IO-Lite: A unified I/O buffering and caching system

Vivek S. Pai Peter Druschel Willy Zwaenepoel
Rice University
Houston, TX 77005

Abstract

This paper presents the design, implementation, and evaluation 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 filesystem, the file cache, and the network subsystem to share a single physical copy of the data safely and concurrently. Protection and security are maintained through a combination of access control and read-only sharing. The various subsystems use (mutable) buffer aggregates to access the data according to their needs. IO-Lite eliminates all copying and multiple buffering of I/O data, and enables various cross-subsystem optimizations. Experiments with a Web server on IO-Lite show performance improvements between 40 and 80% on real workloads.

1 Introduction

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 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 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 size of the document cache for a given main memory size. A reduced cache size, however, likely causes an increased rate of disk accesses and reduced throughput. Finally, lack of support for application-specific cache replacement policies and optimizations like TCP checksum caching 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 share a single physical copy of the data safely and concurrently. IO-Lite achieves this goal by storing buffered I/O data in immutable buffers, whose locations in 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).

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.

1.1 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.

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 network subsystem's send buffers of one or more connections. 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. Also, server applications cannot exercise customized file cache replacement policies.

The outline of the rest of the paper is as follows: Section 2 presents the design of IO-Lite and discusses its operation in a Web server application. Section 3 describes a prototype implementation in a BSD UNIX system. A quantitative evaluation of IO-Lite is presented in Section 4, including performance results with a Web server on real workloads. In Section 5, we present a qualitative discussion of IO-Lite in the context of related work. Section 6 offers conclusions.

2 IO-Lite Design

2.1 Principles: Immutable Buffers and Buffer Aggregates

IO-Lite defines an explicit abstraction for I/O data, called *buffer aggregate*. Buffer aggregates have access semantics appropriate for a unified buffer/caching system. 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.

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. For example, the filesystem cache, applications that access a given file, and the network subsystem that transmits part of that file 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 data buffers inside the buffer aggregate abstraction. Buffer aggregates are instances of an abstract data type (ADT) that represent I/O data.

[1] Some systems avoid this data copying in a transparent manner under certain conditions.

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

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, splitting, and mutating data contained in I/O buffers.

It is important to note that 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 2.7.

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 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.

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* [9] to support safe concurrent sharing. Copy-free I/O facilities that only allow *sequential* sharing [17, 6], where only one protection domain has access to a given buffer at any time, are not suitable for use in caching I/O systems.

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 [11], 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 can be implicitly 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 before the first buffer is deallocated by the receiver(s). Otherwise, buffers can be recycled, and the transfer performance approaches that of shared memory.

IO-Lite ensures access control and protection at the granularity of processes. That is, 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. This is trivial in most cases, except when an incoming packet arrives at a network interface. Here, the network driver must perform an *early demultiplexing* operation to determine the packet's destination. Incidentally, early demultiplexing has been identified by many researchers as a necessary feature for efficiency and quality of service in high-performance networks [19]. With IO-Lite, early demultiplexing is necessary for best performance.

Figure 1 depicts the relationship between VM pages, buffers, and buffer aggregates. IO-Lite buffers are allocated in a region of the virtual address space

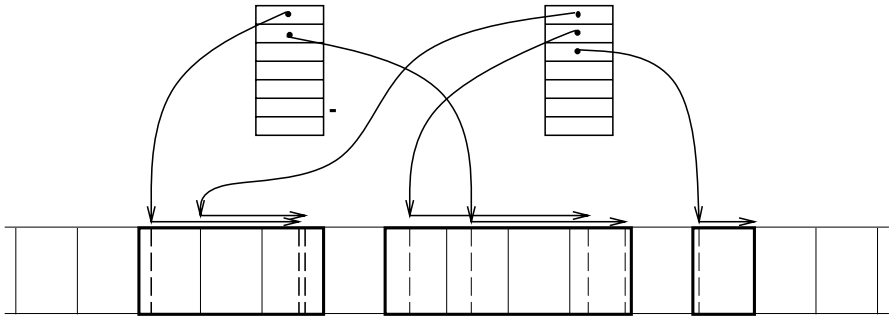

Figure 1: Aggregate buffers and slices

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; that is, 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.

2.3 IO-Lite and Applications

To take *full* advantage of IO-Lite, application programs can 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 the technical report [16].

