An Efficient Zero-Copy I/O Framework for UNIX®

Moti N. Thadani
Yousef A. Khalidi

SMLI TR-95-39 May 1995

Abstract:

Traditional UNIX® I/O interfaces are based on *copy semantics*, where read and write calls transfer data between the kernel and user-defined buffers. Although simple, copy semantics limit the ability of the operating system to efficiently implement data transfer operations. In this paper, we present extensions on the traditional UNIX interfaces that are based on *explicit buffer exchange*. Instead of transferring *data* between user-defined buffers and the kernel, the new extensions transfer data *buffers* between the user and the kernel. We study using the new interfaces in typical application programs, and compare their use to the standard UNIX interfaces. The new interfaces lend themselves to an efficient zero-copy data transfer implementation. We describe such an implementation in this paper, and we examine its performance. The implementation, done in the context of the Solaris™ operating system, is very efficient: for example, on a typical file transfer benchmark, the network throughput was improved by more than 40% and the CPU utilization reduced by more than 20%.

Sun Microsystems
Laboratories, Inc.
A Sun Microsystems, Inc. Business
M/S 29-01
2550 Garcia Avenue
Mountain View, CA 94043

email addresses:
moti.thadani@eng.sun.com
yousef.khalidi@eng.sun.com

An Efficient Zero-Copy I/O Framework for UNIX®

Moti N. Thadani *Yousef A. Khalidi*

Sun Microsystems Laboratories
2550 Garcia Avenue
Mountain View, CA 94043

1 Introduction

The advent of high-speed networks such as Asynchronous Transfer Mode (ATM), and new application domains such as multimedia and video-on-demand are increasing the demands for efficient I/O data transfers. Reducing or eliminating the cost of data-copying between applications and operating system kernel is an important step towards an efficient I/O system.

There has been a fair amount of work published in the literature on reducing data-touching overheads. Such efforts include protocol integrated layer processing [2, 5, 11], high-performance network adapters to eliminate copying between devices and the operating system kernel [6], and restructuring operating system software to minimize data movement [4, 6, 8, 9, 15, 17, 18]. Much of the previous work in this area concentrated on reducing-data copying for network protocol processing, particularly for the TCP network protocol [16].

We are interested in providing an I/O framework for the UNIX® operating system that has the following characteristics:

- The framework should minimize or eliminate all copying of data presented by user applications for output, and for data obtained from I/O devices on input operations.
- The framework should be general and should be applicable to more than networking protocols.
- The framework should integrate well with the UNIX operating system, without major modifications to existing UNIX drivers and Streams modules [1].
- The framework should lend itself to efficient implementation on uniprocessor (UP) and multiprocessor (MP) systems.

Previous systems have addressed some of the above issues, but to our knowledge, there is no one system that meets all of our requirements. For example, there is a lot of work in the area of reducing network protocol processing that, in general, is not applicable to other kinds of I/O [2, 5]. Some systems that implement a zero-copy mechanism such as [4, 9] do not meet our requirement for efficient support on MP systems. The promising work of Druschel and Peterson is mostly geared to network protocols in a non-UNIX environment [8]. Finally, most of the previous systems do not address compatibility with existing Streams modules.

In this paper, we present an efficient zero-copy framework for buffer management and exchange between application programs and the UNIX kernel. Our solution has the following components:

- A buffer management scheme based on the *fast buffers* (*fbufs*) concept described in [8].
- An extension of the basic fbufs concept to allow creation of fbufs out of memory-mapped files.
- Extensions to the UNIX Application Programming Interface (API) in the form of new calls that

explicitly pass fbufs when performing I/O operations, as well as calls to allocate and de-allocate buffers. The API extensions are implemented by a new library, *libfbufs*, plus new system calls.

- Inside the UNIX kernel, fbufs are compatible with Streams internal buffers (mblks), fbufs can be manipulated as mblks by existing Streams modules, and no modifications to the modules are needed to use the buffers.
- Extensions to the device driver kernel support routines to allow drivers to allocate and release fbufs.
- Integration of buffer management with the virtual memory system for allocating and de-allocating memory for the buffers, as well as locking/unlocking the memory when buffers are manipulated by the kernel.

A prototype that incorporates the above components has been implemented in the Solaris™ operating system.[1] The modifications to the kernel were kept to a minimum—a version of the prototype is implemented as a loadable module and no Streams module modifications are needed. The implementation is very efficient: for example, a 1MB file transfer between two machines using *ftp* [15] and an OC3 ATM link achieved a 40% higher throughout, with 20% less CPU utilization than a comparable transfer that used a well-tuned 1-copy TCP implementation (Section 6).

In the next section, we present a brief overview of related work. In Section 2, we discuss the need to extend the basic UNIX I/O interface. Section 3 describes the basic scheme, along with the new API and the various components of the framework. We describe our experience in using the new interface extensions in Section 4. Section 5 describes the implementation within the Solaris operating system. We report in Section 6 on the performance of the system using a set of micro and macro benchmarks. Finally, we present our conclusions in Section 7.

2 Related Work

Several systems have attempted to reduce data touching overhead when performing I/O operations. Data touching overheads include those operations that require processing of the data within a given buffer such as checksumming or copying from one buffer to another. “Raw” disk I/O in UNIX is usually implemented with zero-copy, i.e., the data is transferred from disk to user buffers without an intermediate copy into operating system kernel buffers. However, this kind of I/O is highly specialized. Other efforts are generally focused on network I/O or remote procedure call overhead. There is little evidence of published work done to integrate network and disk I/O, for example.

Previous efforts include integrated layer processing [2, 5, 11], elimination of data-copying by careful design of network adaptors [6], and restructuring operating system software to minimize data movement [4, 6, 8, 9, 15, 17, 18]. These efforts can be effective because optimizing a single data-touching operation can produce a large improvement in performance. Clearly, data-touching overheads are most significant for large I/O transfers [12], and reducing data touching costs can have considerable benefits for such operations as image and video transfers over high-speed networks.

2.1 Single copy implementations

A single copy implementation requires one pass over data sent or received from an I/O device (including checksum computation, if any). Most UNIX I/O systems perform at least one copy between user buffers and the kernel. In the case of network I/O, on the other hand, traditionally two copies (plus a checksum operation) are needed. Given the semantics of UNIX I/O application interfaces and the underlying protocols (e.g., TCP), single copy implementations for network operations are not straightforward. Two recent examples of single copy implementations that are tailored for the TCP protocol [16] are described in [6, 18].

When applied to networking protocols, single copy implementations do have several limitations, including:

- **The amount of physical memory used**
  The amount of physical memory used during a copy is twice the size of the data object being transferred: This can be a significant factor for applications that manipulate large data objects. Copies must be delayed if sufficient physical memory is temporarily unavailable [15]. This contrasts with zero-copy implementations which require only as much memory as the size of the data object.

[1] This work is exploratory in nature and this description does not in any way imply that the interfaces described herein will be incorporated in any publicly available version of Solaris.

- **Insufficient interaction with the protocol layer**
  Insufficient interaction with the protocol layer in the construction of outgoing packets: A single copy implementation copies and computes the checksum of the user data at the system call level. Copying/checksumming is done in units that the underlying protocol can send over the network. The size of the units is an estimate based on upper bounds available at that point. Some of the bounds are the capacity of the largest memory buffers allocatable, the maximum segment size negotiated by TCP when the connection was set up, and the largest protocol window the receiver has advertised. Unfortunately, the bounds can be too large. In a congested network or with an overloaded receiver, the effect (implementation-dependent) of picking a large value can vary from causing unnecessary traffic on the network to increasing processing costs at the sender.

