Fbufs: A High-Bandwidth Cross-Domain Transfer Facility

Peter Druschel and Larry L. Peterson[*]
Department of Computer Science
University of Arizona
Tucson, AZ 85721

Abstract

We have designed and implemented a new operating system facility for I/O buffer management and data transfer across protection domain boundaries on shared memory machines. This facility, called *fast buffers* (fbufs), combines virtual page remapping with shared virtual memory, and exploits locality in I/O traffic to achieve high throughput without compromising protection, security, or modularity. Its goal is to help deliver the high bandwidth afforded by emerging high-speed networks to user-level processes, both in monolithic and microkernel-based operating systems.

This paper outlines the requirements for a cross-domain transfer facility, describes the design of the fbuf mechanism that meets these requirements, and experimentally quantifies the impact of fbufs on network performance.

1 Introduction

Optimizing operations that cross protection domain boundaries has received a great deal of attention recently [2, 3]. This is because an efficient cross-domain invocation facility enables a more modular operating system design. For the most part, this earlier work focuses on lowering *control transfer* latency—it assumes that the arguments transferred by the cross-domain call are small enough to be copied from one domain to another. This paper considers the complementary issue of increasing *data transfer* throughput—we are interested in I/O intensive applications that require significant amounts of data to be moved across protection boundaries. Such applications include real-time video, digital image retrieval, and accessing large scientific data sets.

Focusing more specifically on network I/O, we observe that on the one hand emerging network technology will soon offer sustained data rates approaching one gigabit per second to the end host, while on the other hand, the trend towards microkernel-based operating systems leads to a situation where the I/O data path may intersect multiple protection domains. The challenge is to turn good network bandwidth into good application-to-application bandwidth, without compromising the OS structure. Since in a microkernel-based system one might find device drivers, network protocols, and application software all residing in different protection domains, an important problem is moving data across domain boundaries as efficiently as possible. This task is made difficult by the limitations of the memory architecture, most notably the CPU/memory bandwidth. As network bandwidth approaches memory bandwidth, copying data from one domain to another simply cannot keep up with improved network performance [15, 7].

This paper introduces a high-bandwidth cross-domain transfer and buffer management facility, called *fast buffers* (fbufs), and shows how it can be optimized to support data that originates and/or terminates at an I/O device, potentially traversing multiple protection domains. Fbufs combine 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).

[*]This work supported in part by National Science Foundation Grant CCR-9102040, and ARPA Contract DABT63-91-C-0030.

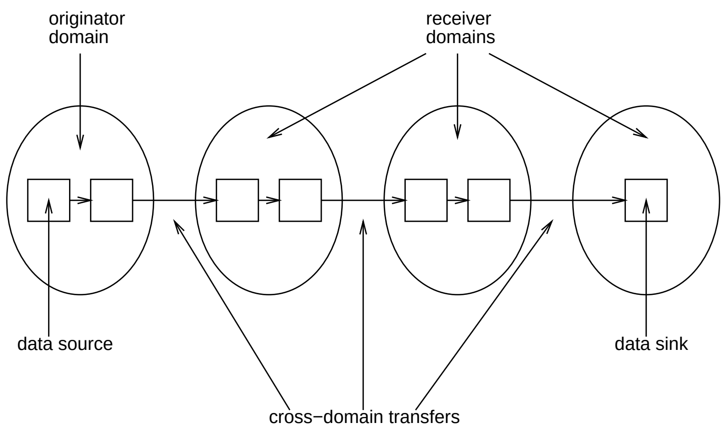

Figure 1: Layers Distributed over Multiple Protection Domains

2 Background

This section outlines the requirements for a buffer management and cross-domain data transfer facility by examining the relevant characteristics of network I/O. It also reviews previous work in light of these requirements. The discussion appeals to the reader's intuitive notion of a data buffer; Section 3 defines a specific representation.

2.1 Characterizing Network I/O

We are interested in the situation where I/O data is processed by a sequence of software layers—device drivers, network protocols, and application programs—that may be distributed across multiple protection domains. Figure 1 depicts this abstractly: data is generated by a source module, passed through one or more software layers, and consumed by a sink module. As the data is passed from one module to another, it traverses a sequence of protection domains. The data source is said to run in the *originator* domain, and the other modules run in *receiver* domains.

