RICE UNIVERSITY

Cache Management in Scalable Network Servers

by

Vivek S. Pai

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

DOCTOR OF PHILOSOPHY

Approved, Thesis Committee:

Willy Zwaenepoel, Chair
Professor of Computer Science

Peter Druschel
Assistant Professor of Computer Science

Peter J. Varman
Associate Professor in Electrical and
Computer Engineering

Alan Cox
Associate Professor of Computer Science

Houston, Texas

November, 1999

Cache Management in Scalable Network Servers

Vivek S. Pai

Abstract

For many users, the perceived speed of computing is increasingly dependent on the performance of network server systems, underscoring the need for high performance servers. Cost-effective scalable network servers can be built on clusters of commodity components (PCs and LANs) instead of using expensive multiprocessor systems. However, network servers cache files to reduce disk access, and the cluster's physically disjoint memories complicate sharing cached file data. Additionally, the physically disjoint CPUs complicate the problem of load balancing. This work examines the issue of cache management in scalable network servers at two levels – per-node (local) and cluster-wide (global).

Per-node cache management is addressed by the IO-Lite unified buffering and caching system. Applications and various parts of the operating system currently use incompatible buffering schemes, resulting in unnecessary data copying. For network servers, overall throughput drops for two reasons – copying wastes CPU cycles, and multiple copies of data compete with the filesystem cache for memory. IO-Lite allows applications, the operating system, file system, and network code to safely and securely share a single copy of data.

The cluster-wide solution uses a technique called Locality-Aware Request Distribution (LARD) that examines the content of incoming requests to determine which node in a cluster should handle the request. LARD uses the request content to dynamically partition the incoming request stream. This partitioning increases the file cache hit rates on the individual nodes, and it maintains load balance in the cluster.

Contents

Illustrations

Chapter 1

Introduction

The advent of the World Wide Web (WWW) and user-friendly Web browsers enabled millions of people to access online information and entertainment on the Internet. Many metrics indicate an exponential growth of Internet usage, and this trend appears to be continuing. Factors that will keep driving this trend include increasing computer penetration into homes, bandwidth increases to the end-user, and new mechanisms for accessing the Internet. The increase in computer penetration results in a larger pool of people using the Internet. Cable modems and digital subscriber lines (DSL) provide one to two orders of magnitude more bandwidth for the end user, allowing each person to access more content in the same amount of time. Wireless devices and specialized Web access terminals allow people to access the Internet in situations where access previously would have been difficult or impossible. All of these factors will continue to fuel the growth of Internet usage.

Network servers are the machines responsible for generating the content that gets delivered to the end user. As a result, the number of users and their rate of activity directly influences the demands placed on network servers. However, network servers must also contend with the demands placed on them by the mechanisms for transmitting content. In the process of displaying a single Web "page" for a user, a Web browser may have to generate multiple requests to a network server. Each "frame," or part of the page, requires a different request, as does each image contained in the page. Therefore, network servers not only have to deal with hundreds or thousands of users simultaneously requesting documents, but they must also deal with multiple requests from each user.

Given the Internet's dependence on network servers, the scalability of network servers could affect the growth and performance of the World Wide Web. Two potential limitations for network servers are the input/output (I/O) systems in current operating systems, and the capacity of individual machines. The function of network servers is intimately tied to various I/O systems – reading requests from clients, loading files from storage, and sending data (text, audio, images, etc.) to clients all require I/O operations. As a result, inefficiency in the I/O system can negatively impact network servers. Network servers also have to contend with the finite processing capacity of individual machines. Accepting incoming connections, interpreting requests, generating responses, and processing outbound data all require processing capacity. These processing requirements place an upper limit on the number of simultaneous requests a single network server can handle.

We address these scalability limitations by focusing on effective buffer and cache management. For the case of individual machines, we examine the issue in the context of the I/O system. We discuss how the design of traditional I/O systems leads to inefficiencies, and we demonstrate how these inefficiencies can be eliminated through a unified I/O system. To scale beyond the capacities of single machines, we use a cluster-based approach, where a group of network servers cooperate to handle a heavy workload. In such a system, a stream of incoming requests is partitioned among the servers in the cluster. Our work focuses on a new request partitioning technique designed to improve the filesystem cache efficiency of the machines in the cluster. This technique improves the performance of the individual machines, increasing the cluster's capacity beyond what can be achieved by existing approaches.

1.1 Network Server Basics

Much of the work performed by network servers focuses on transferring data, making the efficient management of buffers and caches key to overall performance. Consider the case of a client requesting a file from a Web server. The server must determine what file is being requested, load the file from disk if necessary, and send the contents of the file to the client. Each of these steps involves some form of buffering or caching, and this example illustrates why buffers and caches play a central role in network servers.

A disk cache refers to physical memory being used to hold data stored on disk, while a buffer refers to a region of memory used to hold data while it is being transferred. In network servers, effective management of buffers and caches is essential to achieving high performance. Since disks are mechanical devices, accessing data directly from disk incurs long delays, on the order of tens of milliseconds. If every request handled by the network server were forced to retrieve data directly from disk, disk performance would limit overall server performance. By keeping frequently-used disk data in main memory, the server can access the data from memory, avoiding the long delays associated with disk access.

In traditional Unix systems, buffers and caches are separate and identified by their ownership – applications, the filesystem, the network subsystem, and the interprocess communication subsystem. For network servers, this separation is important for two reasons – the buffers and caches consume memory, a limited resource, and the process of copying the data in buffers consumes processor cycles and memory bandwidth, also limited resources.

Intelligent buffer and cache management can provide two benefits for network servers: (1) maximize the effective memory size by reducing the amount of replicated data in buffers, and (2) improve performance by reducing or eliminating data copying between different buffer types. The increase in memory size allows servers to increase the size of the disk cache, thereby improving overall performance by reducing the number of disk accesses.

1.2 The I/O Problem

General-purpose operating systems provide inadequate support for server applications, leading to poor server performance and increased hardware cost of server systems. One source of the problem is lack of integration among the various input-output (I/O) subsystems and the application in general-purpose operating systems. Each I/O subsystem uses its own buffering or caching mechanism, and applications generally maintain their own private I/O buffers. This approach leads to repeated data copying, multiple buffering of I/O data, and other performance-degrading anomalies.

Data copying has three negative effects – wasting CPU cycles, polluting data caches, and consuming memory bandwidth. These problems limit scalability along three axes, which we discuss below. The root cause of the problems is that main memory bandwidths have not kept pace with improvements in CPU speed.

- **CPU cache effectiveness** - For applications with good locality, the impact of the discrepancy between CPU speed and main memory bandwidth is mitigated by the use of high-speed CPU data caches. Since all data accessed by the CPU goes through the data cache, data copying evicts other items from the data cache. To make matters worse, the copied data may not be used again by the CPU, resulting in useful items having been evicted from the cache to make space for useless data. This effect, known as cache pollution, reduces the benefits of CPU data caches.

- **CPU speed** - Networks servers often handle data sets exceeding the size of CPU caches. When this occurs, data copying gains no benefit from the presence of the CPU caches – they are too small to hold the data being read, and are likewise too small to buffer the data being written. As a result, data copying proceeds at memory bandwidth speeds, since all accesses must be satisfied by the main memory. Since the CPU stalls while waiting for the memory subsystem, data copying reduces the benefits of faster CPUs.

- **Multiprocessor performance** - As noted above, data copying consumes main memory bandwidth, which is a limited resource. This effect is particularly trouble on multiprocessor machines, where memory bandwidth is a shared resource. In such cases, data copying by one processor can interfere with the progress of other processors performing memory operations. If multiple CPUs are copying data to/from main memory, the main memory bandwidth is divided amongst the CPUs. In these situations, adding extras CPUs does not improve performance, since each CPU only gets a smaller share of the bandwidth.

For network servers, the negative effects of multiple buffering are equally important. Multiple buffering of data wastes memory, an important resource for network servers. Since all sources of virtual memory in the machine compete for the available physical memory, multiple buffering reduces the memory available in the machine for other uses. In practice, the effect of multiple buffering is to reduce the memory available to the filesystem cache. A reduced cache size causes higher cache miss rates, increasing disk accesses and reducing throughput.

1.3 The Cluster Problem

Existing approaches to increasing network server capacity exhibit poor scalability and/or fare significantly worse in price/performance than commodity machines. Two common approaches for increasing network server capacity are to either use a more powerful single machine, such as a multiprocessor, or to use a set of machines to share the work.

While large-scale multiprocessor systems can provide more performance than uniprocessors, they have several drawbacks that limit their attractiveness in this regard. Large multiprocessors will always have a worse price/performance ratio than commodity machines since their high development costs must be amortized over a lower sales volume. Multiprocessors are also limited in the number of CPUs that a single machine can accomodate, limiting their effectiveness for high-volume network servers. Finally, a single large machine becomes a single point of failure, which limits the attractiveness of multiprocessors for certain applications.

Using a set of commodity machines (a cluster) connected by a high-speed network is a more promising hardware platform than a large multiprocessor for several reasons. Due to the economies of scale, commodity components will maintain a better price/performance ratio than large multiprocessors. Additionally, such a platform is not limited in the number of CPUs that can be physically configured in a single machine. Finally, unlike multiprocessors, the capacity of such a system can be incrementally increased as needed, without requiring significant initial investments to accomodate future growth. The new problem introduced in using such a scheme, however, is sharing the work among all of the machines in the cluster.

Existing approaches to sharing work among a cluster of network servers either sacrifice flexibility or make poor use of the cluster's resources. Two current schemes for distributing work among the nodes of a cluster are manual partitioning of the server content, or load balancing by assigning requests to balance work in the cluster. Manually partitioning the server content requires administrative involvement, and such techniques fare poorly in environments where document popularity can vary widely over time. Load balancing approaches fare better in this respect, but they can increase overall cluster costs because additional hardware is needed to maintain performance. The root cause of the problem is the lack of effective filesystem cache management across the nodes of the cluster.

Because the nodes of a cluster server are disjoint, they waste resources by duplicating effort. The nodes have separate physical memories, operating system instances, and filesystem caches. Without some form of coordination, multiple requests for the same content can be spread across the cluster. When this situation occurs, all of the nodes waste resources: CPU time handling the request, disk time loading the data, and memory space caching the request content. Since popular documents are often requested by many clients, this situation is common in cluster servers.

These effects limit the scalability of the cluster by making poor use of the cluster resources, especially the filesystem caches. Over time, the nodes of the cluster will cache the more popular documents in the server's content. Without any mechanism for coordination between nodes, the filesystem caches of the cluster nodes become near-replicas of each other. As a result, the effective cache size of the cluster is just the size of a single machine. When the workload on the cluster increases and the site content grows, not only must new nodes be added for processing capacity, but all of the existing nodes' memories must be increased to maintain the filesystem cache hit rate.

1.4 Thesis Statement and Contributions

The hypothesis of this dissertation is that the scalability limits in network servers stem from the lack of effective buffer and cache management. Effective buffer and cache management must be applied at two levels – within each machine of the cluster, and between machines in the cluster. Within each machine, the buffer and cache management must encompass the operating system, the filesystem, the network subsystem, the interprocess communication system, and applications. In the cluster-wide case, the filesystem caches in the various nodes must be coordinated.

This goal of this dissertation is to design a *scalable cluster-based network server* by addressing the issue of effective buffer and cache management. We believe that we can meet our goals without abandoning current systems or precluding future growth. Toward these goals, our contributions focus in this work are the following:

- A new kernel subsystem that addresses per-node buffer and cache management, resulting in more effective CPU utilization;

- a new technique for intelligently dispatching requests to nodes of a cluster server, yielding higher filesystem cache performance; and

- a new concurrency architecture for high-performance network server applications.

These new techniques result from two central observations regarding the performance characteristics of network servers:

- **Effective filesystem caches are essential for good network server performance.** Given that much of the work done by network servers revolves around serving files or parts of files to clients, the filesystem cache is a vital part of the operating system. For these kinds of transfers, disk access times are one to two orders slower than memory access times. As a result, network servers can effectively utilize optimizations that improve the filesystem cache size or filesystem cache hit rate.

- **Current I/O system designs limit the benefits of fast CPUs.** Once a network server has reduces its dependence on disk speed through caching files in main memory, the performance of the memory system becomes a limiting factor to request processing throughput. Current I/O systems are not designed to minimize memory traffic, resulting in relative slow memory systems limiting the throughput of much faster CPUs.

Our hardware design uses a cluster of commodity machines as processing nodes connected via high-performance commodity networks, allowing us to scale overall throughput by adding more nodes. The compelling price/performance of commodity components makes them attractive candidates for becoming the building blocks of a scalable server. The software component of our system allows the use of standard operating systems running off-the-shelf server software or the use of slightly-modified server software to gain even better performance. The combination of these features allows for a steady, scalable growth path that is able to take advantage of advances in both hardware and software as performance requirements increase.

1.5 Dissertation Overview

Chapter 2 describes the IO-Lite unified buffering and caching system. The goal of IO-Lite is to unify all buffers in the system, allowing applications, the operating system, the filesystem, the network subsystem, and the interprocess communication system to safely and concurrently share a single copy of data. In doing so, IO-Lite eliminates the data copying that would normally occur when thesis entities communicate. This chapter presents the design of IO-Lite, describes its implementation in the FreeBSD operating system, and evaluates its performance on a set of applications. Since we are primarily interested in network servers, the bulk of the evaluation is presented using a Web server. Finally, we present other approaches to copy avoidance that have been proposed, and discuss their strengths and weaknesses in the context of network server support.

Chapter 3 describes our approach to request distribution in clusters. Clusters of commodity machines connected via high-speed networks provide a cost-effective scalable platform for network servers. However, existing approaches to request distribution in cluster-based network servers focus only on load balance within the cluster. We develop a novel technique that achieves both good balance and improves the locality seen by each node in the cluster. This technique, called Locality-Aware Request Distribution (LARD), is first evaluated via simulation against other request distribution strategies. We then presents results on an actual cluster using a workload derived from actual Web servers.

After discussing optimizations within the operating system and at the cluster level, we focus our attention on the architecture of network server applications. In Chapter 4, we describe a novel concurrency architecture for network servers called Asymmetric Multiple Process Event Driven (AMPED). This architecture is designed to provide high performance across a wide range of workloads. We first compare this architecture to other concurrency architectures, and then describe an implementation of this architecture, the Flash Web server. Finally, we evaluate the performance of the

Flash Web server, both against other Web servers and against other implementations of itself using different concurrency architectures.

Chapter 2

IO-Lite

For many users, the perceived speed of computing is increasingly dependent on the performance of networked server systems, underscoring the need for high performance servers. Unfortunately, general-purpose operating systems provide inadequate support for server applications, leading to poor server performance and increased hardware cost of server systems.

One source of the problem is lack of integration among the various input-output (I/O) subsystems and applications in general-purpose operating systems. Each I/O subsystem uses its own buffering or caching mechanism, and applications generally maintain their own private I/O buffers. This approach leads to repeated data copying, multiple buffering of I/O data, and other performance-degrading anomalies.

Repeated data copying causes high CPU overhead and limits the throughput of a server. Multiple buffering of data wastes memory, reducing the space available for the filesystem cache. A reduced cache size causes higher cache miss rates, increasing the number of disk accesses and reducing throughput. Finally, lack of support for application-specific cache replacement policies [17] and optimizations like TCP checksum caching [40] further reduce server performance.

We present the design, the implementation, and the performance of IO-Lite, a unified I/O buffering and caching system for general-purpose operating systems. IO-Lite unifies *all* buffering and caching in the system to the extent permitted by the hardware. In particular, it allows applications, interprocess communication, the file cache, the network subsystem, and other I/O subsystems to safely and concurrently share a single physical copy of the data. IO-Lite achieves this goal by storing buffered

I/O data in immutable buffers, whose locations in physical memory never change. The various subsystems use mutable buffer aggregates to access the data according to their needs.

The primary goal of IO-Lite is to improve the performance of server applications such as those running on networked (e.g., Web) servers, and other I/O-intensive applications. IO-Lite avoids redundant data copying (decreasing I/O overhead), avoids multiple buffering (increasing effective file cache size), and permits performance optimizations across subsystems (e.g., application-specific file cache replacement and cached Internet checksums).

We introduce a new IO-Lite application programming interface (API) designed to facilitate general-purpose I/O without copying. Applications wanting to gain the maximum benefit from IO-Lite use the interface directly. Other applications can benefit by linking with modified I/O libraries (e.g., *stdio*) that use the IO-Lite API internally. Existing applications can work unmodified, since the existing I/O interfaces continue to work.

A prototype of IO-Lite was implemented in FreeBSD. In keeping with the goal of improving performance of networked servers, our central performance results involve a Web server, in addition to other benchmark applications. Results show that IO-Lite yields a performance advantage of 40 to 80% on real workloads. IO-Lite also allows efficient support for dynamic content using third-party CGI programs without loss of fault isolation and protection.

The outline of the rest of this chapter is as follows: Section 2.1 discusses the design of the buffering and caching systems in UNIX and their deficiencies. Section 2.2 presents the design of IO-Lite and discusses its operation in a Web server application. Section 2.3 describes our prototype IO-Lite implementation in FreeBSD. A quantitative evaluation of IO-Lite is presented in Section 2.4, including performance results with a Web server on real workloads. In Section 2.5, we present a qualitative discussion of IO-Lite in the context of related work, and we conclude in Section 2.6.

2.1 Background

In state-of-the-art, general-purpose operating systems, each major I/O subsystem employs its own buffering and caching mechanism. In UNIX, for instance, the network subsystem operates on data stored in BSD *mbufs* or the equivalent System V *streambufs*, allocated from a private kernel memory pool. The mbuf (or streambuf) abstraction is designed to efficiently support common network protocol operations such as packet fragmentation/reassembly and header manipulation.

The UNIX filesystem employs a separate mechanism designed to allow the buffering and caching of logical disk blocks (and more generally, data from block oriented devices.) Buffers in this *buffer cache* are allocated from a separate pool of kernel memory.

In older UNIX systems, the buffer cache is used to store all disk data. In modern UNIX systems, only filesystem metadata is stored in the buffer cache; file data is cached in VM pages, allowing the file cache to compete with other virtual memory segments for the entire pool of physical main memory.

No support is provided in UNIX systems for buffering and caching at the user level. Applications are expected to provide their own buffering and/or caching mechanisms, and I/O data is generally copied between OS and application buffers during I/O read and write operations[1]. The presence of separate buffering/caching mechanisms in the application and in the major I/O subsystems poses a number of problems for I/O performance:

Redundant data copying: Data copying may occur multiple times along the I/O data path. We call such copying *redundant*, because it is not necessary to satisfy some hardware constraint. Instead, it is imposed by the system's software structure and its interfaces. Data copying is an expensive operation, because it generally proceeds at memory rather than CPU speed and it tends to pollute the data cache.

[1] Some systems transparently avoid this data copying under certain conditions using page remapping and copy-on-write.

Multiple buffering: The lack of integration in the buffering/caching mechanisms may require that multiple copies of a data object be stored in main memory. In a Web server, for example, a data file may be stored in the filesystem cache, in the Web server's buffers, and in the send buffers of one or more connections in the network subsystem. This duplication reduces the effective size of main memory, and thus the size and hit rate of the server's file cache.

Lack of cross-subsystem optimization: Separate buffering mechanisms make it difficult for individual subsystems to recognize opportunities for optimizations. For example, the network subsystem of a server is forced to recompute the Internet checksum each time a file is being served from the server's cache, because it cannot determine that the same data is being transmitted repeatedly.

2.2 IO-Lite Design

2.2.1 Principles: Immutable Buffers and Buffer Aggregates

In IO-Lite, all I/O data buffers are *immutable*. Immutable buffers are allocated with an initial data content that may not be subsequently modified. This access model implies that all sharing of buffers is read-only, which eliminates problems of synchronization, protection, consistency, and fault isolation among OS subsystems and applications. Data privacy is ensured through conventional page-based access control.

Moreover, read-only sharing enables very efficient mechanisms for the transfer of I/O data across protection domain boundaries, as discussed in Section 2.2.2. For example, the filesystem cache, applications that access a given file, and the network subsystem can all safely refer to a single physical copy of the data.

The price for using immutable buffers is that I/O data can not generally be modified in place[2]. To alleviate the impact of this restriction, IO-Lite encapsulates I/O

[2]As an optimization, I/O data can be modified in place if it is not currently shared.

data buffers inside the *buffer aggregate* abstraction. Buffer aggregates are instances of an abstract data type (ADT) that represents I/O data. All OS subsystems access I/O data through this unified abstraction. Applications that wish to obtain the best possible performance can also choose to access I/O data in this way.

The data contained in a buffer aggregate does not generally reside in contiguous storage. Instead, a buffer aggregate is represented internally as an ordered list of <*pointer,length*> pairs, where each pair refers to a contiguous section of an immutable I/O buffer. Buffer aggregates support operations for truncating, prepending, appending, concatenating, and splitting data contained in I/O buffers.

