IO-Lite: A Unified I/O Buffering and Caching System

Vivek S. Pai and Peter Druschel and Willy Zwaenepoel
Rice University

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, the interprocess communication system, the filesystem, the file cache, and the network subsystem to safely and concurrently share a single physical copy of the data. Protection and security are maintained through a combination of access control and read-only sharing. IO-Lite eliminates all copying and multiple buffering of I/O data, and enables various cross-subsystem optimizations. Experiments with a Web server show performance improvements between 40 and 80% on real workloads as a result of IO-Lite.

Categories and Subject Descriptors: D.4.4 [**Software**]: Operating Systems—*Communications Management*; D.4.8 [**Software**]: Operating Systems—*Performance*

General Terms: Communications, Management, Performance

Additional Key Words and Phrases: zero-copy,networking,caching,I/O buffering

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 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

Address: Computer Science Department, 6100 Main, Houston, TX 77005

support for application-specific cache replacement policies [Cao and Felten 1994] and optimizations like TCP checksum caching [Kaashoek et al. 1997] 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 the paper is as follows: Section 2 discusses the design of the buffering and caching systems in UNIX and their deficiencies. Section 3 presents the design of IO-Lite and discusses its operation in a Web server application. Section 4 describes our prototype IO-Lite implementation in FreeBSD. A quantitative evaluation of IO-Lite is presented in Section 5, including performance results with a Web server on real workloads. In Section 6, we present a qualitative discussion of IO-Lite in the context of related work, and we conclude in Section 7.

2. 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.

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.

3. IO-LITE DESIGN

3.1 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 3.2. 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.

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

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 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 3.8.

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.

3.2 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* [Druschel and Peterson 1993] to support safe concurrent sharing. Copy-free I/O facilities that only allow *sequential* sharing [Brustoloni and Steenkiste 1996; Pasquale et al. 1994] 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 [Hutchinson and

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

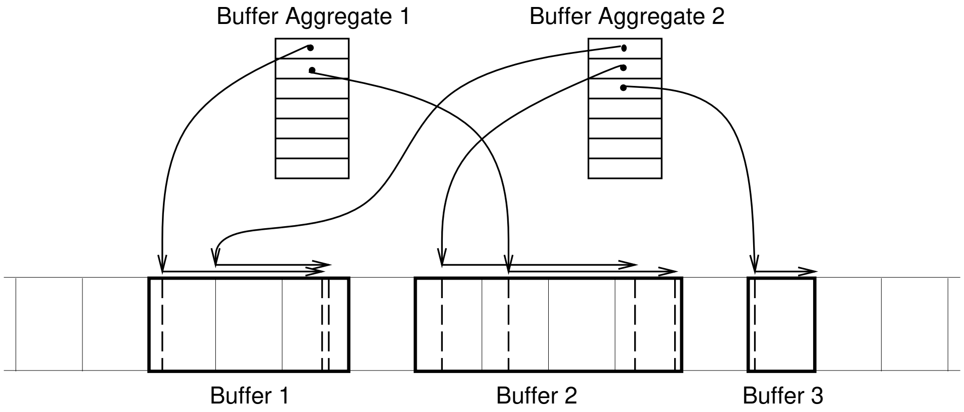

Fig. 1. Aggregate buffers and slices – IO-Lite allocates contiguous buffers in virtual memory. Applications access these buffers through data structures called buffer aggregates, which contain ordered tuples of the form <address, length>. Each tuple refers to a subrange of memory called a *slice*.

Peterson 1991], 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.

3.3 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 3.6.

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

Fig. 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.

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 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.

3.4 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 our technical report [Pai 1999].

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_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 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.

3.5 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.

3.6 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 3.3, 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 [McCanne and Jacobson 1993], 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 [Tennenhouse 1989]. With IO-Lite, as with fbufs [Druschel and Peterson 1993], early demultiplexing is necessary for best performance.

3.7 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 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

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

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. [Cao and Felten 1994].

3.8 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.

3.9 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.

3.10 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.

4. 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 [McKusick et al. 1996]. 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 [McKusick et al. 1996]. 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.

Access Control: To reduce the number of operations and the amount of bookkeeping 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.

5. 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.

