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

Rice University CS Technical Report TR97-294

Vivek S. Pai, Peter Druschel, Willy Zwaenepoel

Rice University

Abstract

This paper presents the design, implementation, and evaluation of IO-Lite, a unified I/O buffering and caching system. 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. Performance measurements show significant performance improvements on Web servers and other I/O intensive applications.

1 Introduction

This paper presents the design, the implementation, and the performance of IO-Lite, a unified I/O buffering and caching system. 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 I/O-intensive applications, including Web servers, etc. IO-Lite avoids redundant data copying (decreasing I/O overhead), avoids multiple buffering (increasing effective file cache size), and permits various performance optimizations across subsystems (e.g., cached Internet checksums).

Many other designs have been proposed to improve I/O performance in operating systems. In particular, various designs exist for copy-free I/O along certain data paths [3, 13, 6, 10, 16, 4]. IO-Lite distinguishes itself from these approaches by using a single paradigm to unify buffering on all data paths in the system, including inter-application, application-to-kernel, kernel-to-application, and inter-kernel data paths without restrictions on alignment of data. Furthermore, IO-Lite fully integrates the file cache, thus avoiding multiple buffering and copying along I/O data paths involving cached files.

Unlike approaches based on extensible kernels [1, 5, 12], IO-Lite uses an application-independent paradigm and provides an unrestricted execution environment, benefiting I/O intensive applications across the board. Existing UNIX applications can take advantage of IO-Lite's performance with modest to no changes. Since IO-Lite does not rely on aggressive extensibility technology, it can be readily integrated into an existing UNIX system.

We have implemented IO-Lite in Digital Unix version 3.2C. We use this implementation to present performance results on DEC AlphaStation 200 4/233 machines connected by FDDI and by 100Mb Ethernet. In keeping with our stated goal of improving performance of I/O-intensive servers, our central performance results involve a Web server, in addition to other benchmark applications. Our Web server can efficiently support general, complex client requests, such as CGI documents, without additional mechanisms. In addition, we present some microbenchmarks, and we also show significant performance gains on other popular I/O-intensive applications such as grep.

The outline of the rest of the paper is as follows. Section 2 briefly discusses background and requirements, and then presents the design of our unified buffering/caching system. In Section 3, we present a qualitative discussion of IO-Lite in the context of related work, using the requirements of a Web server as a framework for comparison. A quantitative evaluation of IO-Lite is presented in Section 4, including results of microbenchmarks and detailed performance results of a Web server and some other common I/O-intensive applications. Section 5 offers some conclusions.

2 IO-Lite Design

2.1 Background: Conventional I/O Buffering

In state-of-the-art UNIX systems, each major I/O subsystem employs its own buffering and caching mechanism. 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 allow the copy-free manipulation of possibly discontiguous buffers. It provides efficient support for common protocol operations such as fragmentation and reassembly of network packets, and the addition and removal of packet headers and trailers.

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

Multiple buffering The lack of integration in the buffering/caching mechanisms may cause multiple copies of the same data item to be stored in main memory. For example, in a Web server, a data file may be stored in the filesystem cache, in the Web server's user-level cache, and it may be held in the network subsystem's transmission buffers for multiple connections, awaiting acknowledgment by the client. This duplication reduces the effective size of main memory, and thus the hit rate of the server's file cache.

Lack of opportunity for 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. This is significant, because this checksum calculation is in many cases the only remaining data-touching operation in the I/O data path, once all redundant data copying has been eliminated.

2.2 Requirements for a Unified Buffer/Cache

We now proceed to define the requirements for a unified buffering/caching system. The primary goals of this system are avoiding redundant data copying

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

and multiple buffering, and facilitating optimizations across subsystems. There are three fundamental requirements for a unified buffering/caching system:

- Avoiding copying and multiple buffering implies that only a *single* physical copy of a data item must exist.
- In order to support caches as part of the unified buffer system, this single physical copy of the data item must be shared *concurrently* among different OS subsystems and applications. This sharing must not compromise protection and safety.
- Avoiding redundant copying further requires that the initial storage location of a data item (in physical memory) be preserved during the item's lifetime in main memory. As a corollary, the initial storage *layout* of a data object (in physical memory) must be preserved throughout its lifetime.

  The storage layout of a large data object (e.g., a file) received from a network device is commonly not contiguous, nor are its fragments necessarily page aligned or page sized. That is, the initial storage layout of data objects received from a network device can be complex. The prevalence of file retrieval across networks in distributed systems makes this an important scenario that must be handled efficiently, i.e., without copying.

Enabling cross-subsystem optimizations requires a fourth condition:

- To facilitate data-dependent optimizations across subsystems, it is necessary for a subsystem to be able to uniquely identify a previously seen data item. For example, given this facility, the TCP transmission function can detect that it is being asked to transmit a data object that it has transmitted before. Thus, it can avoid recomputing the IP checksum and use a cached value instead.