While the underlying I/O buffers are *immutable*, buffer aggregates are *mutable*. To mutate a buffer aggregate, modified values are stored in a newly-allocated buffer, and the modified sections are then logically joined with the unmodified portions through pointer manipulations in the obvious way. The impact of the absence of in-place modifications will be discussed in Section 2.2.8.

In IO-Lite, all I/O data is encapsulated in buffer aggregates. Aggregates are passed among OS subsystems and applications by value, but the associated IO-Lite buffers are passed *by reference*. This approach allows a single physical copy of I/O data to be shared throughout the system. When a buffer aggregate is passed across a protection domain boundary, the VM pages occupied by all of the aggregate's buffers are made readable in the receiving domain.

Conventional access control ensures that a process can only access I/O buffers associated with buffer aggregates that were explicitly passed to that process. The read-only sharing of immutable buffers ensures fault isolation, protection, and consistency despite the concurrent sharing of I/O data among multiple OS subsystems and applications. A system-wide reference counting mechanism for I/O buffers allows safe reclamation of unused buffers.

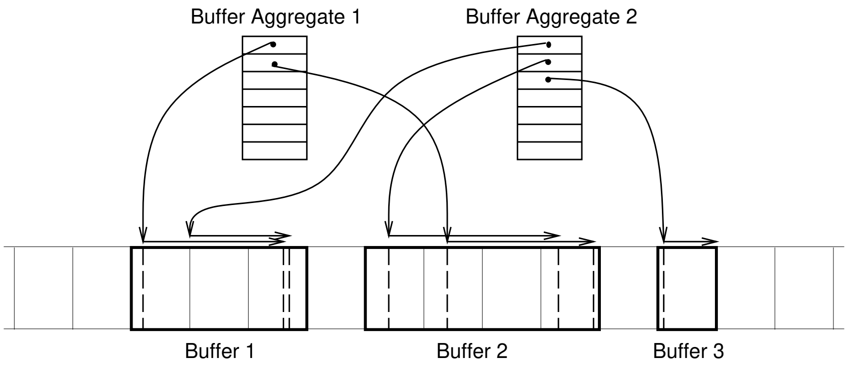

Figure 2.1 : Aggregate buffers and slices – IO-Lite allocates contiguous buffers in virtual memory. Applications access these buffers through data structures called buffer aggregates, which contain ordered tuples of the form <address, length>. Each tuple refers to a subrange of memory called a *slice*.

2.2.2 Interprocess Communication

In order to support caching as part of a unified buffer system, an interprocess communication mechanism must allow safe *concurrent* sharing of buffers. In other words, different protection domains must be allowed protected, concurrent access to the same buffer. For instance, a caching Web server must retain access to a cached document after it passes the document to the network subsystem or to a local client.

IO-Lite uses an IPC mechanism similar to *fbufs* [27] to support safe concurrent sharing. Copy-free I/O facilities that only allow *sequential* sharing [15, 57] are not suitable for use in caching I/O systems, since only one protection domain has access to a given buffer at any time and reads are destructive.

I/O-Lite extends fbufs in two significant directions. First, it extends the fbuf approach from the network subsystem to the filesystem, including the file data cache, thus unifying the buffering of I/O data throughout the system. Second, it adapts the fbuf approach, originally designed for the x-kernel [35], to a general-purpose operating system.

IO-Lite's IPC, like fbufs, combines page remapping and shared memory. Initially, when an (immutable) buffer is transferred, VM mappings are updated to grant the receiving process read access to the buffer's pages. Once the buffer is deallocated, these mappings persist, and the buffer is added to a cached pool of free buffers associated with the I/O stream on which it was first used, forming a lazily established pool of read-only shared memory pages.

When the buffer is reused, no further VM map changes are required, except that temporary write permissions must be granted to the producer of the data, to allow it to fill the buffer. This toggling of write permissions can be avoided whenever the producer is a trusted entity, such as the OS kernel. Here, write permissions can be granted permanently, since a trusted entity is expected to honor the buffer's immutability.

IO-Lite's worst case cross-domain transfer overhead is that of page remapping; it occurs when the producer allocates the last buffer in a particular buffer pool before the first buffer is deallocated by the receiver(s). Otherwise, buffers can be recycled, and the transfer performance approaches that of shared memory.

2.2.3 Access Control and Allocation

IO-Lite ensures access control and protection at the granularity of processes. No loss of security or safety is associated with the use of IO-Lite. IO-Lite maintains cached pools of buffers with a common access control list (ACL), i.e., a set of processes with access to all IO-Lite buffers in the pool. The choice of a pool from which a new IO-Lite buffer is allocated determines the ACL of the data stored in the buffer.

IO-Lite's access control model requires programs to determine the ACL of an I/O data object prior to storing it in main memory, in order to avoid copying or page remapping. Determining the ACL is trivial in most cases, except when an incoming packet arrives at a network interface, as discussed in Section 2.2.6.

Figure 2.1 depicts the relationship between VM pages, buffers, and buffer aggregates. IO-Lite buffers are allocated in a region of the virtual address space called the *IO-Lite window*. The IO-Lite window appears in the virtual address spaces of all protection domains, including the kernel. The figure shows a section of the IO-Lite window populated by three buffers. An IO-Lite buffer always consists of an integral number of (virtually) contiguous VM pages. The pages of an IO-Lite buffer share identical access control attributes: in a given protection domain, either all or none of a buffer's pages are accessible.

Also shown are two buffer aggregates. An aggregate contains an ordered list of tuples of the form <*address, length*>. Each tuple refers to a subrange of memory called a *slice*. A slice is always contained in one IO-Lite buffer, but slices in the same IO-Lite buffer may overlap. The contents of a buffer aggregate can be enumerated by reading the contents of each of its constituent slices in order.

Data objects with the same ACL can be allocated in the same IO-Lite buffer and on the same page. As a result, IO-Lite does not waste memory when allocating objects that are smaller than the VM page size.

2.2.4 IO-Lite and Applications

To take *full* advantage of IO-Lite, application programs use an extended I/O application programming interface (API) that is based on buffer aggregates. This section briefly describes this API. A complete discussion of the API can be found in Appendix A.

IOL_read and IOL_write form the core of the interface (see Figure 2.2). These operations supersede the standard UNIX `read` and `write` operations. (The latter operations are maintained for backward compatibility.) Like their predecessors, the new operations can act on any UNIX file descriptor. All other file descriptor related UNIX systems calls remain unchanged.

The new IOL_read operation returns a buffer aggregate (IOL_Agg) containing at most the amount of data specified as an argument. Unlike the POSIX read, IOL_read may always return less data than requested. The IOL_write operation replaces the data in an external data object with the contents of the buffer aggregate passed as

```
size_t IOL_read(int fd, IOL_Agg **aggr, size_t size);
size_t IOL_write(int fd, IOL_Agg *aggr);
```

Figure 2.2 : IO-Lite I/O API – The IOL_read and IOL_write system calls form the core of the IO-Lite API, and are used by applications to take full advantage of IO-Lite.

an argument.

The effects of IOL_read and IOL_write operations are *atomic* with respect to other IOL_write operations concurrently invoked on the same descriptor. That is, an IOL_read operations yields data that either reflects all or none of the changes resulting from a concurrent IOL_write operation on the same file descriptor. The data returned by an IOL_read is effectively a "snapshot" of the data contained in the object associated with the file descriptor.

Additional IO-Lite system calls allow the creation and deletion of IO-Lite allocation pools. A version of IOL_read allows applications to specify an allocation pool, such that the system places the requested data into IO-Lite buffers from that pool. Applications that manage multiple I/O streams with different access control lists use this operation. The IOL_Agg abstract data type supports a number of operations for creation, destruction, duplication, concatenation and truncation as well as data access.

Language-specific runtime I/O libraries, like the ANSI C *stdio* library, can be converted to use the new API internally. Doing so reduces data copying without changing the library's API. As a result, applications that perform I/O using these standard libraries can enjoy some performance benefits merely by re-linking them with the new library.

2.2.5 IO-Lite and the Filesystem

With IO-Lite, buffer aggregates form the basis of the filesystem cache. The filesystem itself remains unchanged.

File data that originates from a local disk is generally page-aligned and page-sized. However, file data received from the network may not be page-aligned or page-sized, but can nevertheless be kept in the file cache without copying. Conventional UNIX file cache implementations are not suitable for IO-Lite, since they place restrictions on the layout of cached file data. As a result, current UNIX implementations perform a copy when file data arrives from the network.

The IO-Lite file cache has no statically allocated storage. The data resides in IO-Lite buffers, which occupy ordinary pageable virtual memory. Conceptually, the IO-Lite file cache is very simple. It consists of a data structure that maps triples of the form <*file-id, offset, length*> to buffer aggregates that contain the corresponding extent of file data.

Since IO-Lite buffers are immutable, a write operation to a cached file results in the replacement of the corresponding buffers in the cache with the buffers supplied in the write operation. The replaced buffers no longer appear in the file cache. They persist, however, as long as other references to them exist.

For example, assume that an IOL_read operation of a cached file is followed by an IOL_write operation to the same portion of the file. The buffers that were returned in the IOL_read are replaced in the cache as a result of the IOL_write. However, the buffers persist until the process that called IOL_read deallocates them and no other references to the buffers remain. In this way, the snapshot semantics of the IOL_read operation are preserved.

2.2.6 IO-Lite and the Network

With IO-Lite, the network subsystem uses IO-Lite buffer aggregates to store and manipulate network packets.

Some modifications are required to network device drivers. As explained in Section 2.2.3, programs using IO-Lite must determine the ACL of a data object prior to storing the object in memory. Thus, network interface drivers must determine the

I/O stream associated with an incoming packet, since this stream implies the ACL for the data contained in the packet.

To avoid copying, drivers must determine this information from the headers of incoming packets using a packet filter [47], an operation known as *early demultiplexing*. Incidentally, early demultiplexing has been identified by many researchers as a necessary feature for efficiency and quality of service in high-performance networks [61]. With IO-Lite, as with fbufs [27], early demultiplexing is necessary for best performance.

2.2.7 Cache Replacement and Paging

We now discuss the mechanisms and policies for managing the IO-Lite file cache and the physical memory used to support IO-Lite buffers. There are two related issues, namely (1) replacement of file cache entries, and (2) paging of virtual memory pages that contain IO-Lite buffers. Since cached file data resides in IO-Lite buffers, the two issues are closely related.

Cache replacement in a unified caching/buffering system is different from that of a conventional file cache. Cached data is potentially concurrently accessed by applications. Therefore, replacement decisions should take into account both references to a cache entry (i.e., IOL_read and IOL_write operations), as well as virtual memory accesses to the buffers associated with the entry[3].

Moreover, the data in an IO-Lite buffer can be shared in complex ways. For instance, assume that an application reads a data record from file A, appends that record to the same file A, then writes the record to a second file B, and finally transmits the record via a network connection. After this sequence of operations, the buffer containing the record will appear in two different cache entries associated with file A (corresponding to the offset from where the record was read, and the offset at which it was appended), in a cache entry associated with file B, in the network subsystem

[3] Similar issues arise in file caches that are based on memory mapped files.

transmission buffers, and in the user address space of the application. In general, the data in an IO-Lite buffer may at the same time be part of an application data structure, represent buffered data in various OS subsystems, and represent cached portions of several files or different portions of the same file.

Due to the complex sharing relationships, a large design space exists for cache replacement and paging of unified I/O buffers. While we expect that further research is necessary to determine the best policies, our current system employs the following simple strategy. Cache entries are maintained in a list ordered first by current use (i.e., is the data currently referenced by anything other than the cache?), then by time of last access, taking into account read and write operations but not VM accesses for efficiency. When a cache entry needs to be evicted, the least recently used among currently not referenced cache entries is chosen, else the least recently used among the currently referenced entries.

Cache entry eviction is triggered by a simple rule that is evaluated each time a VM page containing cached I/O data is selected for replacement by the VM pageout daemon. If, during the period since the last cache entry eviction, more then half of VM pages selected for replacement were pages containing cached I/O data, then it is assumed that the current file cache is too large, and we evict one cache entry. Because the cache is enlarged (i.e., a new entry is added) on every miss in the file cache, this policy tends to keep the file cache at a size such that about half of all VM page replacements affect file cache pages.

Since all IO-Lite buffers reside in pageable virtual memory, the cache replacement policy only controls how much data the file cache *attempts* to hold. Actual assignment of physical memory is ultimately controlled by the VM system. When the VM pageout daemon selects a IO-Lite buffer page for replacement, IO-Lite writes the page's contents to the appropriate backing store and frees the page.

Due to the complex sharing relationships possible in a unified buffering/caching system, the contents of a page associated with a IO-Lite buffer may have to be written to multiple backing stores. Such backing stores include ordinary paging space, plus one or more files for which the evicted page is holding cached data.

Finally, IO-Lite includes support for application-specific file cache replacement policies. Interested applications can customize the policy using an approach similar to that proposed by Cao et al. [17].

2.2.8 Impact of Immutable I/O buffers

Consider the impact of IO-Lite's immutable I/O buffers on program operation. If a program wishes to modify a data object stored in a buffer aggregate, it must store the new values in a newly-allocated buffer. There are three cases to consider.

First, if every word in the data object is modified, then the only additional cost (over in-place modification) is a buffer allocation. This case arises frequently in programs that perform operations such as compression and encryption. The absence of support for in-place modifications should not significantly affect the performance of such programs.

Second, if only a subset of the words in the object changes value, then the naive approach of copying the entire object would result in partially redundant copying. This copying can be avoided by storing modified values into a new buffer, and logically combining (chaining) the unmodified and modified portions of the data object through the operations provided by the buffer aggregate.

The additional costs in this case (over in-place modification) are due to buffer allocations and chaining (during the modification of the aggregate), and subsequent increased indexing costs (during access of the aggregate) incurred by the non-contiguous storage layout. This case arises in network protocols (fragmentation/reassembly, header addition/removal), and many other programs that reformat/reblock I/O data units. The performance impact on these programs due to the lack of in-place modification is small as long as changes to data objects are reasonably localized.

The third case arises when the modifications of the data object are so widely scattered (leading to a highly fragmented buffer aggregate) that the costs of chaining and indexing exceed the cost of a redundant copy of the entire object into a new, contiguous buffer. This case arises in many scientific applications that read large matrices from input devices and access/modify the data in complex ways. For such applications, contiguous storage and in-place modification is a must. For this purpose, IO-Lite incorporates the *mmap* interface found in all modern UNIX systems. The mmap interface creates a contiguous memory mapping of an I/O object that can be modified in-place.

The use of mmap may require copying in the kernel. First, if the data object is not contiguous and not properly aligned (e.g., incoming network data) a copy operation is necessary due to hardware constraints. In practice, the copy operation is done lazily on a per-page basis. When the first access occurs to a page of a memory mapped file, and its data is not properly aligned, that page is copied.

Second, a copy is needed in the event of a store operation to a memory-mapped file, when the affected page is also referenced through an immutable IO-Lite buffer. (This case arises, for instance, when the file was previously read by some user process using an IOL_read operation). The modified page must be copied in order to maintain the snapshot semantics of the IOL_read operation. The copy is performed lazily, upon the first write access to a page.

2.2.9 Cross-Subsystem Optimizations

A unified buffering/caching system enables certain optimizations across applications and OS subsystems not possible in conventional I/O systems. These optimizations leverage the ability to uniquely identify a particular I/O data object throughout the system.

For example, with IO-Lite, the Internet checksum module used by the TCP and UDP protocols is equipped with an optimization that allows it to cache the Internet checksum computed for each slice of a buffer aggregate. Should the same slice be transmitted again, the cached checksum can be reused, avoiding the expense of a repeated checksum calculation. This optimization works extremely well for network servers that serve documents stored on disk with a high degree of locality. Whenever a file is requested that is still in the IO-Lite file cache, TCP can reuse a precomputed checksum, thereby eliminating the only remaining data-touching operation on the critical I/O path.

To support such optimizations, IO-Lite provides with each buffer a *generation number*. The generation number is incremented every time a buffer is re-allocated. Since IO-Lite buffers are immutable, this generation number, combined with the buffer's address, provides a system-wide unique identifier for the *contents* of the buffer. That is, when a subsystem is presented repeatedly with an IO-Lite buffer with identical address and generation number, it can be sure that the buffer contains the same data values, thus enabling optimizations like Internet checksum caching.

2.2.10 Operation in a Web Server

We start with an overview of the basic operation of a Web server on a conventional UNIX system. A Web server repeatedly accepts TCP connections from clients, reads the client's HTTP request, and transmits the requested content data with an HTTP response header. If the requested content is static, the corresponding document is read from the file system. If the document is not found in the filesystem's cache, a disk read is necessary.

In a traditional UNIX system, copying occurs when data is read from the filesystem, and when the data is written to the socket attached to the client's TCP connection. High-performance Web servers avoid the first copy by using the UNIX mmap interface to read files, but the second copy remains. Multiple buffering occurs because a given document may simultaneously be stored in the file cache and in the TCP retransmission buffers of potentially multiple client connections.

With IO-Lite, all data copying and multiple buffering is eliminated. Once a document is in main memory, it can be served repeatedly by passing buffer aggregates between the file cache, the server application, and the network subsystem. The server obtains a buffer aggregate using the IOL_read operation on the appropriate file descriptor, concatenates a response header, and transmits the resulting aggregate using IOL_write on the TCP socket. If a document is served repeatedly from the file cache, the TCP checksum need not be recalculated except for the buffer containing the response header.

Dynamic content is typically generated by an auxiliary third-party CGI program that runs as a separate process. The data is sent from the CGI process to the server process via a UNIX pipe. In conventional systems, sending data across the pipe involves at least one data copy. In addition, many CGI programs read primary files that they use to synthesize dynamic content from the filesystem, causing more data copying when that data is read. Caching of dynamic content in a CGI program can aggravate the multiple buffering problem: Primary files used to synthesize dynamic content may now be stored in the file cache, in the CGI program's cache as part of a dynamic page, in the Web server's holding buffers, and in the TCP retransmission buffers.

With IO-Lite, sending data over a pipe involves no copying. CGI programs can synthesize dynamic content by manipulating buffer aggregates containing newly-generated data and data from primary files. Again, IO-Lite eliminates all copying and multiple buffering, even in the presence of caching CGI programs. TCP checksums need not be recomputed for portions of dynamically generated content that are repeatedly transmitted.

IO-Lite's ability to eliminate data copying and multiple buffering can dramatically reduce the cost of serving static and dynamic content. The impact is particularly strong in the case when a cached copy (static or dynamic) of the requested content exists, since copying costs can dominate the service time in this case. Moreover, the elimination of multiple buffering frees up valuable memory resources, permitting a larger file cache size and hit rate, thus further increasing server performance.

Web servers use IO-Lite's access control model in a straightforward manner. The various access permissions in a Web server stem from the sources of the data: the filesystem for static files, the CGI applications for dynamic data, and the server process itself for internally-generated data (response headers, redirect responses, etc). Mapping these permissions to the IO-Lite model is trivial – the server process and every CGI application instance have separate buffer pools with different ACLs. When the server process reads a buffer aggregate, either from the filesystem or a CGI process, IO-Lite makes the underlying buffers readable in the server process. When this data is sent by the server to the client, the network subsystem has access to the pages by virtue of being part of the kernel.

Finally, a Web server can use the IO-Lite facilities to customize the replacement policy used in the file cache to derive further performance benefits. To use IO-Lite, an existing Web server need only be modified to use the IO-Lite API. CGI programs must likewise use buffer aggregates to synthesize dynamic content.

2.3 Implementation

IO-Lite is implemented as a loadable kernel module that can be dynamically linked to a slightly modified FreeBSD 2.2.6 kernel. A runtime library must be linked with applications wishing to use the IO-Lite API. This library provides the buffer aggregate manipulation routines and stubs for the IO-Lite system calls.

Network Subsystem: The BSD network subsystem was adapted by encapsulating IO-Lite buffers inside the BSD native buffer abstraction, mbufs. This approach avoids intrusive and widespread source code modifications.

The encapsulation was accomplished by using the mbuf out-of-line pointer to refer to an IO-Lite buffer, thus maintaining compatibility with the BSD network subsystem in a very simple, efficient manner. Small data items such as network packet headers are still stored inline in mbufs, but the performance critical bulk data resides in IO-

Lite buffers. Since the mbuf data structure remains essentially unmodified, the bulk of the network subsystem (including all network protocols) works unmodified with mbuf-encapsulated IO-Lite buffers.

Filesystem: The IO-Lite file cache module replaces the unified buffer cache module found in 4.4BSD derived systems [48]. The bulk of the filesystem code (below the block-oriented file read/write interface) remains unmodified. As in the original BSD kernel, the filesystem continues to use the "old" buffer cache to hold filesystem metadata.

