RICE UNIVERSITY

IO-Lite: A Copy-free UNIX I/O System

by

Vivek S. Pai

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

Master of Science

Approved, Thesis Committee:

Willy Zwaenepoel, Chair
Professor of Computer Science

Peter Druschel
Assistant Professor of Computer Science

Peter J. Varman
Associate Professor in Electrical and
Computer Engineering

Houston, Texas

January, 1997

IO-Lite: A Copy-free UNIX I/O System

Vivek S. Pai

Abstract

Memory copy speed is known to be a significant barrier to high-speed communication. We perform an analysis of the requirements for a copy-free buffer system, develop an implementation-independent applications programming interface (API) based on those requirements, and then implement a system that conforms to the API. In addition, we design and implement a fully copy-free filesystem cache.

Performance tests indicate that our system dramatically outperforms traditional systems on communications-oriented tasks by a factor of 2 to 10. Application programs that have been modified to utilize our copy-free system have also shown reductions in run time, ranging from 10% to nearly 50%.

Acknowledgments

I would like to thank the following:

Rice University and the National Science Foundation for their partial funding of this work through their graduate fellowships.

My advisors Dr. Zwaenepoel and Dr. Druschel for their encouragement, support, ideas, and feedback.

Members of our research group for being sounding boards for ideas, stress relievers, and in the case of Gaurav Banga, unintentional system administrators.

My family and particularly my parents, for their continued support over the years.

Contents

Illustrations

Chapter 1

Introduction

Dramatic increases in microprocessor and network performance have rendered the main memory system a bottleneck in general-purpose computer systems. Particularly affected are programs with poor memory access locality, including I/O intensive applications and programs with large working sets. We discuss the historical cause of these problems and some of the solutions that have been used to handle them. We also discuss why the existing solutions are not complete nor scalable, and we discuss how we propose to handle the problem of limited memory performance. This section concludes with a discussion of our solution and an explains the organization of the rest of this document.

The growing disparity between cache latency and main memory latency has caused programmers to be more concerned with keeping data in cache to the maximum extent possible. Likewise, the growing disparity between cache and main memory bandwidth has forced the authors of performance-critical applications to carefully limit their accesses to main memory. While processor speeds have been doubling roughly every 18 months [17], data copy operations are essentially limited by the speed of main memory, which has been growing only at approximately 7% annually [17]. As a result, limited memory bandwidth has become a bottleneck for many applications [22]. As the CPU/memory speed disparity grew, the prevalence of caches became more widespread, and cache sizes increased as well. Caches were a cheap and simple way of hiding main memory bandwidth/latency problems, since a relatively small cache could absorb a large fraction of the program's dynamic working set. Increasing cache size was a cost-effective way of improving performance for a variety of applications, and today, some off-chip caches are as large as four megabytes. Programs with large working sets, however, have not seen as much improvement from caches, since they have been bound by main memory speed. Scientific applications, multimedia applications, and fileservers require high main-memory bandwidth, so there has been renewed focus on improving bandwidth.

The actual memory bandwidth available to the CPU has been lower than the theoretical peak bandwidth of the memory system for a variety of reasons. The memory system transfers data to the cache one cache line at a time, and the transfer is initiated when the processor fails to find data in the cache. So, between transfers, the process must execute instructions and miss in the caches, which requires some amount of time. In addition to the time required by the processor, some amount of time must be spent to handle the cache misses, to arbitrate the bus, etc., for each transfer. These combined overheads limit the actual throughput of the memory system. Recently, processors have been introduced with instructions aimed at addressing this problem [28, 19]. The new instructions are designed to more effectively load large amounts of memory with less overhead.

It might seem that the new attention paid to memory bandwidth would make the memory issue go away, but that is not the case. Bandwidth has been historically considered something that is easily improvable (from a technical standpoint), but doing so added cost to the design. The traditional approaches to improving bandwidth involved multi-banked memory, wider data buses, and longer cache lines [26], which caused the system price to increase. Longer cache lines also have the potential side effect of reducing cache hit rates since increasing cache line size for a fixed-size cache reduces the number of lines in the cache. The benefit of improved bandwidth had to offset the higher overall price, which inherently limits the widespread applicability of this approach. Another reason it has been neglected in many workstation-class (and even server-class) machines is because the gap between processor cycle time and memory cycle time had not grown large enough such that sufficiently many programs were adversely affected. However, as memory chip densities increase, the attempt to solve the bandwidth issue through multiple banks will diminish in usefulness, since fewer chips will be required to provide the same memory capacity. The other option will be to increase the total amount of memory in the system, which forces an increase in cost. For large systems that need lots of high-capacity memory chips, the multi-bank solution is workable, but for smaller systems, the bandwidth problem is not necessarily so easily solvable. In addition, the physical/electrical differences between DRAM memory and CPU (and SRAM memory) will ensure that the CPU will remain memory-bound for data-intensive applications.

One area where the memory bandwidth problem significantly affects performance is the set of programs that must communicate large amounts of data, either with other programs, or with the operating system. The performance of these programs becomes dominated by the speed of the main memory system, since the working set is larger than the size of caches [6, 7]. When this occurs, improvements in processor speed have little impact on execution time. A problem that limits the performance of these programs is the redundant copying of data that occurs as part of I/O and IPC operations. Despite recent work to reduce such data copying *inside* many commercial operating system kernels [14, 1, 13], a copy is still required whenever data crosses the kernel/user boundary in UNIX-based systems. For programs that communicate with each other, that boundary can be crossed twice for each item communicated.

Some of these issues have been addressed independently (to some extent) in existing operating systems. Redundant data copying in the network code directly affects network latency, so much attention has been paid to reducing the number of copies in the networking code [27, 10]. Likewise, interprocess shared memory [33] was introduced in operating systems as a way of allowing programs to communicate with each other using few data copies. Finally, most Unix-based operating systems support some form of memory-mapped files [15], which allows the operating system and the user code to communicate filesystem data without excessive copying. However, areas of communication still exist where redundant data copying takes place, and the existing mechanisms fail to address these, nor are they obviously expandable outside their target domains.

Rather than suggest more ad-hoc approaches to this problem, we have performed a wholesale evaluation of the problem and suggest a solution that has broad applicability. This work makes two contributions. First, it proposes a new high-performance application programming interface (API) that allows copy-free I/O operations and can be implemented using a variety of cross-domain transfer and buffer-management mechanisms. Second, it introduces IO-Lite, a highly optimized UNIX-based I/O system that implements the new API. IO-Lite eliminates both redundant data copying and other expensive data-moving operations from the entire data path in the common case. This goal is achieved regardless of the data's source and sink device, or the number of user processes traversed by the data.

Recent technological advances in networking, DRAM memories, and disk systems, combined with the emergence of new application domains like multimedia, have created the need for a reexamination of the I/O system, i.e., the part of the operating system responsible for I/O. The bandwidth of commercially available LANs has increased from tens to hundreds of Mbits/sec, and commercial networks in the Gbit/s range are on the horizon [9]. The availability of high-density, low-cost DRAM memory has made possible the use of large file caches. These caches help mitigate the mechanical latencies of disk drives, which used to form the bottleneck for file I/O. As a result of these developments, the amount of processing required on the host CPU to perform I/O operations has become more of a factor in overall performance.

Inefficiencies in the I/O subsystem of today's operating systems are due in part to the copying of I/O data along its path from source to sink device. It is widely recognized that the current UNIX (POSIX) `read`/`write` interface does not lend itself to a *general* copy-free implementation of I/O operations. This thesis proposes a new, high-performance I/O API that allows an implementation to avoid all redundant data copying. For the purposes of this work, *redundant data copying* is the physical movement of data from one location in main memory to another, where (1) the data values are not modified, and (2) the movement is not required to satisfy some hardware constraint. Whenever possible, a copy-free I/O system allows data to be placed into a main memory location by its producer (a CPU or an input device), and to stay in that location until it is consumed (by a CPU or output device). Besides avoiding copy overhead, an I/O system of this type can also reduce main memory consumption and programs' working sets.

A number of previous research prototypes combine copy-free data transfer mechanisms with non-standard APIs to eliminate copying from certain I/O data paths [27, 21, 25, 30, 12]. These previous efforts have two limitations: First, the APIs they use tend to be closely tied to their implementation. Therefore, it is unclear whether they lend themselves as a basis for a standard, implementation-independent, high-performance I/O interface. Second, with the exception of [25], these systems generally only optimize certain data paths under a certain set of circumstances, for instance, I/O of a single application from/to a network interface. Data copying is still needed when data paths involve multiple applications, servers, the filesystem, or pipes.

This work addresses the issue of copy-free IO, both through an elegant solution to the problem as well as an implementation of that solution. Moreover, we provide a framework through which other implementors and other designs can cooperate through a standard Applications Programming Interface (API). In doing so, we attempt to ensure that our solution is not limited solely to the domain of one operating system or one implementation. This work makes contributions to the area of copy-free IO in such areas as API design, disk cache design, and kernel buffer management. Our solution also builds on proven concepts developed in other systems, such as the fbufs [12] system, but it extends that work in new directions. The rest of this thesis is organized as follows: Chapter 2 explains in detail the motivation for building a unified buffer system and copy-free data sharing. Chapter 3 discusses the requirements and attributes of a copy-free I/O system, and how the key elements of the IO-Lite design satisfy those requirements. A new and extended Unix I/O application programming interface (API) that allows copy-free I/O, along with the philosophy behind its design, is presented in chapter 4. Chapter 5 describes an aggressive implementation of the IO-Lite API for the Digital UNIX operating system, and it examines the various components of the implementation. Our solution and implementation are compared to other work in the field in chapter 6. The performance of the system, both using test benchmarks as well as real applications, is evaluated in chapter 7. The key contributions of this work are summarized in chapter 8. The final chapters cover related work and draw some conclusions.

Chapter 2

Detailed motivation

Although some interfaces (shared memory, mapped files) have been introduced to reduce redundant data copying, they do not fully address the problem for a variety of reasons. Part of the problem stems from the internals of the OS and the requirements of interfaces like mmap. Also, the semantics of current interfaces, like the `read` and `write` system calls, force the OS to perform data copying, so these interfaces are not good candidates for high-performance environments.

2.1 OS limitations

The various buffering mechanisms used within UNIX-based operating systems have historically been designed to support only a particular subsystem, and the interfaces between different mechanisms have usually forced copies to take place. For example, UNIX filesystem code relies on the paradigm of having pages of memory represent blocks on disk in a one-to-one mapping [20]. For the case of local disks, this mapping scheme is effective for transfers to/from the disk, since local disks can transfer data contiguously in page-sized blocks.

However, this scheme generally forces data copying when used in conjunction with the `read` and `write` system calls, since the locations specified by the process making the call may not be aligned with the disk data. In special cases, when the locations specified by the process fall on page boundaries, page remapping can be used in conjunction with copy-on-write [14] to avoid data copying. This concept is taken one step further in memory-mapped files, in which case the process explicitly maps the file and accesses the file data directly. The process trades off flexibility in access (the data is now at fixed locations) for higher performance. Most implementations of memory mapped files also place other restrictions on the process. Some of these restrictions include: limiting the ability to increase the file length once the file is mapped, no implicit support for atomic operations, and generally vague, non-existent, or non-uniform specifications about the interaction of memory-mapped files and the regular `read` and `write` system calls.

The networking code uses a different buffer scheme from the filesystem, due to the large differences between the two systems. The network code relies on linked lists of small buffers with support for larger amounts of data "attached" to the buffers. These buffers are called mbufs [20], short for memory buffer. Early networks (and even most recent networks) were unable to send pages of data contiguously, so a page-based system would not have been the best choice when physical memory was a scarce resource. Mbufs are used in the networking code to communicate data to/from the network, but they are also used in interprocess communication through sockets on a single machine. However, one important point about the mbufs is that the memory contained in these buffers (and the memory linked to them) is essentially private to the operating system. Any interaction with code outside of the networking code, therefore, requires a data copy on each boundary crossing in most cases.

2.2 The problem illustrated

Let us closely examine a somewhat simple problem and the data copying associated with it. For the purpose of illustration, consider the case of a user running a http server [3] and a browser on a single machine that gets files from a local fileserver. When a file is to be retrieved, the data is sent over the network and received by the networking code. However, it is in several pieces and is in the private memory of the networking code. So, the operating system copies the data, both to transfer the data into the space designated by the filesystem, and to make it contiguous. Once the data is in the filesystem cache, it must be made available to the http server, and if the server process uses the `read` system call, another copy must be performed. A server that uses `mmap` can avoid this copy. Next, the server must send the data to the client process, and this commonly is done by using the `write` system call and a socket. The networking code will once again perform a copy to put the data into an mbuf-supported format. Finally, the data is sent to the client process, and the operating system once again performs another copy to take the data from the mbuf format and reassemble it as the client process has specified.

In this entire series of steps, the data has to be copied four times (three if `mmap` is used), and the contents of the data are not important to anything but the final client process. However, the copies are performed by the various layers of the system either to meet buffer management concerns (the networking code), layout concerns (the filesystem code) or system call semantics (the `read` system call).

2.3 Extending the current system

Before proposing an entirely new interface to eliminate these copies, it seems natural to ask if there would be a way of modifying the existing interfaces to reduce these copies. To specify a situation that would be most amenable to optimization, let us make three assumptions: the http server memory-maps files, the http server sends data in page-sized chunks, and the client performs page-sized reads. These assumptions are made for illustrative purposes in an attempt to show how much support would be needed to achieve goals similar to what IO-Lite achieves. If these assumptions do not hold, fewer opportunities exist for copy elimination.

The copy from the mbufs into the filesystem page can theoretically be eliminated by trapping accesses to the page and performing the copies lazily. The page can then be "marked" such that the system knows from where to assemble the page's data. The networking code can be modified to "hand off" ownership of mbufs, or at the very least, to keep mbufs in limbo while they are referenced by entities other than the networking code. These two optimizations combined would allow the system to delay or perhaps eliminate the copy from the mbufs to the filesystem. To eliminate the copy on the "outbound path" (from the application to the mbufs), the networking code can be designed to realize that the page can be lazily constructed from mbuf data and to go directly to the mbuf data instead. The mbuf data will then get sent over the socket, and will be used to satisfy the client's read request. However, the copy into the client's address space can not be eliminated, since there is no provision for the client to access the mbuf data.

With all of this theoretical support and under rather idealized circumstances, the data path has been reduced from three copies to only one. However, the issue of additional complexity in the new system has not been addressed, so let us first examine the new burden imposed on the networking code. The buffer-management code now has to understand the concept of outside references, and some form of "permanence" has to be introduced to handle the associated bookkeeping. The data contained by the mbufs (either internally or in external regions) now has a life outside of the networking code, so the mbufs structures must be preserved in some form. Finally, the pages used to hold out-of-line data can no longer be a fixed-size quantity, since the data in the filesystem can be long-lived. So, in addition to the other complexity, the networking code would also have to handle pageable memory. Also, support mechanisms to lazily construct a page from mbufs have to be created and the system has to either trap kernel accesses or explicitly check the page for this special mode before sending its data via new mbufs. Having the kernel accesses cause the page to be lazily constructed is not a good solution, since the goal is to use the backing mbuf data when possible.

