a server tested under LAN conditions [24].

The number of concurrent connections can have a significant impact on server performance [4]. Our next experiment measures the impact of the number of concurrent HTTP connections on our various servers. Persistent connections were used to simulate the effect of long-lasting WAN connections in a LAN-based testbed. We replay the ECE logs with a 90MB data set size to expose the performance effects of a limited file cache size. In Figure 12 we see the performance under Solaris as the number of number of simultaneous clients is increased.

The SPED, AMPED and MT servers display an initial rise in performance as the number of concurrent connections increases. This increase is likely due to the added concurrency and various aggregation effects. For instance, a large number of connections increases the average number of completed I/O events reported in each `select` system call, amortizing the overhead of this operation over a larger number of I/O events.

As the number of concurrent connections exceeds 200, the performance of SPED and AMPED flattens while the MT server suffers a gradual decline in performance. This decline is related to the per-thread switching and space overhead of the MT architecture. The MP model suffers from additional per-process overhead, which results in a significant decline in performance as the number of concurrent connections increases.

7 Related Work

James Hu et al. [17] perform an analysis of Web server optimizations. They consider two different architectures, the multi-threaded architecture and one that employs a pool of threads, and evaluate their performance on UNIX systems as well as Windows NT using the WebStone benchmark.

Various researchers have analyzed the processing costs of the different steps of HTTP request serving and have proposed improvements. Nahum et al. [25] compare existing high-performance approaches with new socket APIs and evaluate their work on both single-file tests and other benchmarks. Yiming Hu et al. [18] extensively analyze an earlier version of Apache and implement a number of optimizations, improving performance especially for smaller requests. Yates et al. [31] measure the demands a server places on the operating system for various workloads types and service rates. Banga et al. [5] examine operating system support for event-driven servers and propose new APIs to remove bottlenecks observed with large numbers of concurrent connections.

The Flash server and its AMPED architecture bear some resemblance to Thoth [9], a portable operating system and environment built using "multi-process structuring." This model of programming uses groups of processes called "teams" which cooperate by passing messages to indicate activity. Parallelism and asynchronous operation can be handled by having one process synchronously wait for an activity and then communicate its occurrence to an event-driven server. In this model, Flash's disk helper processes can be seen as waiting for asynchronous events (completion of a disk access) and relaying that information to the main server process.

The Harvest/Squid project [8] also uses the model of an event-driven server combined with helper processes waiting on slow actions. In that case, the server keeps its own DNS cache and uses a set of "dnsserver" processes to perform calls to the `gethostbyname()` library routine. Since the DNS lookup can cause the library routine to block, only the dnsserver process is affected. Whereas Flash uses the helper mechanism for blocking disk accesses, Harvest attempts to use the `select()` call to perform non-blocking file accesses. As explained earlier, most UNIX systems do not support this use of `select()` and falsely indicate that the disk access will not block. Harvest also attempts to reduce the number of disk metadata operations.

Given the impact of disk accesses on Web servers, new caching policies have been proposed in other work. Arlitt et al. [2] propose new caching policies by analyzing server access logs and looking for similarities across servers. Cao et al. [7] introduce the Greedy DualSize caching policy which uses both access frequency and file size in making cache replacement decisions. Other work has also analyzed various aspects of Web server workloads [11, 23].

Data copying within the operating system is a significant cost when processing large files, and several approaches have been proposed to alleviate the problem. Thadani et al. [30] introduce a new API to read and send memory-mapped files without copying. IO-Lite [29] extends the fbufs [14] model to integrate filesystem, networking, interprocess communication, and application-level buffers using a set of uniform interfaces. Engler et al. [20] use low-level interaction between the Cheetah Web server and their exokernel to eliminate copying and streamline small-request handling. The Lava project uses similar techniques in a microkernel environment [22].

Other approaches for increasing Web server performance employ multiple machines. In this area, some work has focused on using multiple server nodes in parallel [6, 10, 13, 16, 19, 28], or sharing memory across machines [12, 15, 21].

8 Conclusion

This paper presents a new portable high-performance Web server architecture, called asymmetric multi-process event-driven (AMPED), and describes an implementation of this architecture, the Flash Web server. Flash nearly matches the performance of SPED servers on cached workloads while simultaneously matching or exceeding the performance of MP and MT servers on disk-intensive workloads. Moreover, Flash uses only standard APIs available in modern operating systems and is therefore easily portable.

We present results of experiments to evaluate the impact of a Web server's concurrency architecture on its performance. For this purpose, various server architectures were implemented from the same code base. Results show that Flash with its AMPED architecture can nearly match or exceed the performance of other architectures across a wide range of realistic workloads.

Results also show that the Flash server's performance exceeds that of the Zeus Web server by up to 30%, and it exceeds the performance of Apache by up to 50% on real workloads. Finally, we perform experiments to show the contribution of the various optimizations embedded in Flash on its performance.