The original UNIX read and write system calls for files are implemented by IO-Lite for backward compatibility; a data copy operation is used to move data between application buffers and IO-Lite buffers.

VM System: Adding IO-Lite does not require any significant changes to the BSD VM system [48]. IO-Lite uses standard interfaces exported by the VM system to create a VM object that represents the IO-Lite window, map that object into kernel and user process address spaces, and to provide page-in and page-out handlers for the IO-Lite buffers.

The page-in and page-out handlers use information maintained by the IO-Lite file cache module to determine the disk locations that provide backing store for a given IO-Lite buffer page. The replacement policy for IO-Lite buffers and the IO-Lite file cache is implemented by the page-out handler, in cooperation with the IO-Lite file cache module.

IPC System: The IO-Lite system adds a modified implementation of the BSD IPC facilities. This implementation is used whenever a process uses the IO-Lite read/write operations on a BSD pipe or UNIX domain socket. If the processes on both ends of a pipe or UNIX domain socket-pair use the IO-Lite API, then the data transfer proceeds copy-free by passing the associated IO-Lite buffers by reference. The IO-Lite system ensures that all pages occupied by these IO-Lite buffers are readable in the receiving domain, using standard VM operations.

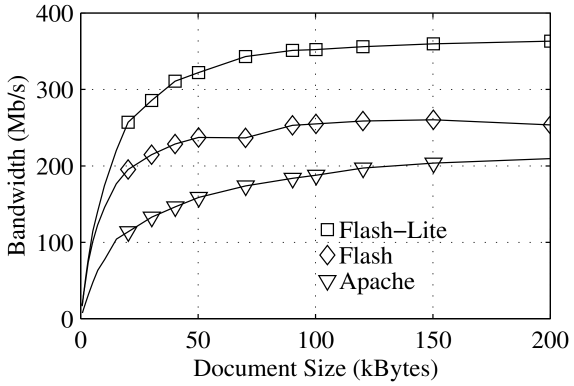

Figure 2.3 : HTTP Single File Test – All clients request the same file from the server, and we observe the aggregate bandwidth generated. This test provides the best-case performance of the servers using nonpersistent connections.

Access Control: To reduce the number of operations and the amount of book-keeping needed by the VM system, IO-Lite performs all access control over groups of pages called chunks. Chunks are fixed-sized regions of virtual memory (currently set to 64 kBytes) that share the same access permissions. When a process requests a new IO-Lite buffer, it is allocated from a chunk with the appropriate ACL. If no available chunk exists, a new chunk is allocated and made writeable in the process's address space. When a process sends a buffer aggregate to another process, IO-Lite makes all of the underlying chunks readable in the receiver's address space.

2.4 Performance

For our experiments, we use a server system with a 333MHz Pentium II PC, 128MB of main memory and five network adaptors connected to a switched 100Mbps Fast Ethernet.

To fully expose the performance bottlenecks in the operating system, we use a high-performance in-house Web server called *Flash* [55]. Flash is an event-driven HTTP server with support for CGI. To the best of our knowledge, Flash is among the fastest HTTP servers currently available. *Flash-Lite* is a slightly modified version

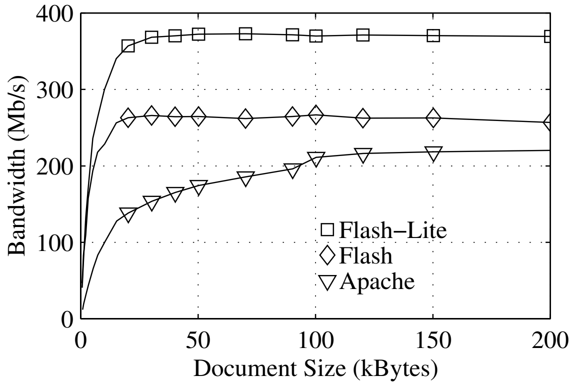

Figure 2.4 : Persistent HTTP Single File Test – Rather than creating a new TCP connection for each transfer, each client requests multiple transfers on an existing connection. Removing the TCP setup/teardown overhead allows even small transfers to achieve significant benefit.

of Flash that uses the IO-Lite API. Flash is an aggressively optimized, experimental Web server; it reflects the best in Web server performance that can be achieved using the standard facilities available in a modern operating system. Flash-Lite's performance reflects the additional benefits that result from IO-Lite.

While Flash uses memory-mapped files to read disk data, Flash-Lite uses the IO-Lite read/write interface to access disk files. In addition, Flash-Lite uses the IO-Lite support for customization of the file caching policy to implement Greedy Dual Size (GDS), a policy that performs well on Web workloads [18]. The modifications necessary for Flash to use IO-Lite were straightforward and simple. Calls to `mmap` to map data files were replaced with calls to `IOL_read`. Allocating memory for response headers, done using `malloc` in Flash, is handled with memory allocation from IO-Lite space. Finally, the gathering/sending of data to the client (via `writev` in Flash) is accomplished with `IOL_write`.

For comparison, we also present performance results with Apache version 1.3.1, a widely used Web server [5]. This version uses mmap to read files and performs substantially better than earlier versions. Apache's performance reflects what can be expected of a widely used Web server today.

All Web servers were configured to use a TCP socket send buffer size of 64KBytes. Access logging was disabled to ensure fairness to all servers. Logging accesses drops Apache's performance by 13-16% on these tests, but only drops Flash/Flash-Lite's performance by 3-5%.

2.4.1 Nonpersistent Connections

In the first experiment, 40 HTTP clients running on five machines repeatedly request the same document of a given size from the server. A client issues a new request as soon as a response is received for the previous request [9]. The file size requested varies from 500 bytes to 200KBytes (the data points below 20KB are 500 bytes, 1KB, 2KB, 3KB, 5KB, 7KB, 10KB and 15 KB). In all cases, the files are cached in the server's file cache after the first request, so no physical disk I/O occurs in the common case.

Figure 2.3 shows the output bandwidth of the various Web servers as a function of request file size. Results are shown for Flash-Lite, Flash and Apache. Flash performs consistently better than Apache, with bandwidth improvements up to 71% at a file size of 20KBytes. This result confirms that our aggressive Flash server outperforms the already fast Apache server.

For files 50KBytes and larger, Flash using IO-Lite (Flash-Lite) delivers a bandwidth increase of 38-43% over Flash and 73-94% over Apache. For file sizes of 5KBytes or less, Flash and Flash-Lite perform equally well. The reason is that at these small sizes, control overheads, rather than data dependent costs, dominate the cost of serving a request.

The throughput advantage obtained with IO-Lite in this experiment reflects only the savings due to copy-avoidance and checksum caching. Potential benefits resulting from the elimination of multiple buffering and the customized file cache replacement are not realized, because this experiment does not stress the file cache (i.e., a single document is repeatedly requested).

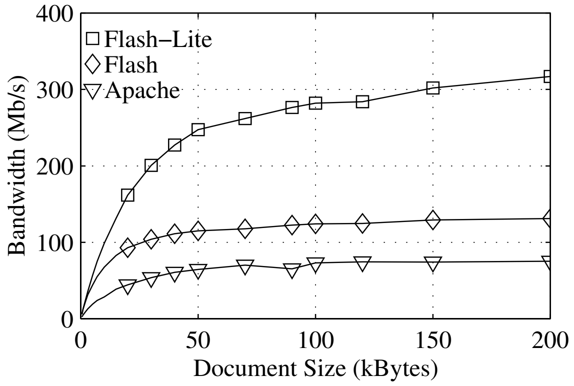

Figure 2.5 : HTTP/FastCGI – Each client requests data from a persistent CGI application spawned by the server. In standard UNIX, the extra copying between the server and the CGI application becomes a significant performance bottleneck.

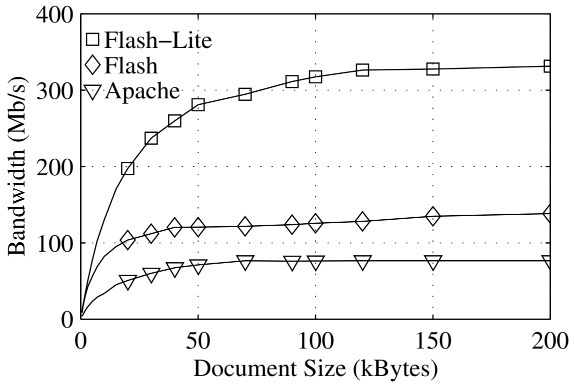

Figure 2.6 : Persistent-HTTP/FastCGI – Each client reuses the TCP connection for multiple CGI requests. Flash and Apache do not receive significant benefits because their performance is limited by the copying between the server and CGI application.

2.4.2 Persistent Connections

The previous experiments are based on HTTP 1.0, where a TCP connection is established by clients for each individual request. The HTTP 1.1 specification adds support for persistent (keep-alive) connections that can be used by clients to issue multiple requests in sequence. We modified both versions of Flash to support persistent connections and repeated the previous experiment. The results are shown in

Figure 2.4.

With persistent connections, the request rate for small files (less than 50KBytes) increases significantly with Flash and Flash-Lite, due to the reduced overhead associated with TCP connection establishment and termination. The overheads of the process-per-connection model in Apache appear to prevent that server from fully taking advantage of this effect.

Persistent connections allow Flash-Lite to realize its full performance advantage over Flash at smaller file sizes. For files of 20KBytes and above, Flash-Lite outperforms Flash by up to 43%. Moreover, Flash-Lite comes within 10% of saturating the network at a file size of only 17KBytes and it saturates the network for file sizes of 30KBytes and above.

2.4.3 CGI Programs

An area where IO-Lite promises particularly substantial benefits is CGI programs. When compared to the original CGI 1.1 standard [1], the newer FastCGI interface [2] amortizes the cost of forking and starting a CGI process by allowing such processes to persist across requests. However, there are still substantial overheads associated with IPC across pipes and multiple buffering, as explained in Section 2.2.10.

We performed an experiment to evaluate how IO-Lite affects the performance of dynamic content generation using FastCGI programs. A test CGI program, when receiving a request, sends a "dynamic" document of a given size from its memory to the Web server process via a UNIX pipe; the server transmits the data on the client's connection. The results of these experiments are shown in Figure 2.5.

The bandwidth of the Flash and Apache servers is roughly half their corresponding bandwidth on static documents. This result shows the strong impact of the copy-based pipe IPC in regular UNIX on CGI performance. With Flash-Lite, the performance is significantly better, approaching 87% of the speed on static content. Also interesting is that CGI programs with Flash-Lite achieve performance better

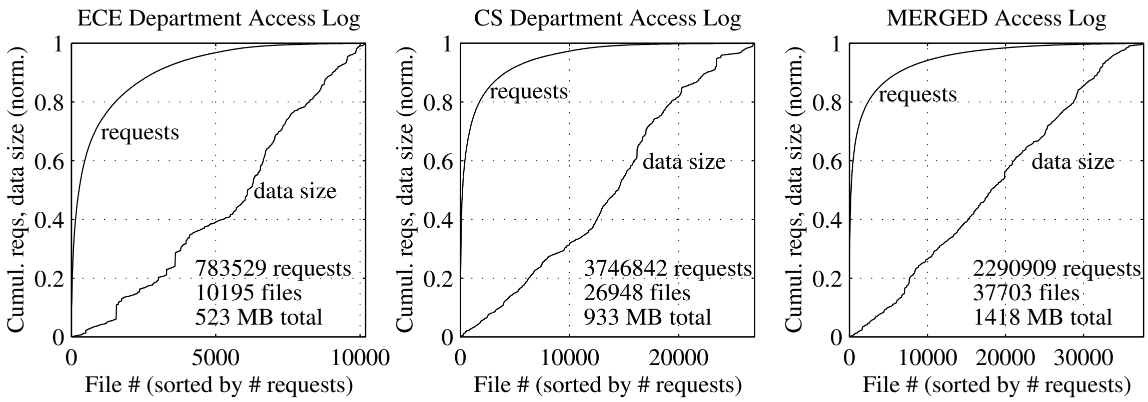

Figure 2.7 : Trace characteristics – These graphs show the cumulative distribution functions for the data size and request frequencies of the three traces used in our experiments. For example, the 5000 most heavily requested files in the ECE access constituted 39% of the total static data size (523 MB) and 95% of all requests.

than static files with Flash.

Figure 2.6 shows results of the same experiment using persistent HTTP-1.1 connections. Unlike Flash-Lite, Flash and Apache cannot take advantage of the efficiency of persistent connections here, since their performance is limited by the pipe IPC.

The results of these experiments show that IO-Lite allows a server to efficiently support dynamic content using CGI programs, without giving up fault isolation and protection from such third-party programs. This result suggests that with IO-Lite, there may be less reason to resort to library-based interfaces for dynamic content generation. Such interfaces were defined by Netscape and Microsoft [38, 52] to avoid the overhead of CGI. Since they require third-party programs to be linked with the server, they give up fault isolation and protection.

2.4.4 Trace-based Evaluation

To measure the overall impact of IO-Lite on the performance of a Web server under more realistic workload conditions, we performed experiments where our experimental server is driven by workloads derived from server logs of actual Web servers. We use logs from various Web servers from Rice University, and extract only the requests for static documents.

For these tests, we use access logs from the Electrical and Computer Engineering department, the Computer Science department, and a combined log from seven Web servers located across the University. We will refer to these traces as ECE, CS, and MERGED, respectively. The MERGED access log represents the access patterns for a hypothetical single Web server hosting all content for the Rice University campus. The average request size in these traces is 23 kBytes for ECE, 20 kBytes for CS, and 17 kBytes for MERGED. The other characteristics of these access logs are shown in Figure 2.7.

Our first test is designed to measure the overall behavior of the servers on various workloads. In this experiment, 64 clients replay requests from the access logs against the server machine. The clients share the access log, and as each request finishes, the client issues the next unsent request from the log. Since we are interested in testing the maximum performance of the server, the clients issue requests immediately after earlier requests complete.

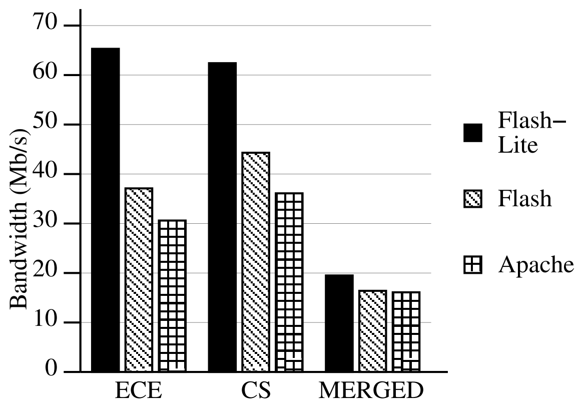

Figure 2.8 : Overall trace performance – In each test, 64 clients were used to replay the entries of the trace. New requests were started immediately after previous requests completed.

Figure 2.8 shows the overall performance of the various servers on our traces. The performance differences between these tests and the single-file test in Section 2.4.1 stem from the nature of the workloads presented to the servers. In the single-file tests, no cache misses or disk activity occur once the file has been brought into memory. In contrast, these traces involve a large number of files, cover large data set sizes, and generate significant disk activity. The combination of these factors reduces the performance of all of the servers. Server performance on these tests is influenced by a variety of factors, including average request size, total data set size, and request locality. Flash-Lite significantly outperforms Flash and Apache on the ECE and CS traces. However, the MERGED trace has a large working set and poor locality, so all of the servers remain disk-bound.

2.4.5 Subtrace Experiments

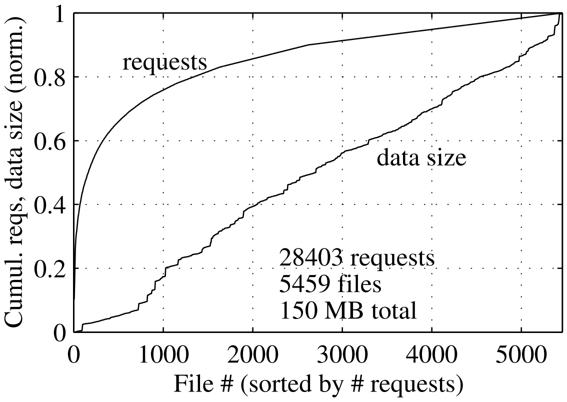

Figure 2.9 : 150MB subtrace characteristics – This graph presents the cumulative distribution functions for the data size and request frequencies of the subtrace we used. For example, we see that the 1000 most frequently requested files were responsible for 20% of the total static data size but 74% of all requests.

Replaying full traces provides useful performance data about the relative behavior of the three servers on workloads derived from real servers' access logs. To obtain more detailed information about server behavior over a wider range of workloads, we experiment with varying the request stream sent to servers. We use a portion of the

MERGED access log that corresponds to a 150MB data set size, and then use prefixes of it to generate input streams with smaller data set sizes. The characteristics of the 150MB subtrace are shown in Figure 2.9.

By using the subtraces as our request workload, our experiment evaluates server performance over a range of dataset sizes (and therefore working set sizes). Employing methodology similar to the SpecWeb [3] benchmark, the clients randomly pick entries from the subtraces to generate requests. Four client machines with 16 clients each are used to generate the workload. Each client issues one request at a time and immediately issues a new request when the previous request finishes. Each data point represents the average aggregate bandwidth generated during a one hour run.

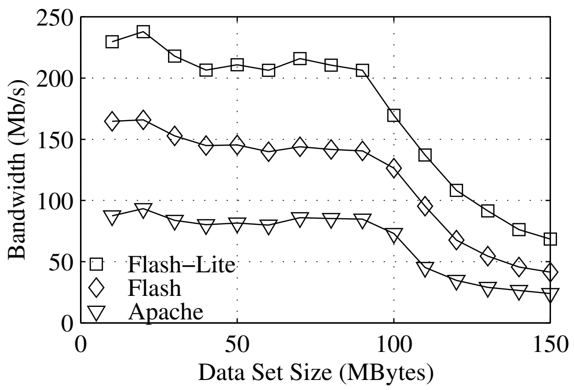

Figure 2.10 : MERGED subtrace performance – This graph shows the aggregate bandwidth generated by 64 clients as a function of data set size. Flash outperforms Apache due to aggressive caching, while Flash-Lite outperforms Flash due to a combination of copy avoidance, checksum caching, and customized file cache replacement.

Figure 2.10 shows the performance in Mb/sec of Flash-Lite, Flash, and Apache on the MERGED subtrace with various data set sizes. For this trace, Flash exceeds the throughput of Apache by 65-88% on in-memory workloads and by 71-110% on disk-bound workloads. Compared to Flash, Flash-Lite's copy avoidance gains an additional 34-50% for in-memory workloads, while its cache replacement policies generate a 44-67% gain on disk-bound workloads.

2.4.6 Optimization Contributions

Flash-Lite's performance gains over Flash stem from a combination of four factors: copy elimination, double-buffering elimination, checksum caching, and a customized cache replacement policy. To quantify the effects of each of these contributions, we performed a set of tests with different versions of Flash-Lite and IO-Lite. Flash-Lite was run both with its standard cache replacement policy (GDS), and with a more traditional least-recently-used (LRU) cache replacement. Likewise, IO-Lite was run with and without the checksum cache enabled. These additional tests were run in the configuration described in Section 2.4.4.

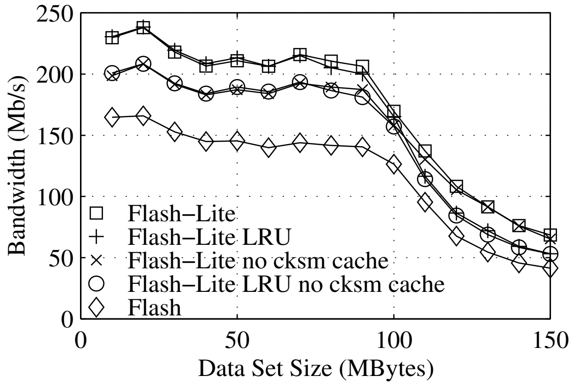

Figure 2.11 : Optimization contributions – To quantify the effects of the various optimizations present in Flash-Lite and IO-Lite, two file cache replacement policies are tested in Flash-Lite, while IO-Lite is run with and without checksum caching.

The results of these additional tests are shown in Figure 2.11, with the results for Flash-Lite and Flash included for comparison. The benefit from copy elimination alone ranges from 21-33% and can be determined by comparing the in-memory performance of Flash with Flash-Lite running on a version of IO-Lite without checksum caching. Checksum caching yields an additional 10-15% benefit for these cases. Using the GDS cache replacement policy provides a 17-28% benefit over LRU on disk-heavy workloads, as indicated by comparing Flash-Lite to Flash-Lite-LRU.