The requirements described above are no simpler (and possibly quite a bit more complicated) than the support required for some form of IO-Lite, and more importantly, the system described is not easily expandable or even as general as IO-Lite. It works well for one particular set of application behaviors, and if the application changes, then the system has little utility. So while it might be theoretically possible to transparently support a reduced-copy system behind standard POSIX calls, the desirability and practicality of doing so is unclear. The system would require a significant increase in kernel complexity, and would not be suitable for the variety of tasks envisioned for IO-Lite. For example, this system would still force copying most of the time when the user process writes to the filesystem, or when processes on the same machine communicate with each other.

Chapter 3

A Copy-free I/O System

IO-Lite consists of many components working together to provide an API to the programmer and to implement this API within a UNIX-based operating system. IO-Lite implements one particular approach to implementing the API, but it is not the only means of doing so. This section discusses the requirements for a copy-free system and how this work addresses some of these requirements. The requirements outlined are those necessary for the most aggressive copy-free implementation, but a system capable of meeting those requirements need not be implemented in a completely aggressive manner. However, understanding the requirements for a fully copy-free system will help us prevent adding something to our design that hinders aggressive implementations. The discussion of requirements and the key elements of the design provides the foundation for the in-depth discussion of the implementation later in this document.

3.1 Requirements of a Copy-free system

There are two fundamental requirements for a copy-free I/O system. First, the storage location of a data item (in physical memory) must be preserved during the item's entire lifetime in main memory, since a change in storage location would require a copy operation. A corollary is that the initial storage *layout* of a data object (in physical memory) must be preserved throughout its lifetime. Second, all program entities (processes, OS) that use a particular data item must access a single, shared copy of the data item.

Note that 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. As a result, OS and applications should be prepared to process I/O data that is neither contiguous nor page-aligned, whenever that is feasible.

It might appear that the requirements of location preservation and sharing of a single physical copy could be met transparently, through the use of the virtual memory management hardware (MMU). That is, the virtual storage location of a data item could be changed, if necessary, by changing address translation tables. Similarly, users of a data item could transparently share a single physical copy through the use of shared mappings and copy-on-write [14]. Transparent support for such operations would imply that the current interfaces could remain unchanged, and programs would notice benefits without any modifications.

While this transparent remapping may work in limited situations, in general, paged virtual memory hardware is of limited practical use when attempting to *transparently* avoid copying of I/O data in UNIX-based systems. Since address translation occurs at page granularity, virtual address relocation (page remapping) requires that the source and target addresses be properly aligned, i.e., when divided by the page size, the two addresses must have the same remainder. Mapping discontiguous fragments into a contiguous address range additionally requires that the size of all but the last fragment is a multiple of the VM page size. Furthermore, the copy-on-write technique is only effective in avoiding physical data copying when the data is shared read-only. Lastly, page remapping and copy-on-write require page table modifications and associated consistency operations; these operations have substantial overheads, particularly in multiprocessor machines [4].

For a practical illustration of how easily page remapping can be "defeated," consider the case of a process reading a file from the network, as shown in Figure 3.1. In the case of FDDI, the maximum transmission unit is 4500 bytes, so a full disk block (assuming 8 kByte blocks) cannot be contained in a single packet. The packets also contain header information, so even if the incoming packets were placed consecutively on a page, the data in them would still not be properly aligned. Optimistic solutions for properly placing data do exist [8, 23], but they do not meet our requirements for full generality. In the event that two logically consecutive packets from within a single disk block are placed on different pages, the remapping cannot work, since the hardware does not allow parts of two physical pages to be mapped to the same logical page. The figure illustrates the worst-case example, where two packets are on improperly aligned parts of different pages and each packet has header information that needs to be stripped. However, cases this extreme are not needed to defeat virtual memory remapping. Even something as simple as a process writing part of a page to the filesystem can defeat the remapping scheme, since the surrounding data for the page would have to come from some other physical page of memory.

Given that VM address translation is not a general solution for copy-free I/O, the semantics for accessing I/O data must be chosen appropriately to allow the preservation of the data's initial storage layout, and to permit safe sharing of a single copy of the data. In particular, the application programming interface must have appropriate semantics.

3.2 Key elements of the IO-Lite design

In order to achieve copy-free I/O, data must be communicated between applications and OS *by reference*. Applications must be prepared to process data stored at a given location and with a given layout. As a second requirement, the access semantics for I/O data must be such that OS and applications can use a single copy of the data without sacrificing protection and security.

IO-Lite's operations for reading and writing of data from/to an I/O descriptor operate on *buffer aggregates*. A buffer aggregate is an abstract data type (ADT) that represents an ordered sequence of data bytes. The data may generally be stored in a number of discontiguous data buffers. The IO-Lite `IOL_read` and `IOL_write` operations pass the buffers in the buffer aggregates by reference. Note, however, that while the buffers themselves are passed by reference, the buffer aggregates may not be, and in the current implementation, they are not. In the current implementation, buffer aggregates are never shared across domains

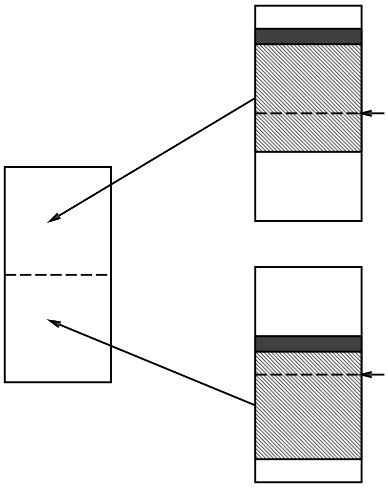

Figure 3.1 Virtual remapping problems

(where domains are various user processes or the kernel). The pass-by-reference semantics combined with the provision for non-contiguous data storage are the IO-Lite API's principal features that allow the preservation of data's initial storage layout.

Copy-free I/O also requires shared access of a single data copy. Unfortunately, concurrent access to I/O data by OS and applications can violate protection and fault isolation among different protection domains. This problem can be addressed elegantly through the use of an access model for I/O data based on *immutable buffers*. Immutable buffers are allocated with an initial data content that may not be subsequently modified by any user of the data. This access model implicitly restricts the sharing of buffers to read-only sharing, which eliminates problems of synchronization, consistency, and fault isolation. Moreover, it permits very efficient I/O system implementations based on variations of shared memory.

IO-Lite's buffer aggregates store data in immutable data buffers. Note that data storage in immutable buffers does not imply that buffer aggregates are immutable. In IO-Lite, a buffer aggregate is not immutable; rather, the buffers holding its present data values are immutable. When an application wishes to modify the contents of a buffer aggregate, modified portions of the data contents are stored in new buffers, and the new buffers are then combined with the old buffers holding unmodified portions of the original data to form a new buffer aggregate. Consider the example illustrated in Figure 3.2.

The original aggregate consists of two distinct immutable buffers. When the process wishes to make a modified version, a new buffer aggregate is created that points to the portions of the existing buffers, plus any new buffers the process wants to include. In the example, portions 1 and 3 of the the original buffers were added to the new buffer aggregate, as was all of buffer 4. So, while the new buffer aggregate represents a different sequence of data, it avoids copying by referring to the original data in immutable buffers. Neither the original buffer aggregate nor the contents of the original buffers change in any way.

Buffer aggregates support operations that permit the efficient, copy-free manipulation of data stored in discontiguous, immutable buffers. The exact functions supported by buffer

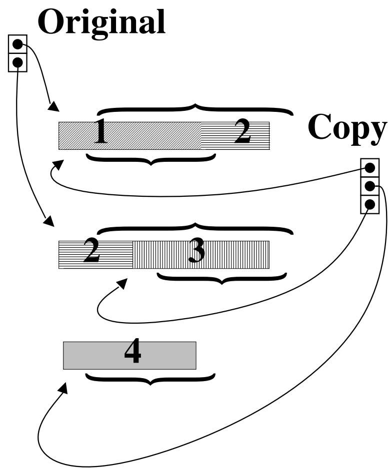

Figure 3.2 Buffer aggregate manipulation

aggregates are explained in more detail in Section 4.1. The implementation of the aggregate ADT uses an appropriate data structure to maintain the aggregate's constituent buffers*. Through the use of an abstract data type, this data structure remains hidden from applications to ensure maximum flexibility in implementing the IO-Lite API.

As we shall argue in Section 4.3, an I/O data access model based on discontiguous, immutable buffers is appropriate for a large class of applications. However, certain applications access or modify data objects in a manner that cannot be efficiently handled by such a data structure. For example, scientific applications often read large matrices from input devices, and access/modify the data in complex ways. For such applications, indexing costs associated with the access of non-contiguous storage may exceed the cost of copying the entire matrix into contiguous storage. Similarly, the cost of allocation and linkage of a small buffer for each modification may outweigh the cost of creating a private copy that can be modified in-place.

To support applications with such data access patterns, there is an alternative access mode for I/O data objects based on memory mapped objects (files). Memory mapped objects are always contiguous, and may be modified in-place. There may be additional costs associated with memory mapping an object, when compared to the standard access method. First, if the initial storage layout of the object is not properly aligned, a physical copy may be required prior to mapping the object. Data received from a network device is frequently misaligned. Second, in-place modifications of a mapped data object may require copying the modified page (copy-on-write) if the object is shared. For this reason, the standard access method should be used whenever possible. Memory mapping satisfies applications that require contiguous and/or in-place modifiable I/O data, but incurs data copying only when actually required.

*Similar data structures are widely used inside the network subsystem of operating systems (e.g., BSD mbufs).

The key attributes of the IO-Lite API can be summarized as follows:

Immutable buffers I/O data is stored in immutable buffers; this facilitates efficient I/O system implementations based on variations of shared memory.

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

Pass by reference I/O data is communicated among operating system, servers, and applications by passing references to buffers via buffer aggregates.

Memory mapped objects Memory mapped objects are provided as an alternative access model for applications with data access patterns that call for contiguous storage and in-place modifications.

Chapter 4

Application Programming Interface

This section describes a new I/O application programming interface that is designed to meet the requirements outlined above. A set of "manual pages" describing the API can be found in Appendix A. First is an overview of the API, and then follows a discussion of the philosophy behind the design of the API. The API has been designed to be minimal, both in the number of functions and their semantics, so some aspects of the API that may seem unusual are explained in this section. The section also contains a discussion of programming using the new API and it closes with an explanation of IO-Lite's security model.

4.1 Overview and description of the IO-Lite API

The API can be logically decomposed into three parts: performing the actual IO, manipulating data structures, and actually examining/modifying the data. The routines that perform the actual IO usually take a file descriptor as a parameter, and as a result, they interact with the rest of the system (either the kernel alone or in addition to other processes). These routines are `IOL_read`, `IOL_write`, `IOL_write_destroy`, `IOL_mmap`, and `IOL_control`, and are shown in figure 4.1. The routines that manipulate data structures are generally those which take an `IOL_Agg` structure as a parameter. These routines manipulate the logical arrangement of IO-Lite data once the space for it has been determined, but they do not actually examine the data directly. These routines are `IOL_create`, `IOL_destroy`, `IOL_concat`, `IOL_split`, `IOL_select`, `IOL_duplicate`, and `IOL_length`, and are shown in figure 4.2. Finally, there is the set of routines that are used to actually examine or modify data, and these are distinguished by the presence of an `IOL_Context` structure in the parameter list. This structure is an "opaque" data structure that is used by the library to maintain state information when the user wishes to perform a traversal of an `IOL_Agg`. Since

```
IOL_read(int fd, IOL_Agg **aggregate, size_t size, size_t *retSize);
IOL_write(int fd, IOL_Agg *aggregate, size_t *retSize);
IOL_write_destroy(int fd, IOL_Agg **aggregate);
IOL_control(int fd, int flags);
IOL_mmap(addr_t *addr, size_t len, int prot, int flags,
         int fd, size_t off);
```

Figure 4.1 IO-Lite I/O API

the IOL_Agg is also an "opaque" structure, the user cannot perform the traversal directly, but must instead call a set of "generator-style" functions, which allow the user to obtain information about the pieces of the buffer aggregate. These functions are IOL_gen_start, IOL_gen_nextRd, IOL_gen_nextWr, IOL_gen_seek, IOL_gen_tell, and IOL_gen_end, and are shown in figure 4.3. Obviously, the rules of thumb given for classifying the types of function are not perfect, since, for example, the IOL_gen_start function requires both IOL_Agg and IOL_Context parameters, but the rules give a logical grouping to the types of functions that comprise the API.

This section merely gives a logical overview of the main functions in the API. The reasoning behind the semantics of these functions can be found in section 4.2, and the details behind their implementation in our prototype can be found in section 5.2.

The core of the "actual IO" part of the interface consists of two operations, IOL_read and IOL_write (see Figure 4.1). These operations supersede the standard UNIX read and write operations, but the latter operations are maintained for backward compatibility and to handle file types not supported by IO-Lite. Like their predecessors, the new operations can act on any UNIX file descriptor. All other UNIX system calls related to file descriptors 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 logic behind allowing these "short reads" is explained in section 4.2. The IOL_write operation replaces the data in an external data object with the contents of the buffer aggregate passed as an argument. IOL_write_destroy has the additional side effect of destroying the aggregate passed as an argument. Although IOL_write_destroy is not strictly necessary from a completeness standpoint, it is a formal part of the API since the combination of a write operation followed by a destroy operation is expected to be quite common, and some systems may be able to perform useful optimizations in this case.

The effects of IOL_read and IOL_write operations are *atomic* with respect to other IOL_write operations concurrently invoked on the same descriptor. That is, an IOL_read operations yields data that either reflects all or none of the changes affected by a concur-

```
IOL_create_allocator(IOL_Allocator **allocator, IOL_Specs *impl_extr);
IOL_destroy_allocator(IOL_Allocator *allocator);
IOL_create(IOL_Allocator *allocator, IOL_Agg **agg, size_t size);
IOL_destroy(IOL_Agg *aggregate);
IOL_concat(IOL_Agg **dst, IOL_Agg *src1, IOL_Agg *src2);
IOL_split(IOL_Agg **dst1, IOL_Agg **dst2, IOL_Agg *src, size_t len);
IOL_select(IOL_Agg **dst, IOL_Agg *src, size_t start, size_t len);
IOL_duplicate(IOL_Agg **dst, IOL_Agg *src);
size_t IOL_length(IOL_Agg *src);
```

Figure 4.2 Buffer Aggregate Operations

```
IOL_gen_start(IOL_Agg *src, IOL_Context **ctxt);
IOL_gen_nextRd(IOL_Context *ctxt, const char **addr, size_t *len);
IOL_gen_nextWr(IOL_Context *ctxt, char **addr, size_t *len);
IOL_gen_end(IOL_Context *ctxt);
IOL_gen_seek(IOL_Context *ctxt, long pos);
IOL_gen_tell(IOL_Context *ctxt, long *pos);
```

Figure 4.3 Generator-style Traversal Operations

rent `IOL_write` operation on the same file descriptor. The data returned by a `IOL_read` is effectively a "snapshot" of the data contained in the object associated with the file descriptor.