Note that although this section considers the general case of multiple protection domains, the discussion applies equally well to systems in which only two domains are involved: kernel and user. Section 4 shows how different transfer mechanisms perform in the two domain case, and Section 5.1 discusses the larger issue of how many domains one might expect in practice.

2.1.1 Networks and Buffers

On the input side, the network adapter delivers data to the host at the granularity of a *protocol data unit* (PDU), where each arriving PDU is received into a buffer.[1] Higher level protocols may reassemble a collection of PDUs into a larger *application data unit* (ADU). Thus, an incoming ADU is typically stored as a sequence of non-contiguous, PDU-sized buffers.

On the output side, an ADU is often stored in a single contiguous buffer, and then fragmented into a set of smaller PDUs by lower level protocols. Fragmentation need not disturb the original buffer holding the ADU; each fragment can be represented by an offset/length into the original buffer.

PDU sizes are network dependent, while ADU sizes are applications dependent. Control overhead imposes a practical lower bound on both. For example, a 1 Gbps link with 4 KByte PDUs results in more than 30,500 PDUs per second. On the other hand, network latency concerns place an upper bound on PDU size, particularly when PDUs are sent over the network without further fragmentation. Similarly, ADU size is limited by application-specific latency requirements, and by physical memory limitations.

2.1.2 Allocating Buffers

At the time a buffer is allocated, we assume it is known that the buffer is to be used for I/O data. This is certainly the case for a device driver that allocates buffers to hold incoming packets, and it is a reasonable expectation to place on application programs. Note that it is not strictly necessary for the application to know that the buffer will eventually find its way to an I/O device, but only that it might transfer the buffer to another domain.

The situation depicted in Figure 1 is oversimplified in that it implies that there exists a single, linear path through the I/O subsystem. In general, data may traverse a number of different paths through the software layers, and as a con-

[1]PDU size may be larger than the network packet size, as is likely in an ATM network. PDUs are the appropriate unit to consider because they are what the end hosts sees.

sequence, visit different sequences of protection domains. We call such a path an *I/O data path*, and say that a buffer belongs to a particular I/O data path. We further assume that all data that originates from (terminates at) a particular communication endpoint (e.g., a socket or port) travels the same I/O data path. An application can therefore easily identify the I/O data path of a buffer at the time of allocation by referring to the communication endpoint it intends to use. In the case of incoming PDUs, the I/O data path to which the PDU (buffer) belongs can often be determined, either by the network adaptor (e.g., by interpreting an ATM cell's VCI and/or adaptation layer info in hardware) or by having the device driver inspect the headers of the arriving PDU prior to the transfer of the PDU into main memory.

Locality in network communication [14] implies that if there is traffic on a particular I/O data path, then more traffic can be expected on the same path in the near future. Consequently, it is likely that a buffer that was used for a particular I/O data path can be reused soon for that same data path.

2.1.3 Accessing Buffers

We now consider how buffers are accessed by the various software layers along the I/O data path. The layer that allocates a buffer initially writes to it. For example, a device driver allocates a buffer to hold an incoming PDU, while an application program fills a newly allocated buffer with data to be transmitted. Subsequent layers require only read access to the buffer. An intermediate layer that needs to modify the data in the buffer instead allocates and writes to a new buffer. Similarly, an intermediate layer that prepends or appends new data to a buffer—e.g., a protocol that attaches a header—instead allocates a new buffer and logically concatenates it to the original buffer using the same buffer aggregation mechanism that is used to join a set of PDUs into a reassembled ADU.

We therefore restrict I/O buffers to be *immutable*—they are created with an initial data content and may not be subsequently changed. The immutability of buffers implies that the originator domain needs write permission for a newly allocated buffer, but it does not need write access after transferring the buffer. Receiver domains need read access to buffers that are passed to them.

Buffers can be transferred from one layer to another with either *move* or *copy* semantics. Move semantics are sufficient when the passing layer has no future need for the buffer's data. Copy semantics are required when the passing layer needs to retain access to the buffer, for example, because it may need to retransmit it sometime in the future. Note that there are no performance advantages in providing move rather than copy semantics since buffers are immutable. This is because with immutable buffers, copy semantics can be achieved by simply *sharing* buffers.