Acknowledgments

We are grateful to Erich Nahum, Jeff Mogul, and the anonymous reviewers, whose comments have helped to improve this paper. Thanks to Michael Pearlman for our Solaris testbed configuration. Special thanks to Zeus Technology for use of their server software and Damian Reeves for feedback and technical assistance with it. Thanks to Jef Poskanzer for the thttpd web server, from which Flash derives some infrastructure. This work was supported in part by NSF Grants CCR-9803673, CCR-9503098, MIP-9521386, by Texas TATP Grant 003604, and by an IBM Partnership Award.

References

[1] Apache. http://www.apache.org

[2] M. F. Arlitt and C. L. Williamson. Web Server Workload Characterization: The Search for Invariants. In *Proceedings of the ACM SIGMETRICS '96 Conference*, pages 126–137, Philadelphia, PA, Apr. 1996.

[3] G. Banga and P. Druschel. Measuring the capacity of a Web server. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, Monterey, CA, Dec. 1997.

[4] G. Banga and P. Druschel. Measuring the capacity of a Web server under realistic loads. *World Wide Web Journal (Special Issue on World Wide Web Characterization and Performance Evaluation)*, 1999. To appear.

[5] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. 3rd USENIX Symp. on Operating Systems Design and Implementation*, Feb. 1999.

[6] T. Brisco. DNS Support for Load Balancing. RFC 1794, Apr. 1995.

[7] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, Monterey, CA, Dec. 1997.

[8] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 Usenix Technical Conference*, Jan. 1996.

[9] D. R. Cheriton. *The Thoth System: Multi-Process Structuring and Portability*. Elsevier Science Publishing Co,. Inc, 1982.

[10] Cisco Systems Inc. LocalDirector. http://www.cisco.com

[11] M. Crovella and A. Bestavros. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. In *Proceedings of the ACM SIGMETRICS '96 Conference*, pages 160–169, Philadelphia, PA, Apr. 1996.

[12] M. Dahlin, R. Yang, T. Anderson, and D. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proc. USENIX Symp. on Operating Systems Design and Implementation*, Monterey, CA, Nov. 1994.

[13] O. P. Damani, P.-Y. E. Chung, Y. Huang, C. Kintala, and Y.-M. Wang. ONE-IP: Techniques for hosting a service on a cluster of machines. *Computer Networks and ISDN Systems*, 29:1019–1027, 1997.

[14] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 189–202, Dec. 1993.

[15] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Copper Mountain, CO, Dec. 1995.

[16] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, San Malo, France, Oct. 1997.

[17] J. C. Hu, I. Pyrali, and D. C. Schmidt. Measuring the impact of event dispatching and concurrency

models on web server performance over high-speed networks. In *Proceedings of the 2nd Global Internet Conference*, Phoenix, AZ, Nov. 1997.

[18] Y. Hu, A. Nanda, and Q. Yang. Measurement, analysis and performance improvement of the Apache web server. In *Proceedings of the 18th IEEE International Performance, Computing and Communications Conference (IPCCC'99)*, February 1999.

[19] IBM Corporation. IBM eNetwork dispatcher. http://www.software.ibm.com/network/dispatcher

[20] M. F. Kaashoek, D. R. Engler, G. R. Ganger, and D. A. Wallach. Server Operating Systems. In *Proceedings of the 1996 ACM SIGOPS European Workshop*, pages 141–148, Connemara, Ireland, Sept. 1996.

[21] H. Levy, G. Voelker, A. Karlin, E. Anderson, and T. Kimbrel. Implementing Cooperative Prefetching and Caching in a Globally-Managed Memory System. In *Proceedings of the ACM SIGMETRICS '98 Conference*, Madison, WI, June 1998.

[22] J. Liedtke, V. Panteleenko, T. Jaeger, and N. Islam. High-performance caching with the Lava hit-server. In *Proceedings of the USENIX 1998 Annual Technical Conference*, New Orleans, LA, June 1998.

[23] S. Manley and M. Seltzer. Web Facts and Fantasy. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, pages 125–134, Monterey, CA, Dec. 1997.

[24] J. C. Mogul. Network behavior of a busy web server and its clients. Technical Report WRL 95/5, DEC Western Research Laboratory, Palo Alto, CA, 1995.

[25] E. Nahum, T. Barzilai, and D. Kandlur. Performance Issues in WWW Servers. submitted for publication.

[26] National Center for Supercomputing Applications. Common Gateway Interface. http://hoohoo.ncsa.uiuc.edu/cgi

[27] Open Market, Inc. FastCGI specification. http://www.fastcgi.com

[28] V. S. 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 8th Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1998. ACM.

[29] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, New Orleans, LA, Feb. 1999.

[30] M. N. Thadani and Y. A. Khalidi. An efficient zero-copy I/O framework for UNIX. Technical Report SMLI TR-95-39, Sun Microsystems Laboratories, Inc., May 1995.

[31] D. Yates, V. Almeida, and J. Almeida. On the interaction between an operating system and Web server. Technical Report TR-97-012, Boston University, CS Dept., Boston MA, 1997.

[32] Zeus Technology Limited. Zeus Web Server. http://www.zeus.co.uk