The `mmap` operation supported in many versions of Unix is an explicit part of the IO-Lite API. Using `IOL_mmap`, applications can obtain a contiguous image of an I/O data object. When mapped for writing, the object can be modified in-place. Accessing a mapped object may force a data copy, and is thus potentially less efficient than performing explicit I/O using `IOL_read`/`IOL_write`. Different memory mappings of the same data object are guaranteed to be consistent. `IOL_write` operations affect mappings of the same data object, just as `IOL_read` operations reflect modification of the same data object through a memory mapping. However, the effect of `IOL_write` operations is not guaranteed to appear *atomic* with respect to a memory mapped image of the same object. Similarly, an `IOL_read` operation is not necessarily *atomic* with respect to concurrent modifications through a memory mapping.

The `IOL_control` operation allows applications to control operational attributes of IO-Lite on a per- file-descriptor basis. For example, an application can provide a hint to the implementation that its integrity is not jeopardized by concurrent modifications of data stored in received buffers. This hint can enable certain aggressive optimizations in VM-based data transfer mechanisms. Volatile fbufs [12] are an example of such an optimization. In this optimization, the creator of the data never loses write permissions, though the creator is expected to not overwrite the pages once they have been passed to the next domain. Not changing the permissions reduces the number of VM operations performed, both in the obvious case when the page is first sent, but also when the page is freed. If the page is no longer in use and the creator needs a new page to fill with data, the page can be recycled, and the permissions for the page are already correct. In producer-consumer situations, the volatile optimization greatly reduced the number of VM operations, since no VM operations are required once a "pipeline" has been established.

Figure 4.2 shows operations supported by the buffer aggregate abstract data type. The `IOL_create` operation creates an aggregate of the requested size. After the allocation, the buffer may be filled with its initial data contents by the application. The `IOL_destroy` operation destroys the aggregate and reclaims the underlying buffers, unless they are referenced by other aggregates. The internal data structure used to represent buffer aggregates is an implementation issue, and is not reflected in the API.

Additional operations provide means to truncate, split, concatenate, and duplicate buffer aggregates. No data is copied in the implementation of these operations. The `IOL_gen` set of operations (shown in figure 4.3) allow the enumeration of the contiguous buffers that comprise a buffer aggregate. Applications use these functions to retrieve data from aggregates, or to initialize the data contents of newly allocated aggregates.

4.2 Philosophy behind the IO-Lite API

Both the IO API and the buffer aggregate operations are fairly minimalist, but this design is intentional. During the design of the API, every call was evaluated from a standpoint of portability. The goals were to ensure that no aspect of any call was tied to one particular implementation, and that no call had semantics that would prevent more or less aggressive implementations of the underlying call. This decision is the motivation behind the "opaque" nature of the `IOL_Agg` structure. The library has direct access to the internals of this structure, but the user programs do not, since the internals of the structure are implementation-dependent.

The semantics of the calls have also been designed to give implementors a wide range of freedom, since the calls explicitly make few "guarantees". As a result, the implementations of system calls and libraries can vary in aggressiveness while still conforming to API. For example, the `IOL_read` call is intentionally specified to return at most the number of bytes requested, which may seem like an odd choice at first. However, there are parallels in the standard Unix API, since `read` calls on sockets can return fewer bytes than requested. The IO-Lite API merely extends these semantics to all reads, not just reads from sockets. An aggressive implementation may decide to take into account the source of the data, the overhead in reading, and the amount of physical memory in the system before doing any actual reading. A naive implementation may just read the amount requested and not consider the effects on the rest of the system. Both implementations can still conform to the API, and the implementations may have different strengths for different applications.

For example, consider the case of attempting extremely large reads, on the order of several times the size of physical memory. This scenario might arise if an application wishes to make changes to a large file while still allowing for the possibility of rollback (undoing the changes). Since the IO-Lite API supports "snapshot" semantics, automatic undo would be a simple task. In a conventional Unix system, the read would require an equal amount of backing memory, and would require that the data actually be read from disk. The process of reading would involve pages being read from disk into main memory, and then paged out to the swap device. In a sophisticated implementation of IO-Lite, the read could be handled in such a way that only the portions of the file that subsequently get overwritten would be loaded in memory. A less sophisticated implementation of IO-Lite, however, might decide to load the data from disk, but could limit the amount loaded so that the scenario described for the conventional Unix system does not occur. The amount loaded might take into account factors such as total physical memory in the system, or the amount of free memory.

The API has also been designed to allow a separation of concerns between the library code and the system code, by allowing the two components of the system to communicate as much or as little information as possible. In naive system-level implementations, the library can handle the details not implemented by the system code, and in more complete system-level implementations, the role of the library code is lessened. Of course, the decision regarding what to put in the library code and what to keep at the system level can be based on many factors. Information kept by the library may allow for certain API calls to run faster, but the implementor may prefer to keep more information at the system level to prevent an errant application from accidentally overwriting its files. It is expected that any user-accessible implementation will keep all of the "critical" data in the system-level code in order to prevent errant/malicious applications from causing system security concerns, but it is conceivable than an embedded system may choose to implement significant portions of the API in the library rather than the system-level code.

An example of this flexibility can be seen in the `IOL_read` call and its ability to return with less data than requested. If the IO-Lite API were implemented on top of other existing copy-minimizing systems, the library writers may decide that a call to `IOL_read` should just return the amount of data in a single contiguous memory region, since some systems only communicate a single region on each system call. For convenience, the library writers might decide to make multiple system calls at once based on the assumption that the program will eventually want all of the data. The current library for IO-Lite makes just one system call, but it can obtain an array of contiguous regions. More aggressive implementations may even allow for "sparse" regions, which do not actually exist in memory but are brought in on demand by the library. The fact that the internals of the `IOL_Agg` structure are not visible to the programmer and the fact that the semantics of the API calls are well-defined allows the implementors a great deal of flexibility, even within a single environment.

All of the flexibility offered to the library would be useless unless the program has some practical way of accessing the data. Since IO-Lite's data structures are implementation-dependent, the process must use function calls to get the information about the location and size of the actual data buffers. The process uses a "generator-style" mechanism to access the data, which means that the library provides a function to set up a "walk-through" of the abstracted data, and functions that iterate through the data, returning one contiguous region of data on each invocation. The library also has a "seek" function that the program can use to go directly to any portion of the data. The API could have also contained "callback-style" functions, where the user provides a function to be called repeatedly with all of the portions of the data, but the generator style is the more flexible of the two styles, and the callback style can be implemented using the generator style if desired.

4.3 IO-Lite Programming Model

Efficient use of the IO-Lite API by applications requires a programming model that differs from that appropriate for the standard POSIX interface. For best performance, applications should access data directly in their I/O buffers, regardless of buffer locations and data layout in the application's address space. Furthermore, applications are not allowed to modify data in I/O buffers in-place.

This change in programming model raises two concerns. First, how difficult is it for newly written applications to use the IO-Lite interface? Second, how difficult is it to convert existing applications to use the new interface? In considering the second question, it is important to realize that existing applications continue to work under IO-Lite, since the old, copy-based interface is retained for backward compatibility. A conversion is only called for if the application could substantially benefit from the increased performance of copy-free I/O. Many existing application do not have high enough I/O demands to warrant such a con-

```
        POSIX                                IO-Lite

allocate a buffer
while not eof                       while not eof
    read data into buffer               read next buffer aggregate
                                        foreach buffer in aggregate
    foreach byte in buffer                  foreach byte in buffer
        read/consume byte                       read/consume byte
                                        destroy buffer aggregate
```

Figure 4.4 IO-Lite input loop

version. Many new applications such as multimedia do, however, call for high performance, copy-free I/O.

```
        POSIX                                IO-Lite

allocate a buffer
while not done                      while not done
                                        alloc buffer aggregate
                                        foreach buffer in aggregate
    foreach location in buffer              foreach location in buffer
        produce and write byte                  produce and write byte
    write buffer data to output         write aggregate
```

Figure 4.5 IO-Lite output loop

Another factor is that many existing applications use a standard, language-specific I/O interface instead of directly using the UNIX/POSIX I/O API. For such applications, a data copy can be eliminated simply by substituting the runtime library implementation with one that uses the IO-Lite API internally. For example, C programs that use the ANSI C stdio library can enjoy performance benefits simply by linking them with a modified implementation of that library.

To assess the difficulty of writing new applications to the IO-Lite interface, consider an application that reads and consumes its input sequentially. “Consuming input” means that the information contained in the input data is converted to another representation and the values in the input data are not retained. An example is a compiler: It parses a source program and produces from this input an internal representation of the program. The input loop of an application of this type in standard UNIX and with the IO-Lite API are shown in Figure 4.4.

Many applications produce sequential output. That is, they produce a stream of output data from some internal data structure. A compiler is again an example of a program of this type. Figure 4.5 is an illustration of the necessary changes to the output loop.

Some application produce output while sequentially consuming their input, without ever creating an intermediate data structure. In this case, the program's main loop is a fusion of the input and output loops described above. Another type of application is a *filter*. Often, substantial portions of the input data stream appear in the output of a filter. The key to writing an efficient filter program with IO-Lite is to avoid copying the entire input data stream into new buffers. This can be accomplished by means of the manipulation operations supported by the buffer aggregate ADT. They allow clipping of parts of the input stream, insertion of new data into the input stream, and alteration of parts of the input stream without requiring a copy of the entire stream.

Finally, there are applications with substantial I/O demands that access and modify input data in complex ways. Scientific applications often have this behavior. Input data sets may be matrices that are randomly accessed and updated during the course of the program's execution. In this case, the costs of dealing with discontiguous storage and immutable buffers would exceed the cost of copying the entire matrix into contiguous, writable storage. With IO-Lite, such applications use the `IOL_mmap` operation to obtain a contiguous, writable image of the input data. An aggressive implementation of the IO-Lite API ensures that data is copied in this case only when actually necessary.

To illustrate the changes necessary for converting a program to use IO-Lite, we consider the case of grep, a program to select lines of text based on a regular expression. Grep operates on single lines of input, but reads data in large regions†. More importantly, it expects that the line of text is contiguous in memory. However, grep does not have a priori knowledge of where line boundaries occur, so it must handle the situation when lines cross block boundaries. In order to do so, grep reads in a block of data and works backward to find the start of the last line. The contents of that line are later shifted to a buffer and are prepended to the next block read. When converting grep to use the IO-Lite API, we follow similar logic. However, since we do not wish to shift much data, we solve the line-break problem by having an extra region were the broken line is copied and made contiguous. In the general case, between any two IO-Lite buffers, three invocations of grep's main processing code are called - one for the majority of the first buffer, one for the reassembled line that originally straddled the two buffers, and the final call for the majority of the second buffer. This process is fairly simple, and does not differ much from what the unmodified grep must do even for the regular case, simply to handle lines breaking across block boundaries.

In our experience, it is quite natural and simple to write applications with sequential I/O to the IO-Lite API. Moreover, applications with complex I/O behavior can use the `IOL_mmap` interface, which requires copying of data only when actually necessary, due to improper alignment or concurrent sharing of the requested I/O data. As seen in the case of grep, the support necessary for grep to function under IO-Lite must already exist for grep to operate in general. It can be argued that other programs which operate on streams of data already have the infrastructure necessary to operate under IO-Lite. Moreover, programs of this nature that cannot accommodate IO-Lite are likely to be poorly behaved even under Unix, since they are likely to expect all data to be contiguous. For example, if grep *required* that all of the data be in one large contiguous region, it would be impractical to use as a

†Optionally, grep can memory-map the file if possible, but since grep also operates on pipes, it must have support for handling smaller pieces of data

filter for automatically-generated data, since it would inherently restrict the amount of data that could be processed.

4.4 IO-Lite Security model

The security model in IO-Lite is almost identical to the security mechanism used in fbufs [12]. The mechanism is based on the concept of communication paths, and relies on pools of pages associated with these paths. In essence, all of the processes along a communication path use pages from the same pools, so that the protection information only needs to be set once. Free pages are kept associated with the communication path so that no VM system changes need to be made once the paths are established.

This security mechanism, when understood and used properly, is no weaker than standard Unix security because of the fact that the buffers in IO-Lite are immutable. A process that passes along data to a recipient can still see the data, but since the recipient cannot change the data buffer, the sender process sees nothing more than what it sent. Normally, any process that produces data will keep write access to the data even once it has been sent to another process. These buffers are called "volatile," because their contents can still be changed by the sender once they have been sent. In some cases, the recipient does not even examine the data in question, so if the sender changes the data after it has been sent, the recipient is unaffected. There are also programs which make only one pass over the data, so changes made to the data would not affect the recipient. If the change is made to data already processed by the recipient, it is never observed, and if the change is made to data that has not been processed, the effect is exactly the same as if that data had been sent originally. Volatile buffers cannot cause any problems for these types of applications. Even more complicated applications can use volatile buffers if the sender is a trusted program or the operating system. The recipient can also specify "nonvolatile" buffers, in which case any write permissions are revoked from the creator process. The cost of removing write permissions, however, leads to lower performance.

The issue of having multiple communications paths through a single application will be handled by using multiple "allocators" per application. So, an application can be part of multiple sets of page pools, and this scenario would not force it to breach security for any of its senders or any of its recipients. As in fbufs, this scheme does imply that early demultiplexing (allocating data in the appropriate pool originally) should take place whenever possible to enjoy the highest performance benefits. In the absence of early demultiplexing, copying can be performed to move data from one pool to another, and while this is less than ideal, the amount of copying involved is never more than the amount of copying that takes place in a standard Unix system. Page swapping, where pages can be transferred from one pool to another, may help in this situation, and we are exploring the possibility of providing such a mechanism, either to the kernel subsystem only, or also to user processes.

Finally, the integration of the Unix filesystem permission model into this system should be outlined. Currently, the filesystem is used to open files, so the filesystem's permission model is inherently accessed. However, the current prototype does implement fine-grained support for separating the process's filesystem data from its other data. It can be argued that a process which understands the security model of IO-Lite does not need to also contend with the filesystem's security overhead. However, if such an integration is desired, file blocks can be mapped into addresses spaces on a per-page or per-file basis, rather than on a per-chunk basis. Furthermore, as regions of memory are written into the file cache, they can be tagged so that future reads of that region through the file system only map the necessary pages rather than the chunk. Conceptually, this process is not difficult, but some careful engineering must be applied to avoid a naive implementation which discards the chunk-based model.

Chapter 5

I/O Lite Implementation

This section discusses possible I/O system implementations that conform to the IO-Lite API. The API was designed to permit a wide range of implementations and approaches to buffer management and cross-domain data transfer. Possible implementations range from simple approaches that avoid copying in some cases, to aggressive implementations that avoid data copying whenever the hardware allows it. We describe and evaluate one such aggressive implementation, an IO-Lite I/O subsystem for Digital UNIX (formerly known as DEC OSF/1).