One of the other benefits of IO-Lite is the extra memory saved by eliminating double-buffering. However, in this experiment, the fast LAN and the relatively small client population results in less than two megabytes of memory being devoted to network buffers. As such, the fact that IO-Lite eliminates double-buffering is not evident in this test. With more clients in a wide area network, the effects of multiple buffering become much more significant, as shown in the next section.

2.4.7 WAN Effects

Our experimental testbed uses a local-area network to connect a relatively small number of clients to the experimental server. This setup leaves a significant aspect of real Web server performance unevaluated, namely the impact of wide-area network delays and large numbers of clients [9]. In particular, we are interested here in the TCP retransmission buffers needed to support efficient communication on connections with substantial bandwidth-delay products.

Since both Apache and Flash use mmap to read files, the remaining primary source of double buffering is TCP's transmission buffers. The amount of memory consumed by these buffers is related to the number of concurrent connections handled by the server, times the socket send buffer size T_{ss} used by the server. For good network performance, T_{ss} must be large enough to accommodate a connection's bandwidth-delay product. A typical setting for T_{ss} in a server today is 64KBytes.

Busy servers may handle several hundred concurrent connections, resulting in significant memory requirements even in the current Internet. With future increases in Internet bandwidth, the necessary T_{ss} settings needed for good network performance are likely to increase, making double buffering elimination increasingly important.

With IO-Lite, however, socket send buffers do not require separate memory since they refer to data stored in IO-Lite buffers[4]. Double buffering is eliminated, and the amount of memory available for the file cache remains independent of the number of concurrent clients contacting the server and the setting of T_{ss}.

[4] A small amount of memory is required to hold mbuf structures.

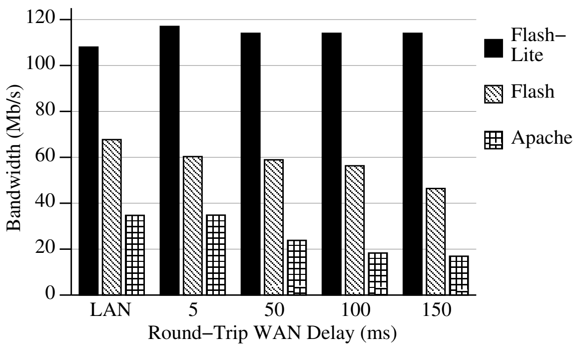

Figure 2.12 : Throughput vs. network delay – In a conventional UNIX system, as the network delay increases, more network buffer space is allocated, reducing the memory available to the filesystem cache. The throughput of Flash and Apache drop due to this effect. IO-Lite avoids multiple buffering, so Flash-Lite is not affected.

To quantify the impact of the memory consumed by the transmission buffers, we configure our test environment to resemble a wide-area network. We interpose a "delay router" between each client machine and the server. Using these delay routers, we can configure the network delay for all data exchanged between the clients and the server. In wide area networks, the extra networking delay increases the time necessary to transmit documents. As a result, the number of simultaneous connections seen by a server increases with the network delay. To keep the server saturated, we linearly scale the number of clients with the network delay, from 64 clients in the LAN (no delay) case to a maximum of 900 clients for the 150ms delay test. We run the tests with a dataset size of 120MB, which is neither entirely disk-bound or CPU-limited.

Figure 2.12 shows the performance of Flash-Lite, Flash, and Apache as a function of network delay. The performance of Flash and Apache drop as the network delay increases. They are affected by the network subsystem dynamically allocating more space as the network delay increases. When this occurs, the memory available to the filesystem cache decreases, and the cache miss rate increases. The 50% drop in Apache is higher than the 33% drop for Flash because Apache also loses extra memory by using a separate server process per simultaneous connection. In contrast, Flash-Lite's

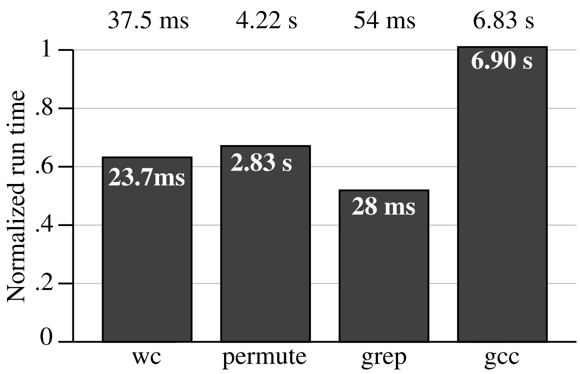

Figure 2.13 : Various application runtimes – The time above each bar indicates the run time of the unmodified application, while the time at the top of the bar is for the application using IO-Lite.

performance actually increases slightly in these tests. It does not suffer the effects of multiple buffering, so only a small amount of additional control overhead is added as the network delay increases. However, the larger client population increases the available parallelism, slightly increasing Flash-Lite's performance.

2.4.8 Other Applications

To demonstrate the impact of IO-Lite on the performance of a wider range of applications, and also to gain experience with the use of the IO-Lite API, a number of existing UNIX programs were converted to use IO-Lite. We modified GNU grep, wc, cat, and the GNU gcc compiler chain (compiler driver, C preprocessor, C compiler, and assembler)[5].

For these programs, the modifications necessary to use IO-Lite were minimal. The cat program was the simplest, since it does not process the data it handles.

[5]Our application performance results presented here differ from our earlier results[56]. Those results were from a prototype of IO-Lite running on Digital UNIX 3.2C on a 233MHz Alphastation 200. The performance differences stem from a combination of more advanced hardware and a different operating system

Its modifications consisted of replacing the UNIX read/write calls with their IO-Lite equivalents. The wc program makes a single pass over the data, examining one character at a time. Converting it involved replacing UNIX read with IOL_read and iterating through the slices returned in the buffer aggregate. Modifying grep was slightly more involved, since it operates in a line-oriented manner. Again, the UNIX read call was replaced with IOL_read. However, since grep expects all data in a line to be contiguous in memory, lines that were split across IO-Lite buffers were copied into dynamically-allocated contiguous memory.

For gcc, rather than modify the entire program, we simply replaced the C *stdio* library with a version that uses IO-Lite for communication over pipes. The C preprocessor's output, the compiler's input and output, and the assembler's input all use the C *stdio* library, and were converted merely by relinking them with an IO-Lite version of the *stdio* library.

Figure 2.13 depicts the results obtained with wc, permute, grep and gcc. The "wc" bar refers to a run of the word-count program on a 1.75 MB file. The file is in the file cache, so no physical I/O occurs. "Permute" generates all possible permutations of 4 character words in a 40 character string. Its output ($10! * 40 = 145152000$ bytes) is piped into the wc program. The "grep" bar refers to a run of the GNU grep program on the same file used for the wc program, but the file is piped into grep from cat instead of being read directly from disk. The "gcc" bar refers to compilation of a set of 27 files (167 KBytes total).

Using IO-Lite in the wc example reduces execution time by 37% since it reads cached files. All data copies between the filesystem cache and the application are eliminated. The remaining overhead in the IO-Lite case is due to page mapping. Each page of the cached file must be mapped into the application's address space when a file is read from the IO-Lite file cache.

The permute program involves producer/consumer communication over a pipe. When this occurs, IO-Lite can recycle the buffers used for interprocess communication. Not only does IO-Lite eliminate data copying between the processes, but it also avoids the VM map operations affecting the wc example. Using IO-Lite in this example reduces execution time by 33%, comparable to that of the wc test. The permute program is more computationally intensive than wc, so the largest source of remaining overhead in this test is the computation itself.

The most significant gain in these tests is for the grep case. Here, IO-Lite is able to eliminate three copies – two due to cat, and one due to grep. As a result, the performance of this test improves by 48%. This gain is larger than the gain in the wc and permute tests, since more data copies are eliminated.

The gcc compiler chain was converted to determine if there were benefits from IO-Lite for more compute-bound applications and to stress the IO-Lite implementation. We observe no performance benefit in this test for two reasons: (1) the computation time dominates the cost of communication, and (2) only the interprocess data copying has been eliminated, but data copying between the applications and the stdio library still exists.

2.5 Related Work

To provide a basis for comparison with related work, we examine how existing and proposed I/O systems affect the design and performance of a Web server. We begin with the standard UNIX (POSIX) I/O interface, and go on to more aggressively optimized I/O systems proposed in the literature.

POSIX I/O: The UNIX/POSIX read/readv operations allow an application to request the placement of input data at an arbitrary (set of) location(s) in its private address space. Furthermore, both the read/readv and write/writev operations have copy semantics, implying that applications can modify data that was read/written from/to an external data object without affecting that data object.

To avoid the copying associated with reading a file repeatedly from the filesystem, a Web server using this interface would have to maintain a user-level cache of Web documents, leading to double-buffering in the disk cache and the server. When serving a request, data is copied into socket buffers, creating a third copy. CGI programs [1] cause data to be additionally copied from the CGI program into the server's buffers via a pipe, possibly involving kernel buffers.

Memory-mapped files: The semantics of mmap facilitate a copy-free implementation, but the contiguous mapping requirement may still demand copying in the OS for data that arrives from the network. Like IO-Lite, mmap avoids multiple buffering of file data in the file cache and the application(s). Unlike IO-Lite, mmap does not generalize to network I/O, so double buffering (and copying) still occurs in the network subsystem.

Moreover, memory-mapped files do not provide a convenient method for implementing CGI support, since they lack support for producer/consumer synchronization between the CGI program and the server. Having the server and the CGI program share memory-mapped files for IPC requires ad-hoc synchronization and adds complexity.

Transparent Copy Avoidance: In principle, copy avoidance and single buffering could be accomplished transparently using existing POSIX APIs, through the use of page remapping and copy-on-write. Well-known difficulties with this approach are VM page alignment problems, and potential writes to buffers by applications, which may defeat copy avoidance by causing copy-on-write faults.

The Genie system [14, 15, 16] addresses the alignment problem and allows transparent copy-free network access under certain conditions. It also introduces an asymmetric interface for copy-free IPC between a client and a server process. Under appropriate conditions, Genie provides copy-free data transfer between network sockets and memory-mapped files.

The benefit of Genie's approach is that some applications potentially gain performance without any source-level changes. However, it is not clear how many applications will actually meet the conditions necessary for transparent copy avoidance. Applications requiring copy avoidance and consistent performance must ensure proper alignment of incoming network data, use buffers carefully to avoid copy-on-write faults, and use special system calls to move data into memory-mapped files.

For instance, Web server applications must be modified in order to obtain Genie's full benefits. The server application must use memory-mapped files, satisfy other conditions necessary to avoid copying, and use new interfaces for all interaction with CGI applications. The CGI applications have three options: remain unmodified and trust the server process not to view private data, page-align and pad all data to be sent to the server to ensure that private data is not viewable, or resort to copying interfaces.

Copy Avoidance with Handoff Semantics: The *Container Shipping* (CS) I/O system [57], Thadani and Khalidi's work [62], and the UVM Virtual Memory System [22] use I/O read and write operations with handoff (move) semantics. Like IO-Lite, these systems require applications to process I/O data at a given location. Unlike IO-Lite, they allow applications to modify I/O buffers in-place. This is safe because the handoff semantics permit only sequential sharing of I/O data buffers—i.e., only one protection domain has access to a given buffer at any time.

Sacrificing concurrent sharing comes at a cost: Since applications lose access to buffers used in write operations, explicit physical copies are necessary if the applications need access to the data after the write. Moreover, when an application reads from a file while a second application is holding cached buffers for the same file, a second copy of the data must be read from the input device. The lack of support for concurrent sharing prevents effectively integrating a copy-free I/O buffering scheme with the file cache.

In a Web server, lack of concurrent sharing requires copying of "hot" pages, making the common case more expensive. CGI programs that produce entirely new data for every request (as opposed to returning part of a file or a set of files) are not affected, but CGI programs that try to intelligently cache data suffer copying costs.

Fbufs: Fbufs is a copy-free cross-domain transfer and buffering mechanism for I/O data, based on immutable buffers that can be concurrently shared. The fbufs system was designed primarily for handling network streams, was implemented in a non-UNIX environment, and does not support filesystem access or a file cache. IO-Lite's cross-domain transfer mechanism was inspired by fbufs. When trying to use fbufs in a Web server, the lack of integration with the filesystem would result in double-buffering. Their use as an interprocess communication facility would benefit CGI programs, but with the same restrictions on filesystem access.

Extensible Kernels: Recent work has proposed the use of of *extensible* kernels [12, 28, 40, 60] to address a variety of problems associated with existing operating systems. Extensible kernels can potentially address many different OS performance problems, not just the I/O bottleneck that is the focus of our work.

The flexibility of extensible kernels allows them to address issues outside of the scope of copy-free systems, such as the setup costs associated with data transfer. For example, the Cheetah Web server [40] in the Exokernel project optimizes connection state maintenance, providing significant benefits for small transfers on a LAN. Performance on large files should be similar for Flash-Lite and Cheetah, since IO-Lite provides the same copy avoidance and checksum caching optimizations.

The drawbacks of extensible kernels stem from the integration between operating system and application functions. In order to gain benefits, server/application writers must implement OS-specific kernel extensions or depend on a third party to provide an OS library for this purpose. These approaches are not directly applicable to existing general-purpose operating systems, and they do not provide an application-independent scheme for addressing the I/O bottleneck. Moreover, these approaches require new safety provisions, adding complexity and overhead.

In particular, CGI programs may pose problems for extensible kernel-based Web servers, since some protection mechanism must be used to insulate the server from poorly-behaved programs. Conventional Web servers and Flash-Lite rely on the operating system to provide protection between the CGI process and the server, and the server does not extend any trust to the CGI process. As a result, the malicious or inadvertent failure of a CGI program will not affect the server.

Monolithic System Calls: Due to the popularity of static content in Web traffic, a number of systems (including Windows NT, AIX, Linux, and later versions of FreeBSD) have included a new system call to optimize the process of handling static documents. The system calls (generally called sendfile or transmitfile) take as parameters the network socket to the client, the file to be sent, and a response header to prepend to the file. These techniques are similar to earlier work done on splicing data streams [29].

The benefit of this approach is that it provides a very simple interface to the programmer. The drawback is the lack of extensibility, especially with respect to dynamic documents. Additionally, some internal mechanism (copy-on-write, exclusive locks) must still be used to ensure applications cannot modify file data that is in transit.

To summarize, IO-Lite differs from existing work in its generality, its integration of the file cache, its support for cross-subsystem optimizations, and its direct applicability to general-purpose operating systems. IO-Lite is a general I/O buffering and caching system that avoids all redundant copying and multiple buffering of I/O data, even on complex data paths that involve the file cache, interprocess communication facilities, network subsystem and multiple application processes.

2.6 Conclusion

This chapter presents the design, implementation, and evaluation of IO-Lite, a unified buffering and caching system for general-purpose operating systems. IO-Lite improves the performance of servers and other I/O-intensive applications by eliminating all redundant copying and multiple buffering of I/O data, and by enabling optimizations across subsystems.

Experimental results from a prototype implementation in FreeBSD show performance improvements between 40 and 80% over an already aggressively optimized Web server without IO-Lite, both on synthetic workloads and on real workloads derived from Web server logs. IO-Lite also allows the efficient support of CGI programs without loss of fault isolation and protection. Further results show that IO-Lite reduces memory requirements associated with the support of large numbers of client connections and large bandwidth-delay products in Web servers by eliminating multiple buffering, leading to increased throughput.

Chapter 3

LARD

Network servers based on clusters of commodity workstations or PCs connected by high-speed LANs combine cutting-edge performance and low cost. A cluster-based network server consists of a front-end, responsible for request distribution, and a number of back-end nodes, responsible for request processing. The use of a front-end makes the distributed nature of the server transparent to the clients. In most current cluster servers the front-end distributes requests to back-end nodes without regard to the type of service or the content requested. That is, all back-end nodes are considered equally capable of serving a given request and the only factor guiding the request distribution is the current load of the back-end nodes.

With *content-based request distribution*, the front-end takes into account both the service/content requested and the current load on the back-end nodes when deciding which back-end node should serve a given request. The potential advantages of content-based request distribution are: (1) increased performance due to improved hit rates in the back-end's main memory caches, (2) increased secondary storage scalability due to the ability to partition the server's database over the different back-end nodes, and (3) the ability to employ back-end nodes that are specialized for certain types of requests (e.g., audio and video).

The *locality-aware request distribution* (LARD) strategy presented in this chapter is a form of content-based request distribution, focusing on obtaining the first of the advantages cited above, namely improved cache hit rates in the back-ends. Secondary storage scalability and special-purpose back-end nodes are not discussed any further in this chapter.

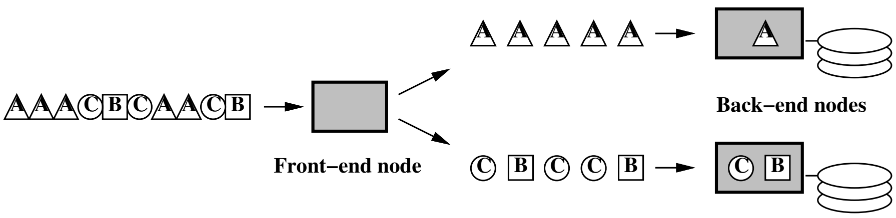

Figure 3.1 : Locality-Aware Request Distribution (LARD) - In this scheme, the front-end node assigns incoming requests to back-end nodes in a manner that increases the cache effectiveness of the back-end nodes.

Figure 3.1 illustrates the principle of LARD in a simple server with two back-ends and three targets[1] (A,B,C) in the incoming request stream. The front-end directs all requests for A to back-end 1, and all requests for B and C to back-end 2. By doing so, there is an increased likelihood that the request finds the requested target in the cache at the back-end. In contrast, with a round-robin distribution of incoming requests, requests of all three targets will arrive at both back-ends. This increases the likelihood of a cache miss, if the sum of the sizes of the three targets, or, more generally, if the size of the working set exceeds the size of the main memory cache at an individual back-end node.

Of course, by naively distributing incoming requests in a content-based manner as suggested in Figure 3.1, the load between different back-ends might become unbalanced, resulting in worse performance. The first major challenge in building a LARD cluster is therefore to design a practical and efficient strategy that *simultaneously* achieves load balancing and high cache hit rates on the back-ends. The second challenge stems from the need for a protocol that allows the front-end to hand off an established client connection to a back-end node, in a manner that is transparent to clients and is efficient enough not to render the front-end a bottleneck. This re-

[1]In the following discussion, the term *target* is being used to refer to a specific object requested from a server. For an HTTP server, for instance, a target is specified by a URL and any applicable arguments to the HTTP `GET` command.

quirement results from the front-end's need to inspect the target content of a request *prior* to assigning the request to a back-end node. This work demonstrates that these challenges can be met, and that LARD produces substantially higher throughput than the state-of-the-art approaches where request distribution is solely based on load balancing, for workloads whose working set exceeds the size of the individual node caches.

Increasing a server's cache effectiveness is an important step towards meeting the demands placed on current and future network servers. Being able to cache the working set is critical to achieving high throughput, as a state-of-the-art disk device can deliver no more than 120 block requests/sec, while high-end network servers will be expected to serve thousands of document requests per second. Moreover, typical working set sizes of web servers can be expected to grow over time, for two reasons. First, the amount of content made available by a single organization is typically growing over time. Second, there is a trend towards centralization of web servers within organizations. Issues such as cost and ease of administration, availability, security, and high-capacity backbone network access cause organizations to move towards large, centralized network servers that handle all of the organization's web presence. Such servers have to handle the combined working sets of all the servers they supersede.

With round-robin distribution, a cluster does not scale well to larger working sets, as *each* node's main memory cache has to fit the entire working set. With LARD, the effective cache size approaches the *sum* of the node cache sizes. Thus, adding nodes to a cluster can accommodate both increased traffic (due to additional CPU power) and larger working sets (due to the increased effective cache size).

This work presents the following contributions:

1. a practical and efficient LARD strategy that achieves high cache hit rates and good load balancing,

2. a trace-driven simulation that demonstrates the performance potential of locality-aware request distribution,

3. an efficient *TCP handoff protocol*, that enables content-based request distribution by providing client-transparent connection handoff for TCP-based network services, and

4. a performance evaluation of a prototype LARD server cluster, incorporating the TCP handoff protocol and the LARD strategy.

The outline of the rest of this chapter is as follows: In Section 3.1 we develop our strategy for locality-aware request distribution. In Section 3.2 we describe the model used to simulate the performance of LARD in comparison to other request distribution strategies. In Section 3.3 we present the results of the simulation. In Section 3.4 we move on to the practical implementation of LARD, particularly the TCP handoff protocol. We describe the experimental environment in which our LARD server is implemented and its measured performance in Section 3.5. We describe related work in Section 3.6 and we conclude in Section 3.7.

3.1 Strategies for Request Distribution

3.1.1 Assumptions

The following assumptions hold for all request distribution strategies considered in this chapter:

- The front-end is responsible for handing off new connections and passing incoming data from the client to the back-end nodes. As a result, it must keep track of open and closed connections, and it can use this information in making load balancing decisions. The front-end is not involved in handling outgoing data, which is sent directly from the back-ends to the clients.