- **Delayed checksum calculation**
  Delayed checksum calculation and its effects on increasing the latency on acknowledgments: In both [6] and [18], checksum computation is combined with a single data copy in each of the receive and transmit paths. The cost of a combined copy and checksum operation is comparable to a copy alone. However, acknowledgments can only be transmitted after the checksum is calculated. Since the checksum is calculated only when data is copied to the application, the latency for acknowledgments may be increased.

  The protocol implementation must be modified to detect that the application has not read data from the connection for an extended period of time (e.g., because the application is waiting for some other resource in the system). This implies overhead to set up additional timers in the best case, and the actual cost of handling timer expiration in the worst case. The data handling in the worst case requires separate checksum and copy operations.

- **Checksum failure**
  Checksum failure results in additional overhead and/or change in semantics: Since the checksum is computed while copying the data into the user buffer, the application buffer may be altered unexpectedly, and applications that expect to retain the original data when a read operation fails will not work correctly. In some implementations, the application program buffer may have to be cleared on a checksum failure to prevent corrupted data from being available to the program or to prevent unauthorized information transfer.

Note that the above two problems can be alleviated with hardware support for checksum handling. However, with hardware checksum support, a more efficient zero-copy implementation is possible.

2.2 Zero-copy implementations

A zero-copy implementation is one in which data from I/O devices is directly received in the user buffer without requiring any access by the kernel (similarly for output between a user buffer and the devices). Note that for most network protocol stacks, including TCP, such a scheme would require checksum computation in the network adapter while data is being received and transmitted.

Several zero-copy implementations have been described in the literature [4, 9, 17]. The implementations typically involve the following virtual memory operations:

- In order to maintain the semantics of the UNIX write system call, the system must do one of the following [7]:

  Either block the user process from modifying the buffer being written, or

  copy the buffer when a modification is attempted by the process while output is being performed (see Section 3.1).

  The first option is undesirable because applications assume that they will continue as soon as the I/O operations is posted (and not necessarily wait until it is completed). The second option is costly, as it requires establishing and removing a copy-on-write mapping on each write operation and relies on the application not reusing buffers; in the case of many legacy applications, an actual copy will occur.
- In order to maintain the semantics of the UNIX read operation, the physical pages holding the data received by the kernel are remapped at the virtual address specified in the read.

Some of these previous systems combine the virtual memory manipulations with checksum support in the network adapters. The major limitation of these systems is in the virtual memory manipulation overheads:

- In the uniprocessor case, the cost of virtual memory manipulations can be comparable to data copy. The performance of schemes that use page remapping and copy-on-write mappings are limited by the time it takes to acquire VM locks, change the mappings in software, and perform any cache and TLB consistency [15, 8].

- The overhead of page re-mapping in multiprocessor systems can be significantly higher than in the uniprocessor case due to translation look-aside buffer shoot-down [7].

A different approach, the *fast buffers* (fbufs) scheme [8], advocates sharing of buffers between the user and the kernel.[2] A key aspect of the scheme is the explicit exchange of buffers when performing I/O operations. Caching of buffer virtual memory mapping is used to take advantage of referential locality. Once data has been exchanged between a the user and the kernel, the buffer is saved for other data exchanges between the two domains. Therefore, page table updates are eliminated in the common case.

To utilize the full potential of the fbufs scheme, system support for two features is needed:

- Hardware checksum support on output and input when interacting with a protocol that requires checksumming the data (e.g., TCP).
- De-multiplexing of incoming data directly into a pre-mapped user buffer at the device driver level (see Section 5.4.1).

3 UNIX Fast Buffers

This section presents the motivation and the concepts that underlie the UNIX fbufs[3] design and implementation. A new application programming interface is discussed along with the underlying buffer management framework and its integration with the Streams I/O subsystem in UNIX. In addition, we extend the basic fbufs mechanism to include file (disk) I/O.

3.1 Why we need to extend the UNIX I/O API

The traditional UNIX input-output interfaces, such as read and write calls, are based on copy semantics, where data is (semantically) copied between the kernel and user-defined buffers. On a read call, the user presents the kernel with a pre-allocated buffer, and the kernel must put the data read into this buffer. On a write call, the user is free to reuse the data buffer as soon as the call returns.

To maintain the above semantics, zero-copy implementations in UNIX have to resort to establishing and removing virtual memory mappings on each operation as described in Section 2. The performance of such page re-mapping implementations is very dependent on the machine configuration, the cache architecture, TLB miss handling costs, and whether the machine is a uniprocessor or a multiprocessor. The performance will always be worse than a zero-copy scheme that does not require page remapping.

It would be highly desirable to simply avoid the overhead of virtual memory/TLB operations on each I/O request. The fbufs approach mentioned in Section 2 is such a scheme, one that does not require *any* virtual memory operations in the common case. As shown in [8], the fbufs scheme performs much better than a page remapping scheme on a DECStation™ 5000/200, a uniprocessor workstation. We suspect the performance difference to be even more pronounced on multiprocessor systems, due to TLB shoot-down costs.

We chose to adapt the fbufs scheme to UNIX since it appeared to be the most promising zero-copy approach. To use an fbufs-like approach, however, an interface that explicitly exchanges buffers is needed. Therefore, we defined new extensions to the UNIX API that enable a very efficient and general zero-copy implementation. After we present the general scheme, we describe the new interfaces in Section 3.3. In Section 4, we show that the new extended interfaces are very easy to use and can coexist with the current interfaces.

3.2 Overview

There is a compelling case for applications to use the fbufs framework. By passing an fbuf (rather than an arbitrary buffer) to the kernel on output, the application is guaranteed a saving in data-copying overheads implicit in the definition of the traditional write system call. Similarly, by receiving data in an fbuf on input, the application is guaranteed a saving in data-copying overheads of the traditional read system call (see Figure 1).

A UNIX application using fbufs must be aware of the need to obtain fbufs and of the transfer of ownership that occurs as part of the I/O operations. Typically, an application may allocate fbufs, generate data in the

[2] The scheme is generally applicable to a set of domains, and not just between the user and kernel domains.

[3] We will use the term fbufs in the rest of the paper to refer to our implementation and extensions of the basic fbufs concept [8].

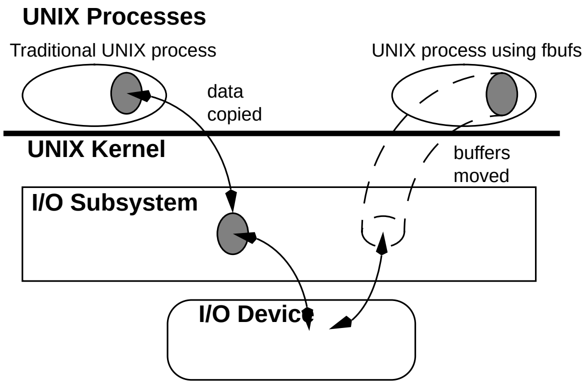

FIGURE 1. UNIX I/O API

The traditional UNIX I/O interface supports data exchange between application and operating system which involves copying each byte of data. The extensions to the API provides for the explicit exchange of buffers (containing data) between application and operating system which eliminates copying.

fbufs, and transfer the fbufs to the kernel as part of output to some device. A second possibility is for the application to obtain from the system an fbuf containing data from a specified device/file. The application may then process the data and either transfer the fbuf back to kernel as part of output (to the same or different device), keep the fbuf, or simply deallocate it.