2.3 Immutable Buffers and Buffer Aggregates

The approach taken by IO-Lite is to define an explicit abstraction for I/O data that has 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 elegantly eliminates problems of synchronization, protection, consistency, and fault isolation among OS subsystems and applications[2]. Moreover, read-only sharing allows very efficient mechanisms for the transfer of I/O data across protection domain boundaries, as discussed in Section 2.4. For example, the filesystem cache, applications that access the same 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[3]. To alleviate the impact of this restriction, IO-Lite encapsulates I/O data in *buffer aggregates*. Buffer aggregates are instances of an abstract data type (ADT) that represents I/O data. 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 2.3.1.

In IO-Lite, all I/O data is encapsulated in buffer aggregates and passed among OS subsystems and ap-

[2] Privacy is ensured through conventional page-based access control.

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

plications *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 trivially guarantees 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.

To take *full* advantage of IO-Lite, application programs can use a modified I/O application programming interface that is based on buffer aggregates. At the heart of this API are two operations that replace the conventional UNIX read/write operations on file descriptors. A full discussion of this API is beyond the scope of this paper, due to space limitations. A technical report contains the relevant information [9].

We are now ready to summarize the key attributes of IO-Lite's unified representation of I/O data as follows:

Immutable buffers I/O data is stored in immutable buffers. This allows safe, concurrent read-only sharing of a single copy of I/O data among all OS subsystems and applications.

Buffer aggregates A buffer aggregate ADT combines multiple discontiguous, immutable buffers. The aggregate supports operations for the efficient access, manipulation, and modification of I/O data objects stored in immutable buffers.

Pass by reference I/O data is communicated among operating system, servers, and applications by passing references to buffer aggregates. This allows highly efficient cross-domain transfer of I/O data, and is consistent with the goal of sharing a single copy of all I/O data.

2.3.1 Impact of immutable I/O buffers

Let us consider the impact of the restriction that I/O buffers cannot be modified in place. 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. Second, if only a subset of the words in the object change values, then this naive approach would result in partial redundant copying. We can avoid this partial copying by storing only 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 cost in this case (over in-place modification) is 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. 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.

The first case arises frequently in programs that perform operations such as compression and encryption. The absence of support for in-place modifications does not affect the performance of these programs at all. The second 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 minimal. The third case arises in many scientific programs 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 an *mmap* interface similar to that 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. Using mmap may require copying. First, if the data object is not contiguous and not properly aligned, like incoming network data, a copy operation is necessary due to hardware constraint. 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 write to a memory-mapped file, where the modified page is also referenced through an immutable IO-Lite buffer, for instance when the file was read using an read operation. The system is forced to copy the modified page in order to maintain the snapshot semantics of read. The copy is again done lazily, upon the first write access to a page.

2.4 Cross-domain data transfer

In order to support caches as part of a unified buffer system, whether it be a system-level file cache or a user-level cache, the cross-domain data transfer 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 a mechanism similar to *fbufs* [4] to achieve this goal. Mechanisms that only allow *sequential* sharing [10, 3] cannot achieve this goal.

IO-Lite, 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. Since the fbuf mechanism maintains cached pools of buffers with a common access control list (i.e., a set of processes with access to the fbufs in the pool), the choice of a pool from which a new fbuf is allocated determines the access control list of the data stored in the fbuf. This implies that programs must determine the access control list of a I/O data object prior to storing it in main memory. 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 [15]. With IO-Lite, to avoid copying entirely, early demultiplexing is necessary.

Figure 1 depicts the relationship between VM pages, buffers, and buffer aggregates. Fbufs are allocated in a region of the virtual called the *fbuf window*, which appears in the virtual address space of all protection domains, including the kernel. The figure shows a section of the fbuf window populated by three fbufs. An fbuf always consists of an integral number of (virtually) contiguous VM pages. The pages of an fbuf share identical access control attributes; that is, in a particular domain, either all or none of an fbuf'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 fbuf, but slices in the same fbuf may overlap. The contents of a buffer aggregate can be enumerated by reading the contents of each of its constituent slices in order.

2.5 IO-Lite File Cache

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. In IO-Lite, buffer aggregates form the basis of the filesystem cache[4]. File data that originates from a local disk will generally be page-aligned and page sized. However, file data

[4]The filesystem itself remains unchanged.