IOL_read and IOL_write form the core of the interface (see Figure 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_read_pool operation allows the application to additionally specify an IO-Lite 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 attributes can use this operation. The IOL_write operation replaces the data in an external data object with the contents of the buffer aggregate passed as 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 affected by a concurrent IOL_write operation on the same file descriptor. The data returned by a 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 buffer pools. The (IOL_Agg) buffer aggregate abstract data type supports a number of operations for creation, destruction, duplication, concatenation and truncation as well as data access.

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

```
size_t IOL_read(int fd, IOL_Agg **aggregate, size_t size);
size_t IOL_read_pool(int fd, IOL_Pool *pool, IOL_Agg **aggregate, size_t size);
size_t IOL_write(int fd, IOL_Agg *aggregate);
```

Figure 2: IO-Lite I/O API

2.4 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 tuples of the form *<file-id, offset>* to buffer aggregates, which contain an extent of the corresponding 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; however, they persist as long as other references to them exist.

For example, assume that a IOL_read operation of a cached file is followed by a 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.

2.5 IO-Lite and the Network

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

Limited modifications are required to network device drivers. As explained in Section 2.2, network interface drivers that allocate IO-Lite buffers for input data must specify the I/O stream for which the buffer is to be used, since this stream implies the ACL fo the data stored in the buffer. Network interface drivers must determine this information from the headers of incoming packets using a packet filter (early demultiplexing.)

2.6 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. This concerns 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 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 fur-

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

ther 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. [7].

2.7 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 change values, then the naive approach of copying the entire object would result in partial 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 system is forced to copy the modified page 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.

2.8 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 was 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 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 optimizations such as this, 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. This allows optimizations such as the Internet checksum caching.

2.9 Operation in a Web Server

In this section, we describe the operation of IO-Lite in a Web server as an example. 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.

Copying occurs as part of the reading of data 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, 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 data from primary files and newly generated data. 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 copying 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 exist, since copying costs can dominate the service time in this case. Moreover, the elimination of multiple copying frees up valuable memory resources. This benefits the file cache size and hit rate, thus further increasing server performance.

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 the IO-Lite API and use buffer aggregates to synthesize dynamic content.

A quantitative evaluation of IO-Lite in the context of a Web server follows in Section 4.

3 Implementation

This section gives an overview of the implementation of the IO-Lite prototype system in a 4.4BSD derived UNIX kernel.

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. This approach maintains 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 [14]. 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. These operations are implemented on top of IOL_read and IOL_write, where a data copy operation is used to move data between application buffer and IO-Lite buffers.

VM System: Adding IO-Lite does not require any significant changes to the BSD VM system [14]. 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.

4 Performance

In this section, we evaluate the performance of a prototype IO-Lite 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.

All experiments use a server system based on a 333Mhz Pentium II PC equipped with 128MB of main memory and five network adaptors 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*. Flash is an event-driven HTTP server with support for CGI. To the best of our knowledge, Flash is among the fastest HTTP server currently available. *Flash-Lite* is a slightly modified version of Flash that uses the IO-Lite API.

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 [8].

For comparison, we also present performance results with Apache version 1.3.1, a widely used Web server [3]. 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.

Flash is an aggressively optimized, experimental Web server; it reflects the best in Web server per-

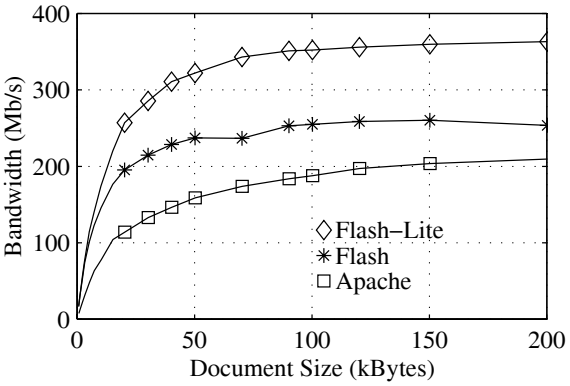

Figure 3: HTTP

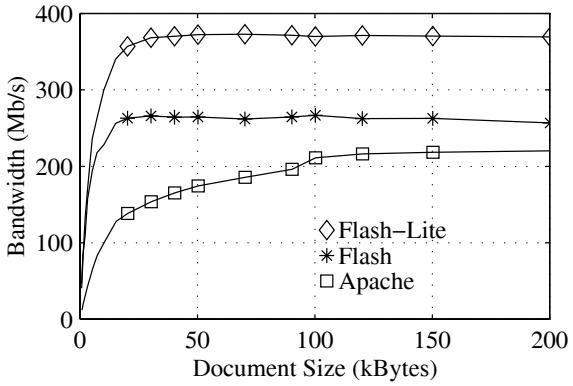

Figure 4: Persistent HTTP

formance 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. All Web servers were configured use a TCP socket send buffer size of 64KBytes; access logging was disabled.