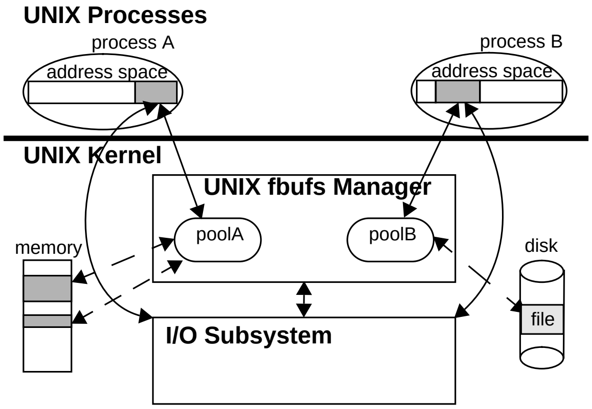

FIGURE 2. Buffer allocation

Fbufs can be allocated by a process or by the I/O subsystem from the fbufs manager. Once allocated, they can be passed from a process to the I/O subsystem or vice-versa, or deallocated and returned to the fbufs buffer pool from which they were allocated. An fbuf is backed by either memory or a disk file.

The following implementation properties of fbufs facilitate this usage model (Figure 2):

- Each fbuf is either owned by an application program or the kernel, or is in an fbufs buffer pool. An fbuf can be transferred from an application program to the kernel or vice versa. Fbufs carry with them information about which fbufs' buffer pool they should be returned.
- Each fbufs' buffer pool is associated with an application program. An application program is associated with at most one fbuf's buffer pool. An application has access to its own pool only, and cannot compromise the security of the kernel or any other process in the system.
- No remapping of virtual addresses is required because the implementation strives to reuse the same set of buffers for each process. Thus, virtual memory translations are cached, and no virtual memory subsystem overheads are incurred in the common case.
- The I/O subsystem (device drivers, Streams modules, and file systems) can allocate fbufs and place incoming data directly in them. Therefore, no copying between buffers is needed.

Finally, we have extended the original fbufs concept to memory mapped files by providing an interface that automatically creates fbufs from portions of application address spaces that are mapped to UNIX files.

3.3 Application programming interface

The fbufs application programming interface provides system calls and library routines to allocate buffers, read data from a device (file) descriptor, and write data to a file (device) descriptor. These interfaces are defined in Table 2.

The **uf_read** and **uf_get** interfaces provide input functions by reading data into a new buffer allocated by the kernel at an address **bufpp*, also allocated by the kernel. The uf_get interface is an alternative to uf_read, applicable only to Streams devices, which inputs as much data as is available at the device, rather than being limited by a parameter.

In the output function, **uf_write**, the buffer at address *bufp* is transferred to the operating system; from the perspective of the application, an implicit **uf_deallocate** occurs before the call returns. The result of attempting to access or modify the buffer **bufp* is undefined after the call is made.

Interface	Description
ssize_t uf_read (int *fd*, void ***bufpp*, size_t *nbytes*)	Transfer buffer containing data from device (file) fd with a maximum size of nbytes from kernel to user.
ssize_t uf_get (int *fd*, void ***bufpp*)	Transfer buffer containing data from device (file) fd from kernel to user.
ssize_t uf_write (int *fd*, const void **bufp*, size_t *nbytes*)	Transfer buffer containing data from user to device (file) fd.
void * uf_allocate (size_t *size*)	Allocate fbuf.
void * uf_deallocate (void * *ptr*, size_t *size*)	Return fbuf.
caddr_t uf_mmap (caddr_t *addr*, size_t *len*, int *prot*, int *flags*, int *fd*, off_t *off*)	Map file fd into the process address space such that the mapped region can be used as fbufs.

Table 1: UNIX fbufs API

The buffer at address *bufp* is either allocated explicitly by a call to the allocator, **uf_allocate**, or implicitly as the result of a uf_read or uf_get. In addition, *bufp* may be a reference to a buffer within a range of the user address space mapped to some other device (file) by using the **uf_mmap** call.

The calls require the buffer passed to them to be an fbuf. If a non-fbufs buffer is supplied by the application, the framework returns an error. We considered an alternative definition of invoking the standard write system call when using uf_write with a non-fbuf buffer, but decided against it because buffer ownership, which is intrinsic to the semantics of the uf_write call, cannot be changed for an arbitrary buffer. For consistency with the uf_write interface, uf_read and uf_get also require an fbuf buffer.

4 Programming Using the New API

The new API required by our fbufs framework makes it necessary to use a slightly different programming interface from the standard UNIX I/O API. We studied some typical applications and found that the changes are usually small and localized and, hence, easy to implement.

4.1 The Mechanics

A library, *libfbufs*, must be linked by applications that use the framework. The fbufs buffer pool for such an application is created as part of process initialization. The I/O subsystem automatically creates associations between the buffer pool and devices accessed by the application that support the new framework. Applications that read data and process it or discard it can use uf_read. Applications that read, optionally filter the data, and write the (filtered) data can use uf_read or uf_mmap followed by uf_write. Applications that generate data and then write or discard it may use uf_allocate, followed by uf_write or uf_deallocate.

Note that an application can use the new API even when the accessed device does not support the fbufs framework. The use of fbufs is transparent to the device when an application uses uf_write. In the uf_read case, each I/O operation will result in an allocation of an fbuf, and a data-copy from kernel buffers to the fbuf performed by the fbufs framework. The resulting cost of the uf_read in this case is comparable to the traditional read, while a significant performance benefit continues to be realized on the uf_write.

4.2 An example

To illustrate the use of uf_read, uf_write and uf_mmap, we converted a representative application, the file transfer program **ftp**, to use the fbufs scheme. The modifications are gratifyingly trivial: A mere 20 lines of the approximately 10,000 lines of C code are affected. The code fragment in Figure 3 shows a read from a network endpoint, followed by a write to a disk file that is changed to a uf_read from the network endpoint and a write to the disk file.

The code fragment in Figure 3 shows the inner loop in ftp, where a read from a disk file followed by a write to a network endpoint, is converted to a uf_mmap of the disk file followed by uf_write.The technique of using mmap rather than read is one that can be applied with the standard UNIX interfaces. Accessing a file using a memory mapping (mmap or uf_mmap) saves one data copy. However, a data copy is still necessary when the mapped data is output using write. In our framework, by combining uf_mmap with uf_write, *all* data copying is eliminated.

```
errno = d = 0;
while ((c = read(fileno(din), buf, sizeof (buf))) > 0) {          Original Code
        if ((d = write(fileno(fout), buf, c)) < 0)
                break;
        bytes += c;
}
```

```
errno = d = 0;                                                       Modified Code
while ((c = uf_read(fileno(din), (void **) &fbuf, FBSIZE)) > 0) {
        d = write(fileno(fout), fbuf, c);
        uf_deallocate (fbuf, c);
        if (d < 0) {
                break;
        }
        bytes += c;
}
```

FIGURE 3. Using uf_read

The original and modified code fragments from the recvrequest routine of the ftp program. A write call is used instead of a uf_write due to a current implementation limitation described in Section 5.5.

```
errno = d = 0;                                                   Original Code
while ((c = read(fileno (fin), buf, sizeof (buf))) > 0) {
        if ((d = write(fileno (dout), buf, c)) < 0)
                break;
        bytes += d;
}
```

```
addr = uf_mmap (NULL, file_size, PROT_READ,                     Modified Code
                MAP_SHARED, fileno (fin), 0);
errno = d = 0;
while (bytes < file_size) {
        c = (FBSIZE < file_size-bytes) ?
                FBSIZE:file_size-bytes;
        if ((d = uf_write(fileno (dout), fbuf, c)) < 0)
                break;
        bytes += d;
        fbuf += c;
}
```

FIGURE 4. Using uf_mmap

The original and modified code fragments from the sendrequest routine of the ftp program.