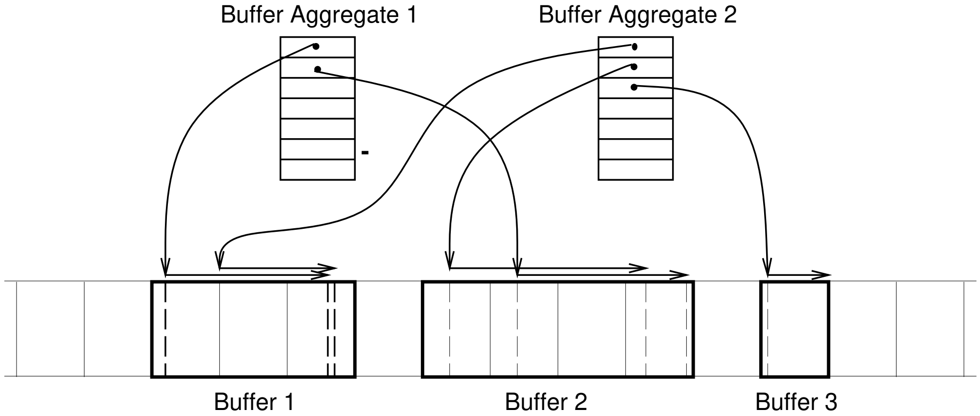

Figure 1 Aggregate buffers and slices

received from the network will not be page-aligned or page-sized, but can nevertheless be kept in the file cache in the same representation as it is received.

The IO-Lite file cache has no statically allocated storage. The data resides in buffers that 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 fbufs 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 a read operation of a cached file is followed by a write operation to the same portion of the file. The buffers that were returned in the read are replaced in the cache as a result of the write. However, they persist until the process that called read deallocates them and no other references to the buffers remain. In this way, the snapshot semantics of the read operation are preserved.

2.5.1 IO-Lite Cache Replacement and Paging

We now turn to a discussion of the mechanisms and policies for managing the IO-Lite file cache and the physical memory used to support I/O 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. First, cached data is potentially concurrently accessed by applications. Therefore, replacement decisions should take into account both references to a cached entry (i.e., cache lookups and insertions), as well as VM memory accesses to the buffers associated with the entry[5]. Second, 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 represent a portion 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.

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

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 only cache lookups and insertions (i.e., I/O read and write operations to cached files). 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. Since 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 controlled by the VM system. When the VM pageout deamon 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.

2.6 Enabling 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 content throughout the system. For example, the TCP protocol with IO-Lite 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, TCP can reuse the cached checksum, avoiding the expense of a repeated checksum calculation. This works extremely well for network servers, which 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 one, 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.

3 Discussion

This section provides a qualitative discussion of IO-Lite in contrast to various other I/O systems. To make the discussion more concrete, we examine how each of the I/O systems affect the design and performance of a Web server. Issues of interest are the I/O data path and the cache of Web documents, attachment of the HTTP response header, and support for Common Gateway Interface (CGI) programs. This section also provides the necessary background for the more quantitative discussion to follow in Section 4.

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.

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, cached data is then copied into network buffers, potentially leading to triple-buffering. Interprocess communication, which affects CGI programs, also suffers, since the same data must be copied from the CGI program into IPC buffers, which are then copied into the server's buffers. Data generated by a CGI program must be copied at least 3 times in transmission, and if the program uses a buffered library like stdio, it will be copied 4 times. One of the few benefits of this interface is the ability to modify a cached HTTP response header in-place, to update the response timestamp, for example.

Memory-mapped files The `mmap` operation can be found in many modern UNIX systems. `mmap`'s conventional share semantics facilitates a copy-free implementation, but the contiguous mapping requirement may still demand copying in the OS for data coming in over the network, as discussed in Section 2. 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 there is no support for producer/consumer synchronization between CGI program and the server. Having the server and the CGI program share memory-mapped files for IPC would require ad-hoc synchronization. The simplicity of socket-based communication is sacrificed to support copy avoidance.

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 [3]. 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 no longer 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 even on 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 [10] and Thadani and Khalidi [16] 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 forces copying "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 will 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.

Approaches Based on Extensible Kernels Recent work has proposed the use of of *extensible* kernels [1, 5, 12] 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, our kernel is fixed and we provide a single application-independent paradigm 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 CGI programs. In our web server, we 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 itself.

Extensible kernels do allow aggressive optimizations like modifying the HTTP response header in-place, incrementally updating a file's TCP checksum information, and storing checksum information on disk. Presumably, the checksum information is kept in different formats for various network adapter MTU sizes, or is kept in a format where incremental computation is inexpensive. In contrast, our approach caches incremental components of the TCP checksum, so that checksums are stored after the first time a block of data is transmitted. When response headers change, the file data's checksums remain unchanged, so the majority of the data is not touched again. Likewise, a variety of network adapters does not pose any problem for our system, since the variations in segment size, etc., will just cause new entries in the checksum cache.

4 Performance

In this section, we evaluate the performance of our prototype implementation of the IO-Lite I/O system. The implementation takes the form of a loadable kernel module that can be dynamically linked to a running Digital UNIX 3.2C kernel. In addition, the implementation includes a runtime library that must be linked with applications wishing to use IO-Lite. This library provides the buffer aggregate manipulation routines and stubs for the IO-Lite-related system calls. In one of the experiments, we also include a modified implementation of the ANSI C stdio library that uses IO-Lite internally.