Recall the key features of the IO-Lite API. First, the location, alignment, and layout of I/O data buffers in an application's address space is not determined by the application. This permits (1) the preservation of data's initial storage location necessary to achieve a copy-free data paths, and (2) the use of a variety of buffer management and data transfer techniques, including page remapping, shared memory, and fbufs, as well as storage of I/O data in memory regions with special properties (e.g, network interface memory or reflective memory).

Second, the IO-Lite API specifies I/O data buffers as being immutable. This gives an implementation wide latitude for sharing buffers among multiple address spaces. As a result, a single physical copy of data can be shared read-only, without giving up protection and fault isolation. Finally, the IO-Lite API hides the data structures and algorithms for referencing buffers and manipulating aggregates from the application, through the use of an abstract data type. This gives implementations complete freedom in managing buffers.

Shared memory channels [27], page remapping schemes such as the Container Shipping I/O system [25], copy-on-write (COW) [14], and fbufs [12] can all be used to implement the IO-Lite API. In each case, the resulting systems will be able to avoid copying whenever the implementation supports a copy-free data path. The IO-Lite I/O system is a very aggressive implementation of the IO-Lite API that seeks to avoid copying whenever the hardware allows it. The IO-Lite I/O system is based on *fbufs* [12], a facility that integrates buffer management and cross-domain data transfer, but extends that previous work 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 to UNIX-based operating systems, with minimal disruption to existing OS code and application programs. The following is a brief discussion of fbufs, followed by a sketch of the integration of fbufs with the existing UNIX networking subsystem, filesystem, virtual memory system, and device drivers.

The fbuf mechanism builds on two well-known techniques for transferring data across protection domains: page remapping and shared memory. It is equally correct to view fbufs as using shared memory (where page remapping is used to dynamically change the set of pages shared among a set of domains), or using page remapping (where pages that have been mapped into a set of domains are cached for use by future transfers). Fbufs are described in detail elsewhere [12].

Fbufs were originally used for network I/O only. Furthermore, the original fbuf implementation was used in conjunction with an x-kernel based communication subsystem [18]. To achieve a unified copy-free I/O system in a UNIX-based environment, the fbuf mechanisms had to be extended and adapted. First, to use fbufs in a BSD UNIX-based network subsystem, the implementation of the BSD *mbufs* was modified to transparently use fbufs. Second, the fbuf system was extended to interoperate with the UNIX filesystem and file cache.

The prototype IO-Lite implementation consists of a user-space library and a kernel module. These two components coupled together provide a working implementation of most of the IO-Lite API. The kernel module itself can be viewed as having separate components to handle its various tasks. One part of the kernel module is responsible for handling the interaction with the library module, since the kernel module provides system calls for the library module. The kernel module also interfaces with the rest of the kernel to interact with the filesystem, the sockets code, and the memory system. Each of these interactions is explained later in this section.

5.1 System calls description

Before a description of how the API is implemented can be presented, the system calls upon which the library relies needs to be discussed. The actual implementation of the interface uses only one real system call upon which the desired system calls are multiplexed. However, the base library code handles the details of this scheme, so this discussion will ignore the mechanics of this process and treat them as multiple system calls.

There are basically five important system calls in the implementation, and they perform the following operations: read, write, allocate space, add references, and remove references. These functions will be discussed in a fair amount of detail in order to illustrate how they can be used to implement the IO-Lite API.

```
int CacheInsert(int fileNumber, struct aggStruct *newAgg);
```

The `CacheInsert` function "writes" data into the specified file/socket. It returns the number of bytes written, or a -1 on error. The second parameter is a pointer to an `aggStruct`, which is a structure understood by both the system code and the library. An `aggStruct` contains an array of (pointer, length) pairs and a pointer to another `aggStruct`. It is defined as:

```
typedef struct aggStruct {
  int as_size;
  struct aggStruct *as_next;
  SliceRef as_slices[MAXSLICEARRAYSIZE];
} AggStruct;
```

The array size in a single `aggStruct` is fixed, and the field `as_size` indicates how many elements are in use. The library can link together `aggStructs` as linked lists and pass them to the `CacheInsert` function. This ability to link together `aggStructs` into chains is important, because this is how arbitrarily large/disjoint items of data can be atomically "written" into the IO-Lite cache.

```
int CacheLookup(int fileNumber, const long length, void *buffer,
                long bufferLenBytes, int *slicesReturned);
```

The `CacheLookup` function performs the "read" from the system. Note, however, that it does not actually read data into the location specified by the `buffer` argument. Since the implementation is copy-free system, the caller cannot specify where the data is to be located. Instead, the buffer is intended to hold an array of "slices," which are <location, length> pairs. The number of slices returned by the system is returned in the `slicesReturned` variable, and the total number of bytes in the result is the return value of the system call.

At first, it might seem strange that `CacheInsert` and `CacheLookup` do not have the same style of parameters, since the Unix `read` and `write` system calls have similar parameter lists. However, this difference is intentional. `CacheInsert` does not require any data to be changed at the user level, whereas `CacheLookup` expects data to be returned. In `CacheLookup`, the caller has no a priori estimate of how many slices will be returned, so it is not possible to allocate exactly as much space as needed for the <location, length> pairs.

This lack of predictability causes a problem for atomic read operations. In the common case, we would like to provide the system a small buffer, that is part of the library's return structure, to place the list of slices. However, in the uncommon case, we may want to provide a very large buffer to accommodate as many slices as the system is willing to return. The `aggStruct` structures are library-level structures, so having the system code allocate them would be impractical. A reasonable solution that is efficient in the common case and always correct is to use a single `aggStruct` as the buffer, and if that is not enough space for the requested data, to make the call again using a very large buffer. This second call will allow for an atomic read operation, and the slices can then be copied into multiple `aggStructs` within the library.

```
int GetFbufSpace(long numBytes, int ownerID, struct sliceRef *buffer,
                 long bufLenBytes);
```

The system also provides a mechanism for the library to allocate space. The `GetFbufSpace` system call has some of the same structure as the `CacheLookup` call, with a buffer space used to hold the resulting slices. The return value of this call is the number of slices actually returned. The amount of space returned can be less than the amount of space requested, especially if the amount of buffer space provided is inadequate.

Space allocation is handled in the kernel for reasons of simplicity and space efficiency. It would be conceivable to have a two-level allocation scheme, where the kernel allocates memory to applications in large chunks and the applications (or library) subdivide the allocated units. While this approach may result in fewer system calls, it is potentially less space efficient and requires memory allocation code in both the kernel and the library. However, based on measurements of the overhead for memory allocation and deallocation in the test applications, there is no compelling need to go to a two-level allocation system.

Finally, the functions to handle reference counting take an array of <position, length> pairs and make note of the change in reference counts. Reference counts are maintained in the kernel code, but since the user-level code can create/remove references, the library can/must explicitly pass information to the kernel when the reference counts must be changed. The reference counts are used by the kernel to determine when memory regions are no longer in use and can be recycled.

5.2 Library implementation details

Now that the underlying system calls have been discussed, how the IO-Lite library is implemented can be examined. It should be clear from the preceding section that the kernel code does not directly export data structures to the library code. The only call where the kernel code specifies the organization of user-level structures is the `CacheInsert` system call, where the parameters do not get modified. So, the library has a fair bit of liberty with respect to the structures it uses to keep track of data. Of course, the IO-Lite API was designed to allow the combination of the library and the system calls a high degree of flexibility, and since the system call interface does not significantly restrict the library's data structures, the library retains most of the flexibility afforded by the API.

The kernel implementation uses `aggStructs` to communicate data to be written. For the sake of convenience, the library code uses this structure in other places though, strictly speaking, it is not required. However, the structure is reasonably flexible and efficient, so it is used internally in the library. The `IOL_Agg` structure mentioned in the library basically contains some housekeeping information along with a pointer to a (potential) chain of `aggStructs`. The housekeeping information speeds up certain accesses by keeping known information like the head and tail of the `aggStruct` chain and the amount of data in the chain. The actual pointers to the data are kept in the `aggStruct` chain. This background information provides the basis for describing how the various API calls are implemented.

`IOL_Err IOL_write(int fd, IOL_Agg *aggregate)` - the library calls `CacheInsert` with the `aggStruct` chain associated with the `IOL_Agg`. In the current implementation, `CacheInsert` should not ever return with a short count when writing to a real file. When dealing with sockets, it can return a short count, theoretically, if an error occurs inside the kernel subsystem. Since this is not expected to be a frequent occurrence, the library just returns errors to the caller.

`IOL_Err IOL_read(int fd, IOL_Agg **aggregate, size_t size)` - most of the time, we expect that the result of `CacheLookup` will fit inside of one `aggStruct`. So, the library optimistically calls `CacheLookup` with the buffer space set to the buffer space inside of an `aggStruct`. If all goes well, the function returns with the amount requested, and the library associates the `aggStruct` with the new `IOL_Agg` to be returned. However, there is the possibility that the buffer is not large enough. Rather than return a "short read," what the library does in this case is to re-issue the call, but this time supply a large external buffer to hold the SliceRefs. Then, multiple `aggStructs` are allocated and the data is copied into them. This scheme optimizes for the common case, while not placing an undue burden on the uncommon case.

It should be noted that the design of `CacheLookup` is what allows this two-tier system. We believe that this "retry" approach is a net benefit, because the library can still perform an atomic read operation while only making two system calls in the worst case. The interface between the library/kernel boundary for this call is extremely simple, which allows the kernel code to remain simple. While it might seem desirable to have a system call that has some fixed-size return buffer and make repeated calls to achieve a large read, it would be difficult to maintain atomic operations in such a situation. For these reasons, the "retry" approach is a good overall solution.

`IOL_Err IOL_write_destroy(int fd, IOL_Agg **aggregate)` - this call is a variant of `IOL_write` with the side effect that the contents of the `IOL_Agg` are deallocated after they have been written. While this call is not necessary from a completeness standpoint, it can be an efficiency optimization on some systems. Since reference counting is handled in the kernel, deallocation could also be performed in the kernel itself. This optimization would reduce the number of system calls needed to perform a combined write and deallocate. However, it is equally correct to have this call perform a write followed by a separate call to handle the deallocation, and this approach is taken simply to reduce the complexity of the kernel code. This call is not directly supported in the kernel code because on the current test applications, the greatest benefit from doing so would never be more than a 2% improvement. So, kernel support for this routine is a low priority.

`IOL_Err IOL_create_allocator(IOL_Allocator **allocator, IOL_Specs *impl_extr)` - this function has not been implemented.

`IOL_Err IOL_destroy_allocator(IOL_Allocator *allocator)` - this function has not been implemented.

`IOL_Err IOL_create(IOL_Allocator *allocator, IOL_Agg **aggregate, size_t size)` - the library allocates `aggStructs` as necessary, and it allocates space by calling the `GetFbufSpace` system call. The `aggStructs` are linked to the `IOL_Agg`, but that is the extent of the work the library performs for this function.

`IOL_Err IOL_destroy(IOL_Agg *aggregate)` - the library code never actually deallocates space explicitly. For this call, the library will deallocate the `IOL_Agg` and it will deallocate any `aggStructs` backing the `IOL_Agg`. Before deallocating the `aggStructs`, the library will call the kernel to decrement the appropriate reference counts. However, only the kernel code can actually reclaim space, and it does this automatically when there are no more references to the space in question.

`IOL_Err IOL_concat(IOL_Agg **dst, IOL_Agg *src1, IOL_Agg *src2)` - a new `IOL_Agg` is created that contains the concatenation of the two `IOL_Agg`s provided. The appropriate reference counts are incremented, and new `aggStructs` are allocated. Neither src1 nor src2 are affected.

`IOL_Err IOL_split(IOL_Agg **dst1, IOL_Agg **dst2, IOL_Agg *src, size_t length)` - the contents of the source `IOL_Agg` are (nondestructively) split into two new `IOL_Agg`s. The length specifies the number of bytes to be placed into the first `IOL_Agg`, and the remainder is put into the second. Again, the reference counts are incremented as appropriate.

`IOL_Err IOL_select(IOL_Agg **dst, IOL_Agg *src, size_t start, size_t length)` - a new `IOL_Agg` is created that contains a subset of the source `IOL_Agg`. The start parameter indicates the number of bytes to skip at the start of the source agg, and the length parameter indicates the number of bytes to keep. As usual, the appropriate reference counts are incremented.

`IOL_Err IOL_duplicate(IOL_Agg **dst, IOL_Agg *src)` - the entire source agg is duplicated, and all reference counts are incremented.

`size_t IOL_length(IOL_Agg *src)` - one of the fields in the implementation of the `IOL_Agg` structure is a length field, and this function simply returns the contents of that field. Since the `IOL_Agg` structure is by definition "opaque," the library must have a well-defined function to get the length of an `IOL_Agg`. It is conceivable that some other implementation of the IO-Lite API may not have a length field in the `IOL_Agg` structure, or may decide to evaluate the length field lazily.

`IOL_Err IOL_gen_start(IOL_Agg *src, IOL_Context **ctxt)` - this function performs the setup necessary to make further "generator" calls. The `IOL_Context` structure, just like the `IOL_Agg`, is also "opaque," so the library implementors have a fair bit of freedom implementing the entire set of generator calls. In the implementation, the context structure points to the aggregate. The library keeps track of the current location, both in terms of bytes from the start of the aggregate as well as the actual "slice" it is handling. A reference count inside of the `IOL_Agg` structure is incremented so that the `IOL_Agg` cannot entirely disappear while a generator is active.

`IOL_Err IOL_gen_end(IOL_Context *ctxt)` - when the program is finished with a generator-style traversal of an `IOL_Agg`, this function should be called. It decrements the associated aggregate's reference count and potentially frees the aggregate if needed.

`IOL_Err IOL_gen_nextRd(IOL_Context *ctxt, const char **addr, size_t *length)`,
`IOL_Err IOL_gen_nextWr(IOL_Context *ctxt, char **addr, size_t *length)` - these functions are called by the program in order to iterate through the `IOL_Agg` originally associated with the `IOL_Context`. On every call, the start address and length of the next contiguous region is returned and the internal pointer is updated. For the case of `IOL_gen_nextWr`, the library checks to make sure that the aggregate is still a candidate for writing.

`IOL_Err IOL_gen_seek(IOL_Context *ctxt, long pos)` - given an active generator context, this function moves the internal pointer to a specified position from the start of the underlying aggregate. Using this function, a process can essentially randomly access the data within the context without having to manually walk through (and ignore) the data prior to the point of interest. This function is not only practical from the standpoint of the user, but in a system where the contents of the `IOL_Agg` are lazily evaluated, seeking directly to the spot in question can be significantly more efficient than walking the aggregate and ignoring the data prior to the spot in question.