Consider the case where a buffer is passed out of the originator domain. As described above, there is no reason for a correct and well behaved originator to write to the buffer after the transfer. However, protection and security needs generally require that the buffer/transfer facility *enforce* the buffer's immutability. This is done by reducing the originator's access permissions to read only. Suppose the system does not enforce immutability; such a buffer is said to be *volatile*. If the originator is a trusted domain—e.g., the kernel that allocated a buffer for an incoming PDU—then the buffer's immutability clearly need not be enforced. If the originator of the buffer is not trusted, then it is most likely an application that generated the data. A receiver of such a buffer could fail (crash) while interpreting the data if the buffer is modified by a malicious or faulty application. Note, however, that layers of the I/O subsystem generally do not interpret outgoing data. Thus, an application would merely interfere with its own output operation by modifying the buffer asynchronously. The result may be no different if the application had put incorrect data in the buffer to begin with.

There are thus two approaches. One is to enforce immutability of a buffer; i.e. the originator loses its write access to the buffer upon transferring it to another domain. The second is to simply assume that the buffer is volatile, in which case a receiver that wishes to interpret the data must first request that the system raise the protection on the buffer in the originator domain. This is a no-op if the originator is a trusted domain.

Finally, consider the issue of how long a particular domain might keep a reference to a buffer. Since a buffer can be passed to an untrusted application, and this domain may retain its reference for an arbitrarily long time, it is necessary that buffers be pageable. In other words, the cross-domain transfer facility must operate on pageable, rather than wired (pinned-down) buffers.

2.1.4 Summary of Requirements

In summary, by examining how buffers are used by the network subsystem, we are able to identify the following set of requirements on the buffer management/transfer system, or conversely, a set of restrictions that can reasonably be placed on the users of the transfer facility.

- The transfer facility should support both single, contiguous buffers, and non-contiguous aggregates of buffers.

- It is reasonable to require the use of a special buffer allocator for I/O data.

- At the time of allocation, the I/O data path that a buffer will traverse is often known. For such cases, the transfer facility can employ a data path-specific allocator.

- The I/O subsystem can be designed to use only immutable buffers. Consequently, providing only copy semantics is reasonable.

- The transfer facility can support two mechanisms to protect against asynchronous modification of a buffer by the originator domain: eagerly enforce immutability by raising the protection on a buffer when the originator transfers it, or lazily raise the protection upon request by a receiver.

- Buffers should be pageable.

Section 3 gives the design of a cross-domain transfer facility that supports (exploits) these requirements (restrictions).

2.2 Related Work

The unacceptable cost of copying data from one buffer to another is widely recognized. This subsection reviews literature that addresses this problem.

2.2.1 Page Remapping

Several operating systems provide various forms of virtual memory (VM) support for transferring data from one domain to another. For example, the V kernel and DASH [4, 19] support *page remapping*, while Accent and Mach support *copy-on-write* (COW) [8, 1]. Page remapping has move rather than copy semantics, which limits its utility to situations where the sender needs no further access to the transferred data. Copy-on-write has copy semantics, but it can only avoid physical copying when the data is not written by either the sender or the receiver after the transfer.

Both techniques require careful implementation to achieve good performance. The time it takes to switch to supervisor mode, acquire necessary locks to VM data structures, change VM mappings—perhaps at several levels—for each page, perform TLB/cache consistency actions, and return to user mode poses a limit to the achievable performance. We consider two of the more highly tuned implementations in more detail.

First, Tzou and Anderson evaluate the remap facility in the DASH operating system [19]. The paper reports an incremental overhead of 208μsecs/page on a Sun 3/50. However, because it measures a ping-pong test case—the same page is remapped back and forth between a pair of processes—it does not include the cost of allocating and deallocating pages. In practice, high-bandwidth data flows in one direction through an I/O data path, requiring the source to continually allocate new buffers and the sink to deallocate them. The authors also fail to consider the cost of clearing (e.g., filling with zeros) newly allocated pages, which may be required for security reasons.

So as to update the Tzou/Anderson results, and to quantify the impact of these limitations, we have implemented a similar remap facility on a modern machine (DecStation 5000/200). Our measurements show that it is possible to achieve an incremental overhead of 22μsecs/page in the ping-pong test, but that one would expect an incremental overhead of somewhere between 42 and 99μsecs/page when considering the costs of allocating, clearing, and deallocating buffers, depending on what percentage of each page needed to be cleared.