All experiments were performed on DEC AlphaStation 200 4/233 machines equipped with 64MB of main memory, FDDI and Fast Ethernet network adaptors. The model 200 4/233 has a 233Mhz 21064A CPU with 16KB L1 instruction and data caches, and a 512 Kbyte combined L2 cache. Our machines run Digital UNIX version 3.2C with vendor-supplied performance patches for busy servers[6].

This section has three parts. In the first part, we evaluate the throughput of IO-Lite on local IPC using Unix domain sockets and Internet domain sockets.

[6] These patches fix some well-known performance problems in BSD-derived network subsystems, which greatly affect Web server performance. In particular, they add a hash-table based PCB table, they increase the limit on the listen backlog (to a default of 1024), and they make the TCP TIME_WAIT period a tunable parameter (set to 1 second in our case)

We then evaluate the performance of IO-Lite in the context of an HTTP server with full support for CGI scripts. Finally, we present performance results with a number of common I/O-intensive applications, such as grep and gcc.

4.1 Microbenchmarks

Figure 2 shows the results of a microbenchmark where a process repeatedly allocates a buffer, writes one word into each page of the buffer, and sends the buffer through a UNIX domain stream socket to a second process. The second process receives the data, reads one word in each page, and deallocates the buffer. The stream socket send and receiver buffer sizes were set to 256 KBytes. The graphs correspond to the throughput achieved with IO-Lite versus the standard POSIX interface for varying transfers sizes (the page size is 8 KBytes).

The two curves shown for IO-Lite, "IO-Lite-untrusted" and "IO-Lite-trusted", correspond to cases where the originator of the data is an (un)trusted process. The difference lies in the overhead for removing write permission for transmitted pages from an untrusted process. In all experiments reported in the remainder of this paper, data originates from the (trusted) OS kernel, and therefore the IO-Lite-trusted figures apply.

As expected, the copy-free IO-Lite system outperforms standard UNIX by a considerable margin. Moreover, our results compare favorably with those reported for a similar experiment with the Container Shipping system [10][7]. When transferring 128 KBytes (16 pages) at a time, CS achieves only 88 MBytes/s. The CS API requires that a received buffer is explicitly mapped into the receiver's address space before it can be accessed. When the receiver does not map or access the received data, a throughput of 248 Mbytes/s was reported by the CS system. This is still considerably lower than the 490 MB/s achieved with IO-Lite-vol, where the data is accessed by the receiver.

[7] Container Shipping results were reported for a DEC 3000/800 system, a machine with identical architecture as our AlphaStation 200 4/233, and comparable performance. The 3000/800 server has a slightly slower CPU clock rate (200Mhz), smaller L1 caches (8KB), but larger L2 cache (2 MB).

The difference in performance between CS and IO-Lite in this experiment is related to their transfer semantics. The CS handoff semantics require its implementation to map and unmap buffer pages on each transfer. The IO-Lite semantics, on the other hand, allow its implementation to dynamically allocate a pool of shared memory pages that can be recycled. After a short startup period, no VM map changes are necessary during data transfers. (With an untrusted data source, two map changes are necessary per use of a buffer, independent of the number of transfers.) We expect that the resulting performance difference would be even more pronounced on a multiprocessor, where TLB shootdown adds considerable cost to VM map changes [2].

The results of a network loopback test are shown in Figure 3. Two test processes on the same machine exchange data as in the previous test, except INET domain stream sockets were used, i.e., the TCP/IP protocol suite is used for communication. The packets are routed via the machine's FDDI interface, which means that a MTU of 4352 bytes is used by TCP/IP. Of course, the data never reaches the network interface, since communication is local. Again, IO-Lite outperforms the standard UNIX I/O system dramatically. However, the absolute throughput numbers are more modest, due to the overheads in the TCP/IP protocol stack. We attribute this to cache and TLB effects. With Unix, performance is dominated by the copy speed, which in turn depends on L2 cache hit rate. The copy speed diminishes as the transfer size approaches a substantal fraction of the L2 cache (512 KBytes). With IO-Lite, no copying occurs, but accesses to the transferred data pages still generate TLB pressure, causing an increasing rate of TLB misses for larger transfer sizes.

4.2 Web Servers

Our main experiment evaluates the performance of a Web server using IO-Lite, versus the same server using memory-mapped files. As a basis for comparison, we also provide performance results for the Harvest cache running in httpd accelerator mode and for thttpd, a freeware web server [11].