4.3 General techniques

A widely-used model in client-server computing is to have a port monitor process on the server that accepts requests on multiple ports. Such a port monitor typically authenticates the service requester and then spawns the real service provider task to handle the service, passing to the service provider the file descriptor on which communication with the service requester is to occur. The service provider program then asserts ownership over the file descriptor (to correctly receive event notification and signals). This presents an opportunity for the I/O subsystem to associate the inherited file descriptor with the fbufs pool of the service provider task, and thus enjoy the performance benefits of the new framework for this style of programming.

Note that some applications, for example, the **rlogind** program, fork a child process that handles the tasks of input from the user and output to the network, while the parent retains responsibility for input from the network and output to the user. Such applications require two processes to access the same I/O device (the TCP stream or socket). While our framework is capable of supporting this mode (see Section 5.5), we believe that a better solution is to multi-thread such applications.

5 Implementation

The implementation is logically divided into three components: a library, a buffer pool manager, and a system call component, as shown in Figure 5.

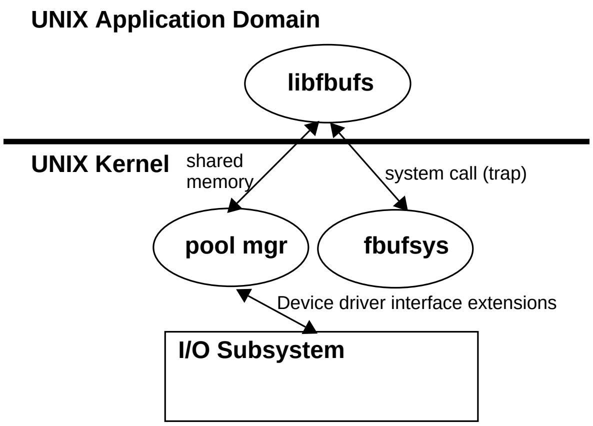

FIGURE 5. UNIX fbufs implementation

The buffer pool manager consists of an application library, libfbufs, and a kernel pool manager which coordinate allocation and deallocation of fbufs via a shared memory interface. The library invokes the system call component fbufsys, via a trap, to transfer fbufs between kernel and application. Device driver interface extensions allow the I/O subsystem to allocate fbufs in the kernel.

5.1 libfbufs

The libfbufs library, running in the application domain, provides the fbuf interfaces to UNIX applications. The application allocator/deallocator interfaces uf_allocate and uf_deallocate are implemented in the library and share data structures with the kernel pool manager (Section 5.2). The uf_read, uf_get, uf_write, uf_mmap and uf_unmap interfaces are provided as traps to a new system call with appropriate arguments (Section 5.3).

5.2 Buffer pool manager

Each application program that uses fbufs is associated with a different instance of the buffer pool manager. The buffer pool manager is responsible for allocation of memory, tracking the allocation/deallocation of individual fbufs, managing mappings between user and kernel addresses and conversion between fbufs and Streams mblks.

5.2.1 Allocation of memory

Each buffer pool is composed of a number of segments. A segment represents a contiguous range of user addresses that is backed by a filesystem object capable of supporting the standard mmap interface. The buffer pool manager is itself a special instance of such a filesystem object and is the supplier of *anonymous* memory [10]. The allocation of physical memory is deferred until the latest possible moment, until either the application program accesses an fbuf (causing a page fault), or the application transfers an fbuf to the kernel, or an I/O subsystem allocates an fbuf from the pool. The allocation is done in units of blocks, the size of a block being a tunable parameter.

5.2.2 Tracking allocation/deallocation of individual fbufs

Internally, fbufs are represented as 5-tuples of <context, user-address, device, offset, length>. The memory used to hold the contents of the buffer is allocated only when the contents are accessed, not necessarily when the buffer is allocated, nor when the buffer is passed between domains. The distinction provides greater power and flexibility to the fbufs framework, since the representation can be the foundation for a lazy evaluation of I/O requests (Section 7).

Fbufs can be allocated and deallocated by the kernel and the application program. To make this possible, the buffer manager has a kernel allocator/deallocator component and a user library allocator/deallocator component. In both domains, the allocator/deallocator components access and manipulate a list of free fbufs. The interesting problem is to make the free list consistent and the manipulations atomic with respect to the two domains.

We implement the list as a bit string that is mapped in both the kernel and the application address space. In the bit string, a 1 represents a free page and a 0 represents an allocated page. The string is accessed in units of words (using the native word size of the architecture), each of which contains page bits. A lock protects access to each word to manage concurrent access. The lock on each word is held for only a few instructions, enough to determine that sufficient free pages exist to satisfy an allocation request or to set the bits that represent the pages being freed. The kernel allocator iterates over the entire range of spin locks, and if a spin lock is not available on the first try, the next spin lock is attempted. The allocator retries the first spin lock only after a pass over the entire range of spin locks. Since normally only one spin lock is held by a process at a time, this procedure increases the probability that the allocation will succeed with limited busy waiting in the operating system code path. After a tunable number of iterations on the bitmap string, the kernel allocator backs off and returns a failure indication. By limiting the number of memory words examined for free bits to one, the duration that a lock is held is made extremely small and using spin locks becomes feasible.

The kernel deallocator does not have the freedom of choosing any bitmap string (as does the allocator) when performing its operation. If the kernel deallocator fails to acquire the appropriate spin lock within a tunable number of iterations it queues deallocations on an internal deferred free list and attempts the deallocation at a later point.

Note that contending from the kernel for resources that can be allocated by a process via spin locks requires special attention since the application program can be preempted while holding the lock (or worse, the application may be malicious or incompetent). Therefore, the kernel allocator and deallocator take the special precaution of backing off after a tunable number of tries to avoid spinning waiting for a blocked or malicious process.