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 [4]. 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 3 shows the output bandwidth of the various Web server 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 confirms that our aggressive Flash server outperforms the already fast Apache server.

Flash using IO-Lite (Flash-Lite) delivers a bandwidth increase of up to 43% over Flash and up to 137% 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).

4.1 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 4.

With persistent connections, the request rate for small files (less then 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.

4.2 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 processes 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 2.9.

We performed an experiment to evaluate how IO-Lite affects the performance of dynamic content gen-

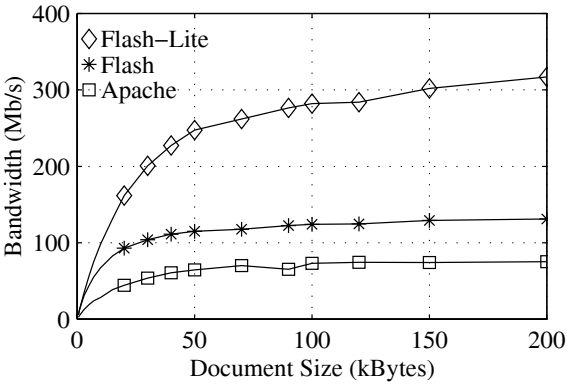

Figure 5: HTTP/Fast CGI

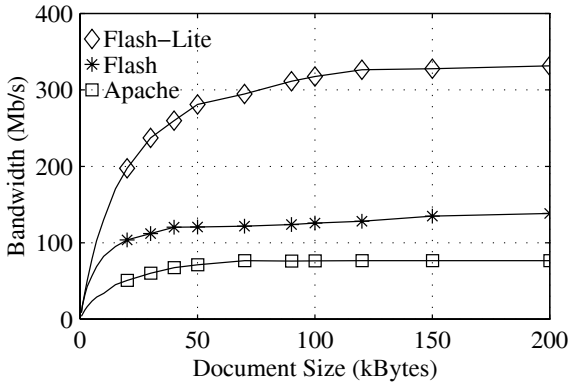

Figure 6: P-HTTP/Fast CGI

eration using Fast CGI programs. A test CGI program, when receiving a request, sends a "dynamic" document of a given size from its memory to the server process via the UNIX pipe; the server transmits the data on the client's connection. The results of these experiments are shown in Figure 5.

The bandwidth of the Flash and Apache servers is roughly half their corresponding bandwidth on static documents. This 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 than static files with Flash.

Figure 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 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 [15, 12] 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.

4.3 Performance On Real Workload

The previous experiments use an artificial workload. In particular, they use a set of requested documents that fits into the server's main memory cache. As a result, these experiments only quantify the increase in performance due to the elimination of CPU overhead with IO-Lite. They do not demonstrate possible secondary performance benefits due to the increased availability of main memory that results from IO-Lite's elimination of double buffering. Increasing the amount of available memory allows a larger set of documents to be cached, thus increasing the server cache hit rate and performance. Finally, since the cache is not stressed in these experiments, possible performance benefits due to the customized file cache replacement policy used in Flash-Lite are not exposed.

	Apache	Flash	Flash-Lite
Requests/sec	524	617	866
Ratio	1.0	1.18	1.65

Table 1: Rice Trace

To measure the combined impact of IO-Lite on the performance of a Web server under realistic workload conditions, we performed experiments where our experimental server is driven by a workload derived from server logs of an actual Web server. We use logs from Rice University's Computer Science departmental Web server. Only requests for static documents were extracted from the logs.

Table 1 show the relative performance in requests/sec of Flash-Lite, Flash, and Apache on the Rice CS department trace. Flash exceeds the throughput of Apache by 18% on this trace. Flash-Lite gains 65% throughput over Apache and 40% over Flash. This demonstrates the effectiveness of IO-Lite under realistic workload conditions, where the set of requested documents exceeds the cache size and disk accesses occur. The average request size in this trace is about 17KBytes.