`IOL_Err IOL_gen_tell(IOL_Context *ctxt, long *pos)` - given an active context, this function returns the location of the internal pointer in terms of bytes from the start of the underlying aggregate. This function can be used in conjunction with `IOL_gen_seek` to move the internal pointer relative to its current position.

`IOL_Err IOL_control(int fd, int flags)` - this function passes its arguments directly to the kernel code through a system call.

`IOL_Err IOL_mmap(addr_t *addr, size_t len, int prot, int flags, int fd, size_t off)` - this support for function has not been implemented in the kernel code. However, this function is intended to mimic the standard Unix `mmap` function.

5.3 Kernel module description

The IO-Lite kernel subsystem interfaces with several of the other parts of the kernel, including the disk interface, sockets, the virtual memory system, the pager demon, and the process exit facility. The memory system is used both to allocate space for the IO-Lite address space and to control access through protection changes. The disk subsystem is accessed for reading and writing files, and will eventually be used to page parts of the IO-Lite address space. The kernel system interfaces with the sockets code to send and receive IO-Lite data over sockets. In the current implementation, the IO-Lite address space is fixed and wired in memory, but eventually, the pager daemon will be used to dynamically control the allocation of physical pages. Finally, the interface to the process exit facility is crucial to the IO-Lite kernel code to prevent poorly behaved applications (and applications that exit unexpectedly) from holding references to memory in the IO-Lite region at exit time.

5.3.1 Memory system

When the IO-Lite subsystem is loaded and activated, it allocates a large region of memory and "wires" it so that the kernel marks it as unpageable. This memory is made visible to user processes and contains the actual data stored by IO-Lite. The memory used by IO-Lite for its internal data structures is allocated separately and is not visible to user processes. When processes make their first calls to the IO-Lite system, the IO-Lite memory is added to their address spaces at a fixed location. This memory is originally mapped such that the processes have no read or write access to it, but the protections are changed as the processes make calls.

Memory is allocated to processes in "chunks," which are fixed-sized blocks of pages. At most one process can ever have write permission for a given chunk, and this is generally the process that allocated the chunk. When other processes gain read access to data, they gain read permission to all of the chunks containing the data. In the current implementation, processes can give each other access to data directly by sending data over sockets, or they can give each other access indirectly by sharing file data. Read permissions are handled on a per-chunk basis for efficiency purposes, but since the security model is based on the concept of allocators, security is not compromised in the process. In the case of non-volatile buffers, write permissions are dynamically manipulated on a per-page basis, but read permissions are always handled by chunks.

Although permissions are handled on a per-chunk basis when possible, chunks are not the minimum unit of allocation. Buffers are the allocation units on which reference counting takes place and can be at most one chunk in size. Buffers are contiguous, are always a multiple of the page size, and never cross chunk boundaries. An allocation request can be fulfilled by many different buffers in different chunks. Once all of the references (from all processes and the kernel) to a buffer have been removed, the buffer is freed and its associated pages are returned to the process that allocated it. Associated with each process are the free pages and chunks it owns. The system uses this "affinity" in order to reduce the number of protection changes it must make.

While a process runs, any requests for memory made on its behalf in the kernel code are first satisfied from the process's list of free pages, if any are available. If no free pages exist for the process, new chunks are allocated to the process. If the process passes references to the memory, either to other processes or to the kernel, those entities also increment the reference counts on the memory. Processes (usually the IO-Lite library) can voluntarily reduce the reference counts on buffers, and a well-behaved process should relinquish all of its reference counts before it quits. However, to ensure that no process can hold resources beyond its lifetime (either through sloppy programming or an unexpected termination), the kernel module interfaces to the kernel's process exit code and reclaims any pages held only by the exiting process.

The implementation does handle some of the exceptional cases posed by the allocation and deallocation scheme, but it currently does little in the way of garbage collection. For instance, it must handle the possibility that a process frees the last reference to a page that was owned by a now-terminated process. Had the owner process been alive, the page would have returned to the owner process's free list. Once a process dies, the kernel reassembles free pages into chunks where possible and reclaims all free chunks. As the remaining outstanding pages are eventually freed, they are tabulated and when all of the pages of a chunk have been freed, the chunk is reclaimed by the kernel.

Currently, no run-time garbage collection or scavenging of pages is performed, which may be a useful feature. In a complete IO-Lite system, such scavenging is not expected to be as useful, since there will be a large, pageable space for IO-Lite memory. However, the current prototype uses a small, wired space, so scavenging may have appreciable benefit. For example, if process's memory needs vary greatly over time, the current implementation will never reclaim any of the process's unused memory. A scavenging system could reclaim chunks that had not been used in a certain amount of time. However, the current implementation makes no attempt to aggregate free pages into chunks, so even if there were a scavenging system for free chunks, the possibility of degenerate cases would still exist.

As mentioned earlier, these issues may not appear in a complete system since some sort of mechanism to limit per-process resource consumption would exist, and the pageability of the IO-Lite space would reduce concerns about exhausting memory. However, even in a complete system scavenging may have some utility, since it could potentially reduce the paging in the system by fulfilling requests for new memory with free chunks previously owned by other processes. An intelligent pager interface can alleviate the need for a scavenger by determining if a page is unused and then just not saving/restoring its contents in that situation. The pager approach would still consume virtual address space, but it has the added benefit that individual pages would effectively be scavenged instead of just whole chunks.

The current implementation avoids pager issues by having all of the IO-Lite memory region marked as unpageable. Since the kernel code does not have to interact with the pager code, all of the IO-Lite memory is allocated as an "anonymous" memory object, which would mean (under normal circumstances) that the memory is backed by swap space. However, once the memory has been allocated, standard kernel functions are used to manipulate its protections and the kernel treats it as any other piece of memory. The one area that has proved to be troublesome in this regard is the mapping interface. The IO-Lite memory is mapped into every process which uses IO-Lite, but it would be desirable to also map the memory into the kernel's address space. Unfortunately, in Digital UNIX, it is not possible to map a region both into kernel and user space simultaneously, so some compromises are made. The IO-Lite memory region is mapped into the user spaces, and the kernel is forced to refer to pages by their (almost always discontiguous) physical addresses. Although this scheme is clearly suboptimal, it has not yet presented any significant obstacles.

5.3.2 Filesystem interface

Conventional UNIX buffer cache implementations are not suitable for a copy-free I/O system, since they place restrictions on the layout of cached file data. Not surprisingly, current Unix implementation perform a copy when file data arrives from the network. In IO-Lite, fbufs form the basis of the filesystem cache[‡]. File data that originates from a local disk will generally be page-aligned and page-sized. However, file data 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 kernel module is intended to "piggyback" onto the filesystem to accomplish its interaction with files, or, more specifically, the IO-Lite kernel module is not integrated with the filesystem code, but instead directly calls specific low-level routines in the filesystem code to accomplish what it requires. The filesystem code for any particular filesystem (as opposed to the Virtual File System code) is not amenable to a "clean" sharing of responsibilities, so there were few options when designing the IO-Lite kernel module. The IO-Lite filesystem interface is not designed to replace a standalone filesystem, since the goal is to cause as little disruption to the kernel as possible. From a practical standpoint, it would not be useful to have a filesystem type accessible only from IO-Lite. Instead, we wish to take existing files and filesystems and have IO-Lite bring file data into memory in a form that it can handle. IO-Lite does not require any changes to the filesystem's metadata nor does it require any changes to the way a file is physically/logically stored on disk. It only change the manner in which a file is retrieved or written.

As a result of these requirements, the prototype bypasses the normal filesystem code when reading or writing pages from a disk. Due to the nature of the IO-Lite system and the goal of avoiding copying, there will be separate IO-Lite interfaces for different filesystem. The current implementation interfaces with the UFS/FFS code, but has the support functions to allow the addition of other filesystems. The support functions are filesystem-independent and rely on "driver" functions that must interface directly to each filesystem type supported.

While it would be desirable to have a "clean" method for interfacing with the filesystem, doing so is prevented by the tight coupling of the filesystem and the memory management system in the Digital UNIX OS. Like most other modern Unix OS's, it uses a "unified" system for handling the interface between the filesystem cache and the rest of the virtual memory system. In early Unix systems, the filesystem cache was statically allocated and generally of a fixed (but tunable) size. Recent implementations have kept a static cache only for filesystem metadata, and have made filesystem data blocks occupy pages that compete with all other pages controlled by the virtual memory system and its associated paging

[‡]The on-disk representation of the filesystem remains unchanged.

system. In either case, file data is page-aligned, page-sized, and usually overwritten in place. These three conditions for data are obviously at odds with the requirements for a copy-free IO system.

One of the side-effects of this close coupling between the filesystem cache and the virtual memory system is that many of the data structures used to handle the filesystem's read/write support functions are closely tied to the virtual memory system, and attempting to "cleanly" add low-level support for a new paradigm is practically impossible. Given that we wish to utilize existing filesystems, the aforementioned complications lead to a compromise solution where the regular filesystem code is used to handle everything related to filesystem metadata, and IO-Lite code drives the reading/writing of filesystem data blocks. IO-Lite communicates with the metadata-handling code to determine the logical block numbers of the blocks the processes wish to read or write, and then it direct the low-level code to perform the reads/writes to the IO-Lite kernel memory rather than to memory controlled by the VM system.

The fbuf-based file cache has no statically allocated storage. The data resides in fbufs that would occupy ordinary pageable virtual memory in a complete IO-Lite implementation. The file cache data structures merely map <file-id, offset> tuples to fbufs that contain the corresponding file data. Since fbufs are immutable, a `IOL_write` operation to a cached file results in the replacement of the corresponding fbufs in the cache with the fbufs supplied in the `IOL_write` operation.

The replaced fbufs no longer appear in the file cache; however, they persist as long as other references to them exist. For example, assume a `IOL_read` from a cached file is followed by a `IOL_write` (with different data) to the same portion of the file. The fbufs that were returned in the `IOL_read` are replaced in the cache as a result of the `IOL_write`. However, they persist in memory until the process that called `IOL_read` deallocates them, and no other references to the fbufs remain. In this way, the snapshot semantics of `IOL_read` are preserved.

One side-effect of the implementation shortcut is that the prototype implementation's filesystem cache can be thought of as "sitting alongside" the regular filesystem cache, and the lack of cooperation at the data-block level can mean that the contents of the two caches may not be the same. Since this is just a prototype, no attempt is made to synchronize the data maintained by the two. In a production system, such a situation would not exist because all reading and writing of regular data would be performed by the IO-Lite system only.

5.3.3 File cache

Our filesystem cache is somewhat different from most caches in that it is designed to handle arbitrary boundaries for the data elements it contains. Most caches deal with blocks of file data, but since this cache is to be copy-free, the restriction that data must kept in blocks is not applicable. As a result, entries in the cache take the form of <start address, length> pairs which are mapped to <file, offset, length> triples. Because of the zero-copy approach, the same pair can be mapped to multiple triples, even within the same file. The triples, of course, cannot overlap in coverage.

This break with traditional cache-management systems has resulted in numerous other changes. For example, a fetch-on-write strategy is not needed for the cache, and in the current implementation, one is not used. Normally, any read or write to part of a file not in the cache causes the calling process to be blocked while the appropriate disk block is retrieved. In the case of reads, the reasoning is obvious, since data not in memory cannot be returned to the process. However, in the case of writes, the reasoning is not so obvious. Microprocessors, for example, do not, in general, have to block for writes anymore, so it seems a little strange that processes should block while doing disk writes.

In theory, a system could be designed where processes are not forced to block in such situations. However, as we shall see, the support for handling such situations is not trivial, and the complexity of such a system must be weighed against its expected benefits. The POSIX `write` system call requires copy semantics, so the process is free to overwrite the buffer when the call returns. In order to have a non-blocking write in this situation, the kernel would have to change page protections such that any attempt by the process to overwrite the buffer would be caught and handled. The kernel could copy the entire page, copy just the portions of the page used as write buffers, or just block the process until the writes complete. The kernel would also have to keep track of all of the outstanding writes and would also have to keep track of them with respect to incoming read requests, since those reads would have to see the results of any successful writes which had taken place before them. The complexity of non-blocking writes is non-trivial, and the benefit from them is not expected to be significant, so implementors choosing to not support them is not surprising. Instead, the block is loaded from disk, the data to be written is copied over parts of the newly-loaded block, and the processes resume operation.

In a regular filesystem cache, every page of the cache represents some range of a file, and there is no overlap between ranges. This simple mapping reduces the complexity of many operations, since there exists a unique place where data can be located. The arbitrary boundaries in the IO-Lite cache prevent such a simple mapping system from being used, so some of the operations relating to the cache must be handled differently than in a traditional cache. Inserting data, retrieving data, reading in disk data, and writing out dirty data are all examples of operations that are handled differently in our cache.

The "insert" operation places data into the cache and de-links any data which previously occupied the space designated occupied the new insertion. The new data may partially overlap with existing data, so the information on existing regions in the cache may have to be "trimmed" or split to accommodate the new data. The cache is organized as a collection of "extents," which are indexed through a hashing scheme based on file identifier and position within the file. Since the extents are allowed to have arbitrary boundaries, the hashing scheme is based on blocks within the file, and the same extent can be hashed to multiple entries in the table if it crosses block boundaries. Alternately, some form of tree-based storage could be used on a per-file basis, but for the prototype, the hashing scheme is sufficient. The tree-based scheme could potentially be more efficient for inserts, since only one entry would have to be recorded regardless of extent size.

The retrieval process is non-trivial because of the way extents are managed. The kernel must potentially search multiple hash bins (for large requests), sort the results, and then determine if there exist gaps in the coverage. The searching process is not necessarily as onerous as might be imagined, especially if large extents exist. Large extents are indexed from multiple hash bins, but the searching process can stop early if it detects that the extent covers the entire request. The retrieval process triggers reading from disk in the event that gaps in the coverage area exist in the cache. The blocks are read from disk, and rather than being inserted in their entirety, only those portions of the blocks that do not already exist in the cache are inserted. This process of searching for cache data when loading disk blocks is obviously more complicated than the fetch-on-write case, since extant data must not be replaced. Issues of locking and races are avoided by using a "compute on demand" model, where the "gap list" is recomputed immediately before the block is inserted into the cache.

The same "compute on demand" model is used to write dirty data to disk. In most UNIX-based systems, a process issues a `sync()` system call periodically to write the updated contents of the file cache to disk. The dirty data is already in-place, and the blocks to be written correspond to the dirty pages. However, since contiguity is not enforced in any way for file data, there is not a scheme to instantly determine the data to be written to disk. If the data were tagged as it were written to the cache, complications would arise when data is overwritten. Instead, when data is written to the cache, the system keeps track of which logical blocks of the file need to be written to disk, and when actually performing the disk write, it does the equivalent of a "gather" operation. In essence, it performs an on-demand lookup operation to determine what data needs to be written to disk.