The improvement from 208μsecs/page (Sun 3/50) to 22μsecs/page (Dec 5000/200) might be taken as evidence that page remapping will continue to become faster at the same rate as processors become faster. We doubt that this extrapolation is correct. Of the 22μsecs required to remap another page, we found that the CPU was stalled waiting for cache fills approximately half of the time. The operation is likely to become more memory bound as the gap between CPU and memory speeds widens.

Second, the Peregrine RPC system [11] reduces RPC latency by remapping a single kernel page containing the request packet into the server's address space, to serve as the server thread's runtime stack. The authors report a cost of only 4μsecs for this operation on a Sun 3/60. We suspect the reason for this surprisingly low number is that Peregrine can remap a page merely by modifying the corresponding page table entry. This is because in the V system—upon which Peregrine is based—all VM state is encoded only in the Sun's physical page tables. Portability concerns have caused virtually all modern operating systems to employ a two-level virtual memory system. In these systems, mapping changes require the modification of both low-level, machine dependent page tables, and high-level, machine-independent data structures. Moreover, unlike the Sun 3/60, most modern architectures (including the DecStation) require the flushing of the corresponding TLB entries after a change of mappings. Both DASH and the fbuf mechanism described in the next section are implemented in a two-level VM system.

Peregrine overlaps the receipt of one PDU from the Ethernet with copying the previous PDU across the user/kernel boundary. However, this strategy does not scale to either high-speed networks, or microkernel-based systems. As network bandwidth approaches memory bandwidth, contention for main memory no longer allows concurrent reception and copying—possibly more than once—at network speeds.

2.2.2 Shared Memory

Another approach is to statically share virtual memory among two or more domains, and to use this memory to transfer data. For example, the DEC Firefly RPC facility uses a pool of buffers that is globally and permanently shared among all domains [16]. Since all domains have read and write access permissions to the entire pool, protection and security are compromised. Data is copied between the shared buffer pool and an application's private memory. As another example, LRPC [2] uses argument stacks that are pairwise shared between communicating protection domains. Arguments must generally be copied into and out of the argument stack.

Both techniques reduce the number of copies required, rather than eliminating copying. This is sufficient to improve the latency of RPC calls that carry relatively small amounts of data, and to preserve the relatively low bandwidth of Ethernet LANs. The fbuf mechanism has the different goal of preserving the bandwidth afforded by high-speed networks at the user level. Fbufs complement a low-latency RPC mechanism.

The bottom line is that using statically shared memory to eliminate *all* copying poses problems: globally shared memory compromises security, pairwise shared memory requires copying when data is either not immediately consumed or is forwarded to a third domain, and group-wise shared memory requires that the data path of a buffer is always known at the time of allocation. All forms of shared memory may compromise protection between the sharing domains.

Several recent systems attempt to avoid data copying by transferring data directly between UNIX application buffers and network interface [5, 17]. This approach works when data is accesses only in a single application domain. A substantial amount of memory may be required in the network adapter when interfacing to high-bandwidth, high-latency networks. Moreover, this memory is a limited resource dedicated to network buffering. With fbufs, on the other hand, network data is buffered in main memory; the network subsystem can share physical memory dynamically with other subsystems, applications and file caches.

3 Design

This section describes the design of an integrated buffer and data transfer mechanism. It begins by introducing a basic mechanism, and then evolves the design with a series of optimizations. Some of the optimizations can be applied independently, giving rise to a set of implementations with different restrictions and costs.

3.1 Basic Mechanism

I/O data is stored in buffers called *fbufs*, each of which consists of one or more contiguous virtual memory pages. A protection domain gains access to an fbuf either by explicitly allocating the fbuf, or implicitly by receiving the fbuf via IPC. In the former case, the domain is called the *originator* of the fbuf; in the latter case, the domain is a *receiver* of the fbuf.

An abstract data type can be layered on top of fbufs to support buffer aggregation. Such abstractions typically provide operations to logically join one or more buffers into an aggregate, split an aggregate into separate buffers, clip data from one end of an aggregate, and so on. Examples of such aggregation abstractions include x-kernel *messages* [10], and BSD Unix *mbufs* [12]. For the purpose of the following discussion, we refer to such an abstraction as an *aggregate object*, and we use the x-kernel's directed acyclic graph (DAG) representation depicted in Figure 2.