- The front-end limits the number of outstanding requests at the back-ends. This approach allows the front-end more flexibility in responding to changing load on the back-ends, since waiting requests can be directed to back-ends as capacity becomes available. In contrast, if we queued requests only on the back-end nodes, a slow node could cause many requests to be delayed even though other nodes might have free capacity.

- Any back-end node is capable of serving any target, although in certain request distribution strategies, the front-end may direct a request only to a subset of the back-ends.

3.1.2 Aiming for Balanced Load

In state-of-the-art cluster servers, the front-end uses *weighted round-robin* request distribution [21, 37]. The incoming requests are distributed in round-robin fashion, weighted by some measure of the load on the different back-ends. For instance, the CPU and disk utilization, or the number of open connections in each back-end may be used as an estimate of the load.

This strategy produces good load balancing among the back-ends. However, since it does not consider the type of service or requested document in choosing a back-end node, each back-end node is equally likely to receive a given type of request. Therefore, each back-end node receives an approximately identical working set of requests, and caches an approximately identical set of documents. If this working set exceeds the size of main memory available for caching documents, frequent cache misses will occur.

3.1.3 Aiming for Locality

In order to improve locality in the back-end's cache, a simple front-end strategy consists of partitioning the name space of the database in some way, and assigning request for all targets in a particular partition to a particular back-end. For instance, a hash function can be used to perform the partitioning. We will call this strategy *locality-based* [LB].

A good hashing function partitions both the name space and the working set more or less evenly among the back-ends. If this is the case, the cache in each back-end should achieve a much higher hit rate, since it is only trying to cache its subset of the working set, rather than the entire working set, as with load balancing based approaches. What is a good partitioning for locality may, however, easily prove a poor choice of partitioning for load balancing. For example, if a small set of targets in the working set account for a large fraction of the incoming requests, the back-ends serving those targets will be far more loaded than others.

3.1.4 Basic Locality-Aware Request Distribution

The goal of LARD is to combine good load balancing and high locality. We develop our strategy in two steps. The basic strategy, described in this subsection, always assigns a single back-end node to serve a given target, thus making the idealized assumption that a single target cannot by itself exceed the capacity of one node. This restriction is removed in the next subsection, where we present the complete strategy.

Figure 3.2 presents pseudo-code for the basic LARD. The front-end maintains a one-to-one mapping of targets to back-end nodes in the `server` array. When the first request arrives for a given target, it is assigned a back-end node by choosing a lightly loaded back-end node. Subsequent requests are directed to a target's assigned back-end node, unless that node is overloaded. In the latter case, the target is assigned a new back-end node from the current set of lightly loaded nodes.

A node's load is measured as the number of active connections, i.e., connections that have been handed off to the node, have not yet completed, and are showing request activity. Observe that an overloaded node will fall behind and the resulting queuing of requests will cause its number of active connections to increase, while the

```
while (true)
  fetch next request r;
  if server[r.target] = null then
    n, server[r.target] ← {least loaded node};
  else
    n ← server[r.target];
    if (n.load > T_high && ∃ node with load < T_low) ||
        n.load ≥ 2 * T_high then
      n, server[r.target] ← {least loaded node};
  send r to n;
```

Figure 3.2 : The Basic LARD Strategy - In the basic strategy, each target is only assigned to a single destination node at any given time.

number of active connections at an underloaded node will tend to zero. Monitoring the relative number of active connections allows the front-end to estimate the amount of “outstanding work” and thus the relative load on a back-end without requiring explicit communication with the back-end node.

The intuition for the basic LARD strategy is as follows: The distribution of targets when they are first requested leads to a partitioning of the name space of the database, and indirectly to a partitioning of the working set, much in the same way as with the strategy purely aiming for locality. It also derives similar locality gains from doing so. Only when there is a significant load imbalance do we diverge from this strategy and re-assign targets. The definition of a “significant load imbalance” tries to reconcile two competing goals. On one hand, we do not want greatly diverging load values on different back-ends. On the other hand, given the cache misses and disk activity resulting from re-assignment, we do not want to re-assign targets to smooth out only minor or temporary load imbalances. It suffices to make sure that no node has idle resources while another back-end is dropping behind.

We define T_{low} as the load (in number of active connections) below which a back-end is likely to have idle resources. We define T_{high} as the load above which a node is likely to cause substantial delay in serving requests. If a situation is detected where a node has a load larger than T_{high} while another node has a load less than T_{low}, a target is moved from the high-load to the low-load back-end. In addition, to limit the delay variance among different nodes, once a node reaches a load of $2T_{high}$, a target is moved to a less loaded node, even if no node has a load of less than T_{low}.

If the front-end did not limit the total number of active connections admitted into the cluster, the load on all nodes could rise to $2T_{high}$, and LARD would then behave like WRR. To prevent this, the front-end limits the sum total of connections handed to all back-end nodes to the value $S = (n-1) * T_{high} + T_{low} - 1$, where n is the number of back-end nodes. Setting S to this value ensures that at most $n-2$ nodes can have a load $\geq T_{high}$ while no node has load $< T_{low}$. At the same time, enough connections are admitted to ensure all n nodes can have a load above T_{low} (i.e., be fully utilized) and still leave room for a limited amount of load imbalance between the nodes (to prevent unnecessary target reassignments in the interest of locality).

The two conditions for deciding when to move a target attempt to ensure that the cost of moving is incurred only when the load difference is substantial enough to warrant doing so. Whenever a target gets reassigned, our two tests combined with the definition of S ensure that the load difference between the old and new targets is at least $T_{high} - T_{low}$. To see this, note that the definition of S implies that there must always exist a node with a load $< T_{high}$. The maximal load imbalance that can arise is $2T_{high} - T_{low}$.

The appropriate setting for T_{low} depends on the speed of the back-end nodes. In practice, T_{low} should be chosen high enough to avoid idle resources on back-end nodes, which could cause throughput loss. Given T_{low}, choosing T_{high} involves a tradeoff. $T_{high} - T_{low}$ should be low enough to limit the delay variance among the back-ends to acceptable levels, but high enough to tolerate limited load imbalance and short-term load fluctuations without destroying locality.

Simulations to test the sensitivity of our strategy to these parameter settings show that the maximal delay difference increases approximately linearly with $T_{high} - T_{low}$. The throughput increases mildly and eventually flattens as $T_{high} - T_{low}$ increases. Therefore, T_{high} should be set to the largest possible value that still satisfies the desired bound on the delay difference between back-end nodes. Given a desired maximal delay difference of D secs and an average request service time of R secs, T_{high} should be set to $(T_{low} + D/R)/2$, subject to the obvious constraint that $T_{high} > T_{low}$. The setting of T_{low} can be conservatively high with no adverse impact on throughput and only a mild increase in the average delay. Furthermore, if desired, the setting of T_{low} can be easily automated by requesting explicit load information from the back-end nodes during a "training phase". In our simulations and in the prototype, we have found settings of $T_{low} = 25$ and $T_{high} = 65$ active connections to give good performance across all workloads we tested.

3.1.5 LARD with Replication

A potential problem with the basic LARD strategy is that a given target is served by only a single node at any given time. However, if a single target causes a back-end to go into an overload situation, the desirable action is to assign several back-end nodes to serve that document, and to distribute requests for that target among the serving nodes. This leads us to the second version of our strategy, which allows replication.

Pseudo-code for this strategy is shown in Figure 3.3. It differs from the original one as follows: The front-end maintains a mapping from targets to a *set* of nodes that serve the target. Requests for a target are assigned to the least loaded node in the target's server set. If a load imbalance occurs, the front-end checks if the requested document's server set has changed recently (within K seconds). If so, it picks a lightly loaded node and adds that node to the server set for the target. On

```
while (true)
   fetch next request r;
   if serverSet[r.target] = ∅ then
      n, serverSet[r.target] ← {least loaded node};
   else
      n ← {least loaded node in serverSet[r.target]};
      m ← {most loaded node in serverSet[r.target]};
      if (n.load > T_high && ∃ node with load < T_low) ||
            n.load ≥ 2T_high then
         p ← {least loaded node};
         add p to serverSet[r.target];
         n ← p;
      if |serverSet[r.target]| > 1 &&
            time() - serverSet[r.target].lastMod > K then
         remove m from serverSet[r.target];
   send r to n
   if serverSet[r.target] changed in this iteration then
      serverSet[r.target].lastMod ← time();
```

Figure 3.3 : LARD with Replication - In this approach, the majority of targets are assigned to a single destination node, but the most heavily requested targets will be assigned to multiple nodes for the duration of their heavy activity.

the other hand, if a request target has multiple servers and has not moved or had a server node added for some time (K seconds), the front-end removes one node from the target's server set. This ensures that the degree of replication for a target does not remain unnecessarily high once it is requested less often. In our experiments, we used values of $K = 20$ secs.

3.1.6 Discussion

As will be seen in Sections 3.3 and 3.5, the LARD strategies result in a good combination of load balancing and locality. In addition, the strategies outlined above have several desirable features. First, they do not require any extra communication between the front-end and the back-ends. Second, the front-end need not keep track of any frequency of access information or try to model the contents of the caches of the back-ends. In particular, the strategy is independent of the local replacement policy used by the back-ends. Third, the absence of elaborate state in the front-end makes it rather straightforward to recover from a back-end node failure. The front-end simply re-assigns targets assigned to the failed back-end as if they had not been assigned before. For all these reasons, we argue that the proposed strategy can be implemented without undue complexity.

In a simple implementation of the two strategies, the size of the `server` or `serverSet` arrays, respectively, can grow to the number of targets in the server's database. Despite the low storage overhead per target, this can be of concern in servers with very large databases. In this case, the mappings can be maintained in an LRU cache, where assignments for targets that have not been accessed recently are discarded. Discarding mappings for such targets is of little consequence, as these targets have most likely been evicted from the back-end nodes' caches anyway.

3.2 Simulation

To study various request distribution policies for a range of cluster sizes under different assumptions for CPU speed, amount of memory, number of disks and other parameters, we developed a configurable web server cluster simulator. We also implemented a prototype of a LARD-based cluster, which is described in Section 3.5.

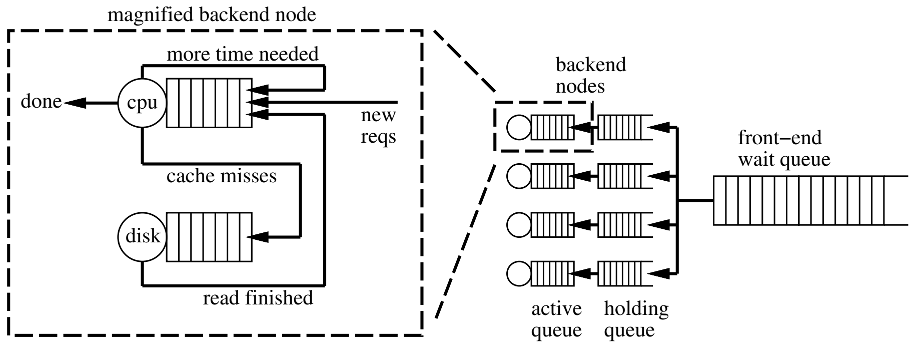

Figure 3.4 : Cluster Simulation Model - A front-end queue controls the number of equests in the back-end nodes. Within each node, the active requests can be rocessed at the CPU and disk queues.

3.2.1 Simulation Model

The simulation model is depicted in Figure 3.4. Each back-end node consists of a CPU and locally-attached disk(s), with separate queues for each. In addition, each node maintains its own main memory cache of configurable size and replacement policy. For simplicity, caching is performed on a whole-file basis.

Processing a request requires the following steps: connection establishment, disk reads (if needed), target data transmission, and connection teardown. The assumption is that front-end and networks are fast enough not to limit the cluster's performance, thus fully exposing the throughput limits of the back-ends. Therefore, the front-end is assumed to have no overhead and all networks have infinite capacity in the simulations.

The individual processing steps for a given request must be performed in sequence, but the CPU and disk times for differing requests can be overlapped. Also, large file reads are blocked, such that the data transmission immediately follows the disk read for each block. Multiple requests waiting on the same file from disk can be satisfied with only one disk read, since all the requests can access the data once it is cached in memory.

The costs for the basic request processing steps used in our simulations were derived by performing measurements on a 300 Mhz Pentium II machine running FreeBSD 2.2.5 and an aggressive experimental web server. Connection establishment and teardown costs are set at 145μs of CPU time each, while transmit processing incurs 40μs per 512 bytes. Using these numbers, an 8 KByte document can be served from the main memory cache at a rate of approximately 1075 requests/sec.

If disk access is required, reading a file from disk has a latency of 28 ms (2 seeks + rotational latency). The disk transfer time is 410μs per 4 KByte (resulting in approximately 10 MBytes/sec peak transfer rate). For files larger than 44 KBytes, an additional 14 ms (seek plus rotational latency) is charged for every 44 KBytes of file length in excess of 44 KBytes. 44 KBytes was measured as the average disk transfer size between seeks in our experimental server. Unless otherwise stated, each back-end node has one disk.

The cache replacement policy we chose for all simulations is Greedy-Dual-Size (GDS), as it appears to be the best known policy for Web workloads [18]. We have also performed simulations with LRU, where files with a size of more than 500KB are never cached. The relative performance of the various distribution strategies remained largely unaffected. However, the absolute throughput results were up to 30% lower with LRU than with GDS.

3.2.2 Simulation Inputs

The input to the simulator is a stream of tokenized target requests, where each token represents a unique target being served. Associated with each token is a target size in bytes. This tokenized stream can be synthetically created, or it can be generated by processing logs from existing web servers.

One of the traces we use was generated by combining logs from multiple departmental web servers at Rice University. This trace spans a two-month period. Another trace comes from IBM Corporation's main web server (www.ibm.com) and represents server logs for a period of 3.5 days starting at midnight, June 1, 1998.

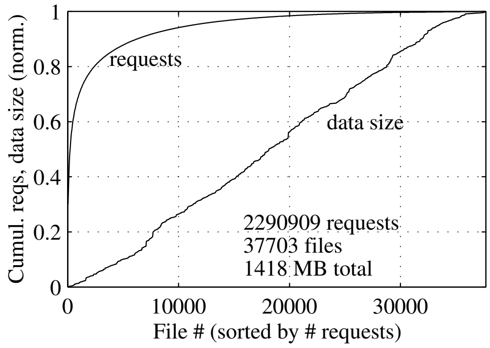

Figure 3.5 : Rice University Trace - This trace combines the logs from multiple departmental Web servers at Rice University. This trace is also described as the MERGED trace in Section 2.4.4

.

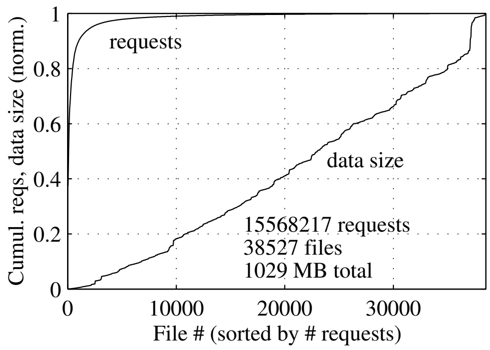

Figure 3.6 : IBM Trace - The IBM trace represents accesses to www.ibm.com and represents accesses over a period of 3.5 days. In broad terms, it has a higher locality of reference than the Rice trace, and it also has a smaller average file size.

Figures 3.5 and 3.6 show the cumulative distributions of request frequency and size for the Rice University trace and the IBM trace, respectively. Shown on the x-axis is the set of target files in the trace, sorted in decreasing order of request frequency. The y-axis shows the cumulative fraction of requests and target sizes, normalized to the total number of requests and total data set size, respectively. The data set for the Rice University trace consist of 37703 targets covering 1418 MB of space, whereas the IBM trace consists of 38527 targets and 1029 MB of space. While the data sets in both traces are of a comparable size, it is evident from the graphs that the Rice trace has much less locality than the IBM trace. In the Rice trace, 560/705/927 MB of memory is needed to cover 97/98/99% of all requests, respectively, while only 51/80/182 MB are needed to cover the same fractions of requests in the IBM trace.

This difference is likely to be caused in part by the different time spans that each trace covers. Also, the IBM trace is from a single high-traffic server, where the content designers have likely spent effort to minimize the sizes of high frequency documents in the interest of performance. The Rice trace, on the other hand, was merged from the logs of several departmental servers.

As with all caching studies, interesting effects can only be observed if the size of the working set exceeds that of the cache. Since even our larger trace has a relatively small data set (and thus a small working set), and also to anticipate future trends in working set sizes, we chose to set the default node cache size in our simulations to 32 MB. Since in reality, the cache has to share main memory with OS kernel and server applications, this typically requires at least 64 MB of memory in an actual server node.

3.2.3 Simulation Outputs

The simulator calculates overall throughput, hit rate, and underutilization time. Throughput is the number of requests in the trace that were served per second by the entire cluster, calculated as the number of requests in the trace divided by the simulated time it took to finish serving all the requests in the trace. The request arrival rate was matched to the aggregate throughput of the server.

The cache hit ratio is the number of requests that hit in a back-end node's main memory cache divided by the number of requests in the trace. The idle time was measured as the fraction of simulated time during which a back-end node was underutilized, averaged over all back-end nodes.

Node underutilization is defined as the time that a node's load is less than 40%

of T_{low}. This value was determined by inspection of the simulator's disk and CPU activity statistics as a point below which a node's disk and CPU both had some idle time in virtually all cases. The overall throughput is the best summary metric, since it is affected by all factors. The cache hit rate gives an indication of how well locality is being maintained, and the node underutilization times indicate how well load balancing is maintained.

3.3 Simulation Results

We simulate the four different request distribution strategies presented in Section 3.1.

1. weighted round-robin [WRR],
2. locality-based [LB],
3. basic LARD [LARD], and
4. LARD with replication [LARD/R].

In addition, observing the large amount of interest generated by global memory systems (GMS) and cooperative caching to improve hit rates in cluster main memory caches [24, 30, 43], we simulate a weighted round-robin strategy in the presence of a global memory system on the back-end nodes. We refer to this system as WRR/GMS. The GMS in WRR/GMS is loosely based on the GMS described in Feeley et al. [30].

We also simulate an idealized locality-based strategy, termed LB/GC, where the front-end keeps track of each back-end's cache state to achieve the effect of a global cache. On a cache hit, the front-end sends the requests to the back-end that caches the target. On a miss, the front-end sends the request to the back-end that caches the globally "oldest" target, thus causing eviction of that target.

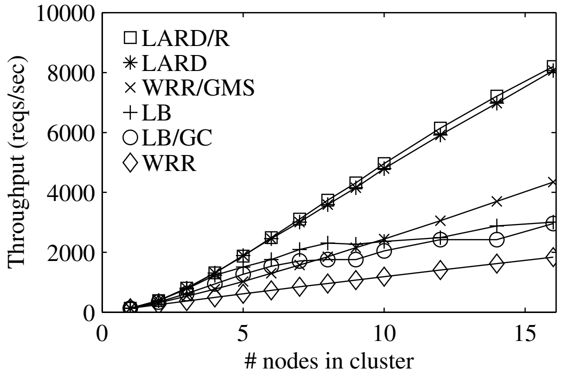

Figure 3.7 : Throughput on Rice Trace - Both LARD and LARD/R significantly outperform the other strategies on the Rice University workload. This result is due to the effects of improved cache hit rate and good load balancing in the cluster.

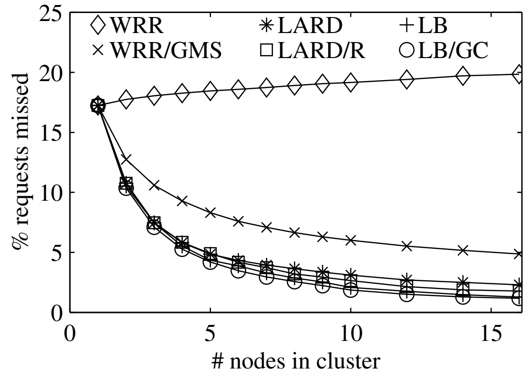

Figure 3.8 : Cache Miss Ratio on Rice Trace - The LARD schemes achieve a miss ratio comparable to the purely locality-based schemes. The WRR approach does not obtain any cache benefit from the addition of extra nodes.