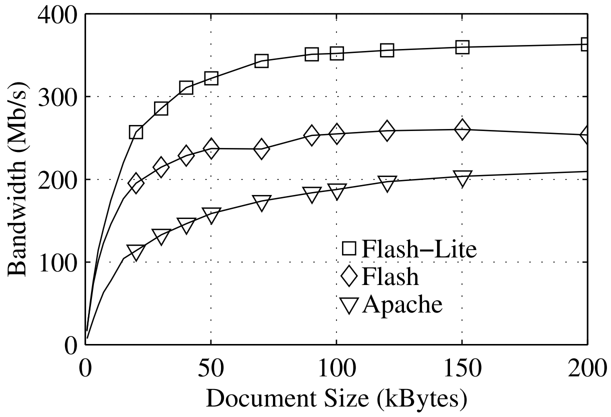

Fig. 3. HTTP Single File Test – All clients request the same file from the server, and we observe the aggregate bandwidth generated. This test provides the best-case performance of the servers using nonpersistent connections.

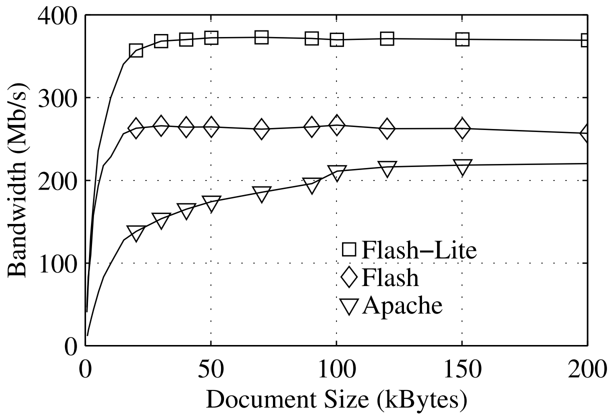

Fig. 4. Persistent HTTP Single File Test – Rather than creating a new TCP connection for each transfer, each client requests multiple transfers on an existing connection. Removing the TCP setup/teardown overhead allows even small transfers to achieve significant benefit.

To fully expose the performance bottlenecks in the operating system, we use a high-performance in-house Web server called *Flash* [Pai et al. 1999a]. 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 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 [Cao and Irani 1997]. 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 [Apache]. 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%.

5.1 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 [Banga and Druschel 1999]. 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 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 re-

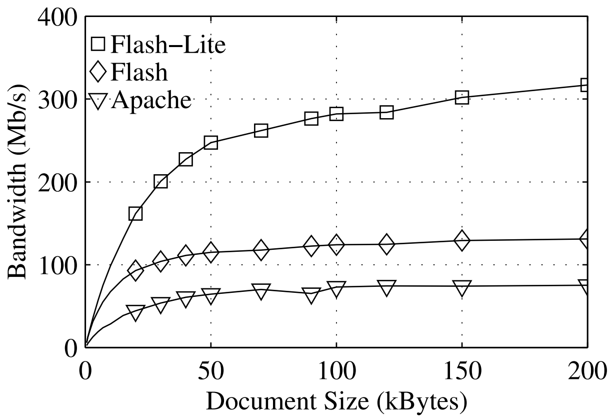

Fig. 5. HTTP/FastCGI – Each client requests data from a persistent CGI application spawned by the server. In standard UNIX, the extra copying between the server and the CGI application becomes a significant performance bottleneck.

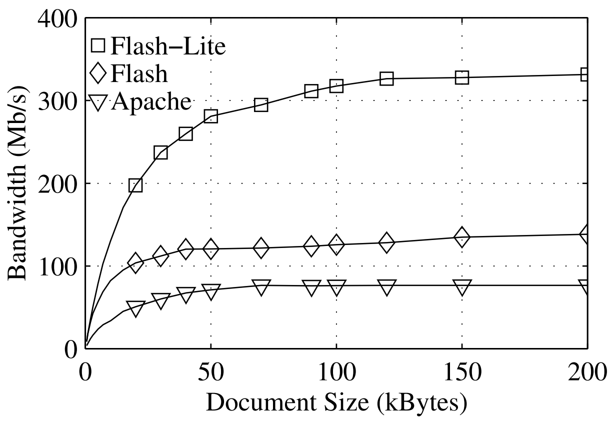

Fig. 6. Persistent-HTTP/FastCGI – Each client reuses the TCP connection for multiple CGI requests. Flash and Apache do not receive significant benefits because their performance is limited by the copying between the server and CGI application.

sulting 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).

5.2 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 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.

5.3 CGI Programs

An area where IO-Lite promises particularly substantial benefits is CGI programs. When compared to the original CGI 1.1 standard [CGI], the newer FastCGI interface [Fas] 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 3.10.

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 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 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 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 [ISAPI ; NSAPI] 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.