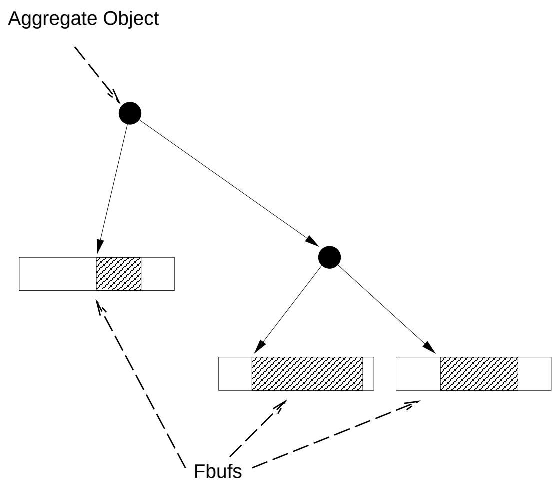

Figure 2: Aggregate Object

A virtual page remapping facility logically copies or moves a set of virtual memory pages between protection domains by modifying virtual memory mappings. We use a conventional remap facility with copy semantics as the baseline for our design. The use of such a facility to transfer an aggregate object involves the following steps.

1. Allocate an Aggregate Object (Originator)
   (a) Find and allocate a free virtual address range in the originator (per-fbuf)
   (b) Allocate physical memory pages and clear contents (per-page)
   (c) Update physical page tables (per-page)

2. Send Aggregate Object (Originator)
    (a) Generate a list of fbufs from the aggregate object (per-fbuf)
    (b) Raise protection in originator (read only or no access) (per-fbuf)
    (c) Update physical page tables, ensure TLB/cache consistency (per-page)
3. Receive Aggregate Object (Receiver)
    (a) Find and reserve a free virtual address range in the receiver (per-fbuf)
    (b) Update physical page tables (per-page)
    (c) Construct an aggregate object from the list of fbufs (per-fbuf)
4. Free an Aggregate Object (Originator, Receiver)
    (a) Deallocate virtual address range (per-fbuf)
    (b) Update physical page table, ensure TLB/cache consistency (per-page)
    (c) Free physical memory pages if there are no more references (per-page)

Note that a receiver that forwards an aggregate object to another domain would also perform the actions in step 2.

Even in a careful implementation, these actions can result in substantial overhead. For example, a simple data path with two domain crossings requires six physical page table updates for each page, three of which may require TLB/cache consistency actions. Moreover, each allocated physical page may need to be cleared—i.e., filled with zeroes—for security reasons.

3.2 Optimizations

The following set of optimizations are designed to eliminate the per-page and per-fbuf costs associated with the base remapping mechanism.

3.2.1 Restricted Dynamic Read Sharing

The first optimization places two functional restrictions on data transfer. First, only pages from a limited range of virtual addresses can be remapped. This address range, called the *fbuf region*, is globally shared among all domains. Note that sharing of an address range does not imply unrestricted access to the memory that is mapped into that range. Second, write accesses to an fbuf by either a receiver, or the originator while a receiver is holding a reference to the fbuf, are illegal and result in a memory access violation exception.

The first restriction implies that an fbuf is mapped at the same virtual address in the originator and all receivers. This eliminates the need for action (3a) during transfer. Note that the DASH remap facility uses a similar optimization. Shared mapping at the same virtual address also precludes virtual address aliasing, which simplifies and speeds up the management of virtually tagged caches, in machines that employ such caches. The second restriction eliminates the need for a copy-on-write mechanism. These restrictions require a special buffer allocator and immutable buffers.

3.2.2 Fbuf Caching

This optimization takes advantage of locality in interprocess communication. Specifically, we exploit the fact that once a PDU or ADU has followed a certain data path—i.e., visited a certain sequence of protection domains—more PDUs or ADUs can be expected to travel the same path soon.

Consider what happens when a PDU arrives from the network. An fbuf is allocated in the kernel, filled, and then transferred one or more times until the data is consumed by the destination domain. At this point, the fbuf is mapped with read only permission into the set of domains that participate in an I/O data path. Ordinarily, the fbuf would now be unmapped from these domains, and the physical pages returned to a free memory pool. Instead, write permissions are returned to the originator, and the fbuf is placed on a free list associated with the I/O data path. When another packet arrives for the same data path, the fbuf can be reused. In this case, no clearing of the buffers is required, and the appropriate mappings already exist.