3.3.1 Rice University Trace

Figures 3.7, 3.8, and 3.9 show the aggregate throughput, cache miss ratio, and idle time as a function of the number of back-end nodes for the combined Rice University trace. WRR achieves the lowest throughput, the highest cache miss ratio, but also the lowest idle time (i.e., the highest back-end node utilization) of all strategies. This confirms our reasoning that the weighted round-robin scheme achieves good load balancing (thus minimizing idle time). However, since it ignores locality, it suffers

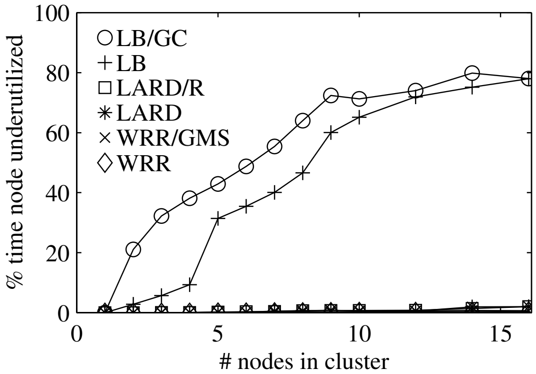

Figure 3.9 : Idle Time on Rice Trace - The LARD schemes achieve load balance in the cluster, demonstrated by their low idle times. The purely locality-based approaches show significant degradation as the cluster size increases.

many cache misses. This latter effect dominates, and the net effect is that the server's throughput is limited by disk accesses. With WRR, the effective size of the server cache remains at the size of the individual node cache, independent of the number of nodes. This can be clearly seen in the flat cache miss ratio curve for WRR.

As expected, both LB schemes achieve a decrease in cache miss ratio as the number of nodes increases. This reflects the aggregation of effective cache size. However, this advantage is largely offset by a loss in load balancing (as evidenced by the increased idle time), resulting in only a modest throughput advantage over WRR.

An interesting result is that LB/GC, despite its greater complexity and sophistication, does not yield a significant advantage over the much simpler LB. This suggests that the hashing scheme used in LB achieves a fairly even partitioning of the server's working set, and that maintaining cache state in the front-end may not be necessary to achieve good cache hit ratios across the back-end nodes. This partly validates the approach we took in the design of LARD, which does not attempt to model the state of the back-end caches.

The throughput achieved with LARD/R exceeds that of the state-of-the-art WRR on this trace by a factor of 3.9 for a cluster size of eight nodes, and by about 4.5 for sixteen nodes. The Rice trace requires the combined cache size of eight to ten nodes to hold the working set. Since WRR cannot aggregate the cache size, the server remains disk bound for all cluster sizes. LARD and LARD/R, on the other hand, cause the system to become increasingly CPU bound for eight or more nodes, resulting in superlinear speedup in the 1-10 node region, with linear, but steeper speedup for more than ten nodes. Another way to read this result is that with WRR, it would take a ten times larger cache in each node to match the performance of LARD on this particular trace. We have verified this fact by simulating WRR with a tenfold node cache size.

The reason for the increased throughput and speedup can also be clearly seen in the graphs for idle time and cache miss ratio. LARD and LARD/R achieve average idle times around 1%, while achieving cache miss ratios that decrease with increasing cluster size and reach values below 4% for eight and more nodes in the case of LARD, going down to 2% at sixteen nodes in the case of LARD/R. Thus, LARD and LARD/R come close to WRR in terms of load balancing while simultaneously achieving cache miss ratios close to those obtained with LB/GC. Thus, LARD and LARD/R are able to translate most of the locality advantages of LB/GC into additional server throughput.

The throughput achieved with LARD/R exceeds that of LARD slightly for seven or more nodes, while achieving lower cache miss ratio and lower idle time. While WRR/GMS achieves a substantial performance advantage over WRR, its throughput remains below 50% of LARD and LARD/R's throughput for all cluster sizes.

3.3.2 Other Workloads

Figure 3.10 shows the throughput results obtained for the various strategies on the IBM trace (www.ibm.com). In this trace, the average file size is smaller than in the Rice trace, resulting in much larger throughput numbers for all strategies. The higher locality of the IBM trace demands a smaller effective cache size to cache the working

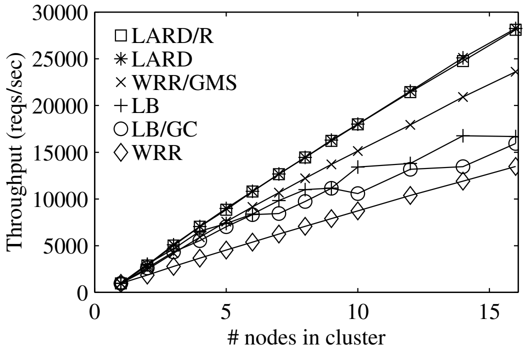

Figure 3.10 : Throughput on IBM Trace - The various strategies exhibit the same qualitative behavior on the IBM trace as with the Rice trace. The absolute numbers are higher, since the IBM trace has a lower average file size.

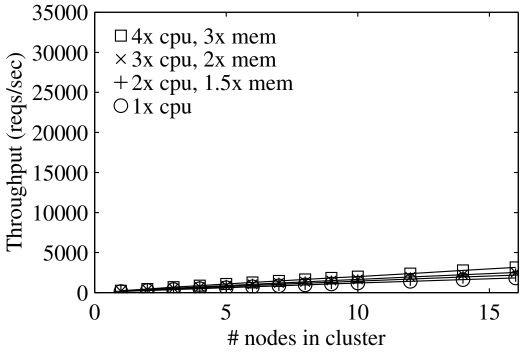

Figure 3.11 : WRR vs CPU - Adding additional CPU performance in the WRR scheme produces little benefit, since the back-end nodes remain disk-bound.

set. Thus, LARD and LARD/R achieve superlinear speedup only up to 4 nodes in this trace, resulting in a throughput that is slighly more than twice that of WRR for 4 nodes and above.

WRR/GMS achieves much better relative performance on this trace than on the Rice trace and comes within 15% of LARD/R's throughput at 16 nodes. However, this result has to be seen in light of the very generous assumptions made in the simulations about the performance of the WRR/GMS system. It was assumed that maintaining the global cache directory and implementing global cache replacement

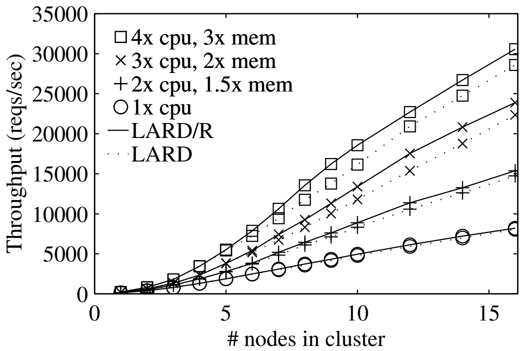

Figure 3.12 : LARD vs CPU - The LARD schemes scale well as CPU speed increases, since the higher filesystem cache hit rates keep the back-end nodes CPU-bound.

has no cost.

The performance of LARD/R only slightly exceeds that of LARD on the Rice trace and matches that of LARD on the IBM trace. The reason is that neither trace contains high-frequency targets that can benefit from replication. The highest frequency files in the Rice and IBM traces account for only 2% and 5%, respectively, of all requests in the traces. However, it is clear that real workloads exist that contain targets with much higher request frequency (e.g. www.netscape.com). To evaluate LARD and LARD/R on such workloads, we modified the Rice trace to include a small number of artifical high-frequency targets and varied their request rate between 5 and 75% of the total number of requests in the trace. With this workload, the throughput achieved with LARD/R exceeds that of LARD by 0-15%. The most significant increase occurs when the size of the "hot" targets is larger than 20 KBytes and the combined access frequency of all hot targets accounts for 10-60% of the total number of requests.

We also ran simulations on a trace from the IBM web server hosting the Deep Blue/Kasparov Chess match in May 1997. This trace is characterized by large numbers of requests to a small set of targets. The working set of this trace is very small and achieves a low miss ratio with a main memory cache of a single node (32 MB).

This trace presents a best-case scenario for WRR and a worst-case scenario for LARD, as there is nothing to be gained from an aggregation of cache size, but there is the potential to lose performance due to imperfect load balancing. Our results show that both LARD and LARD/R closely match the performance of WRR on this trace. This is reassuring, as it demonstrates that our strategy can match the performance of WRR even under conditions that are favorable to WRR.

3.3.3 Sensitivity to CPU and Disk Speed

In our next set of simulations, we explore the impact of CPU speed on the relative performance of LARD versus the state-of-the-art WRR. We performed simulations on the Rice trace with the default CPU speed setting explained in Section 3.2, and with twice, three and four times the default speed setting. The [1x] speed setting represents a state-of-the-art inexpensive high-end PC (300 MHz Pentium II), and the higher speed settings project the speed of high-end PCs likely to be available in the the next few years. As the CPU speed increases while disk speed remains constant, higher cache hit rates are necessary to remain CPU bound at a given cluster size, requiring larger per-node caches. We made this adjustment by setting the node memory size to 1.5, 2, and 3 times the base amount (32 MB) for the [2x], [3x] and [4x] CPU speed settings, respectively.

As CPU speeds are expected to improve at a much faster rate than disk speeds, one would expect that the importance of caching and locality increases. Indeed, our simulations confirm this. Figures 3.11 and 3.12, respectively, show the throughput results for WRR and LARD/R on the combined Rice University trace with different CPU speed assumptions. It is clear that WRR cannot benefit from added CPU at all, since it is disk-bound on this trace. LARD and LARD/R, on the other hand, can capitalize on the added CPU power, because their cache aggregation makes the system increasingly CPU bound as nodes are added to the system. In addition, the results indicate the throughput advantage of LARD/R over LARD increases with

CPU speed, even on a workload that presents little opportunity for replication.

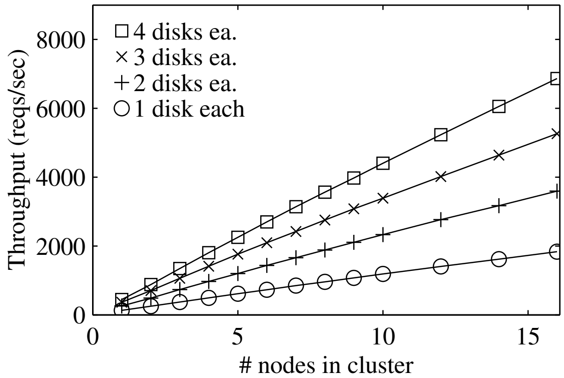

Figure 3.13 : WRR vs disks - Since WRR is do heavily disk bound on the Rice trace, the addition of extra disks on each back-end node improves its performance significantly.

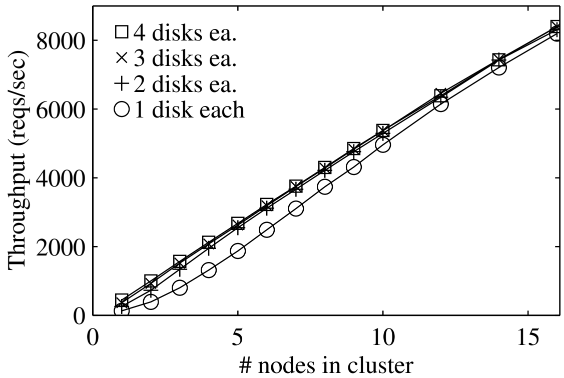

Figure 3.14 : LARD/R vs disks - Adding additional disks in the LARD case produces little benefit, since the lower cache miss rates cause the disks to be less of a bottleneck.

In our final set of simulations, we explore the impact of using multiple disks in each back-end node on the relative performance of LARD/R versus WRR. Figures 3.13 and 3.14, respectively, show the throughput results for WRR and LARD/R on the combined Rice University trace with different numbers of disks per back-end node. With LARD/R, a second disk per node yields a mild throughput gain, but additional disks do not achieve any further benefit. This can be expected, as the increased cache effectiveness of LARD/R causes a reduced dependence on disk speed.

WRR, on the other hand, greatly benefits from multiple disks as its throughput is mainly bound by the performance of the disk subsystem. In fact, with four disks per node and 16 nodes, WRR comes within 15% of LARD/R's throughput. However, the are several things to note about this result. First, the assumptions made in the simulations about the performance of multiple disks are generous. It is assumed that both seek and disk transfer operations can be fully overlapped among all disks. In practice, this would require that each disk is attached through a separate SCSI bus/controller.

Second, it is assumed that the database is striped across the multiple disks in a manner that achieves good load balancing among the disks with respect to the workload (trace). In our simulations, the files were distributed across the disks in round-robin fashion based on decreasing order of request frequency in the trace[2].

Finally, WRR has the same scalability problems with respect to disks as it has with memory. To upgrade a cluster with WRR, it is not sufficient to add nodes as with LARD/R. Additional disks (and memory) have to be added to all nodes to achieve higher performance.

3.3.4 Delay

While most of our simulations focus on the server's throughput limits, we also monitored request delay in our simulations for both the Rice University trace as well as the IBM trace. On the Rice University trace, the average request delay for LARD/R is less than 25% that of WRR. With the IBM trace, LARD/R's average delay is one half that of WRR.

[2]Note that replicating the entire database on each disk as an approach to achieving disk load balancing would require special OS support to avoid double buffering and caching of replicated files and to assign requests to disks dynamically based on load.

3.4 TCP Connection Handoff

In this section, we briefly discuss our TCP handoff protocol and present some performance results with a prototype implementation. A full description of the protocol is beyond the scope of this dissertation. The TCP handoff protocol is used to hand off established client TCP [63] connections between the front-end and the back-end of a cluster server that employs content-based request distribution.

A handoff protocol is necessary to enable content-based request distribution in a client-transparent manner. This is true for any service (like HTTP) that relies on a connection-oriented transport protocol like TCP. The front-end must establish a connection with the client to inspect the target content of a request *prior* to assigning the connection to a back-end node. The established connection must then be handed to the chosen back-end node. State-of-the-art commercial cluster front-ends (e.g., [21, 37]) assign requests without regard to the requested content and can therefore forward client requests to a back-end node prior to establishing a connection with the client.

Our handoff protocol is transparent to clients and also to the server applications running on the back-end nodes. That is, no changes are needed on the client side, and server applications can run unmodified on the back-end nodes. Figure 3.15 depicts the protocol stacks on the clients, front-end, and back-ends, respectively. The handoff protocol is layered on top of TCP and runs on the front-end and back-end nodes. Once a connection is handed off to a back-end node, incoming traffic on that connection (principally acknowledgment packets) is forwarded by an efficient forwarding module at the bottom of the front-end's protocol stack.

The TCP implementation running on the front-end and back-ends needs a small amount of additional support for handoff. In particular, the protocol module needs to support an operation that allows the TCP handoff protocol to create a TCP connection at the back-end without going through the TCP three-way handshake. Likewise, an operation is required that retrieves the state of an established connection and

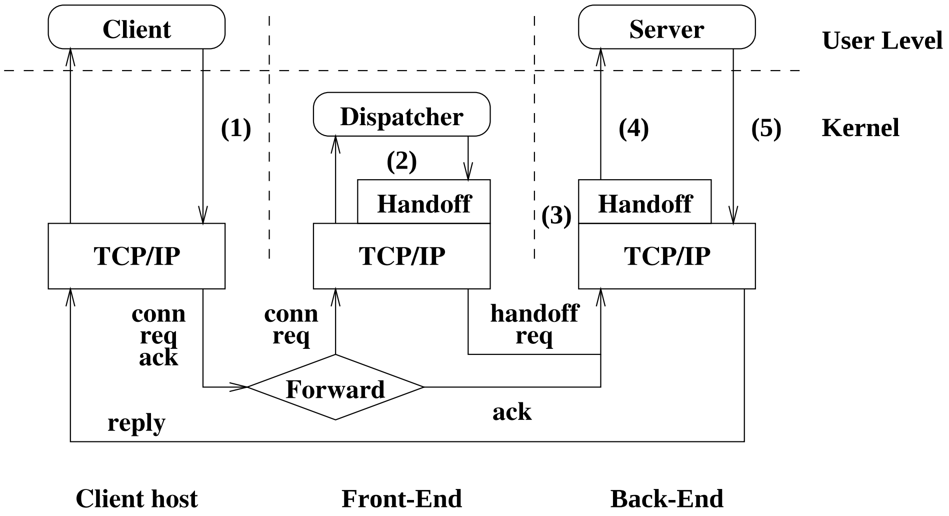

Figure 3.15 : TCP connection handoff - The front-end node passes all incoming traffic to the appropriate back-end node. All outbound data is sent from the back-end directly to the client.

destroys the connection state without going through the normal message handshake required to close a TCP connection.

Figure 3.15 depicts a typical scenario: (1) a client connects to the front-end, (2) the dispatcher at the front-end accepts the connection and hands it off to a back-end using the handoff protocol, (3) the back-end takes over the established connection received by the handoff protocols, (4) the server at the back-end accepts the created connection, and (5) the server at the back-end sends replies directly to the client. The dispatcher is a software module that implements the distribution policy, e.g. LARD.

Once a connection is handed off to a back-end node, the front-end must forward packets from the client to the appropriate back-end node. A single back-end node that fully utilizes a 100 Mb/s network sending data to clients will receive at least 4128 acknowledgments per second (assuming an IP packet size of 1500 and delayed TCP ACKs). Therefore, it is crucial that this packet forwarding is fast.

The forwarding module is designed to allow very fast forwarding of acknowledgment packets. The module operates directly above the network interface and executes in the context of the network interface interrupt handler. A simple hash table lookup is required to determine whether a packet should be forwarded. If so, the packet's header is updated and it is directly transmitted on the appropriate interface. Otherwise, the packet traverses the normal protocol stack.

Results of performance measurements with an implementation of the handoff protocol are presented in Section 3.5.2.

The design of our TCP handoff protocol includes provisions for HTTP 1.1 persistent connections, which allow a client to issue multiple requests. The protocol allows the front-end to either let one back-end serve all of the requests on a persistent connection, or to hand off a connection multiple times, so that different requests on the same connection can be served by different back-ends. However, further research is needed to determine the appropriate policy for handling persistent connections in a cluster with LARD. We have not yet experimented with HTTP 1.1 connections as part of this work.

3.5 Prototype Cluster Performance

In this section, we present performance results obtained with a prototype cluster that uses locality-aware request distribution. We describe the experimental setup used in the experiments, and then present the results.

3.5.1 Experimental Environment

Our testbed consists of 7 client machines connected to a cluster server. The configuration is shown in Figure 3.16. Traffic from the clients flows to the front-end (1) and is forwarded to the back-ends (2). Data packets transmitted from the back-ends to the clients bypass the front-end (3).

The front-end of the server cluster is a 300MHz Intel Pentium II based PC with 128MB of memory. The cluster back-end consists of six PCs of the same type and configuration as the front-end. All machines run FreeBSD 2.2.5. A loadable kernel module was added to the OS of the front-end and back-end nodes that implements the TCP handoff protocol, and, in the case of the front-end, the forwarding module. The clients are 166MHz Intel Pentium Pro PCs, each with 64MB of memory.

The clients and back-end nodes in the cluster are connected using switched Fast Ethernet (100Mbps). The front-end is equipped with two network interfaces, one for communication with the clients, one for communication with the back-ends. Clients, front-end, and back-end are connected through a single 24-port switch. All network interfaces are Intel EtherExpress Pro/100B running in full-duplex mode.

The Apache-1.2.4 [5] server was used on the back-end nodes. Our client software is an event-driven program that simulates multiple HTTP clients. Each simulated HTTP client makes HTTP requests as fast as the server cluster can handle them.

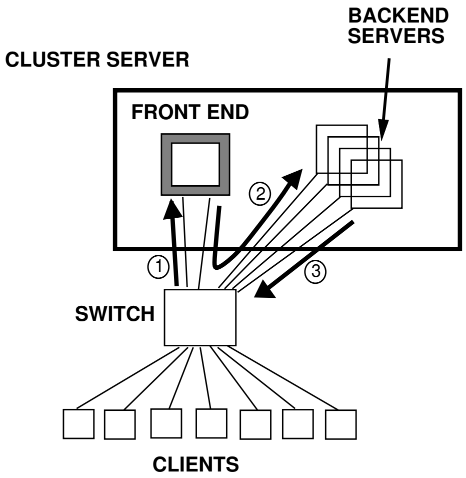

Figure 3.16 : Experimental Testbed - All clients and servers in our testbed are connected via switched Fast Ethernet. However, only the address of the front-end node is visible to the client machines.

3.5.2 Front-end Performance Results

Measurements were performed to evaluate the performance and overhead of the TCP handoff protocol and packet forwarding in the front-end. *Handoff latency* is the added latency a client experiences as a result of TCP handoff. *Handoff throughput* is the maximal rate at which the front-end can accept, handoff, and close connections. *Forwarding throughput* refers to the maximal aggregate rate of data transfers from all back-end nodes to clients. Since this data bypasses the front-end, this figure is limited only by the front-end's ability to forward acknowledgments from the clients to the back-ends.