5.4 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

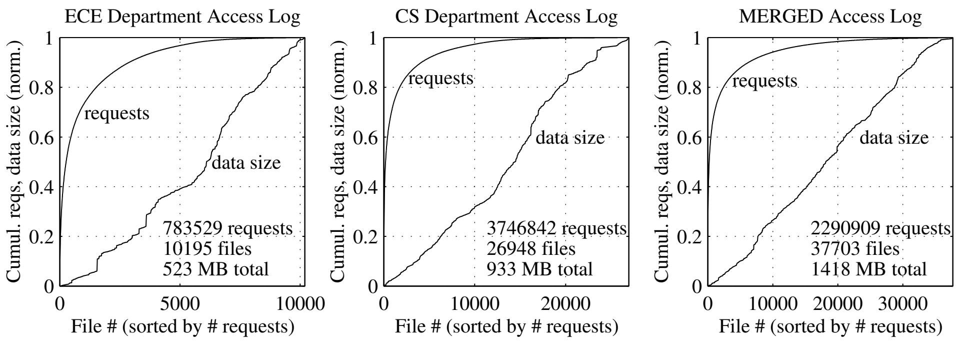

Fig. 7. Trace characteristics – These graphs show the cumulative distribution functions for the data size and request frequencies of the three traces used in our experiments. For example, the 5000 most heavily requested files in the ECE access constituted 39% of the total static data size (523 MB) and 95% of all requests.

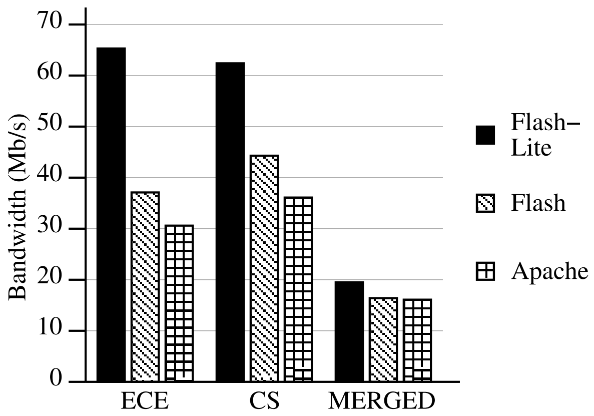

Fig. 8. Overall trace performance – In each test, 64 clients were used to replay the entries of the trace. New requests were started immediately after previous requests completed.

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 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.

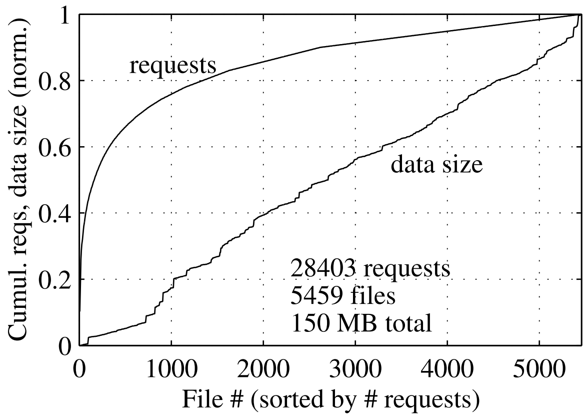

Fig. 9. 150MB subtrace characteristics – This graph presents the cumulative distribution functions for the data size and request frequencies of the subtrace we used. For example, we see that the 1000 most frequently requested files were responsible for 20% of the total static data size but 74% of all requests.

Figure 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 5.1 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.

5.5 Subtrace Experiments

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 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 [Spe] 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.

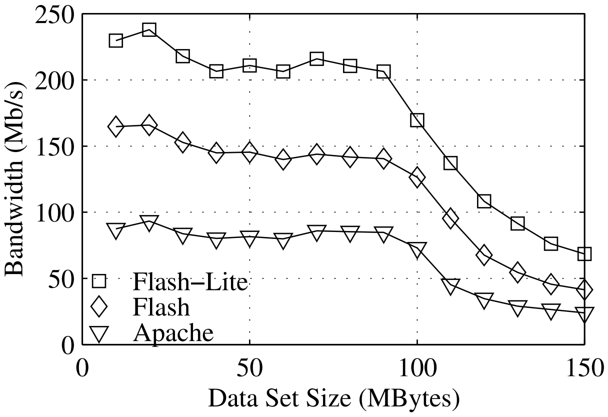

Fig. 10. MERGED subtrace performance – This graph shows the aggregate bandwidth generated by 64 clients as a function of data set size. Flash outperforms Apache due to aggressive caching, while Flash-Lite outperforms Flash due to a combination of copy avoidance, checksum caching, and customized file cache replacement.