To fully expose the performance bottlenecks in the operating system, we started with an already fast HTTP server (thttpd), and made extensive modi-

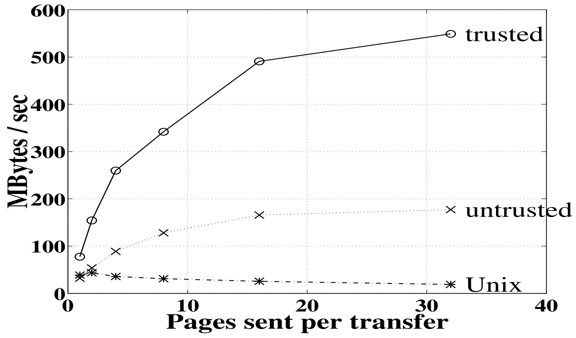

Figure 2 UNIX domain socket throughput

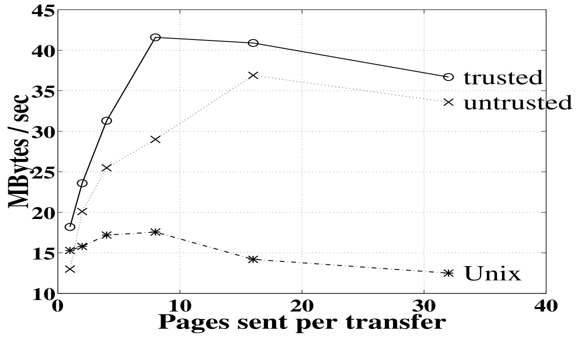

Figure 3 UNIX INET loopback throughput

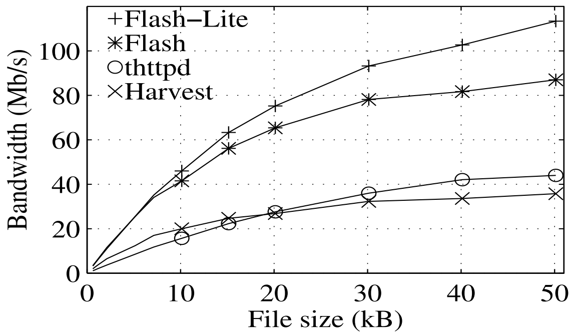

Figure 4 HTTP transfer speed

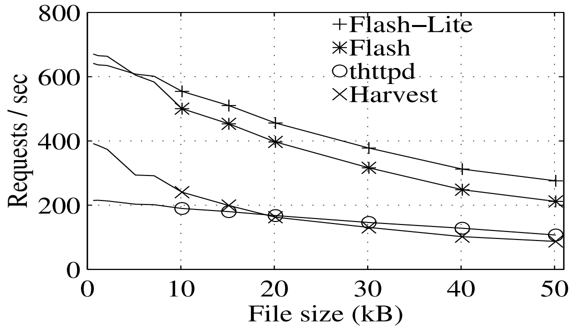

Figure 5 HTTP request rate

fications to eliminate performance problems in the server. The resulting program uses an event-driven loop that invokes the `select` system call to determine when data can be read or written on client connections and to manage interaction with CGI applications. All communication operations in the server are handled in a non-blocking fashion. These changes alone allowed us to more than double the performance of the original thttpd server, and the resulting server was dubbed "Flash". We also added IO-Lite support via small regions of conditionally-included code. This server was dubbed "Flash-Lite".

In our first experiment, several HTTP clients running on two machines repeatedly request the same document of a given size from the Web server machine. The file size requested was varied from 500 bytes to 50k bytes (the data points below 10kB are 500 bytes, 1kB, 2kB, 5kB and 7 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. Our HTTP client is a process that opens multiple connections to the server and uses a `select`-based loop to efficiently manage all the connections. We originally tried using the Webstone HTTP benchmark [17], but the load caused the client machines to reach CPU saturation well before the server machine. All tests were conducted using 32 or more concurrent client connections and access logging was disabled in all the servers. The primary source of connections was the client connected to the server via the FDDI network. However, when that client was saturated and the server showed idle time, we also enabled client connections from a machine connected via a less efficient Fast Ethernet network[8]. Figure 4

[8] The MTU size on the Ethernet card is roughly one-third the MTU of the FDDI card, which causes more processing to occur on the server machine.

shows the output bandwidth of the Web server as a function of request file size and Figure 5 presents the same measurements in terms of requests completed per second.

The two curves at the bottom represent the performance of Harvest and thttpd. Above those are Flash and Flash with IO-Lite. Both versions of Flash outperform Harvest and thttp by a considerable margin. For small file sizes, Flash's increased performance is primarily due to its efficient request-handling. As file sizes increase, the disparity becomes greater due to the reduction in copying. Harvest copies data once internally and then the operating system performs an additional copy. In thttpd and Flash, the files are mmaped, so neither perform any internal copies, but still encounter copying in the operating system. When IO-Lite support is added to Flash, the copying is eliminated, and the effects of this reduced overhead increase with the file size. Because both our FDDI client machine and the FDDI network reached saturation before the Flash-Lite server, we began adding Fast Ethernet clients to the Flash-Lite case for file sizes beyond 10 kBytes[9].