4.4 WAN Effects

Our experimental testbed uses a low-delay LAN to connect a relatively small number of the clients

to the experimental server. This leaves a significant aspect of real Web server performance unevaluated, namely the impact of wide-area network delays and large numbers of clients [4]. 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 significantly. This makes it increasingly important to eliminate double buffering.

The BSD UNIX network subsystem dynamically allocates mbufs to hold data in socket buffers. When the server is contacted by a large number of clients concurrently and the server transmits on each connection an amount of data equal or larger than T_{ss}, then the system may be forced to allocate sufficient mbufs to hold T_{ss} bytes for each connection. Moreover, in FreeBSD and other BSD-derived system, the size of the mbuf pools is never decreased. That is, once the mbuf pool has grown to a certain size, its memory is permanently unavailable for other uses, such as the file cache.

To quantify this effect, we performed an experiment where an increasing number of "slow" background clients contact the server. These clients request a document, but are slow to read the data from their end of the TCP connection, which has a small receive buffer (2KB). This forces the server to buffer data in its socket send buffers and simulates the effect of WAN connections on the server.

As the number of clients increases, more memory is used to hold data in the server's socket buffers, increasing memory pressure and reducing the size of the file cache. 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}.

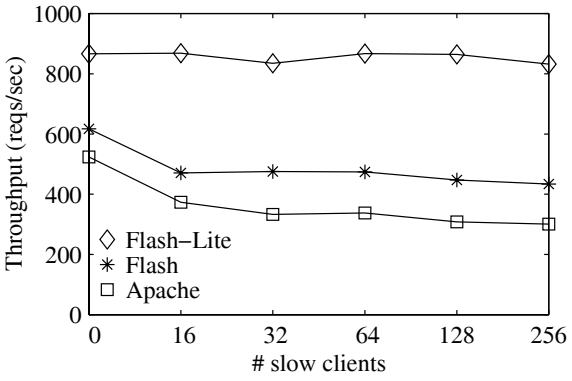

Figure 7: Throughput vs. #clients

Figure 7 shows the performance of Apache, Flash and Flash-Lite as a function of the number of slow clients contacting the server. As expected, Flash-Lite remains unaffected by the number of slow clients contacting the server, up to experimental noise. Apache suffers up to 42% and Flash up to 30% throughput loss as the number of clients increases, reducing the available cache size. For 16 slow clients and more, Flash-Lite is close to 80% faster than Flash; for 32 slow clients and more, Flash-Lite is 150% faster than Apache.

The results confirm IO-Lite's ability to eliminate double buffering in the network subsystem. This effect gains in importance both as the number of concurrent clients and the setting of T_{ss} increases. Future increases in Internet bandwidth will require larger T_{ss} settings to achieve good network utilization.

4.5 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 and some new programs were written to use IO-Lite. We modified GNU grep, wc, cat, and the GNU gcc compiler chain (compiler driver, C preprocessor, C compiler, and assembler). Figure 8 depicts the results obtained with grep, wc, and permute. The "wc" 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 8 character words in a 80 character string. Its output ($10! * 80 = 290304000$ bytes) is piped into the "wc" program. The "grep" bar refers to a run of the GNU grep program on the same file

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

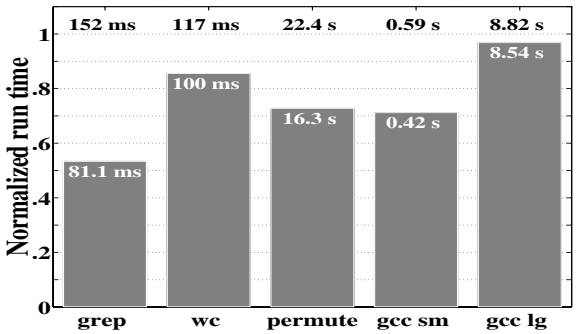

Figure 8: Various application runtimes

used for the "wc" program, but the file is piped into wc from cat instead of being read directly from disk.

Improvements in runtime of approximately 15% result from the use of IO-Lite for wc, since it reads cached files. When a file is read from the IO-Lite file cache, each page of the cached file must be mapped into the application's address space. For the "permute" program the improvement is more significant (24%). The reason is that a pipeline is involved in the latter program. Whenever local interprocess communication occurs, the IO-Lite implementation can recycle buffers, avoiding all VM map operations. Finally, in the "grep" case, the overhead of multiple copies is eliminated, so the IO-Lite version is able to eliminate 3 copies (one due to "grep", and two due to "cat").