Compared to a standard Unix cache, our cache model requires somewhat more complexity, but not excessively so. The implementation choices in the prototype for this "free-form" cache are suitable for testing, but may be insufficient for large systems. In such environments, more database-like structures, such as B-trees, can yield improved performance, as Silicon Graphics has demonstrated in their XFS filesystem [29]. Furthermore, some policy decisions, such as when copying is beneficial, have been avoided in order to produce a purely copy-free implementation. However, avoiding copying for small amounts of data is not justified from a performance standpoint, so a production system should introduce some small amount of copying. Breaking away from a direct-map approach between cache pages and disk blocks has required some new approaches to cache management, but we believe that our solutions are fairly elegant and relatively straightforward from a conceptual standpoint.

5.3.4 Supporting `mmap`

The mmap operation requires special attention, since its requirements and semantics conflict with an fbuf-based representation of file data. First, memory mapping a file requires that the file's data resides in page-sized, page-aligned buffers. Data stored in fbufs may not meet this requirement, for instance, when it was received from a network interface. Second, memory mapped files may be modified in-place, which conflicts with the immutability of the fbuf-based representation.

The first case occurs when an mmap operation is invoked for a file whose data is currently stored using a layout not suitable for mapping it into contiguous storage. Here, a copy operation is necessary due to hardware constraint; it typically occurs when a file whose data was originally received on a network interface is mmap'ed. 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. The second case is that of a write to a memory-mapped file, where the modified page is also referenced through an immutable fbuf representation. This typically occurs when the file was read using an `IOL_read` operation. The system is forced to copy the modified page, in order to maintain the snapshot semantics of `IOL_read`. The copy is again done lazily, upon the first write access to a page.

It can be shown that the amount of data copied in this scheme is bounded by the amount of data written using write(). In a regular filesystem, all of the data written would be copied, so the data copying in this scheme is not excessive. In regular use, however, we expect that the actual amount of data copying will be very low. The worst case for this approach would be a scenario where two or more processes share a file using mmap as well as the `IOL_read` and `IOL_write` interfaces. Furthermore, the processes would also have to interleave their accesses to the file in order to cause the most work for the filesystem. However, this scenario is very unlikely to occur in real code for a variety of reasons. The interaction between mmap and the standard file system is not defined in many cases, and appears to be very system dependent. Furthermore, programs that share a file and expect to cooperate with each other are likely to access the file in the same manner. Even if two programs shared a file and accessed it in different modes, they would probably share some other communication or synchronization channel to ensure cooperation. For these reasons, we expect the "conversions" to be a low frequency event.

However, it should be noted that not only does our approach provide better coherency than found in many Unix variants, but it also provides it at a very reasonable cost. Imagine a "bad" scenario where two processes have a page mmap'ed and both have written to it. A third process reads part of the page using the `IOL_read` system call, causing an aggregate structure to point to part of the fbuf with the mmap page. At this point, writes to the page by the first two processes must be trapped, since the third process expects immutable data in the fbuf. At this point, the page is essentially in copy-on-write mode, and if either of the first two processes write to the page directly, a copy must be performed. However, only the part of the page represented by the third process's fbuf needs to be copied. Once again, the amount of data copied is equivalent to the amount that would have been copied in a regular filesystem.

The decision to have the third process refer to the same physical data is a policy decision that can easily be changed in the implementation according to the machine characteristics and the expected use of this mechanism. If this type of sharing is common and its performance affects the overall performance of the system, it may make sense to perform eager copying. However, this is a policy decision and in no way affects the semantics of the operations.

5.3.5 Sockets communication

Though sockets are accessed by user processes as though they were associated with the filesystem, inside the kernel the two types of file descriptors are treated quite differently. The sockets code is extremely modular by design, and various interfaces and protocols added to Unix implementations over time have resulted in much of the necessary support code to handle transferring IO-Lite data over sockets.

Theoretically, references to fbufs could be sent through networking code, since fbufs were originally designed for high-speed network communication. However, integrating the fbufs facility with the UNIX network subsystem would be difficult due to the fact that BSD's native buffer abstraction, *mbufs*, is used pervasively throughout the BSD UNIX network subsystem [32]. Using fbufs without requiring intrusive and widespread source code modifications is therefore possible only by encapsulating fbufs inside the mbuf abstraction.

Unix sockets code uses mbufs to transfer data between various layers. Mbufs have a small data area to efficiently store and manipulate small amounts of data, and they can also point to large regions of data in other parts of memory. This "out-of-line data" capability was originally used to avoid copying large amounts of data, but has since evolved to allow handling data not originating with the mbufs system. In Digital UNIX, the mbufs code supports a per-mbuf "free" routine to handle disposal of the the out-of-line data when the mbuf is freed.

The desired encapsulation can easily be accomplished by using the mbuf out-of-line pointer to refer to an fbuf. This flexibility is used to treat mbufs referencing IO-Lite data just as any other reference. This approach maintains compatibility with the BSD network subsystem in a very simple, efficient manner, and it allows the IO-Lite code to also keep track of references properly. Small data items such as network packet headers are still stored inline in mbufs, but the performance-critical bulk data resides in fbufs. Since the mbuf data structure remains essentially unmodified, the bulk of the network subsystem works unmodified with mbuf-encapsulated fbufs.

The out-of-line data can be up to a page in size in Digital UNIX, so large regions of data to be sent are broken into pages and each page is sent using a separate mbuf. The sockets code treats the IO-Lite data just as any other data, and the IO-Lite code is able to correctly handle the interfacing since it provides its own "free" routine to the sockets code. As a result of this system, IO-Lite data can be sent over sockets to code not using IO-Lite, at which point the receiving code may have to perform a copy. When the receiving code is also using IO-Lite and uses the IO-Lite library to perform a read, the system can detect that the data was sent from an IO-Lite-aware application and avoid the copying. In fact, the system also recognizes the fact that a large data region was sent through the sockets code as individual pages and reassembles the data into its original contiguous form. This optimization not only prevents fragmentation when two processes are communicating using IO-Lite, but it also reduces the processing overhead (read permissions, address space translations, etc.) involved with the the transfer, since fewer regions are handled.

Limited modifications are required to device drivers. The fbufs system gains efficiency by repeatedly using buffers on the same I/O stream. Therefore, unallocated fbufs are maintained in pools associated with I/O streams. Device drivers that allocate buffers for input data must be modified to specify, at the time of allocation, the I/O stream for which the buffer is to be used. Network interface drivers must determine this information from the headers of incoming packets. For other drivers, the I/O stream used is usually implicit in the I/O request that the driver receives from higher layers of the OS.

Chapter 6

Related Work

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. This makes it very difficult, in general, to deliver data to the requested location without copying. 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. Likewise, these semantics can not in general be implemented without physical copying of data.

As a result of this known limitation, several alternate I/O interfaces have been proposed and implemented. We next discuss a number of I/O APIs that are in current use, or have been proposed in the literature. In particular, we highlight how they compare with the new IO-Lite API in terms of their suitability as a standard, high-performance I/O interface.

The `mmap` operation can be found in many modern UNIX systems. It allows an application to map an external data object (e.g., a file) at a contiguous range of addresses in its address space. The mapping usually has share semantics, that is, store operations to the mapped memory locations directly modify the associated data object[§]. As already discussed, while the share semantics of `mmap` facilitate a copy-free implementation, the requirement of a contiguous mapping may still demand copying. Specifically, `mmap` can be implemented without data copying if and only if the data can be placed contiguously within memory pages as it arrives from the source device. This is generally the case for data that is read from a local disk, but not necessarily the case for data that arrives on a network.

Data copying as part of I/O operations is widely recognized as an impediment to performance [24, 9, 25, 30, 27]. Many researchers have recognized that a copy-free I/O data path cannot be achieved in general without a change to the UNIX read/write I/O interface, and have proposed non-standard interfaces [27, 21, 25, 30, 12]. In section 2.3, we have discussed one possible means of trying to make the UNIX interface copy-free and why we feel that such an effort is not likely to yield useful results. For the amount of work involved, the situation of such transparent schemes to avoid copying is precarious at best, since simple changes in the program can defeat them. Moreover, if the programmer is required to make changes to exploit the system to some degree, it makes sense to make such a scheme more open and resilient to minor modifications. The IO-Lite API is an attempt to consolidate these earlier efforts, and propose a general, implementation-independent interface that can be used as a basis for a standard high-performance I/O API.

[§]Optionally, private mappings are possible, such that modification of the mapped image do not affect the external data object.

A number of research projects have produced optimized I/O systems that eliminate data copying from certain data paths, usually the I/O data path between a single application and the network adaptor [27, 30, 2, 16, 11]. Section 2.1 shows why such systems, while useful in some situations, are not sufficient for general-purpose copy elimination. As previously discussed, such specialized systems are not easily extensible beyond their original domains. To the best of our knowledge, Container Shipping [25] and IO-Lite are the only systems that provide general, copy-free I/O. The CS implementation is constrained by its API semantics to use page remapping. Thadani and Khalidi's system [30] is based on a simplified form of fbufs; copy-free data transfer between user processes is not supported.

Techniques for copy avoidance range from shared memory channels [27, 16], page remapping [25, 31], copy-on-write [14], fbufs [30, 11], to user-space mappings of network adaptors [2, 11]. The IO-Lite API was designed to allow efficient implementation with each of these techniques for copy avoidance.

The *Container Shipping* (CS) I/O system [25] uses I/O read and write operations with handoff (move) semantics. A read operation produces, in the application's address space, a multi-part buffer that contains the requested data. A write operation removes the multi-part buffer passed as an argument from the application's address space. While holding a buffer, an application can modify the buffer's contents in-place. The multi-part buffer data structure (an array of pointers, each referring to a contiguous sub-buffer) is defined as part of the API, and applications manipulate this data structure directly.

Like IO-Lite, the CS API requires applications to process I/O data at a given location and with a potentially discontiguous storage layout. Unlike IO-Lite, the CS API allows applications to modify I/O buffers in-place. This is safe because the CS API's 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. In essence, the designers of the CS API trade off the ability to permit concurrent sharing of I/O buffers for the added flexibility of writable buffers.

Sacrificing concurrent sharing comes at a cost: Since an application loses access to a buffer that it passes 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 applications reads from a file while a second application is holding buffers for the same file, a second copy of the data must be read from the input device. The first situation arises whenever an application needs to retain data it writes to an output device. The latter scenario occurs, for instance, in network servers that employ multiple server processes to allow concurrency in handling client requests. When several clients concurrently request the same document, CS requires that multiple copies of the document are read from the server's local disk.

Thadani and Khalidi [30] propose an I/O API that has similarities with both the CS API and our IO-Lite API. Like the CS API, their API provides I/O read and write operations with handoff semantics. In addition, T&K's API provides a special mmap operation such that pointers into a mapped region can be used as arguments to their I/O write operation. Unlike both CS and IO-Lite, their API has no provisions for multi-part buffers. To avoid copying, the T&K read operation returns only as much data as can be obtained from the input device contiguously.

The T&K API interface is based on the same trade-off with respect to buffer access semantics as the CS API, i.e., it allows in-place modification of buffers at the cost of the inability to support concurrent sharing. When compared to the CS and IO-Lite API, it generally requires a larger number of system calls to read and write data that originates from a network device, since only contiguous data fragments can be read or written. This is because network data arrives generally in the form of independent fragment packets, and the T&K API does not support multi-part buffers.

Brustoloni and Steenkiste [5] implement several different sets of semantics for transferring data between applications and the operating system in their Genie system. They conclude that the performance of the various strategies other than actual copy is so close that "emulated copy" makes sense from a standpoint of maintaining the current interfaces. They obviously implement many strategies in order to evaluate the performance of the various schemes, but where IO-Lite differs from their work is that they do not consider interprocess communication, nor do they interact with the filesystem. For the range of applications that IO-Lite was designed to handle, emulated copy semantics is not sufficient.

While both the CS and the T&K APIs have influenced our design, we feel that neither of them in their current form is suitable as a basis for a standard, high-performance I/O interface. Their most significant common shortcoming is the handoff semantics. These semantics reflect an implementation choice, which trades off concurrent sharing for the ability to modify buffers in-place. Unfortunately, this constrains any implementation of these APIs to allow only sequential sharing of buffers. Transfer mechanisms based on shared memory cannot be safely used. A standard API should not reflect such implementation choices, and in particular, it should not preclude aggressive implementations that attempt to avoid data copying in all cases.

The IO-Lite API permits safe concurrent sharing of buffers, and thus lends itself to implementations based on variations of shared memory, including fbufs. Such implementation not only avoid copying, but also eliminate most VM operations. VM mapping changes can have substantial costs, particularly on shared-memory multiprocessor machines [4]. Moreover, the IO-Lite API can also be implemented using simpler mechanisms based on page remapping, such as the Container Shipping I/O system. In other words, the IO-Lite API allows a range of implementations with differing complexities and various degrees of aggressiveness in avoiding data copying. This, in turn, ensures that the API can be implemented with reasonable effort on a wide range of OS platforms.

The IO-Lite API's encapsulation of buffers in an abstract data type provides additional flexibility. Implementations can modify the data structures, locking protocols, and referencing schemes for I/O buffers without disturbing the API. Furthermore, applications can only reference I/O buffers by using the appropriate operations exported by the buffer aggregate ADT. This allows aggressive optimization such as lazy attachment and mapping of I/O buffers into the application's address space.

Chapter 7

Results

The performance of our prototype implementation of the IO-Lite I/O system is evaluated in this section. The implementation takes the form of a loadable kernel module that can be dynamically linked to a running Digital UNIX OS kernel. In addition, the implementation includes a runtime library that must be linked with any application that wishes to use IO-Lite. This library implements the buffer aggregate ADT, provides stubs for the IO-Lite related system calls, and includes 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 and FDDI 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.

7.1 Transfer microbenchmark

Figure 7.1 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. This test is designed to determine the rate

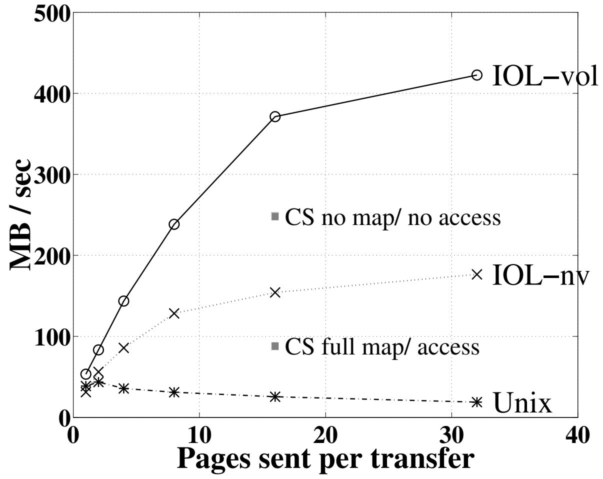

Figure 7.1 UNIX domain socket throughput

at which data can be "made available" between two processes. The stream socket send and receiver buffer sizes are 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). "IOL-vol" and "IOL-nv" correspond to runs with the volatile fbufs optimization [12] enabled and disabled, correspondingly. This optimization can be safely used whenever the originator process of the data is a trusted process (i.e., the kernel), or the receiving process has explicitly enabled the optimization using `IOL_control`.