Persistent connections For small files, Web server performance is limited by the overhead of establishing and tearing down TCP connections. In HTTP 1.0, each request requires its own TCP connection that lasts only for the duration of the request. The HTTP 1.1 specification adds support for persistent (keep-alive) connections, where the same TCP connection can be used by a client for multiple requests. We modified both versions of Flash to support persistent connections and repeated the same experiments. The results[10] are shown in Figures 6 and 7.

On each of the 32 persistent connections that each client machine uses, a new request is issued as soon as a response is received for the previous request. With persistent connections, the request rate for small files nearly triples and the IO-Lite version of Flash is able to saturate the FDDI network on 10 kByte requests. Beyond this point, more client load is generated by using the Fast Ethernet network. Note that our results for persistent and non-persistent HTTP in our simple experiment bracket the performance of a Web server under realistic conditions, even if all clients use HTTP 1.1. This is because a connection can only be reused for temporally closely spaced requests from a client to a particular server.

Common Gateway Interface (CGI) programs One area where IO-Lite's unified buffer approach promises particularly substantial benefit is in the area of Common Gateway Interface (CGI) programs. These are programs that run separately from the server and execute requests on behalf of the client. The results are communicated to the server, which then sends them back to the client. CGI applications are one area that may cause significant problems for kernel-based Web servers, since CGI programs require a general execution environment, and their behavior is not as predictable (or ensurable) as the behavior of the server itself. Furthermore, it is desirable to provide CGI programs a rich environment while protecting the server from errant programs. For these reason, highly optimized Web servers (such as NetApp's server [18, 14], and those based on extensible kernels typically revert to an unoptimized, general execution for CGI programs.

CGI programs are difficult to support efficiently for two reasons—the CGI program must communicate data back to the server, and most implementations of CGI require the server to fork and execute a new process for each request. We support a form of long-lived CGI program where a CGI application does not have to exit after processing just a single request. In our model, the application uses a simple library to establish a connection and wait on incoming requests, but otherwise behaves like a regular CGI application. Our approach maintains the simplicity of the CGI 1.1 standard [8] while gaining many of the benefits of the non-forking CGI approaches [7]. As with regular files, we handle all aspects of CGI processing in the server in a non-blocking manner.

Since CGI programs rely on some form of interprocess communication to return data to the server, and that same data is then sent to the client, they should gain significantly from using IO-Lite. We ran a series of experiments where the CGI application partially mimics the server itself and returns a dummy file of

[9] Had we used the Fast Ethernet as the primary network for the small file tests, the lower latency would have improved the Flash and Flash-Lite request rates to 712 and 683 requests/sec, respectively.

[10] The anomalous dips in the graphs (Flash transfer speed at 20 kBytes and Flash-Lite request rate at 5 kBytes) were caused by TCP anomalies related to silly-window avoidance.