The gcc compiler chain was converted mainly to determine if there were benefits from IO-Lite for more compute-bound applications and to stress the IO-Lite implementation. We expected that a compiler is too compute-intensive to benefit substantially from I/O performance improvements. Rather than modify the entire program, we simply replaced the stdio library with a version that uses IO-Lite for communication over pipes. Interestingly, converting the compiler to use IO-Lite actually led to a measurable performance improvement. The improvement is mainly due to the fact that IO-Lite allows efficient communication through pipes. Although the standard gcc has an option that causes the use of pipes instead of temporary files for communication between the compiler's various stages, various inefficiencies in the handling of pipes actually caused a significant slowdown, so the baseline gcc numbers used for comparison are for gcc running without pipes. Since IO-Lite can handle pipes very efficiently, unexpected performance improvements resulted from its use. The "gcc sm" and "gcc lg" bars refer to compiles of a 1200 Byte and a 206 KByte file, respectively.

The "grep" and "wc" programs read their input sequentially, and were converted to use the IO-Lite API. 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 stdio library. The preprocessor (cpp) uses mmap to read its input.

5 Related Work

This section discusses related work. In particular, 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. In typical use, this interface and its semantics can not generally be implemented without physical copying of data.

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 file cache and 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 CGI program and the server. Having the server and the CGI program share memory-mapped files for IPC requires ad-hoc synchronization and sacrifices the simplicity of socket-

based communication.

Transparent Copy Avoidance: In principle, copy avoidance and single buffering can be accomplished transparently, through the use of page remapping and copy-on-write. Well-known difficulties with this approach are alignment restrictions and the overhead of VM data structure manipulations.

Dealing with VM alignment restrictions leads to the idea of *input alignment* as used in the *emulated copy* technique in Genie [6]. Here, the idea is to try and align system buffers with the application's data buffers, allowing page swapping even if application buffers are not page-aligned. To allow proper alignment of the system buffers, the application's read operation must be posted before the data arrives in main memory. If this is not possible, the application can query the kernel for the page offset of the system's buffers, and align its buffers accordingly. Input alignment is not transparent to applications. Emulated copy achieves transparency only for one side of an IPC channel. While this is well-suited for communication between an application and an OS kernel/server, it cannot transparently support general copy-free IPC among application processes.

IO-Lite does not attempt to provide transparent copy avoidance. I/O-intensive applications can be written/modified to use the IO-Lite API. Legacy applications with less stringent performance requirements can be supported in a backward-compatible fashion at the cost of a copy operation, as in conventional systems. By giving up transparency and in-place modifications, IO-Lite can support general copy-free I/O, including IPC and data paths that involve the file cache.

Since transparent copy-avoidance approaches cannot allow concurrent sharing, they are not suitable for a unified buffering/caching mechanism, as cached file data cannot be concurrently shared. This would lead to multiple buffering in a Web server. Moreover, the lack of general copy-free IPC hampers the performance of CGI programs.

Copy Avoidance with Handoff Semantics: The *Container Shipping* (CS) I/O system [17] and Thadani and Khalidi [20] 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 permits 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 an application loses access to a buffer that it passed as an argument to a write operation, an explicit physical copy is necessary if the application needs 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. This demonstrates that the lack of support for concurrent sharing prevents an effective integration of 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 the double-buffering problem. 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 [5, 10, 13, 18] 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.

In contrast to extensible kernels, IO-Lite is directly applicable to existing general-purpose operating systems and provides an application-independent scheme for addressing the I/O bottleneck. Our approach avoids the complexity and the overhead of new safety provisions required by extensible kernels. It also relieves the implementors of servers and applications from having to write OS-specific kernel extensions.

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.

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.

6 Conclusion

This paper 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 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.

References

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

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

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

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

[5] 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*, Copper Mountain, CO, Dec. 1995.

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

[7] 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.

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

[9] 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.

[10] 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*, Copper Mountain, CO, Dec. 1995.

[11] 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.

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

[13] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. B. no, 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*, San Malo, France, Oct. 1997.

[14] 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.

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

[16] V. S. Pai, P. Druschel, and W. Zwaenepoel. I/O-Lite: A unified I/O buffering and caching system. Technical Report 98-331, Department of Computer Science, Rice University, 1998.

[17] 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.

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

[19] 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.

[20] 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.