Fbuf caching eliminates actions (1a-c), (2a-b), and (4a-c) in the common case where fbufs can be reused. It reduces the number of page table updates required to two, irrespective of the number of transfers. Moreover, it eliminates expensive clearing of pages, and increases locality of reference at the level of TLB, cache, and main memory. The optimization requires that the originator is able to determine the I/O data path at the time of fbuf allocation.

3.2.3 Integrated Buffer Management/Transfer

Recall that the aggregate object abstraction is layered on top of fbufs. However, the transfer facility described up to this point transfers fbufs, not aggregate objects across protection boundaries. That is, an aggregate object has to be translated into a list of fbufs in the sending domain (2a), this list is then passed to the kernel to effect a transfer, and the aggregate object is rebuilt on the receiving side (3c). Note that in this case, any internal data structures maintained by the aggregate object (e.g., interior DAG nodes) are stored in memory that is private to each domain. One consequence of this design is that the fbufs can be used to transfer any data across a domain boundary, and that a different representation for aggregated data can be used on either side of the boundary.

Consider now an optimization that incorporates knowledge about the aggregate object into the transfer facility, thereby eliminating steps (2a) and (3c). The optimization integrates buffer management and cross-domain data transfer facility by placing the entire aggregate object into fbufs. Since the fbuf region is mapped at the same virtual address in all domains, no internal pointer translations are required. During a send operation, a reference to the root node of the aggregate object is passed to the kernel. The kernel inspects the aggregate and transfers all fbufs in which reachable nodes reside, unless shared mappings already exist. The receiving domain receives a reference to the root node of the aggregate object. Steps (2a) and (3c) have therefore been eliminated.

3.2.4 Volatile fbufs

Under the previous optimizations, the transport of an fbuf from the originator to a receiver still requires two physical page table updates per page: one to remove write permission from the originator when the fbuf is transferred, and one to return write permissions to the originator after the fbuf was freed by all the receivers.

The need for removing write permissions from the originator can be eliminated in many cases by defining fbufs to be volatile by default. That is, a receiver must assume that the contents of a received fbuf may change asynchronously unless it explicitly requests that the fbuf be *secured*, that is, write permissions are removed from the originator. As argued in Section 2.1.3, removing write permissions is unnecessary in many cases.

When the volatile fbuf optimization is applied in conjunction with integrated buffer management, an additional problem arises. Since the aggregate object (e.g., a DAG) is stored in fbufs, and a receiving domain must traverse the DAG to access the data, the receiver may be vulnerable to asynchronous changes of the DAG. For example, a bad pointer could cause a protocol in the kernel domain to fail while traversing the DAG in order to compute a checksum.

The problem is solved in the following way. First, receivers verify that DAG pointers reference locations within the fbuf region (this involves a simple range check). Second, receivers check for cycles during DAG traversals to avoid infinite loops. Third, read accesses by a receiver to an address within the fbuf region for which the receiver has no permissions are handled as follows. The VM system maps a page at the appropriate location in the offending domain, initializes the page with a leaf node that contains no data, and allows the read to complete. Thus, invalid DAG references appear to the receiver as the absence of data.

3.2.5 Summary

The optimizations described above eliminate all per-page and per-fbuf costs associated with cross-domain data transfer in the common case—when the data path can be identified at fbuf allocation time, an appropriate fbuf is already cached, and when removing write permissions from the originator is unnecessary. Moreover, in the common case, no kernel involvement is required during cross-domain data transfer. Our facility is therefore well suited for use with user-level IPC facilities such as URPC [3], and other highly optimized IPC mechanisms such as MMS [9].

3.3 Implementation Issues

A two-level allocation scheme with per-domain allocators ensures that most fbuf allocations can be satisfied without kernel involvement. A range of virtual addresses, the fbuf region, is reserved in each protection domain, including the kernel. Upon request, the kernel hands out ownership of fixed sized chunks of the fbuf region to user-level protection domains. The fbuf region is pageable like ordinary virtual memory, with physical memory allocated lazily upon access. Fbuf allocation requests are fielded by *fbuf allocators* locally in each domain. These allocators satisfy their space needs by requesting chunks from the kernel as needed. Deallocated fbufs are placed on the appropriate allocator's free list, which is maintained in LIFO order.

Since fbufs are pageable, the amount of physical memory allocated to fbufs depends on the level of I/O traffic compared to other system activity. Similarly, the amount of physical memory allocated to a particular data path's fbufs is determined by its recent traffic. The LIFO ordering ensures that fbufs at the front of the free list are most likely to have physical memory mapped to them. When the kernel reclaims the physical memory of an fbuf that is on a free list, it discards the fbuf's contents; it does not have to page it out to secondary storage.