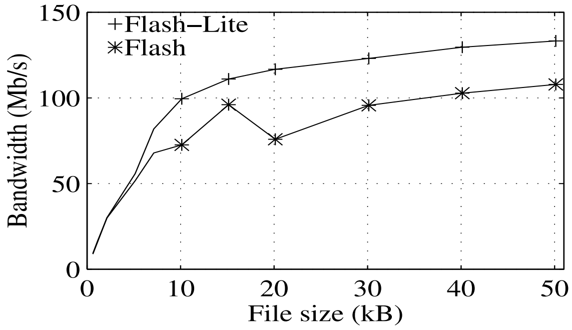

Figure 6
Persistent HTTP
transfer speed

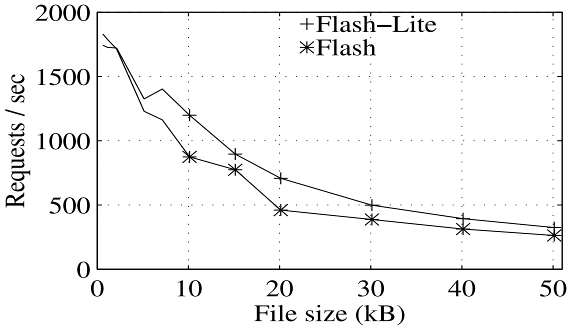

Figure 7
Persistent HTTP
request rate

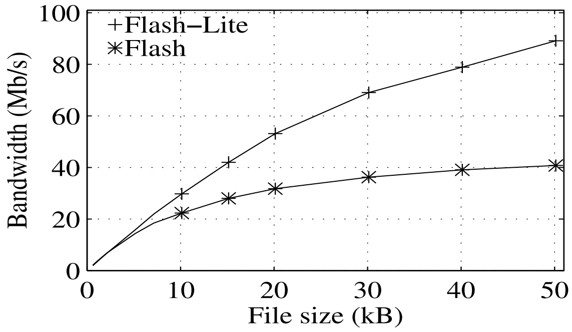

Figure 8 CGI
transfer speed

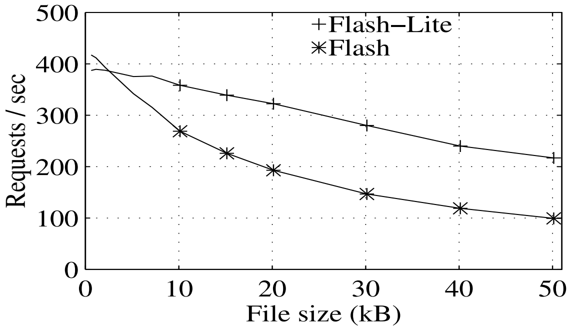

Figure 9 CGI request rate

the requested size. The results of these experiments are shown in Figures 8 and 9. This experiment attempts to measure just the overhead introduced by our CGI model and interprocess communication. For the regular Flash server, both the bandwidth and request rate values are less than half of what the server achieves serving files. With Flash-Lite, the performance is significantly better, approaching 80% of the speed of serving regular files. Also interesting is that CGI programs on Flash-Lite achieve performance comparable to regular files being served by Flash without IO-Lite.

CGI over persistent connections We also measured the case when the client has a persistent connection to the server and makes multiple CGI requests. While the performance of CGI programs using Flash-Lite increased significantly, the performance under regular Flash did not. The difference for large files is over a factor of two. Clearly, the copy overheads (both interprocess and from the server to the kernel) limit the regular Flash case.

In Flash, CGI programs run as separate processes and the operating system provides the server (and itself) protection from ill-behaved processes. The server never blocks for CGI processes and it optionally uses a configurable timeout parameter to kill any CGI process that runs for too long, so we do not have to concern ourselves with the behavior of CGI processes, nor do we have to place any undue restrictions on them. Flash places no "trust" in the CGI process, meaning that an errant CGI program does not cause Flash itself to fail in any way. Yet, IO-Lite affords high performance to CGI request handling.

Projected performance for faster networks The performance results obained with our Web server are limited to some extent by the capacity and characteristics of the network used in our testbed (one FDDI and one Fast Ethernet). Detailed measure-

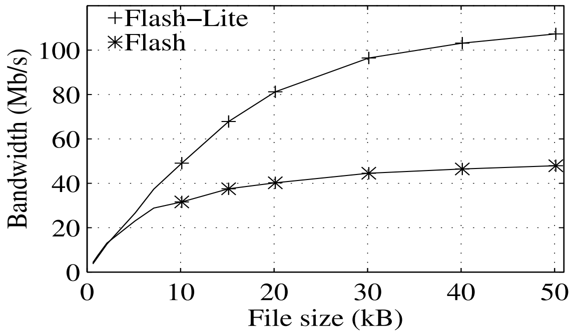

Figure 10
Persistent
connection CGI
transfer speed

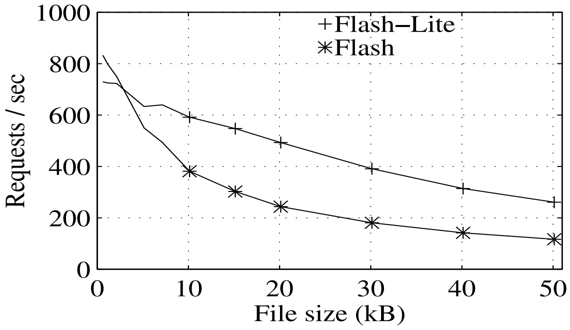

Figure 11
Persistent
connection CGI
request rate

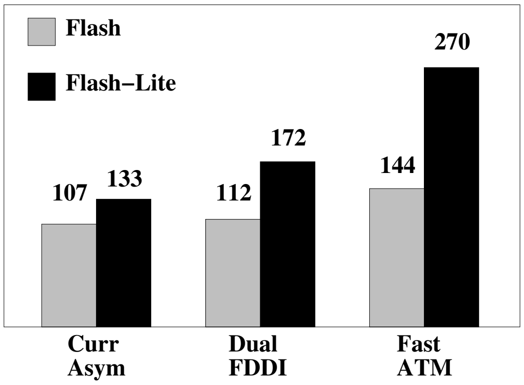

Figure 12 Projected Output Bandwidth (Mbits/s)

ments with an instrumented kernel show that the TCP per-segment processing overheads place a significant load on the server CPU. In particular, the load placed on the CPU while transmitting at a given bandwidth on Fast Ethernet exceeds the overhead of transmitting on FDDI by almost a factor of two. This is caused in large part by the much smaller maximal transfer unit (MTU) of Fast Ethernet (1500 versus 4352 bytes for FDDI), which forces TCP to transmit almost three times as many segments for a given amount of data. Since the benefits of using IO-Lite are limited by these overheads, we calculated projected throughput numbers for networks with larger MTUs, using measured costs of transmitting segments in our testbed. Figure 12 shows the expected output bandwidth of a persistent-connection HTTP server with and without IO-Lite, for a file size of 15KBytes, and assuming (a) one FDDI and one Fast Ethernet (actual testbed setup), (b) two FDDI networks, and (c) one OC-12 ATM network with an MTU of 9188 bytes. The results predict that the benefit of using IO-Lite increase significantly when using a network with a larger MTU. Note that these projections assume that no other system bottlenecks (e.g., memory bandwidth, I/O bus) are exposed at these rates.