5.2.3 Locking/unlocking fbuf memory

The UNIX Streams and other kernel modules assume that the buffers they access are wired in memory and that the kernel will not have to handle any page fault

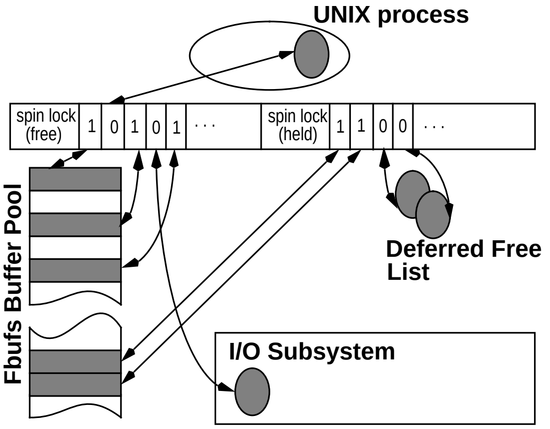

FIGURE 6. Tracking UNIX fbufs

Fbufs are tracked by means of a bit string mapped in both the kernel and the application address spaces. An fbuf that is marked available in the bit string will always be found in the fbufs buffer pool. An fbuf marked unavailable in the bit string is either held by the application or the I/O subsystem or on the deferred free list.

when accessing the buffers. To avoid the cost of locking the fbuf memory on each output operation (and unlocking on each input operation), our implementation locks a subset of the fbufs pool in memory such that, in the common case, no locking/unlocking of fbuf memory is needed. To guard against application allocating fbufs and not freeing them, the framework limits the total amount of memory that an application can dedicate to fbufs, and additionally, unlocks memory pages backing fbufs held by an application beyond a tunable threshold. A per-process FIFO unlocking policy is used.

5.2.4 Managing mappings between user and kernel addresses

In the general case, the user and kernel addresses of the buffer will be different. The buffer manager maintains the mappings and makes it possible to determine one from the other. Particular implementations may be able to dispense with this functionality as suggested in Section 5.5

5.2.5 Converting between fbufs and Streams mblks

The buffer manager also provides mechanisms to convert between fbufs and buffers of existing I/O frameworks such as the Streams mblks. This facility makes it easy to integrate fbufs with existing I/O drivers, network protocol stacks, and other Streams modules.

5.3 System calls

There is one new system call, the **fbufsys** system call which provides the functionality of the uf_read, uf_get, uf_write and uf_mmap interfaces. The uf_read, uf_get and uf_write interfaces interact with the buffer pool manager to convert between fbufs and Streams mblks. These interfaces are responsible for passing the resulting fbufs to the application and the resulting mblk to the underlying device/file. The uf_mmap interface creates a new segment in the buffer pool which is backed by the file being mapped.

5.4 Impact on the I/O subsystem

Device drivers (and other I/O subsystem components) that take advantage of fbufs must be modified slightly to deliver the higher performance afforded by the framework. The changes, which are minor, are oriented around allocation and management of fbufs.

5.4.1 Differences in the I/O environment

Traditionally, a device driver requests a buffer from the operating system when I/O is initiated by a process. It does not care where the buffer comes from and it does not distinguish between the processes that request the I/O. In the fbufs case, the device driver must request buffers from the specific buffer pool that corresponds to the process for which the I/O is initiated. This is reflected in the interface to the fbufs based mblk allocator (Section 5.4.2).

Note that for optimal use, the I/O subsystem must be able to determine the correct buffer pool to use either before initiating input, as in the case of disk or tape I/O (data arrives synchronously or on request from these devices), or by examining some part of the data received, as in the case of network or serial line I/O (data arrives asynchronously on these devices). Disk and tape device need no special hardware features in order to be used with an fbufs system; however, network devices must be able to examine some part of the data packet and determine the correct buffer pool to use while the packet is being received. Note that the scheme is still very useful even if this hardware support is not available, as discussed in Section 8.

Another change required by the fbufs framework is due to the fact that a buffer pool ceases to exist when the owning process exits. Traditional device drivers do not have to worry about the buffer pool being destroyed. On the other hand, when using fbufs, a device driver must be prepared to release buffers allocated from a particular buffer pool when requested by the framework. This requires a small amount of housekeeping in the device driver implementation. Note that the device driver can have no use for buffers that are associated with a process that has exited.

Many systems have separate address spaces for the operating system and for I/O, i.e., the addresses seen by I/O devices are typically different from the addresses seen by the device drivers. Device support routines are provided by the operating system for device drivers to translate between the two domains. In a traditional UNIX system, the addresses of buffers used in an I/O operation are fairly arbitrary. However, in an fbufs system, the same buffers are frequently reused for I/O to the same device. Device drivers can therefore be optimized to take advantage of this referential locality by caching translations between kernel and I/O addresses of fbufs and thus avoiding the expensive translation routines.

5.4.2 Device driver interface

The fbufs mechanism provides the interfaces shown in Table 2 for the I/O subsystem to allocate and free fbufs.

Interface	Description
void *uf_register (void *procref, void(* proc_exit_callback) (void *))	Interface for I/O subsystem components to initiate use of fbufs.
void uf_deregister (void *handle, void (* proc_exit_callback) (void *))	Interface for I/O subsystem components to terminate use of fbufs.
mblk_t *uf_allocb (void *handle, int size)	Interface for Streams drivers to allocate fbufs based mblks.

Table 2: UNIX fbufs Device Driver Interface

An I/O subsystem component (such as a device driver) that uses an fbufs pool must register itself with the fbufs framework. Registering with the framework returns to the driver a handle on the pool used by the process. That handle is in turn used when allocating new fbufs. The device driver calls the uf_deregister when it is done using the pool. Note that a call-back routine is established as part of registering with the framework. The call-back routine is supplied by the device driver and is called by the buffer manager when the process using this buffer pool exits. The call-back routine in uf_deregister is used as a reverse handle to identify the driver to the framework.

Fbufs-based mblks are allocated using uf_allocb, and they may be deallocated using the standard Streams routine **freeb** [1]. When deallocated, the framework tries to keep the fbuf memory locked and associated with the same pool it was allocated from.

5.5 Implementation notes and restrictions

The shared nature of the buffers implies that the framework does not prevent an application from interfering with its own input/output. This might happen, for example, if the application modifies an output buffer after transferring it to the kernel (as in initiating output). The results are undefined if an application attempts to modify a buffer after it hands it to the kernel (basically, it will appear as if the application had provided incorrect data in the first place.) In no circumstance will the kernel be affected if the application clobbers its own data.

The fbufs pool is partitioned by process id to ensure that sharing occurs only between individual applications and the kernel, not between any arbitrary applications. Thus, the buffer pool is not inherited across a fork system call. This scheme can be extended relatively easily, for example by associating fbufs pools with process group ids or with keys, to allow sets of processes to share fbufs. Such an implementation would allow cooperating processes such as the rlogind program (described in Section 4.3) to be supported without requiring their restructuring into multi-threaded applications. Our current implementation does not support this functionality because such support requires modifying a kernel data structure (the proc structure) which would require the entire operating system to be recompiled. We wanted to avoid modifying the kernel proper in this prototyping effort.

The implementation currently restricts the uf_read and uf_write calls to work with stream devices only. Relaxing the restriction to allow their use with other types of file objects is relatively straightforward. However, achieving zero-copy performance with some file system implementations (e.g., UFS) may not be easy. Therefore, the implementation of uf_read and uf_write should be extended to work with non-stream devices, but perhaps they can perform a copy when the underlaying device/file system does not lend itself to a zero-copy implementation.

As noted in Section 5.2, the implementation needs to be translated between user and kernel fbufs addresses. One way to eliminate the translation cost is to use the same virtual addresses for fbufs in the user program and the kernel. In a 32-bit address space, it is generally difficult to allocate a variable-sized buffer at the same virtual address in these two domains. However, it may be possible in a 64 bit address space to reserve a large range of addresses for fbufs use in the kernel, as well as in each application. This range then can be partitioned for use of applications in a non-overlapping way by the kernel while eliminating the need for any address translation.