Figure 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.

5.6 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 5.4.

The results of these additional tests are shown in Figure 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

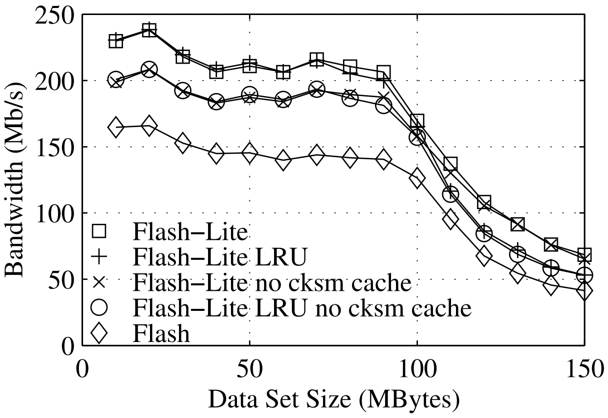

Fig. 11. Optimization contributions – To quantify the effects of the various optimizations present in Flash-Lite and IO-Lite, two file cache replacement policies are tested in Flash-Lite, while IO-Lite is run with and without checksum caching.

multiple buffering become much more significant, as shown in the next section.

5.7 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 [Banga and Druschel 1999]. 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}.

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

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

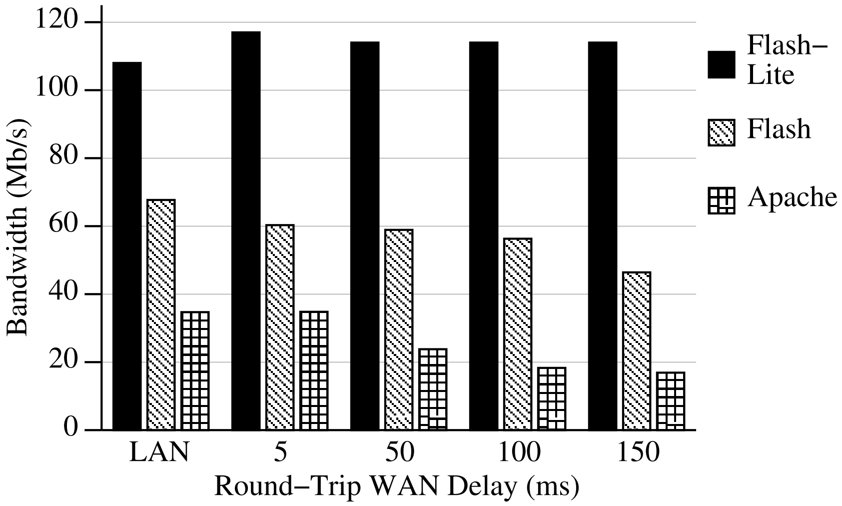

Fig. 12. Throughput vs. network delay – In a conventional UNIX system, as the network delay increases, more network buffer space is allocated, reducing the memory available to the filesystem cache. The throughput of Flash and Apache drop due to this effect. IO-Lite avoids multiple buffering, so Flash-Lite is not affected.

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 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 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.

5.8 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

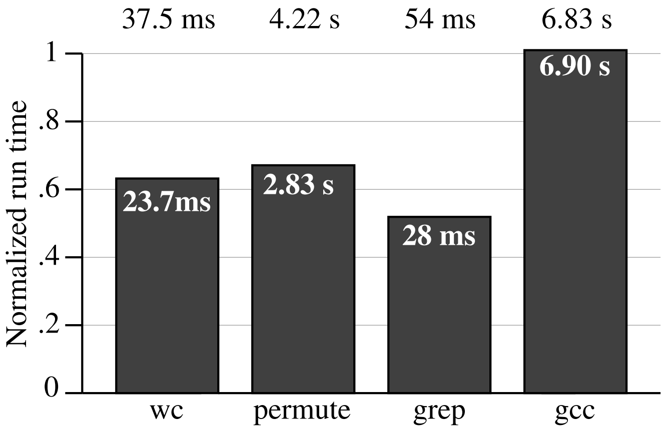

Fig. 13. Various application runtimes – The time above each bar indicates the run time of the unmodified application, while the time at the top of the bar is for the application using IO-Lite.

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. 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 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

[5]Our application performance results in this paper differ from our earlier results[Pai et al. 1999b]. 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

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.