The measured handoff latency is 194 μsecs and the maximal handoff throughput is approximately 5000 connections per second. Note that the added handoff latency is insignificant, given the connection establishment delay over a wide-area network. The measured ACK forwarding overhead is 9 μsecs, resulting in a theoretical maximal forwarding throughput of over 2.5 Gbits/s. We have not been able to measure such high throughput directly due to lack of network resources, but the measured remaining CPU idle time in the front-end at lower throughput is consistent with this figure. Further measurements indicate that with the Rice University trace as the workload, the handoff throughput and forwarding throughput are sufficient to support 10 back-end nodes of the same CPU speed as the front-end.

Moreover, the front-end can be relatively easily scaled to larger clusters either by upgrading to a faster CPU, or by employing an SMP machine. Connection establishment, handoff, and forwarding are independent for different connections, and can be easily parallelized [65]. The dispatcher, on the other hand, requires shared state and thus synchronization among the CPUs. However, with a simple policy such as LARD/R, the time spent in the dispatcher amounts to only a small fraction of the handoff overhead (10-20%). Therefore, we fully expect that the front-end performance can be scaled to larger clusters effectively using an inexpensive SMP platform equipped with multiple network interfaces.

3.5.3 Cluster Performance Results

A segment of the Rice University trace was used to drive the prototype cluster. A single back-end node running Apache can deliver about 167 req/sec on this trace. On cached, small files (less than 8 KB), an Apache back-end can complete about 800 req/sec.

The Apache Web server relies on the file caching services of the underlying operating system. FreeBSD uses a unified buffer cache, where cached files are competing with user processes for physical memory pages. All page replacement is controlled by FreeBSD's pageout daemon, which implements a variant of the clock algorithm [48]. The cache size is variable and depends on main memory pressure from user applications. In our 128 MB back-ends, memory demands from kernel and Apache server processes leave about 100 MB of free memory. In practice, we observed file cache sizes between 70 and 97 MB.

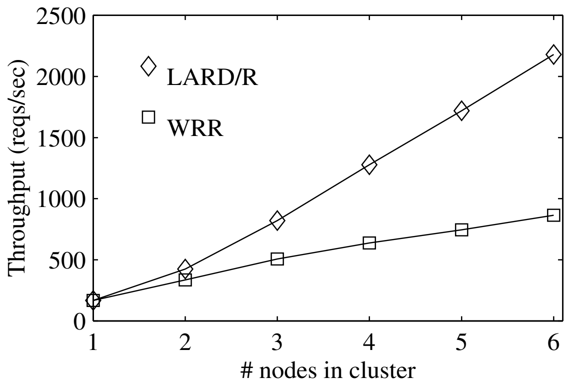

Figure 3.17 : HTTP Throughput (Apache) - Our results in the actual cluster closely match what we expected from our simulations. The WRR approach stays disk bound on this workload, while the cache aggregation in LARD results in higher performance.

We measure the total HTTP throughput of the server cluster with increasing numbers of back-end nodes and with the front-end implementing either WRR or LARD/R. The results are shown in Figure 3.17 and confirm the predictions of the simulator. The throughput achieved with LARD/R exceeds that of WRR by a factor of 2.5 for six nodes. Running LARD/R on a cluster with six nodes at maximal throughput and an aggregate server bandwidth of over 280 Mb/s, the front-end CPU was 60% utilized. This is consistent with our earlier projection that a single CPU front-end can support 10 back-ends of equal CPU speed.

3.6 Related Work

Much current research addresses the scalability problems posed by the Web. The work includes cooperative caching proxies inside the network, push-based document distribution, and other innovative techniques [7, 19, 26, 42, 45, 59]. Our proposal addresses the complementary issue of providing support for cost-effective, scalable network servers.

Network servers based on clusters of workstations are starting to be widely used [31]. Several products are available or have been announced for use as front-end nodes in such cluster servers [21, 37]. To the best of our knowledge, the request distribution strategies used in the cluster front-ends are all variations of weighted round-robin, and do not take into account a request's target content. An exception is the Dispatch product by Resonate, Inc., which supports content-based request distribution [58]. The product does not appear to use any dynamic distribution policies based on content and no attempt is made to achieve cache aggregation via content-based request distribution.

Hunt et al. proposed a TCP option designed to enable content-based load distribution in a cluster server [34]. The design has not been implemented and the performance potential of content-based distribution has not been evaluated as part of that work. Also, no policies for content-based load distribution were proposed. Our TCP handoff protocol design was informed by Hunt et al.'s design, but chooses the different approach of layering a separate handoff protocol on top of TCP.

Fox et al. [31] report on the cluster server technology used in the Inktomi search engine. The work focuses on the reliability and scalability aspects of the system and is complementary to our work. The request distribution policy used in their systems is based on weighted round-robin.

Loosely-coupled distributed servers are widely deployed on the Internet. Such servers use various techniques for load balancing including DNS round-robin [13], HTTP client re-direction [4], Smart clients [66], source-based forwarding [25] and hardware translation of network addresses [21]. Some of these schemes have problems related to the quality of the load balance achieved and the increased request latency. A detailed discussion of these issues can be found in Goldszmidt and Hunt [37] and Damani et al. [25]. None of these schemes support content-based request distribution.

IBM's Lava project [44] uses the concept of a "hit server". The hit server is a specially configured server node responsible for serving cached content. Its specialized OS and client-server protocols give it superior performance for handling HTTP requests of cached documents, but limits it to private Intranets. Requests for uncached documents and dynamic content are delegated to a separate, conventional HTTP server node. Our work shares some of the same goals, but maintains standard client-server protocols, maintains support for dynamic content generation, and focuses on cluster servers.

3.7 Conclusion

We present and evaluate a practical and efficient locality-aware request distribution (LARD) strategy that achieves high cache hit rates and good load balancing in a cluster server. Trace-driven simulations show that the performance of our strategy exceeds that of the state-of-the-art weighted round-robin (WRR) strategy substantially. On workloads with a working set that does not fit in a single server node's main memory cache, the achieved throughput exceeds that of WRR by a factor of two to four.

Additional simulations show that the performance advantages of LARD over WRR increase with the disparity between CPU and disk speeds. Also, our results indicate that the performance of a hypothetical cluster with WRR distribution and a global memory system (GMS) falls short of LARD under all workloads considered, despite generous assumptions about the performance of a GMS system.

We also propose and evaluate an efficient TCP handoff protocol that enables LARD and other content-based request distribution strategies by providing client-transparent connection handoff for TCP-based network services, like HTTP. Performance results indicate that in our prototype cluster environment and on our workloads, a single CPU front-end can support 10 back-end nodes with equal CPU speed as the front-end. Moreover, the design of the handoff protocols is expected to yield scalable performance on SMP-based front-ends, thus supporting larger clusters.

Finally, we present performance results from a prototype LARD server cluster that incorporates the TCP handoff protocol and the LARD strategy. The measured results confirm the simulation results with respect to the relative performance of LARD and WRR.

In this chapter, we have focused on studying HTTP servers that serve static content. However, caching can also be effective for dynamically generated content [39]. Moreover, resources required for dynamic content generation like server processes, executables, and primary data files are also cacheable. While further research is required, we expect that increased locality can benefit dynamic content serving, and that therefore the advantages of LARD also apply to dynamic content.

Chapter 4

Flash

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 [67] 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

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

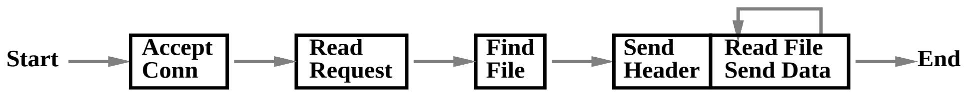

Figure 4.1 : Simplified Request Processing Steps - The minimum processing necessary for handling a single request for a regular file (static content) is shown in this figure.

the MT architecture on the Microsoft Windows NT operating system.

This chapter 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 chapter is structured as follows: Section 4.1 explains the basic processing steps required of all Web servers and provides the background for the following discussion. In Section 4.2, we discuss the asynchronous multi-process event-driven (AMPED), the 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 4.3 before discussing the implementation of the Flash Web server in Section 4.4. Using real and synthetic workloads, we evaluate the performance of all four server architectures and the Apache and Zeus servers in Section 4.5.

4.1 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 chapter 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 4.1, 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 4.2.

In addition to its architecture, the performance of a Web server implementation is also influenced by various optimizations, such as caching. In Section 4.4, we discuss specific optimizations used in the Flash Web server.

4.2 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.

4.2.1 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

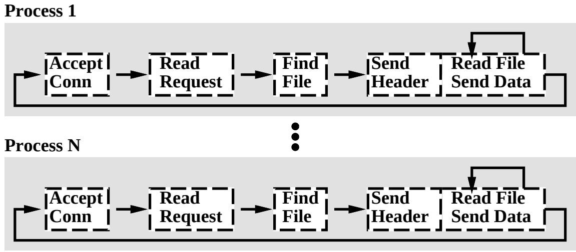

Figure 4.2 : Multi-Process - In the MP model, each server process handles one request at a time. Processes execute the processing stages sequentially.

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.

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 4.2 illustrates the MP architecture.

4.2.2 Multi-threaded

Multi-threaded (MT) servers, depicted in Figure 4.3, 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

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

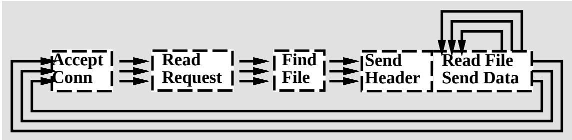

Figure 4.3 : Multi-Threaded - The MT model uses a single address space with multiple concurrent threads of execution. Each thread handles a request.

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.

4.2.3 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 4.4 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,

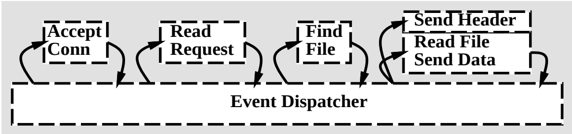

Figure 4.4 : Single Process Event Driven - The SPED model uses a single process to perform all client processing and disk activity in an event-driven manner.

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.

4.2.4 Asymmetric Multi-Process Event-Driven

The Asymmetric Multi-Process Event-Driven (AMPED) architecture, illustrated in Figure 4.5, combines 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.

4.3 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.

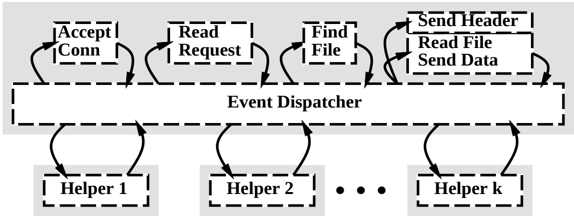

Figure 4.5 : Asymmetric Multi-Process Event Driven - The AMPED model uses a single process for event-driven request processing, but has other helper processes to handle some disk operations.

4.3.1 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.

4.3.2 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.

4.4 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.

4.4.1 Overview

The Flash Web server implements the AMPED architecture described in Section 4.2. 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 [51], 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 4.1 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.

4.4.2 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.

4.4.3 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.

4.4.4 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.

4.4.5 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.

4.4.6 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 [2] 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.

4.4.7 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.

4.5 Performance Evaluation

In this section, we present experimental results that compare the performance of the different Web server architectures presented in Section 4.2 on real workloads. Furthermore, we present comparative performance results for Flash and two state-of-the-art Web servers, Apache [5] and Zeus [67], 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 chapter, 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 [8]. Each simulated HTTP client makes HTTP requests as fast as the server can handle them.

4.5.1 Synthetic Workload

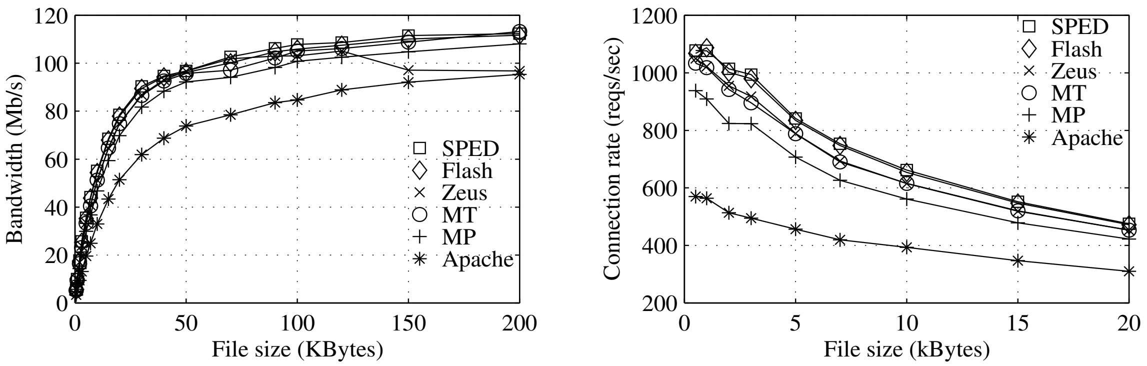

Figure 4.6 : Solaris single file test — On this trivial test, server architecture seems to have little impact on performance. The aggressive optimizations in Flash and Zeus cause them to outperform Apache.

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

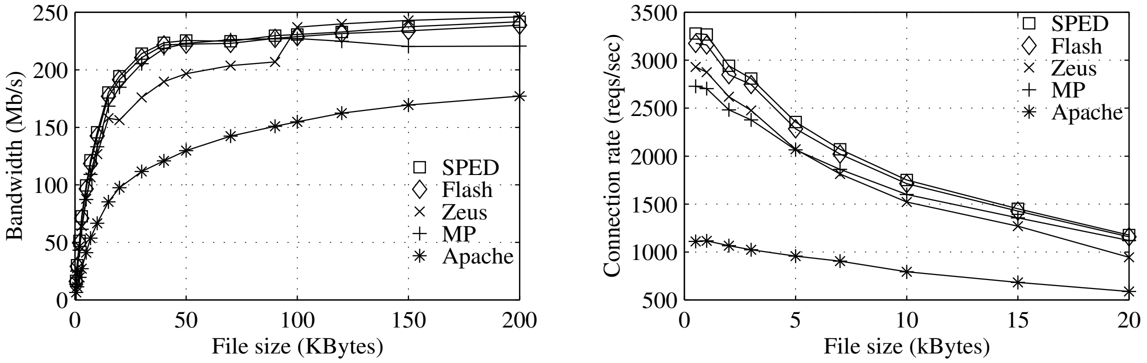

Figure 4.7 : FreeBSD single file test — The higher network performance of FreeBSD magnifies the difference between Apache and the rest when compared to Solaris. The shape of the Zeus curve between 10 kBytes and 100 kBytes is likely due to the byte alignment problem mentioned in Section 4.4.5.

to perform at their highest capacity, since the requested file is cached in the server's main memory. The results are shown in Figures 4.6 (Solaris) and 4.7 (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 4.4.5.

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.

4.5.2 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.

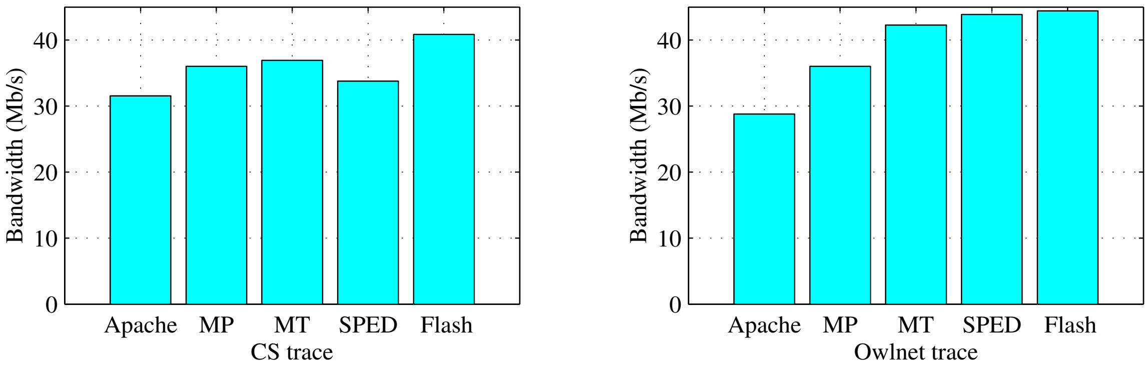

Figure 4.8 : Performance on Rice Server Traces/Solaris - The performance of the various servers replaying real traces differs from the microbenchmarks for two reasons – the traces have relatively small average file sizes, and large working sets that induce heavy disk activity.

Figure 4.8 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.

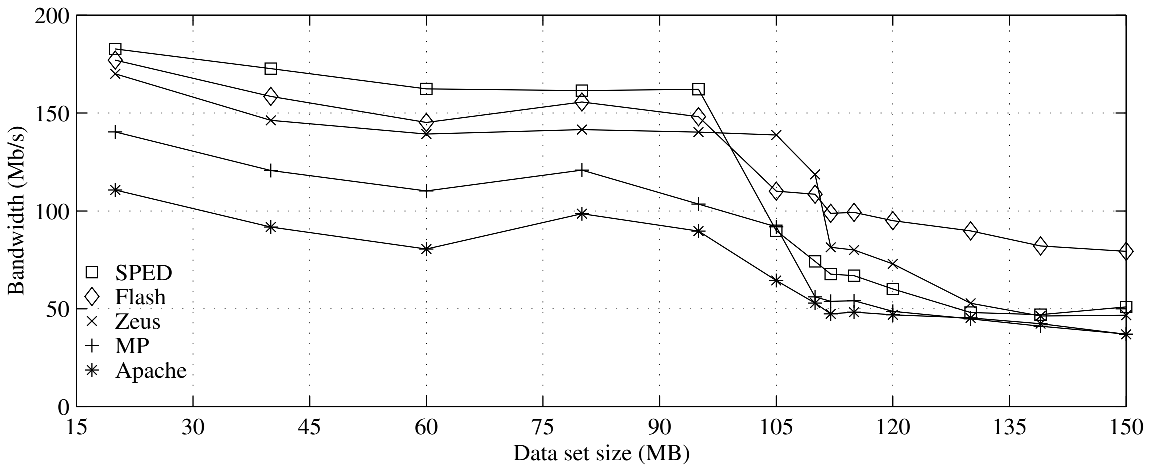

Figure 4.9 : FreeBSD Real Workload - The SPED architecture is ideally suited for cached workloads, and when the working set fits in cache, Flash mimics Flash-SPED. However, Flash-SPED's performance drops drastically when operating on disk-bound workloads.

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 4.9 (BSD) and 4.10 (Solaris) shows the performance, measured as the total output bandwidth, of the various servers under real workload and various dataset

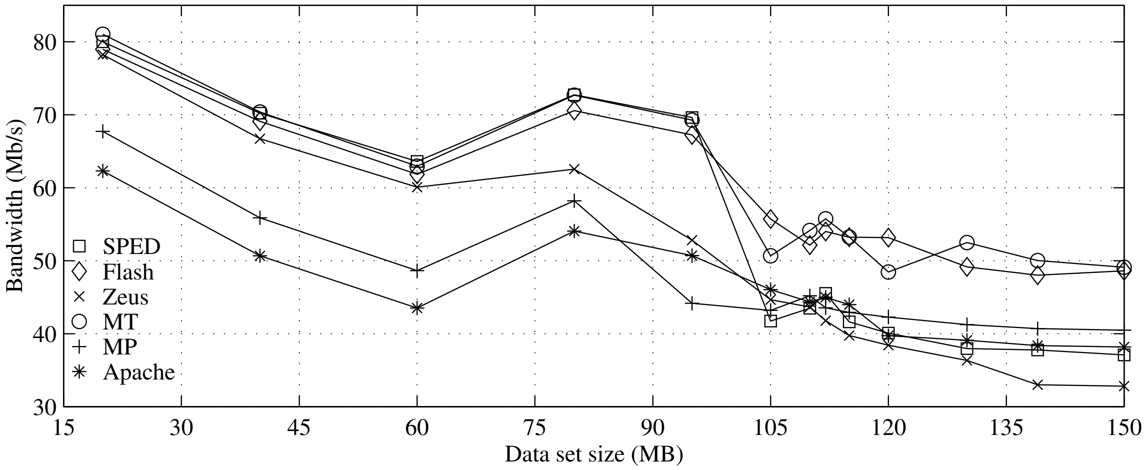

Figure 4.10 : Solaris Real Workload - The Flash-MT server has comparable performance to Flash for both in-core and disk-bound workloads. This result was achieved by carefully minimizing lock contention, adding complexity to the code. Without this effort, the disk-bound results otherwise resembled Flash-SPED.

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 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. Performance 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.