When a message is deallocated and the corresponding fbufs are owned by a different domain, the reference is put on a list of deallocated external references. When an RPC call from the owning domain occurs, the reply message is used to carry deallocation notices from this list. When too many freed references have accumulated, an explicit message must be sent notifying the owning domain of the deallocations. In practice, it is rarely necessary to send additional messages for the purpose of deallocation.

When a domain terminates, it may hold references to fbufs it has received. In the case of an abnormal termination, the domain may not properly relinquish those references. Note, however, that the domain is part of an I/O data path. Thus, its termination will cause the destruction of a communication endpoint, which will in turn cause the deallocation of all associated fbufs. A terminating domain may also be the originator of fbufs for which other domains hold references. The kernel will retain chunks of the fbuf region owned by the terminating domain until all external references are relinquished. An incorrect or malicious domain may fail to deallocate fbufs it receives. This would eventually cause the exhaustion of the fbuf region's virtual address range. To prevent this, the kernel limits the number of chunks that can be allocated to any data path-specific fbuf allocator.

4 Performance

This section reports on several experiments designed to evaluate the performance of fbufs. The software platform used in these experiments consists of CMU's Mach 3.0 microkernel (MK74) [1], augmented with a network subsystem based on the University of Arizona's *x*-kernel (Version 3.2) [10]. The hardware platform consists of a pair of DecStation 5000/200 workstations (25MHz MIPS R3000), each of which was attached to a prototype ATM network interface board, called Osiris, designed by Bellcore for the Aurora Gigabit testbed [6]. The Osiris boards were connected by a null modem, and support a link speed of 622Mbps.

The x-kernel based network subsystem consists of a *protocol graph* that can span multiple protection domains, including the Mach microkernel. Proxy objects are used in the x-kernel to forward cross-domain invocations using Mach IPC. The x-kernel supports a *message* abstract data type similar to the one shown in Figure 2. The message type is immutable—instances are created with an initial content that cannot be subsequently changed.[2] All protocols and device drivers deal with network data in terms of the message abstraction.

Incorporating fbufs into this environment required the following work: the Mach microkernel was modified to provide virtual memory support for fbufs, x-kernel messages were modified to use fbufs rather than malloc'ed buffers, proxy objects were upgraded to use the fbuf data transfer facility, and an Osiris device driver was written that uses fbuf based x-kernel messages. Several new approaches—described in a forthcoming paper—were required to reduce interrupt and other overhead in the driver.

[2]This is true as long as protocols access the message only through the operations exported by the message abstraction.

The first experiment quantifies the performance of fbuf transfers across a single protection boundary. A test protocol in the originator domain repeatedly allocates an x-kernel message, writes one word in each VM page of the associated fbuf, and passes the message to a dummy protocol in the receiver domain. The dummy protocol touches (reads) one word in each page of the received message, deallocates the message, and returns. Table 1 shows the incremental per-page costs—independent of IPC latency—and the calculated asymptotic bandwidths.

There are four things to notice about these numbers. First, the cached/volatile case performs an order of magnitude better than the uncached or non-volatile cases. It also performs an order of magnitude better than the Tzou/Anderson page remapping mechanism re-implemented on the same hardware. Second, the per-page overhead of cached/volatile fbufs is due to TLB misses caused by the accesses in the test and dummy protocols. TLB misses are handled in software in the MIPS architecture. Third, the cost for clearing pages in the uncached case is not included in the table. Filling a page with zeros takes 57μsecs on the DecStation. Fourth, the relatively high per-page overhead for the Mach COW facility is partly due to its lazy update strategy for physical page tables, which causes two page faults for each transfer.

The second experiment measures throughput as a function of message size. The results are shown in Figure 3. Unlike Table 1, the throughput rates shown for small messages in these graphs are strongly influenced by the control transfer latency of the IPC mechanism; it is not intrinsic to the buffer transfer facility. As before, Mach's native transfer facility has been included for comparison; it uses data copying for message sizes of less than 2KBytes, and COW otherwise.