The processes read and write a small amount of data to demonstrate that the pages in question are being mapped and accessed. Some systems that are designed to perform some of the same functions as IO-Lite have different access models (and performance characteristics) based on whether the data being transferred can be touched by the processes communicating. IO-Lite avoids this two-tiered approach, instead allowing full performance with the option of accessing the data. The processes could have read and written all of the data, rather than just a small amount per page, but then the test would be measuring bandwidth of the various layers of the memory hierarchy. Such a producer/consumer test is presented as a separate benchmark.

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 [25]¶. 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 370 MB/s achieved with IOL-vol, where the data is accessed by the receiver.

The difference in performance between CS and IO-Lite in this experiment is directly attributable to the different semantics of their respective APIs. The CS API's handoff semantics force its implementation to map and unmap buffer pages on each transfer. The IO-Lite semantics, on the other hand, allow its fbuf-based 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 volatile fbufs disabled, 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.

The difference in performance between IO-Lite and Unix is another matter entirely. Since the `read/write` system calls perform copies, the throughput of this test is entirely bandwidth-limited by the memory hierarchy. If only a small amount of data is being transferred, the first- and second-level caches are effective. An interesting feature of this test is that the Unix performance results decrease with the number of pages transferred. One would normally assume that the performance numbers would improve as the amount transferred increases, since any setup cost would be amortized over larger regions. However, the effect of the memory hierarchy dominates this test, and as transfer size increases, the data comes

¶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).

from lower levels of the memory hierarchy. With the largest transfer sizes, the test gains very little benefit from the caches, and the throughput is tied almost entirely to the main memory bandwidth of the machine.

The IO-Lite performance numbers in the "volatile fbufs" case scale quite well as the number of pages transferred increases. This scaling is due to the fact that the per-transfer overhead costs are incurred less frequently. However, past the point of 16 pages per transfer, the per-page overheads, combined with the limitations of buffer sizes, become more of a factor and performance levels off. In the case of "nonvolatile fbufs," the per-page overheads are higher due to the access permission changes, and the per-page costs quickly dominate, beginning to flatten the curve after the 8 pages/transfer mark.

7.2 Network microbenchmark

The results of a network loopback test are shown in Figure 7.2. 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 maximum transmission unit (MTU) of 4352 bytes is used by TCP/IP. Of course, the data never reaches the network interface, since communication is local. However, the FDDI interface was specified as the communication endpoint since the choice of interface does have an effect on how the data is "fragmented" into mbufs by the networking code. When using the "loopback" interface for the tests the throughput values for all three cases dropped. The loopback interface on the test machine has a MTU of 1536 bytes, which causes more processing to occur for "fragmentation" of the data. The fragmentation being discussing is not necessarily a physical fragmentation of the data, but instead a logical fragmentation of the data. In the networking layer, the mbufs used to refer to the data are created to hold no more than the MTU size for these tests. So, a smaller MTU causes more mbufs to be created and then reassembled,

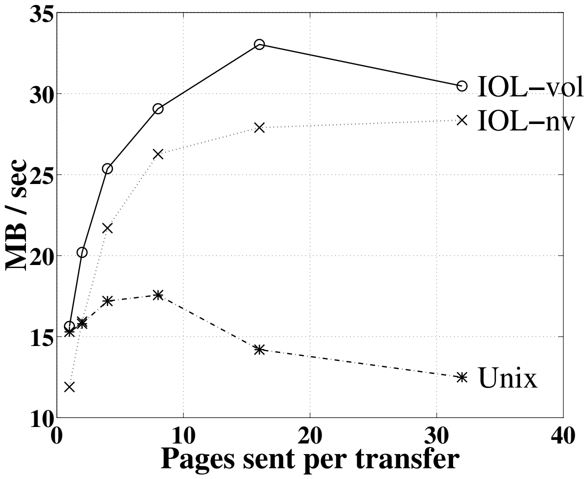

Figure 7.2 UNIX INET loopback throughput

and the resultant increase in processing overhead reduces the throughput. In order to get the highest throughput possible on this system, the FDDI port is specified, since it has the largest MTU size.

Again, IO-Lite outperforms the standard UNIX I/O system dramatically, but the absolute throughput numbers are more modest, due to the overheads in the TCP/IP protocol stack. Data is sent between processes on the same machine to eliminate the physical network speed from this test. However, this test is not a perfect simulation of an infinite-speed network for two reasons. Were this really a test of such a network, the sending and receiving processes would be on different machines. With the two processes on a single machine, the same processor has to perform both the send and receive processing. The other reason this test is not perfect is that the number of copies in this test is not the same as the number of copies in a real network. For example, the IO-Lite tests can avoid all copies in this test (assuming that the network code performs no copies), but in a real network, the data would have to be copied from main memory onto the networking hardware. So, this test also removes some of the memory bandwidth effects that would be seen in an infinite-speed network test.

The results do show some other interesting features. The shape of the Unix curve is interesting because it first rises then drops. The explanation behind this shape is simple, and relates directly to the startup overhead and the cache hierarchy. As more pages are sent in each transfer, the startup overhead is amortized over more pages and the throughput increases. However, the memory hierarchy effects are seen past 8 pages/transfer, so the curve rises at first, then falls. Our second-level cache is 256 kBytes, so the marked decrease at 16 pages/transfer makes sense. Pages are 8 kBytes each on this system, so 128 kBytes are being transferred each time. The data has to be copied from the sender process into the kernel, and then copied into the receiver process. Three sets of 128 kBytes each are involved, which overflows the second-level cache. Likewise, the curve for IOL-vol also reaches a peak before falling again. The minor drop seems to be cache-related, even though it takes place past 16 pages/transfer. Since the IO-Lite cases do not have to copy data, the cache effects will be seen for the bookkeeping structures, and the point at which the cache overflows is not so easily determinable. The IOL-nv case does not exhibit the same behavior with respect to a decrease in performance late in the curve. However, its performance begins to flatten past 8 pages/transfer, so the lack of a decrease past 16 pages/transfer is not surprising.

One final interesting point about the graph is that the IOL-nv case for 1 page/transfer exhibits lower performance than the Unix test. At first, this seems disappointing, since a copying system seems to perform better than a copy-free system for small amounts of data. While it is true that there will be some breakeven point between copying and the structures needed for a copy-free system, this case does not necessarily measure that breakeven point. The transfer of single pages requires so little memory that the entire test should fit into the second-level cache of our system. The data being copied should fit, for the most part, into the first-level cache. So, in some sense, the Unix figures for the small cases are inflated. The IOL-nv case also has to manipulate page protections, which the Unix case does not do, since the buffers in the Unix case are already part of the address spaces of the respective processes. The "penalty" for using nonvolatile fbufs, is, unsurprisingly, a higher breakeven point with respect to copying. The "true" cost of copying in real programs is likely to be higher than what has been measured, since the data is not always going to be coming from the higher levels of the memory hierarchy. The throughput of IOL-nv at one page, even with the high startup overheads, is still on par with the main-memory copy throughput (which will be lower than the Unix throughput for 32 pages/transfer).

7.3 Application benchmarks

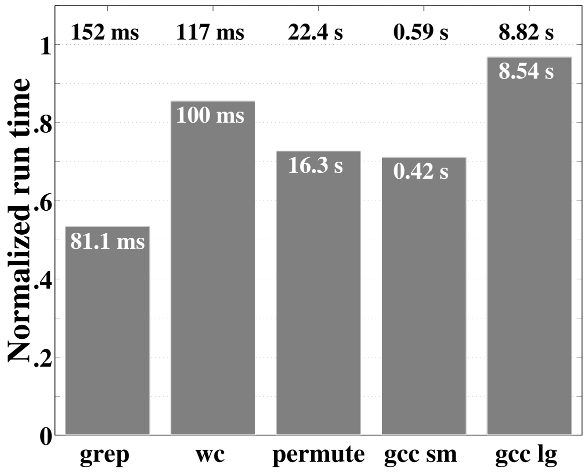

Figure 7.3 Various application runtimes

To demonstrate the impact of copy-free I/O on the performance of real 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 7.3 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, so the performance of wc is affected by the mapping. For the "permute" program the improvement is more significant (24%). The reason for the improved performance is that a pipeline is involved in the latter program between the producer and the consumer. Whenever local interprocess communication occurs, the fbuf-based IO-Lite implementation can recycle buffers, avoiding all VM map operations once the initial mappings have been established. 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 volatile fbuf optimization is enabled in all cases, because the kernel is the originator of file data (grep and wc), or the receiver can safely use volatile fbufs (permute).

The gcc compiler chain was converted mainly to test the appropriateness of the IO-Lite API, to see how hard it is to convert a substantial implementation, and to stress the IO-Lite implementation. A compiler is too compute intensive to benefit substantially from avoiding data copying in its input/output. Interestingly, converting the compiler to use IO-Lite actually led to a noticeable performance improvement. The improvement is mainly due to the fact that IO-Lite allows efficient communication through pipes. The standard gcc has an option that causes the use of pipes instead of temporary files for communication between the compiler's various stages. The option avoids unnecessary disk operations; however, various inefficiencies in the handling of pipes overwhelm the expected benefits. As a result, the IO-Lite version is compared versus the non-piped invocation of gcc. 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 file 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, as outlined in Section 4.3. 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 the IO-Lite-based version of stdio library and making function calls to enable the 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.

The final experiment evaluates the performance of an ftp server based on IO-Lite. A client, connected to the server via a dedicated FDDI network, repeatedly requests a large file transfer from the server. The file is in the server's file cache. Figure 7.4 shows the measured throughput and server CPU utilization during the transfer for 3 cases - the regular ftp server, the server modified to memory-map the file, and finally, for the server modified to use IO-Lite. In all cases, the server attempts to send 64 kBytes of data at a time [‖], and it also resizes the TCP send and receive buffers to their maximum. Since the regular ftp server is able to almost saturate the FDDI network, most of performance improvement in the memory-map and IO-Lite cases is in the form of reduced CPU utilization, which benefits concurrently executing applications.

The CPU utilization figures show the benefits of IO-Lite and copy reduction. The transition from the regular server to the memory-mapped case eliminates one copy - from the filesystem to the user code. The elimination of this single copy results in a reduction in CPU utilization of roughly 20%. Without using IO-Lite, this reduction is effectively the only optimization available to the programmer, since the filesystem buffers are not shared by the network. The outbound path still has to copy the filesystem data into the network buffers, and this copy is eliminated by using IO-Lite, since the buffers used to hold data can be shared by the various components of the operating system. This reduction in copying causes another 20% reduction in CPU utilization. A look at the CPU utilization of the regular case indicates that the CPU is effectively at 100% utilization, so a faster network would have little effect in this case. In contrast, on a faster network, the use of IO-Lite

[‖] We increased the size of this buffer from the standard 8 kBytes in order to raise the bandwidth utilization of the copying case. We found 64 kBytes to be a reasonable size for minimizing various system overheads.

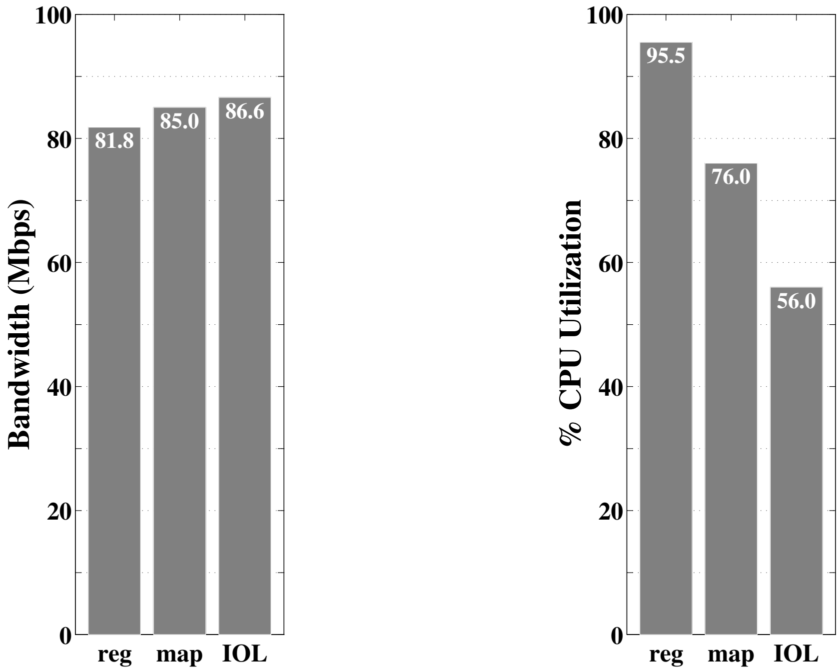

Figure 7.4 FTP server performance

would result in a substantial throughput improvement, since under IO-Lite, the CPU has free cycles on the FDDI network. Our result in the ftp test is roughly comparable to that reported by Thadani and Khalidi [30]. Their results were obtained on a slower machine and a faster network. Therefore, improvements in their system lead to a more substantial throughput gain, whereas improvements in our experimental setup lead to a hefty reduction in CPU utilization.

7.4 Resource consumption

An obvious question raised by the unification of the buffering systems is what effect it has on resource consumption. Three resources that are of interest are CPU time, memory bandwidth, and memory usage. The first two are clearly interdependent since copying fewer bytes of data results in using less memory bandwidth and less CPU processing. The ftp tests demonstrate this effect clearly, since the only real difference between the regular ftp code and the IO-Lite ftp code is the amount of data being copied. We also expect less copying to occur in the filesystem tests, although we expect some additional CPU overhead to manipulate the internal data structures and reference counting. The reduction in CPU usage and memory bandwidth usage is fairly straightforward and simple to understand. However, the issue of memory consumption is slightly more complicated.

Because of the historical scarcity of memory, careful programmers (and operating system writers) paid special attention to keeping memory consumption small. For example, grep reads page-sized blocks of data when not memory-mapping files, and interprocess communication buffers are very often on the order of 8 kBytes. These relatively small buffer sizes are intended to limit memory usage. Replacing these buffers with IO-Lite buffers, therefore, only reduces the memory consumption of these programs by a modest amount, since the same buffers are shared by different applications. In some cases, such as when grep is able to memory-map the source file, the use of IO-Lite does not result in any memory savings, since memory-mapped files also act as shared buffers**. The low degree of memory savings is not surprising given that well-written programs have already paid attention to reducing memory usage. However, given that memory capacity is cheap, whereas memory bandwidth and CPU performance are expensive, the fact that IO-Lite helps reduce CPU usage and bandwidth usage more than it helps reduce memory consumption is not terribly disappointing.