Other benefits Note that our Web server experiments only quantify the (primary) increase in performance due to the elimination of CPU overhead. 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. We did not attempt to quantify this effect because it is difficult to do in our experimental testbed, which uses a low-delay LAN. Assuming the use of mmap to read files, the remaining source of double buffering in a Web server 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 average bandwidth*delay product of the network connections. Busy servers may handle several hundred connections, resulting in significant memory requirements even in the current Internet. With emerging future high-bandwidth networks, these requirements will sharply increase, making it imperative to eliminate double buffering.

4.3 Applications

To demonstrate the impact of copy-free I/O 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, and the GNU gcc compiler chain (compiler driver, C preprocessor, C compiler, and assembler). Figure 13 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 used for the "wc" program, but the file is piped 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

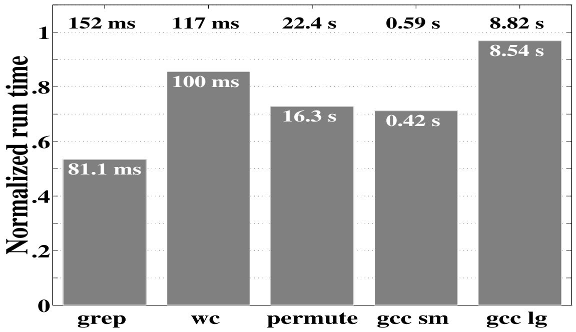

Figure 13 Various application runtimes

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 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 could be converted to use the IO-Lite API easily. 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 our IO-Lite-based version of stdio library and making function calls to enable our library. The preprocessor's (cpp) input operation turned out to be less cooperative. In an initial pass over the code, the preprocessor replaces trigraphs with single characters, and shifts the remainder of the file. More efficient and elegant approaches for handling trigraphs are possible, and these approaches could easily accommodate the IO-Lite semantics. However, given an existing implementation, we decided to take the easy route and convert cpp's input operation by using mmap to create a private mapping of input files.

5 Conclusion

This paper makes two contributions—first, it presents the design, implementation, and evaluation of IO-Lite, a unified buffering and caching system, and second, it compares IO-Lite with other alternatives in the context of a highly demanding driver, such as a web server. Though many different approaches to solving the I/O problem have been suggested, we feel that IO-Lite is aggressive enough to compete with the best of the performance-driven approaches, while at the same time, it is general enough to be practically implementable in standard operating systems. IO-Lite provides a strong conceptual framework in which to make decisions regarding optimization, and we have shown that IO-Lite can elegantly handle optimizations like copy-free interprocess communication, TCP checksum avoidance, and double-buffering elimination.

We believe that IO-Lite is well-suited for demanding applications, and we have provided results that show IO-Lite enjoying significant performance gains, especially in areas like CGI processing, where interprocess communication, the filesystem, and the network system are all involved. The analysis of our results also attempts to show not only what gains were made, but under what conditions gains can be expected. As obvious bottlenecks are eliminated (e.g. moving to persistent HTTP connections), the role of general-purpose optimization will become greater in achieving high performance, and we believe that IO-Lite provides a good framework for such optimization.

References

[1] 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.

[2] D. L. Black, R. F. Rashid, D. B. Golub, C. R. Hill, and R. V. Baron. Translation lookaside buffer consistency: A software approach. In *Third Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 113–122, Boston, Massachusetts (USA), Apr. 1989. ACM.

[3] 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.

[4] 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.

[5] 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.

[6] C. Maeda and B. Bershad. Protocol service decomposition for high-performance networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, Dec. 1993.

[7] Open Market. FastCGI specification. http://www.fastcgi.com/.

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

[9] V. Pai. IO-Lite: A copy-free UNIX I/O system. Technical Report TR97-269, Rice University, Jan. 1997.

[10] 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.

[11] J. Poskanzer. thttpd - tiny/turbo/throttling HTTP server. http://www.acme.com/software/thttpd/.

[12] 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.

[13] J. M. Smith and C. B. S. Traw. Giving applications access to Gb/s networking. *IEEE Network*, 7(4):44–52, July 1993.

[14] K. L. Swartz. Migrating to a web filer. http://www.netapp.com/technology/level3/-3012.html.

[15] 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.

[16] 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.

[17] G. Trent and M. Sake. WebSTONE: The first generation in HTTP server benchmarking. http://www.sgi.com/Products/WebFORCE/-WebStone/paper.html.

[18] A. Watson. Multiprotocol data access: NFS, CIFS, and HTTP. http://www.netapp.com/technology/level3/-3014.html.