6 Performance Evaluation

We measured the performance of our framework in two basic ways, a set of micro-benchmarks to measure the latency of I/O operations, and a set of macro-benchmarks to show how throughput and CPU utilization in real applications are affected by the improvement in latency.

The hardware configuration we used consisted of two uniprocessor 50MHz SPARCstation™ 10 workstations, connected back-to-back via ATM adapters capable of 140Mb/s (line rate). Enough memory was used in each machine to avoid any paging during the experiments.

Solaris 2.4 was the base operating system on each system. The ATM device driver was modified to the minimum necessary to use fbufs. Our modifications did not include caching of translations between kernel and I/O addresses as described in Section 5.4.1. We expect that such a change would further improve the performance of an fbufs system. Note that no changes were made to TCP/IP or any other module in the system.

In every benchmark, the fbufs costs were compared with the cost of the corresponding operation using the standard Streams framework.

6.1 Micro-benchmarks

The first metric is a set of micro-benchmarks that measures the cost of primitive operations. The results of the micro-benchmarks are presented in graphs of latency (or time taken for the operation) vs. size of data.

The first benchmark, **rw**, measures the latency of the input and output operations. It has four parts: a *discard* mode for fbufs and for Streams, and a *loopback* mode for fbufs and for Streams. In the discard mode for fbufs, an fbuf is allocated, transferred to the kernel via uf_write, converted to a Streams mblk and freed by the test driver. In the Streams case for the same mode, data in a static buffer is passed as a parameter to the write system call which allocates a Streams mblk, copies the data into the mblk; the mblk is then freed by the test driver. These two components provide a measure of the cost of the output operation using fbufs and Streams respectively.

The loopback mode for fbufs and for Streams is used to obtain a measure of the cost of an output operation followed by an input operation. In the fbufs case, an fbuf is allocated, transferred to the kernel via uf_write, converted to a Streams mblk, transferred back to the application via uf_read, and then freed by the application by invoking uf_deallocate. The Streams version of the same mode passes data in a static buffer to the write system call which allocates a Streams mblk and copies the data into the mblk. The mblk is then passed back to the application. When the application issues a read request, the data is copied from the mblk into the user buffer, and the mblk is freed. The cost of the output operation alone (obtained from the discard mode of the benchmark) can then be subtracted from the cost of the loopback operation to determine the cost of input using fbufs and Streams.

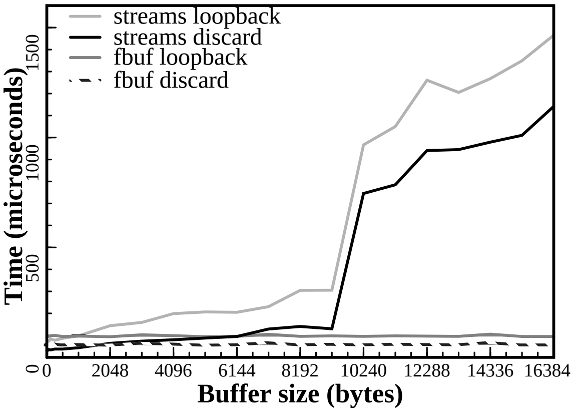

FIGURE 7. **rw** benchmark

The benchmark operates in two modes. In discard mode, the application does a write (or a uf_write) on a Streams driver that frees the resulting Streams message without touching the data. In loopback mode, the Streams driver passes the message back to the application which alternates write and read calls.

The results are shown in Figure 7. Note that the fbufs implementation has a fixed cost per operation. On the other hand, the standard read/write operations incur a per-byte cost in data-copying that increases the cost of each operation linearly with the size of the buffer used.

Note the large discontinuity in the latency of the Streams framework. The jump at around 9216 bytes is due to the mblk allocator falling back on the default kernel memory allocation scheme, rather than its own buffer caches. In contrast, the fbufs framework shows an almost invariant latency with increasing buffer size. At the 8192 byte mark, the fbufs framework has 34% of the latency of the Streams framework. The loopback case doubles the per-byte cost in the Streams case and doubles the per-buffer cost in the fbufs case, which at the 8192 byte mark results in an fbufs latency of just 31% of the comparable Streams framework number. Clearly, the results are even more encouraging with larger buffers. Note that even if the Streams buffer allocator is extended to operate efficiently beyond 9216 bytes, one cannot escape the per-byte copying costs.

The second micro-benchmark, **mapwr**, measures the combination of the mapping of a filesystem object into an application and using the output operation. In the fbufs case, it involves opening a file, mapping the file using uf_mmap to create an fbuf and transferring it to the kernel, converting it to a Streams mblk, and deallocating it. In the Streams mode, the benchmark opens a disk file, maps the file using the standard mmap call and writes the buffer to a Stream. The write allocates a Streams mblk, copies the file data into the mblk and passes the mblk down the stream where it is eventually deallocated. The results of running this benchmark are shown in Figure 8.

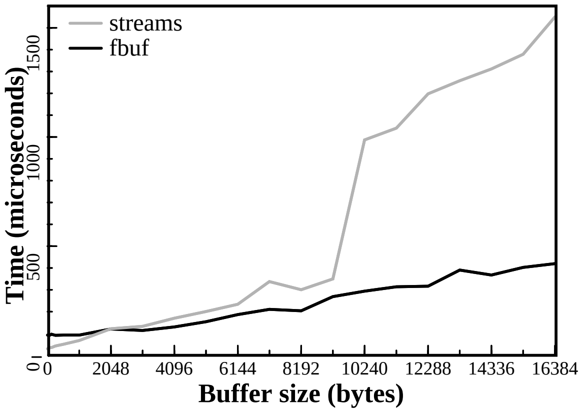

FIGURE 8. maprw benchmark

The benchmark does an mmap (uf_mmap) of a large (~1MByte) file and uses write (uf_write) to output the entire mapped data to a Streams driver. The resulting Streams messages are freed by the Streams driver without touching the data. The file is pre-loaded into the operating system buffer cache to avoid disk I/O.

There are some interesting points to be made about the graph in Figure 8. First, the cost of each operation increases with the size of the buffer in both the Streams and the fbufs case. For the fbufs case, the per-byte cost is due to the file system and not to the fbufs framework. Real I/O from disk is avoided by preloading the disk file contents memory before running the benchmarks. Second, the large discontinuity in the latency of the Streams framework is similar to that seen in Figure 7, and is due to the same reason. Third, at the 8192 byte mark, which is within the optimal range of the Streams allocator, the latency in the fbufs implementation is 68% of the latency of the Streams implementation.

6.2 Macro-benchmarks

The second metric consists of two macro-benchmarks. In both cases, the benchmarks use the TCP/IP network protocols. Note that in order to obtain the full benefit of a zero-copy implementation, the protocol checksum must be computed by the network adapter hardware, and in our prototype we assumed the existence of such hardware in all experiments below.

Our first macro-benchmark, **ttcp**, is a de facto TCP/IP throughput benchmark. We modified the ttcp program by replacing the read and write calls by uf_read and uf_write calls, respectively, to measure TCP throughput in an fbufs based system. We used the file transfer program **ftp** as our second benchmark. The program was modified as described in Section 4.2 to use the new interfaces. In the benchmark, a large (~1MByte) file was transferred from one system to a sink device on a second system to obtain a measure of filesystem to network throughput. The file was preloaded into memory before running the benchmarks.