4.5.3 Flash Performance Breakdown

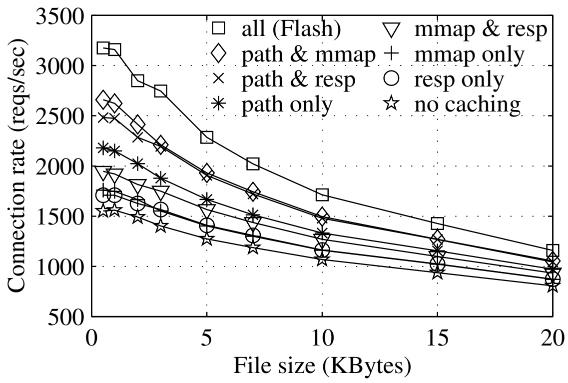

Figure 4.11 : Flash Performance Breakdown - Without optimizations, Flash's small-file performance would drop in half. The eight lines show the effect of various combinations of the caching optimizations.

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 4.11 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.

4.5.4 Performance under WAN conditions

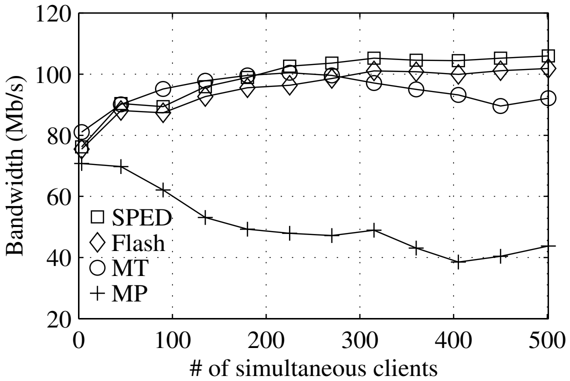

Figure 4.12 : Adding clients - The low per-client overheads of the MT, SPED and AMPED models cause stable performance when adding clients. Multiple application-level caches and per-process overheads cause the MP model's performance to drop.

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 [49].

The number of concurrent connections can have a significant impact on server performance [10]. 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 4.12 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.

4.6 Related Work

James Hu et al. [32] 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. [50] 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. [33] extensively analyze an earlier version of Apache and implement a number of optimizations, improving performance especially for smaller requests. Yates et al. [64] measure the demands a server places on the operating system for various workloads types and service rates. Banga et al. [11] 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 [20], 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 [19] 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. [6] propose new caching policies by analyzing server access logs and looking for similarities across servers. Cao et al. [18] 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 [23, 46].

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. [62] introduce a new API to read and send memory-mapped files without copying. IO-Lite [56] extends the fbufs [27] model to integrate filesystem, networking, interprocess communication, and application-level buffers using a set of uniform interfaces. Engler et al. [41] 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 [44].

Other approaches for increasing Web server performance employ multiple machines. In this area, some work has focused on using multiple server nodes in parallel [13, 21, 25, 31, 36, 54], or sharing memory across machines [24, 30, 43].

4.7 Conclusion

This chapter 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.

Thanks to Ed Costello, Cameron Ferstat, Alister Lewis-Bowen and Chet Murthy, for their help in obtaining the IBM server logs.

We are grateful to our OSDI shepherd Greg Minshall and the anonymous OSDI and TOCS reviewers, whose comments have helped to improve this paper. Thanks to Michael Svendsen for his help with the testbed configuration. 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.

Appendix A

API manual pages

`IOL_concat` - buffer aggregate concatenation

`IOL_control` - manipulation of control options

`IOL_create` - buffer aggregate allocation

`IOL_create_allocator` - allocator creation

`IOL_destroy` - buffer aggregate deallocation

`IOL_destroy_allocator` - allocator destruction

`IOL_duplicate` - buffer aggregate duplication

`IOL_gen_end` - generator-style traversal shutdown

`IOL_gen_nextRd` - generator-style traversal for reads

`IOL_gen_nextWr` - generator-style traversal for writes

`IOL_gen_seek` - moving to arbitrary locations in a generator context

`IOL_gen_start` - generator-style traversal setup

`IOL_gen_tell` - determining current position of generator-style traversal

`IOL_length` - buffer aggregate length query

`IOL_mmap` - map a file into memory

`IOL_read` - reading data from an object

`IOL_select` - buffer aggregate range selection

`IOL_split` - buffer aggregate splitting

`IOL_write` - writing data to an object

`IOL_write_destroy` - writing data to an object and deallocating the buffer aggregate

`IOL_create_allocator`, `IOL_destroy_allocator` - allocator creation and destruction

SYNOPSIS

```
IOL_Err IOL_create_allocator(allocator, impl_extra);
IOL_Allocator **allocator;
IOL_Specs *impl_extra;
```

```
IOL_Err IOL_destroy_allocator(allocator);
IOL_Allocator *allocator;
```

DESCRIPTION

An allocator defines a pool of memory with identical access control attributes. That is, all buffers handed out by an allocator can be accessed by the same set of domains. Allocators need to be created and destroyed only by applications that write to multiple data objects (data sinks) that have different access permissions. Most applications can use only the preallocated default allocator `IOL_DEF_ALLOC`.

`IOL_create_allocator` creates a new allocator object and returns a pointer to it in `allocator`. Any special properties for the new allocator can be marked using the implementation-defined `IOL_Specs` structure, which is pointed to be the `impl_extra` parameter. This parameter can be set to `NULL` if a standard allocator is desired.

`IOL_destroy_allocator` takes an allocator specified by `allocator` and makes it invalid. Any buffers that have already been created using this allocator are not affected. Only allocators created by calls to `IOL_create_allocator` can be destroyed, so attempting to destroy the default allocator is not allowed.

`IOL_create`, `IOL_destroy` - buffer aggregate allocation and deallocation

SYNOPSIS

```
IOL_Err IOL_create(allocator, aggregate, size);
IOL_Allocator *allocator;
IOL_Agg **aggregate;
size_t size;

IOL_Err IOL_destroy(aggregate);
IOL_Agg *aggregate;
```

DESCRIPTION

A buffer aggregate (`IOL_Agg`) is an abstract data type that encapsulates an ordered sequence of data bytes. In general, the data may be stored in a number of discontiguous buffers.

`IOL_create` creates a new buffer aggregate with size set to `size` bytes, and returns a pointer to the new aggregate in `aggregate`. The aggregate's initial data contents are unpredictable. If `size` is zero, no memory is allocated to the aggregate. The aggregate uses the allocator specified by `allocator` to satisfy its memory needs.

`IOL_destroy` destroys the buffer aggregate pointed to by `aggregate` and reclaims any underlying buffers, unless they are referenced by other aggregates. If active generator contexts reference the aggregate, the actual destruction is delayed until all relevant generator contexts become inactive. In this "zombie" period, no new contexts may be opened for this aggregate, nor may this aggregate be used for other operations.

IOL_concat, IOL_split, IOL_select, IOL_create_from_vec, IOL_duplicate, IOL_length - buffer aggregate manipulation operations

SYNOPSIS

```
IOL_Err IOL_concat(dst, src1, src2);
IOL_Agg **dst;
IOL_Agg *src1;
IOL_Agg *src2;

IOL_Err IOL_split(dst1, dst2, src, length);
IOL_Agg **dst1;
IOL_Agg **dst2;
IOL_Agg *src;
size_t length;

IOL_Err IOL_select(dst, src, start, length);
IOL_Agg **dst;
IOL_Agg *src;
size_t start;
size_t length;

IOL_Err IOL_create_from_vec(dst, vec, numItems);
struct IOL_Agg **dst;
struct IOL_Vec *vec;
int numItems;

IOL_Err IOL_duplicate(dst, src);
IOL_Agg **dst;
IOL_Agg *src;
```

```
size_t IOL_length(src);
IOL_Agg *src;
```

DESCRIPTION

`IOL_concat` creates a new buffer aggregate that contains the concatenation of the contents of the buffer aggregate pointed to by `src1` with the contents of the buffer aggregate pointed to by `src2`. A pointer to the new buffer aggregate is returned in `dst`.

`IOL_split` takes a source buffer aggregate pointed to by `src` and creates two new buffer aggregates containing the split contents of the original. Pointers to the new buffer aggregates are returned in `dst1` and `dst2`. The buffer aggregate pointed to by `*dst1` contains the prefix of the contents of `src` with a length of `length` bytes. The buffer aggregate pointed to by `*dst2` contains the postfix of the contents of `src` starting at (including) byte offset `length`. If `length` is greater than or equal to the length of the source aggregate, then `*dst1` will contain a copy of the source aggregate, and `*dst2` will contain a zero-length aggregate. If `length` is zero, then `*dst1` will contain a zero-length aggregate and `*dst2` will contain a copy of the source.

`IOL_select` creates a new buffer aggregate and returns a pointer to it in `dst`. The new buffer aggregate contains a subrange of the contents of `src` starting at (including) byte offset `start`, and having a length of `length` bytes. The value for `start` must be a non-negative number less than the length of the source aggregate, or else an error is returned. If the end of the range to be selected exceeds the end of the source aggregate, the range is truncated.

`IOL_create_from_vec` creates a new buffer aggregate and returns a pointer to it in `dst`. The elements of the buffer aggregate are the specified ranges from the vector of source aggregates, specified by `vec`. Conceptually, this operation is equivalent to concatenating the results of multiple `IOL_select` operations.

`IOL_duplicate` creates a new buffer aggregate and returns a pointed to it in `dst`.

The new buffer aggregate contains the same contents as `src`.

`IOL_length` returns the number of bytes of data contained in the buffer aggregate `src`.

IOL_gen_start, IOL_gen_nextRd, IOL_gen_nextWr, IOL_gen_end, IOL_gen_seek, IOL_gen_tell - examining and modifying buffer aggregate contents

SYNOPSIS

```
IOL_Err IOL_gen_start(src, ctxt);
IOL_Agg *src;
IOL_Context **ctxt;
```

```
IOL_Err IOL_gen_nextRd(ctxt, addr, length);
IOL_Context *ctxt;
const char **addr;
size_t *length;
```

```
IOL_Err IOL_gen_nextWr(ctxt, addr, length);
IOL_Context *ctxt;
char **addr;
size_t *length;
```

```
IOL_Err IOL_gen_end(ctxt);
IOL_Context *ctxt;
```

```
IOL_Err IOL_gen_seek(ctxt, pos);
IOL_Context *ctxt;
long pos;
```

```
IOL_Err IOL_gen_tell(ctxt, pos);
IOL_Context *ctxt;
long *pos;
```

DESCRIPTION

`IOL_gen_start` returns a pointer to a context object in `ctxt` that can be used to generate (in order) the <pointer,length> pairs for the buffers contained in the aggregate pointed to by `src`.

`IOL_gen_nextRd` and `IOL_gen_nextWr` return the <pointer,length> pair of the next (i-th) buffer, relative to the context object pointed to by `ctxt`. The starting address of the i-th buffer is returned in the `addr` parameter, and `length` is filled with the number of (contiguous) bytes of data stored in the i-th buffer. `IOL_gen_nextRd` allows reading of buffers only and can be applied to any aggregate. `IOL_gen_nextWr` allows writing of buffers and can be applied only to a newly allocated aggregate (with `IOL_create`) *before* any manipulation operations or `IOL_write` operations have been invoked using the new aggregate as an argument.

`IOL_gen_end` destroys the context object pointed to by `ctxt`.

`IOL_gen_seek` moves the "internal pointer" to the byte position specified by the `pos` parameter. Subsequent calls to the other generator functions will return the buffer at that position. The byte position is relative to the start of the buffer aggregate.

`IOL_gen_tell` returns the value of the "internal pointer" in the space specified by the `pos` parameter. The value of the pointer is the byte position of the first byte of the buffer returned by the next call to `IOL_gen_nextRd` or `IOL_gen_nextWr`.

IOL_read, IOL_write, IOL_write_destroy, IOL_write_vec, IOL_mmap, IOL_control - input and output operations

SYNOPSIS

```
IOL_Err IOL_read(fd, aggregate, size, *retSize);
int fd;
IOL_Agg **aggregate;
size_t size; size_t *retSize;

IOL_Err IOL_write(fd, aggregate, *retSize);
int fd;
IOL_Agg *aggregate; size_t *retSize;

IOL_Err IOL_write_destroy(fd, aggregate, *retSize);
int fd;
IOL_Agg **aggregate; size_t *retSize;

IOL_Err IOL_write_vec(fd, vec, numItems, retSize);
int fd;
struct IOL_Vec *vec;
int numItems;
size_t *retSize;

IOL_Err IOL_mmap(addr, len, prot, flags, fd, off);
addr_t *addr;
size_t len;
int prot;
int flags;
int fd;
size_t off;
```

```
IOL_Err IOL_control(fd, flags);
int fd;
int flags;
```

DESCRIPTION

`IOL_read` reads up to `size` bytes of data from the data source referred to by file descriptor `fd`, and creates a new buffer aggregate. A pointer to the new aggregate is returned in `aggregate`. The value returned in `retSize` is the number of bytes actually read.

`IOL_write` writes the contents of the buffer aggregate pointed to by `aggregate` to the data sink referred to by file descriptor `fd`. `IOL_write_destroy` behaves like `IOL_write`, but destroys `aggregate` after the write has completed. If all of the data is not written (a short write), then `aggregate` is not destroyed. The number of bytes written is returned in `retSize`.

`IOL_write_vec` performs the IO-Lite equivlant of the Unix `writev` system call. As input, it takes a vector of buffer aggregates with ranges describing the part of each aggregate to be written. This information is passed in the `vec` array. Like `IOL_write`, the number of bytes written is returned in `retSize`.

`IOL_mmap` maps the specified file into memory at the specified address. The arguments passed to `IOL_mmap` are the same as those for most Unix versions of `mmap`. `IOL_control` allows implementation-defined control options to be set for the file descriptor.

Concurrent `IOL_read` and `IOL_write` operations to/from the same object are serializable. In other words, the effect of an `IOL_write` operation is atomic with respect to `IOL_read` and other `IOL_write` operations. However, `IOL_read` and `IOL_write` operations are *not* atomic with respect to concurrent accesses through a memory mapping (mmap) of the same data object.

An `IOL_read` with concurrent store operations through a memory mapping of the same portion of a data object is *not* guaranteed to be serializable. The following assertions can be made about the effect of an `IOL_read` operation from a region of an object that is concurrently modified through a memory mapping (mmap):

The data returned by `IOL_read` is guaranteed to reflect *all* store operations that occurred *before* the begin of the `IOL_read` operation and *none* of the changes that occur *after* the `IOL_read` returns. The data returned by `IOL_read` may reflect the effects of an arbitrary subset of the store operations that occur *during* the `IOL_read` operation.

An `IOL_write` operation with concurrent accesses through a memory mapping of the same portion of a data object is *not* necessarily serializable. The following assertions can be made about the effect of an `IOL_write` operation to a region of an object that is concurrently accessed through a memory mapping (mmap):

Load operations from the memory mapping are guaranteed to reflect *none* of the effects of the `IOL_read` *before* the IOL_READ begins, and *all* of the effects of the `IOL_write` *after* the `IOL_write` completes. *While* the `IOL_write` operation is in progress, load operations may reflect an arbitrary subset of the effects of the concurrent `IOL_write`.

Effects of store operations to the memory mapping that occur *before* the start of an `IOL_write` are guaranteed to be superseded by the effects of the `IOL_write`. Effects of store operations to the memory mapping that occur *after* the `IOL_write` completes are guaranteed to supersede the effects of the `IOL_write`. No assumptions can be made about the effect of store operations that occur *during* an `IOL_write` to the same portion of the data object.

Bibliography

[1] The common gateway interface. http://hoohoo.ncsa.uiuc.edu/cgi/.

[2] FastCGI specification. http://www.fastcgi.com/.

[3] SpecWeb96 specification. http://www.spec.org/osg/web96/.

[4] D. Andresen et al. SWEB: Towards a Scalable WWW Server on MultiComputers. In *Proccedings of the 10th International Parallel Processing Symposium*, Apr. 1996.

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

[6] 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.

[7] G. Banga, F. Douglis, and M. Rabinovich. Optimistic Deltas for WWW Latency Reduction. In *Proceedings of the 1997 Usenix Technical Conference*, Jan. 1997.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 267–284, Copper Mountain, CO, Dec. 1995.

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

[14] J. C. Brustoloni. Interoperation of copy avoidance in network and file I/O. In *Proceedings of the IEEE Infocom Conference*, pages 534–542, New York, Mar. 1999.

[15] J. C. Brustoloni and P. Steenkiste. Effects of buffering semantics on I/O performance. In *Proc. 2nd USENIX Symp. on Operating Systems Design and Implementation*, pages 277–291, Seattle WA (USA), Oct. 1996.

[16] J. C. Brustoloni and P. Steenkiste. User-level protocol servers with kernel-level performance. In *Proceedings of the IEEE Infocom Conference*, pages 463–471, San Francisco, Mar. 1998.

[17] P. Cao and E. Felten. Implementation and performance of application-controlled file caching. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation*, pages 165–177, 1994.

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

pages 193–206, Monterey, CA, Dec. 1997.

[19] 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.

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

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

[22] C. D. Cranor and G. M. Parulkar. The UVM virtual memory system. In *Proceeding of the Usenix 1999 Annual Technical Conference*, pages 117–130, Monterey, CA, June 1999.

[23] 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.

[24] 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.

[25] 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.

[26] P. Danzig, R. Hall, and M. Schwartz. A case for caching file objects inside internetworks. In *Proceedings of the SIGCOMM '93 Conference*, Sept. 1993.

[27] 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.

[28] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 251–266, Copper Mountain, CO, Dec. 1995.

[29] K. Fall and J. Pasquale. Exploiting in-kernel data paths to improve i/o throughput and cpu availability. In *Proceedings of the 1993 Winter Usenix Conference*, Jan. 1993.

[30] 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.

[31] 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.

[32] 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 *Proc. 2nd Global Internet Conf.*, Nov. 1997.

[33] 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.

[34] G. Hunt, E. Nahum, and J. Tracey. Enabling content-based load distribution for scalable services. Technical report, IBM T.J. Watson Research Center, May 1997.

[35] N. C. Hutchinson and L. L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*,

17(1):64–76, Jan. 1991.

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

[37] IBM Corporation. IBM interactive network dispatcher. http://www.ics.raleigh.ibm.com/ics/isslearn.htm.

[38] Microsoft Corporation ISAPI Overview. http://www.microsoft.com/msdn/sdk/platforms/doc/sdk/internet/src/isapimrg.htm.

[39] A. Iyengar and J. Challenger. Improving web server performance by caching dynamic data. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, Monterey, CA, Dec. 1997.

[40] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceno, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. MacKenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, pages 52–65, San Malo, France, Oct. 1997.

[41] 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.

[42] T. M. Kroeger, D. D. Long, and J. C. Mogul. Exploring the bounds of Web latency reduction from caching and prefetching. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, Monterey, CA, Dec. 1997.

[43] 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.

[44] 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.

[45] G. R. Malan, F. Jahanian, and S. Subramanian. Salamander: A push-based distribution substrate for Internet applications. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, Monterey, CA, Dec. 1997.

[46] 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.

[47] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX '93 Winter Conference*, pages 259–269, Jan. 1993.

[48] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System.* Addison-Wesley Publishing Company, 1996.

[49] 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.

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

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

[52] Netscape Server API. http://www.netscape.com/newsref/std/server_api.html.

[53] V. S. Pai. Buffer and cache management in scalable network servers. Technical Report 99-349, Department of Computer Science, Rice University, 1999.

[54] 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.

[55] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proceeding of the Usenix 1999 Annual Technical Conference*, pages 199–212, Monterey, CA, June 1999.

[56] V. S. Pai, P. Druschel, and W. Zwaenepoel. I/O-Lite: A unified I/O buffering and caching system. In *Proc. 3rd USENIX Symp. on Operating Systems Design and Implementation*, pages 15–28, Feb. 1999.

[57] J. Pasquale, E. Anderson, and P. K. Muller. Container Shipping: Operating system support for I/O-intensive applications. *IEEE Computer*, 27(3):84–93, Mar. 1994.

[58] Resonate Inc. Resonate dispatch. http://www.resonateinc.com.

[59] M. Seltzer and J. Gwertzman. The Case for Geographical Pushcaching. In *Proceedings of the 1995 Workshop on Hot Topics in Operating Systems*, 1995.

[60] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. 2nd USENIX Symp. on Operating Systems Design and Implementation*, pages 213–227, Seattle, WA, Oct. 1996.

[61] D. L. Tennenhouse. Layered multiplexing considered harmful. In H. Rudin and R. Williamson, editors, *Protocols for High-Speed Networks*, pages 143–148, Amsterdam, 1989. North-Holland.

[62] 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.

[63] G. Wright and W. Stevens. *TCP/IP Illustrated Volume 2*. Addison-Wesley, Reading, MA, 1995.

[64] 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.

[65] D. J. Yates, E. M. Nahum, J. F. Kurose, and D. Towsley. Networking support for large scale multiprocessor servers. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, Philadelphia, Pennsylvania, May 1996.

[66] B. Yoshikawa et al. Using Smart Clients to Build Scalable Services. In *Proceedings of the 1997 Usenix Technical Conference*, Jan. 1997.

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