6. 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 [CGI] 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 [Brustoloni 1999; Brustoloni and Steenkiste 1996; Brustoloni and Steenkiste 1998] 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 [Pasquale et al. 1994], Thadani and Khalidi's work [Thadani and Khalidi 1995], and the UVM Virtual Memory System [Cranor and Parulkar 1999] 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 [Bershad et al. 1995; Engler et al. 1995; Kaashoek et al. 1997; Seltzer et al. 1996] 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 [Kaashoek et al. 1997] 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 [Fall and Pasquale 1993].

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.

7. 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 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.

ACKNOWLEDGMENTS

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.

REFERENCES

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

FastCGI specification. http://www.fastcgi.com/.

SpecWeb96 specification. http://www.spec.org/osg/web96/.

Apache. Apache. http://www.apache.org/.

Banga, G. and Druschel, P. 1999. Measuring the capacity of a Web server under realistic loads. *World Wide Web Journal (Special Issue on World Wide Web Characterization and Performance Evaluation) 2*, 1 (May), 69–83.

Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C., and Eggers, S. 1995. 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), pp. 267–284.

Brustoloni, J. C. 1999. Interoperation of copy avoidance in network and file I/O. In *Proceedings of the IEEE Infocom Conference* (New York, March 1999), pp. 534–542.

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

Brustoloni, J. C. and Steenkiste, P. 1998. User-level protocol servers with kernel-level performance. In *Proceedings of the IEEE Infocom Conference* (San Francisco, March 1998), pp. 463–471.

CAO, P. AND FELTEN, E. 1994. Implementation and performance of application-controlled file caching. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation* (1994), pp. 165–177.

CAO, P. AND IRANI, S. 1997. Cost-aware WWW proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)* (Monterey, CA, Dec. 1997), pp. 193–206.

CRANOR, C. D. AND PARULKAR, G. M. 1999. The UVM virtual memory system. In *Proceeding of the Usenix 1999 Annual Technical Conference* (Monterey, CA, June 1999), pp. 117–130.

DRUSCHEL, P. AND PETERSON, L. L. 1993. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles* (Dec. 1993), pp. 189–202.

ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE, J. 1995. 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), pp. 251–266.

FALL, K. AND PASQUALE, J. 1993. Exploiting in-kernel data paths to improve I/O throughput and CPU availability. In *Proceedings of the 1993 Winter Usenix Conference* (Jan. 1993), pp. 327–333.

HUTCHINSON, N. C. AND PETERSON, L. L. 1991. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering 17*, 1 (Jan.), 64–76.

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

KAASHOEK, M. F., ENGLER, D. R., GANGER, G. R., BRICENO, H. M., HUNT, R., MAZIÈRES, D., PINCKNEY, T., GRIMM, R., JANNOTTI, J., AND MACKENZIE, K. 1997. Application performance and flexibility on exokernel systems. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles* (San Malo, France, Oct. 1997), pp. 52–65.

MCCANNE, S. AND JACOBSON, V. 1993. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX '93 Winter Conference* (Jan. 1993), pp. 259–269.

MCKUSICK, M. K., BOSTIC, K., KARELS, M. J., AND QUARTERMAN, J. S. 1996. *The Design and Implementation of the 4.4BSD Operating System.* Addison-Wesley Publishing Company.

NSAPI. Netscape Server API. http://www.netscape.com/newsref/std/server_api.html.

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

PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. 1999a. Flash: An efficient and portable Web server. In *Proceeding of the Usenix 1999 Annual Technical Conference* (Monterey, CA, June 1999), pp. 199–212.

PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. 1999b. I/O-Lite: A unified I/O buffering and caching system. In *Proc. 3rd USENIX Symp. on Operating Systems Design and Implementation* (Feb. 1999), pp. 15–28.

PASQUALE, J., ANDERSON, E., AND MULLER, P. K. 1994. Container Shipping: Operating system support for I/O-intensive applications. *IEEE Computer 27*, 3 (March), 84–93.

SELTZER, M. I., ENDO, Y., SMALL, C., AND SMITH, K. A. 1996. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. 2nd USENIX Symp. on Operating Systems Design and Implementation* (Seattle, WA, Oct. 1996), pp. 213–227.

TENNENHOUSE, D. L. 1989. Layered multiplexing considered harmful. In H. RUDIN AND R. WILLIAMSON Eds., *Protocols for High-Speed Networks* (Amsterdam, 1989), pp. 143–148. North-Holland.

THADANI, M. N. AND KHALIDI, Y. A. 1995. An efficient zero-copy I/O framework for UNIX. Technical Report SMLI TR-95-39 (May), Sun Microsystems Laboratories, Inc.