Both ttcp and ftp handled data in 8192 byte buffers. The available hardware was capable of delivering 140Mb/s (instead of the standard 155Mb/s). Accounting for data link, network and transport layer encapsulation overhead (AAL5, IP and TCP respectively), yields an effective theoretical maximum hardware throughput of 125Mb/s.

The results of the fbufs-based benchmarks were compared with the throughput of Solaris 2.4 TCP/IP in Figure 9. Solaris 2.4 uses a Streams-based implementation that uses a single copy plus checksum scheme. As can be seen in the figure, ttcp shows a 35% improvement over the single copy plus checksum implementation. For ftp, the percentage gain is a hefty 42%.

CPU utilization was measured during the benchmark runs in order to better gauge the effects of the new framework. CPU utilization was determined by a utility program that periodically (in this case, every 5 seconds) issues a vmstat system call to obtain the values of various activity counters while the ttcp benchmark is running (Table 3).

Node	Solaris 2.4 CPU Utilization	fbufs CPU Utilization
receiver	99+%	89%
sender	85%	62%

Table 3: CPU Utilization for ttcp

As can be seen from the table, CPU utilization is reduced at both sender and receiver ends when using fbufs. We note that the maximum throughput achieved is less than the calculated maximum, even though the CPU utilization is less than 100%. Examining the flow of packets over the network shows that suboptimal TCP protocol window and acknowledgment handling accounts for the idle time on the wire, even though the CPU is not fully utilized in the fbufs case.

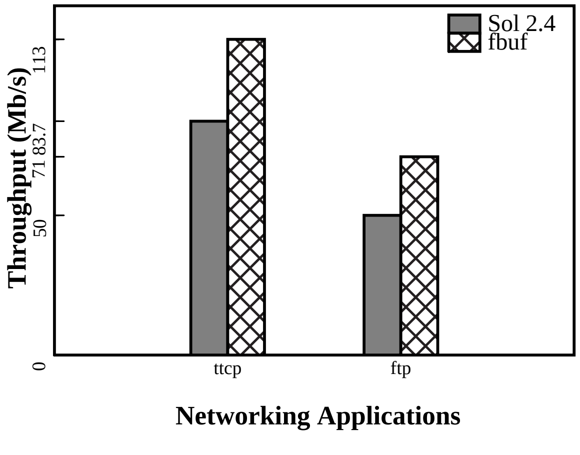

FIGURE 9. Throughput benchmarks

The benchmark ttcp gives a measure of TCP throughput. The file transfer program ftp transferred a large (~1MByte) file from one system to /dev/null on a second to obtain a measure of disk to network throughput. The file is preloaded into the memory before running the benchmark. Fbufs have 35% and 42% higher throughput than the 1-copy Solaris 2.4 implementation for ttcp and ftp, respectively.

6.3 Performance projections

We demonstrate the relevance of our work to future, faster systems by projecting the throughput of the ttcp-ATM benchmark on such systems. The simple model described in Appendix A extrapolates processing costs on systems built with faster CPUs coupled to faster network adapters to demonstrate the effectiveness of our framework.

In the model, we calculate the network latency and the average computational cost of an 8KByte data packet for end-to-end transfer when running the ttcp benchmark on a future system. The model provides a method to calculate the projected end-to-end latency. We can then compute the projected throughput (PT) of the ttcp benchmark for an fbufs-based system and a single-copy system.

We assume in our model that future systems will provide speed-up by a factor of M for data copying operations and by a factor of C for all other operations. We believe that it is reasonable to assume that network speeds will scale at least as fast as CPU speeds. Hence, we limit our projections to values of (C <= Network speed-up).

The throughput advantage of the fbufs scheme is then estimated as

$$\text{Advantage} = (PT_{fbufs} - PT_{1\text{-}copy}) / PT_{1\text{-}copy}$$

or substituting values obtained from Appendix A,

$$\text{Advantage} = (459 + 280{*}C/M) / 492 - 1$$

Consider a future system built around an OC12 ATM network giving a line rate of 622Mb/s and a network speed-up factor of ~4. For the simple case of a speed-up factor of 4,

$$C = M = 4$$
$$\text{Advantage} = (459 + 280) / 492 - 1 = 50\%$$

For another view, let us assume a highly aggressive speed-up factor of 10 for data-copying while using a factor of 4 for other operations,

C = 4
M =10
Advantage = (459 + 280 * 4 / 10) / 492 – 1 = 16%

In fact, it is only with a speed-up factor of 34 for data copying that the advantage of using the fbufs scheme reduces to 0!

7 Conclusions and Future Work

We have presented an efficient I/O mechanism based on a buffer exchange mechanism that has the advantage of being free of MMU manipulations when compared with page re-mapping schemes. A valuable extension to the basic idea is that of creating fast buffers from filesystem files.

A new application programming interface has been defined to take advantage of the mechanism. We have shown that programming the new API is simple, and converting existing applications to use the new interface is straightforward.

New utility routines for device drivers and operating system components have been designed. In order to realize the full potential of the fbufs framework, we require a device adapter capable of de-multiplexing incoming data into a pre-assigned user buffer, and a modified device driver that interacts with the fbufs allocator. However, even with devices that do not support this capability, and with unmodified device drivers, our framework delivers a performance benefit on the output operation. This is possible because the framework converts fbufs to Streams mblks during the output operation making the fbufs usage transparent to device drivers.

The framework has been prototyped in Solaris, a commercial implementation of the UNIX operating system. A set of micro benchmarks and two macro benchmarks/networking applications have been run to demonstrate that significant performance gains are achievable.

We have projected our experimental results to faster systems (CPUs and network media) to show that the fbufs- based framework will have increasing significance. The fbufs-based scheme continues to perform better than a copy-based scheme, even when extremely aggressive improvements in hardware capabilities for data copying operations are assumed. As the gap between the speeds of the CPU and memory systems widens, there is more of a need to avoid copying data altogether. Note also that as demonstrated by our results, an fbufs solution is insensitive to the size of buffers, in contrast to a data-copying approach in which costs increase linearly with the size buffer used. Applications and I/O subsystems that can handle larger data units will benefit increasingly from using fbufs.

We did not directly compare the performance of our implementation to zero-copy implementations that use virtual memory remapping (Section 2.2) because we could not find such an implementation that ran within the operating system we used. However, it is clear that such implementations require MMU/TLB manipulations on *each* I/O operation. In contrast, our scheme does not touch the MMU system in the common case. As CPUs become faster, the relative cost of MMU manipulations will increase [3], which will make implementations that touch the MMU on every operation even less attractive.

Future extensions of this work include using fbufs for device-to-device data-copying. The semantics of fbufs API allows implementations to delay filling a buffer on input operations. If such an fbuf is then output to a device that can Direct MemoryAccess (DMA) directly from the input device, a direct device-to-device data transfer can be performed without the data ever being copied to main memory. Applications such as video servers, which typically read from a high bandwidth data source and write to a sink without actually accessing the data, would be prime candidates to take advantage of this lazy evaluation of I/O requests.