While IO-Lite may not reduce permanent memory consumption significantly in well-designed applications, it can reduce transient memory consumption, especially within the operating system. Consider the network and interprocess communications buffers - these buffers are used to copy data from user space for short periods of time. However, when applications become communication-bound, these "temporary" buffers effectively become permanent, since they are constantly in use. In this scenario, the use of IO-Lite can reduce memory consumption, since no separate network buffers are needed. Reliable protocols like the very popular TCP/IP protocol need to keep sent data until the receipt of the data has been acknowledged, so the amount of buffer space used can become large. The size of the send and receive buffers can usually be controlled by the application, and their maximum sizes are usually on the order of 256 kBytes per buffer per connection. However, as the bandwidth-delay product for networks grows, and as applications become more communications-dependent, these buffer sizes will need to increase to prevent starvation. As this occurs, the use of IO-Lite will provide greater benefits, since the sharing of application buffers with the network code will result in a greater space savings along with the previously mentioned reductions in CPU usage and memory bandwidth consumption.

The previous discussion illustrates another important aspect of IO-Lite, which is that while IO-Lite can benefit existing applications, it can also enable new features and systems which would otherwise have been difficult to implement. Just as the integration of filesystem cache pages and regular VM pages eliminated the need to statically "tune" the cache size, the unification of buffer mechanisms permitted by IO-Lite allows the rethinking of previous assumptions. Programs like grep no longer have to read in small amounts, because if the data already exists, no additional memory is consumed in order for grep to access it. Likewise, limits on interprocess communication buffers and network buffers can be relaxed or disappear entirely, since the buffers are no longer owned by those domains, but instead are part of the applications themselves. Applications that use the filesystem to store temporary data as a means of avoiding the overheads associated with interprocess communication can profitably switch to pipes. The benefits from doing so are numerous:

- Parallelism - If a series of steps are run using pipes instead of temporary files, the processes can be run in parallel on a multiprocessor. Even on a uniprocessor, some benefit is obtained by allowing one process to run while another is blocked on I/O, etc.

**Our copy-based ftp example would save 64 kBytes going to either mmap or IO-Lite, since we increased the buffer size from 8 kBytes to 64 kBytes

In contrast, in the case of using temporary files, each step of the process must finish before the next starts.

- No disk interaction - File creation, data flushing, and file deletion are events that are handled synchronously in most UNIX-based operating systems, and for small files, these operations dominate the cost of the actual data transfer itself. Processes that use pipes can avoid these slow events.
- Memory usage - When using pipes, the amount of memory needed to transfer the inter-process data is lessened, since the pages used to transfer the data can be freed and recycled as soon as the data is consumed. An example of this effect is the "permute" test program - it generates over 270 MBytes of data, but only a very small amount of that data is actually extant at any one time, so relatively few pages are used for transferring the data.
- No cache pollution - This effect is a corollary of not having the data go through the filesystem, but it is an important corollary. When a large amount of data goes through the filesystem, it can force out other data in the filesystem cache. This "cache pollution" can be mitigated by filesystem hints, or by code within the filesystem itself that tries to detect this case and minimize its effects. However, using pipes prevents the adverse effects from the start.

Chapter 8

Contributions of this work summarized

This work makes several contributions to the area of high-performance I/O, both in design and implementation. From a design standpoint, we have proposed an implementation-independent API that can be used in conjunction with a variety of copy-minimizing systems, and that can serve as a standard interface for programmers and system implementors. We have also designed and implemented a copy-free disk cache, and we believe this is the first disk cache that can support totally copy-free operation. This project also makes a number of more evolutionary steps, such as the first implementation of fbufs in a UNIX-based system, and the extension of fbufs to support filesystems.

The detailed design of the filesystem cache is explained in section 5.3.3 and the API is explained in section 4. To the best of our knowledge, our filesystem cache is the first cache of its type that supports arbitrary boundaries, and it does so in an elegant and relatively straightforward manner. The API has been designed to be flexible and usable, while not hindering performance or implementation choices. We have shown one possible implementation of the API, and the performance results lend support to the idea that the API does not hinder the performance of the underlying implementation. We have also argued that the IO-Lite API should be realizable on other systems, and in section 4.2, we provide support for why we believe that the API can be "universal."

Performance tests, especially in the area of sending data through the networking system, have demonstrated that the copy-free approach is effective and efficient enough to move the bottleneck to other parts of the operating system. We have also shown that converting real programs to use IO-Lite is possible and practical for IO-intensive programs. Furthermore, libraries like the stdio library can be converted to use IO-Lite, thereby providing benefit to programs without directly rewriting the programs. We believe that library usage of IO-Lite will be an attractive option for libraries designed to be included in high-performance programs.

Finally, it should be noted that this system has been designed to be implementable on a variety of existing systems, and the prototype implementation works with an existing UNIX-based system. We have not required special hardware support for this system, nor have we designed anything that requires special OS support. We have implemented our system with just a small kernel module and an application library. Though our system is only a prototype and does not contain all of the features expected in a production version of the system, it is usable and provides concrete proof of the concepts behind the system.

Chapter 9

Future work

While we feel significant progress has been made on IO-Lite, some new questions have been raised and the implementation is not complete. In this section, we discuss what is left to be designed, what is yet to be implemented, and new areas to be explored. Obviously design and new research are important, but the issue of implementation is crucial in this area of research, since these systems are too complex for accurate performance evaluation using only theoretical systems.

In the area of design, there are still some open issues relating to the VM system, specifically in the area of writing pages to disk. Since there is no direct mapping from pages in memory to pages on disk, the design does not lead to any obvious answers for how to handle sending pages to secondary storage. We have considered this issue carefully and have determined a guiding principle for the decision-making process about writing to disk. Essentially, we want to view the data on disk as being part of our collection of data, and we want to view data as being "persistent objects." We already view the data in memory in this manner, and we keep reference counts to determine when a buffer has "died." Likewise, we want to elegantly extend this concept to the filesystem, so that we can make decisions about when data on disk can be overwritten safely. We also have the option of tuning the swapping decisions to eagerly swap out pages that have not been used in some time.

We have also considered the idea of lazy attachment of disk data and the systems to support it. In such a system, a process could perform a disk read several times larger than physical memory, and the data would be read as needed. However, the snapshot semantics of the read would be maintained. In most current UNIX-based systems, if a process attempts this, the pages would be read from disk, loaded into memory, and then written out to the swap device. The same data would exist in two places originally, until either the disk file or the process's buffers were modified. Lazy attachment is a means of removing the redundant data from such a scenario and having the operating system intelligently provide data to the process as needed. Such a scenario is possible because of the use of abstract data types in the IO-Lite API and a tight coupling between the library and the kernel module in the implementation.

Though further work is planned on the implementation, the result is not expected to be a full production system, but rather just a proof of concept, intended only to deliver a usable test platform. Some areas that need to be addressed are device drivers, security models, support for `IOL_mmap` and multiprocessor issues. Device driver support and extending the security model should not require much design effort, since our design will be largely based on the design used in the fbufs system. Implementing device drivers for use with fbufs will be necessary to derive the full benefits of an fbuf-capable system, and fleshing out the security system will be necessary to test more complicated programs. The "pool allocators" concept is well-understood and is borrowed from fbufs, so no major impediments to implementation are expected. As mentioned earlier, the current implementation uses only a single allocator per process, but extending that model should be simple. Regarding the support for `IOL_mmap`, the path for the implementation has been outlined, and various issues of handling consistency, etc., have already been addressed in this work. Support for multiprocessors is an issue that will require some patience but not necessarily much design. The prototype uses immutable data storage where possible, and single ownership otherwise, so we expect that we will have to perform little locking in our system. However, multiprocessor support can range anywhere from fine-grained parallelism to coarse-grained locking, so this issue is one of patience rather than pure design.

One interesting issue arising from IO-Lite is the idea of implied semantics and their restrictiveness. We have seen that the standard POSIX `read` system call implies location of data, amount of data, contiguity of data, and atomicity. IO-Lite relaxes the restriction on where the data is to be placed, the contiguity of data, and in some sense, the amount of data returned. Our read-equivalent call, however, goes one step beyond the normal POSIX call and also allows "short reads" under non-error circumstances. Our current implementation, however, does not utilize this feature, and the library hides a short read by performing the read again with a longer return buffer. It would be interesting, though, to see what programs really need the implied atomicity of a read and to see if there is a meaningful way of providing other alternatives. For files that are not being shared, or when some other mechanism already exists to ensure serialized access, the implied atomicity is superfluous.

Chapter 10

Conclusion

This work makes two contributions. First, it proposes a new, high-performance I/O API whose semantics permit a wide range of implementations. Most notably, the API's unique data access model based on immutable buffers facilitates shared-memory based implementations that are both safe and highly efficient. Unlike other VM-based data transfer techniques, shared memory based facilities can avoid all VM map modifications associated with the transfer of data in the common case. This is important, because VM map modifications, while more efficient than data copying, still have significant costs. This is of particular concern on multiprocessors, where expensive TLB operations and potential cache consistency actions are required. Our preliminary experience suggests that applications can be written and modified to use the IO-Lite API with relative ease.

Second, we describe and evaluate an aggressive implementation of the IO-Lite API that eliminates all redundant data copying along I/O data paths, irrespective of the data's source and sink, and the number of user processes traversed. Experiments show that I/O-intensive applications enjoy significant performance benefits from the use of IO-Lite. Improvements are most pronounced in cases where user-to-user communications occurs (through pipes or UNIX domain sockets), since the fbuf-based implementation can usually effect data transfer without any VM map manipulations in this case. This is important, for instance, for multimedia applications, which usually involve high-volume local communication with a window manager.

A public release of the IO-Lite prototype systems for DEC Alpha systems running Digital UNIX V3.2C is planned for the Winter of 1996.

Appendix A

Appendix: API manual pages

`IOL_concat` - buffer aggregate concatenation
`IOL_control` - manipulation of control options
`IOL_create` - buffer aggregate allocation
`IOL_create_allocator` - allocator creation
`IOL_destroy` - buffer aggregate deallocation
`IOL_destroy_allocator` - allocator destruction
`IOL_duplicate` - buffer aggregate duplication
`IOL_gen_end` - generator-style traversal shutdown
`IOL_gen_nextRd` - generator-style traversal for reads
`IOL_gen_nextWr` - generator-style traversal for writes
`IOL_gen_seek` - moving to arbitrary locations in a generator context
`IOL_gen_start` - generator-style traversal setup
`IOL_gen_tell` - determining current position of generator-style traversal
`IOL_length` - buffer aggregate length query
`IOL_mmap` - map a file into memory
`IOL_read` - reading data from an object
`IOL_select` - buffer aggregate range selection
`IOL_split` - buffer aggregate splitting
`IOL_write` - writing data to an object
`IOL_write_destroy` - writing data to an object and deallocating the buffer aggregate

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

SYNOPSIS

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

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

DESCRIPTION

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

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

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

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

SYNOPSIS

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

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

DESCRIPTION

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

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

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

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

SYNOPSIS

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

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

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

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

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

DESCRIPTION

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

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

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

IOL_duplicate creates a new buffer aggregate and returns a pointed to it in dst. The new buffer aggregate contains the same contents as src.

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

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

SYNOPSIS

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

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

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

IOL_Err IOL_gen_end(ctxt);
IOL_Context *ctxt;

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

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

DESCRIPTION

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

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

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

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

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

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

SYNOPSIS

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

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

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

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

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

DESCRIPTION

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

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

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

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

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

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

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

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

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

Bibliography

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for Unix development. In *Proceedings of the USENIX Summer '86 Conference*, July 1986.

[2] A. Bas, V. Buch, W. Vogels, and T. von Eicken. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 40–53, 1995.

[3] T. Berners-Lee, R. Fielding, and H. Hrystyk. Hypertext transfer protocol – http/1.0. Request for Comments 1945, MIT Laboratory for Computer Science (Berners-Lee, Frystyk), UC Irvine (Fielding), May 1996.

[4] 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.

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

[6] D. C. Burger, J. R. Goodman, and A. Kagi. The declining effectiveness of dynamic caching for general-purpose microprocessors. Technical Report CS-TR-95-1261, University of Wisconsin, Madison, WI, 1995.

[7] D. C. Burger, J. R. Goodman, and A. Kagi. Quantifying memory bandwidth limitations of current and future microprocessors. Technical Report CS-TR-96-1295, University of Wisconsin, Madison, WI, 1996.

[8] J. Carter and W. Zwaenepoel. Optimistic implementation of bulk data transfer protocols. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (Sigmetrics '89)*, pages 61–69, May 1989.

[9] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the SIGCOMM '90 Symposium*, pages 200–208, Sept. 1990.

[10] C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley. Afterburner. *IEEE Network*, 7(4):36–43, July 1993.

[11] P. Druschel, B. S. Davie, and L. L. Peterson. Experiences with a high-speed network adaptor: A software perspective. In *Proceedings of the SIGCOMM '94 Conference*, pages 2–13, London, UK, Aug. 1994.

[12] 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.

[13] K. Fall and J. Pasquale. Exploiting in-kernel data paths to improve I/O throughput and cpu availability. In *Proc. Usenix Winter Technical Conf.*, San Diego CA (USA), Jan. 1993.

[14] R. Fitzgerald and R. F. Rashid. The integration of virtual memory management and interprocess communication in Accent. *ACM Transactions on Computer Systems*, 4(2):147–177, May 1986.

[15] R. P. Gingell, J. P. Moran, and W. A. Shannon. Virtual memory architecture in SunOS. In *Proceedings of the 1984 Summer Usenix Conference*, Phoenix, AZ (USA), June 1984. USENIX.

[16] R. Govindan and D. P. Anderson. Scheduling and IPC mechanisms for continuous media. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 68–80, October 1991.

[17] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., Palo Alto, California, 1990.

[18] N. C. Hutchinson and L. L. Peterson. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan. 1991.

[19] Intel Corporation, Santa Clara, California. *Intel Architecture MMX Technology Programmer's Reference Manual*, 1996.

[20] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley Publishing Company, 1989.

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

[22] J. D. McCalpin. A survey of memory bandwidth and machine balance in current high performance computers. *IEEE Technical Committee on Computer Architecture (TCCA) Newsletter*, Dec. 1995.

[23] S. O'Malley, M. Abbott, N. Hutchinson, and L. Peterson. A transparent blast facility. *Journal of Internetworking*, 1(2):57–75, Dec. 1990.

[24] J. K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proc. Summer 1990 USENIX Conf.*, pages 247–256, Anaheim, CA (USA), June 1990. USENIX.

[25] 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.

[26] D. J. Shippy and T. W. Griffith. POWER2 fixed-point, data cache, and storage control units. *IBM Journal of Research and Development*, 38(5):503–524, 1994.

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

[28] Sun Microsystems, Inc. Ultrasparc, the visual instruction set (VIS): On chip support for new-media processing. White Paper Whitepaper 95-022, Sun Microsystems, Inc., Mtn. View, CA, 1996.

[29] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *USENIX 1996 Annual Technical Conference*, pages 1–14, San Diego, CA, Jan. 1996.

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

[31] S.-Y. Tzou and D. P. Anderson. The performance of message-passing using restricted virtual memory remapping. *Software—Practice and Experience*, 21:251–267, Mar. 1991.

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

[33] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, pages 63–76, Oct. 1987.