Finally, we plan to investigate ways to provide the benefits of fbufs without modifying existing applications. One possibility is to use the fbufs mechanism in the implementation of the standard UNIX I/O *stdio* library. Only the implementation of library need be modified to use the fbufs API. Applications linked to the library will be unaffected, yet will notice improved performance.

8 Acknowledgments

We thank Jose Bernabeu-Auban, Bruce Curtis, and Vlada Matena for their help in preparing this paper.

9 References

[1] AT&T. *UNIX System V Streams Programmer's Guide* (1987).

[2] Abott, Mark B. and Larry L. Peterson. "Increasing Network Throughput by Integrating Protocol Layers." *IEEE/ACM Transactions on Networking* (1993).

[3] Anderson Thomas, Henry Levy, Brian Bershad, and Edward Lazowska, "The Interaction of Architecture and Operating System Design," *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)* (1991), 108-120.

[4] Cheriton, D.R. "The V Distributed System." *Communications of the ACM*, vol. 31, no. 3 (March 1988).

[5] Clark, David D. and D.L. Tennenhouse. "Architectural Considerations for a New Generation of Protocols." *Proceedings of the SIGCOMM Symposium on Communication Architectures and Protocols* (September 1990).

[6] Dalton, Chris, Greg Watson, David Banks, Costas Calamvokis, Aled Edwards, and John Lumley. "Afterburner—A network-independent card provides architectural support for high-performance protocols." *IEEE Network* (July 1993).

[7] Druschel, Peter, Mark B. Abbott, Michael A. Pagels, and Larry L. Peterson. "Network Subsystem Design." *IEEE Network* (July 1993).

[8] Druschel, Peter and Larry L. Peterson. "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility." *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (December 1993).

[9] Fitzgerald, R. and R. F. Rashid. "The Integration of Virtual Memory Management and Interprocess Communication in Accent." *ACM Trans on Comp. Sys.*, 4,2 (May 1986).

[10] Gingell, R. A., J. P. Moran and W. A. Shannon, "Virtual Memory Architecture in SunOS," *Proceedings of the USENIX Conference* (May 1987).

[11] Jacobson, Van. "Efficient Protocol Implementation." *ACM SIGCOMM '90 tutorial* (September 1990).

[12] Kay, Jonathan and Joseph Pasquale. "The Importance of Non-Data Touching Processing Overheads in TCP/IP." *Proceedings of the SIGCOMM'93 Conference on Communications Architectures, Protocols and Applications* (October 1993), 259-269.

[13] Leffler, S, M. McKusick, M. Karels and J. Quarterman. *The Design and Implementation of the 4.3BSD Unix Operating System* (1989).

[14] Pasquale, Joseph, Eric Anderson and P. Keith Muller. "Container Shipping—Operating System Support for I/O-Intensive Applications." *Computer* (March 1994).

[15] Postel Jon, "File Transfer Protocol", *Request for Comments—Network Information Center, SRI* (1985).

[16] Postel Jon. "Transmission Control Protocol." *Request for Comments—793 Network Information Center, SRI* (1981).

[17] Tzou, S.-Y. and D. P. Anderson. "The Performance of Message Passing Using Restricted Virtual Memory Remapping." *Software—Practice and Experience 21* (March 1991).

[18] Solaris 2.4 1-copy TCP reference.

10 Appendix A - Performance Projection Model

The average end-to-end latency (EL) of an 8KByte data packet over a TCP connection is computed thus

$$EL = 8KBytes / Throughput$$

This formula factors in the encapsulation overheads of the protocols, as well as the cost of acknowledgments sent on the network. For our test system, we compute from our test results

$$EL_{fbufs} = 8KBytes / 113Mb/s = 553\mu s$$
$$EL_{1\text{-}copy} = 8KBytes / 83.7Mb/s = 746\mu s$$

Network latency (NL) on the other hand is computed by using the formula

$$NL = (8KByte + Encapsulation\ Overhead) / Throughput$$
$$NL_{current} = 9116\ Bytes / 140\ Mb/s = 497\mu s$$

The average computational cost (CC) of 8KBytes of data on our test system is computed using the formula

$$CC = 8KByte / Throughput * CPU\text{-}Utilization$$

Our experiments show that the receiver has a higher utilization than the sender, and we use the higher figure (since the throughput will always be limited by the slower side). From our test results we have

$$CC_{fbufs} = 8KBytes / 113\ Mb/s * 89\% = 492\mu s$$
$$CC_{1\text{-}copy} = 8KBytes / 83.7\ Mb/s * 99\% = 739\mu s$$

In the case of the fbufs-based system, a negligible fraction of the total cost is due to data-copying, while in a single copy system, ~280µs are attributable to data-copying (from tracing a kernel running the benchmark).

We observe that when $CC > NL$, $EL \cong CC$, i.e., when the protocol processing cost on the CPU is greater than the time taken to send the data over the network medium, the end-to-end latency is approximately the same as the protocol processing cost. This is intuitively correct and analogous to the throughput of a pipelined processor being limited to that of its slowest component. We will use this observation to derive EL for our future systems based on a computed value of CC.

Network latency for OC12 ATM

$$NL_{OC12} = 9116\ Bytes / 622\ Mb/s = 112\mu s$$

Projected computation costs (PC)

$$PC_{fbufs} = 492\mu s / C$$
$$PC_{1\text{-}copy} = (739\text{-}280)\mu s / C + 280 / M$$
or $$PC_{1\text{-}copy} = 459\mu s / C + 280 / M$$

Note that for values of ($PC < NL_{OC12}$) throughput of both frameworks will converge to the throughput of the underlying network (622Mb/s in this case). In that case, the performance advantage will be in terms of lower CPU utilization. However, protocol windowing, interrupt processing costs and acknowledgment latency become more important when ($PC < NL_{OC12}$) making our simple model less reliable.

Since we are using simple extrapolation, we know that for values of ($C <= 622/140$) or ($C <= 4$), i.e., for CPU speed-up less than or equal to the network speed-up, the relation ($PC > NL_{OC12}$) holds true. This allows us to use the simple estimation for end-to-end latency (PEL).

$$PEL = PC$$
$$PEL_{fbufs} = 492\mu s / C$$
$$PEL_{1\text{-}copy} = 459\mu s / C + 280 / M$$

Throughput is then projected by:

$$PT = 8KByte / PEL$$
$$PT_{fbufs} = 8KBytes / (492\mu s / C)$$
$$PT_{1\text{-}copy} = 8KBytes / (459\mu s / C + 280 / M)$$

About the authors

Moti N. Thadani is a Staff Engineer at Sun Microsystems Laboratories. His interests include computer networking and distributed and object systems. He received a BTech in Electrical Engineering from the Indian Institute of Technology, New Delhi and an M.S. in Computer Science from Tulane University.

Yousef A. Khalidi is currently a Senior Staff Engineer and Principal Investigator at Sun Microsystems Laboratories. His interests include operating systems, distributed object-oriented software, computer architecture, and high-speed networking. He is one of the principal designers of the Spring operating system, and a co-winner of Sun’s Presidential Award in 1993. He has a Ph.D. in Information and Computer Science from Georgia Institute of Technology, where he was one of the principal designers of the Ra and Clouds operating systems.

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@eng.sun.com>.
For distribution issues, contact Amy Tashbook, Assistant Editor <amy.tashbook@eng.sun.com>.