For small messages—under 4KB—the performance break down is as follows. For message sizes under 2KB, Mach's native data transfer facility is slightly faster than uncached or non-volatile fbufs; this is due to the latency associated with invoking the virtual memory system, which we have not optimized in our current implementation. However, cached/volatile fbufs outperform Mach's transfer facility even for very small message sizes. Consequently, no special-casing is necessary to efficiently transfer small messages.

The third experiment demonstrates the impact of fbufs on network throughput by taking protocol processing overhead into account. It is also valuable because it is a macro experiment; i.e., it more accurately reflects the effects of the processor's instruction and data caches. A test protocol in the originator domain repeatedly creates an x-kernel message, and sends it using a UDP/IP protocol stack that

	incremental per-page cost (μsecs)	asymptotic throughput (Mbps)
fbufs, cached/volatile	3	10,922
fbufs, volatile	21	1,560
fbufs, cached	29	1,130
fbufs	37	886
Mach COW	144	228
Copy	316	104

Table 1: Incremental per-page costs

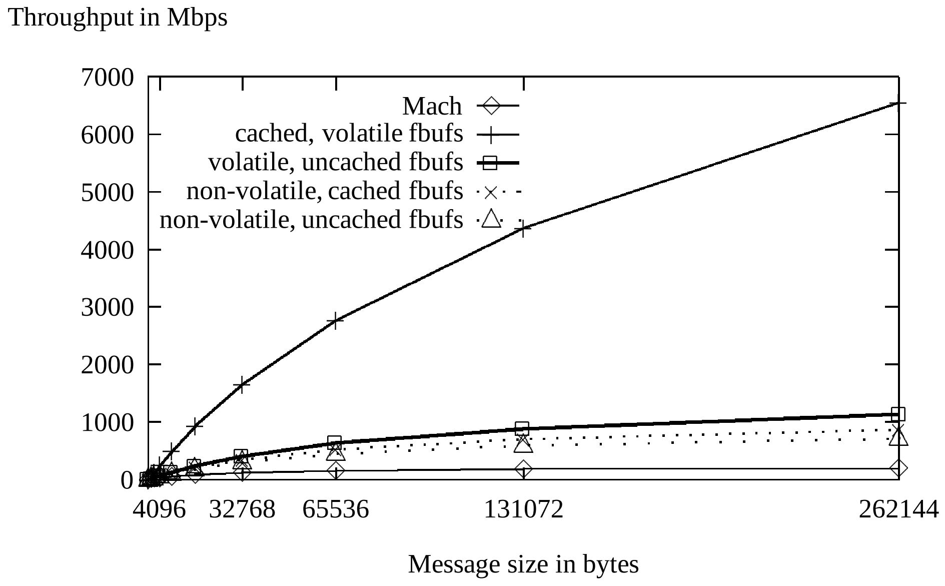

Figure 3: Throughput of a single domain boundary crossing

resides in a network server domain.[3] IP fragments large messages into PDUs of 4KBytes. A local loopback protocol is configured below IP; it turns PDU's around and sends them back up the protocol stack. Finally, IP reassembles the message on the way back up, and sends it to a receiver domain that contains the dummy protocol from the first experiment. The use of a loopback protocol rather than a real device driver simulates an infinitely fast network. Thus, the experiment ignores the effects of limited I/O bus bandwidth and network bandwidth in currently available commercial hardware.

For comparison purposes, we have performed the same experiment with all components configured into a single protection domain, rather than three domains. By comparing the results we can quantify the impact of domain boundaries on network throughput. Figure 4 shows the measured throughput in each case. The anomaly in the single domain graph is caused by a fixed fragmentation overhead that sets in for messages larger than 4KBytes. This cost is gradually amortized for messages much larger than 4KBytes. Moreover, this peak does not occur in the multiple domain cases due to the dominance of cross-domain latency for 4KByte transfers.

There are four things to observe about these results. First, the use of cached fbufs leads to a more than twofold improvement in throughput over uncached fbufs for the entire range of message sizes. This is significant since the performance of uncached fbufs is competitive with the fastest page remapping schemes. Second, we considered only a single domain crossing in either direction; this corresponds to the structure of a monolithic system. In a microkernel-based system, it is possible that additional domain crossings would occur. Third, for message sizes of 64KBytes and larger, the cached fbuf throughput is more than 90% of the throughput for a data path that involves no domain boundary crossings. This is of practical importance since large messages are common with high-bandwidth ap-

[3] UDP and IP have been slightly modified to support messages larger than 64KBytes.

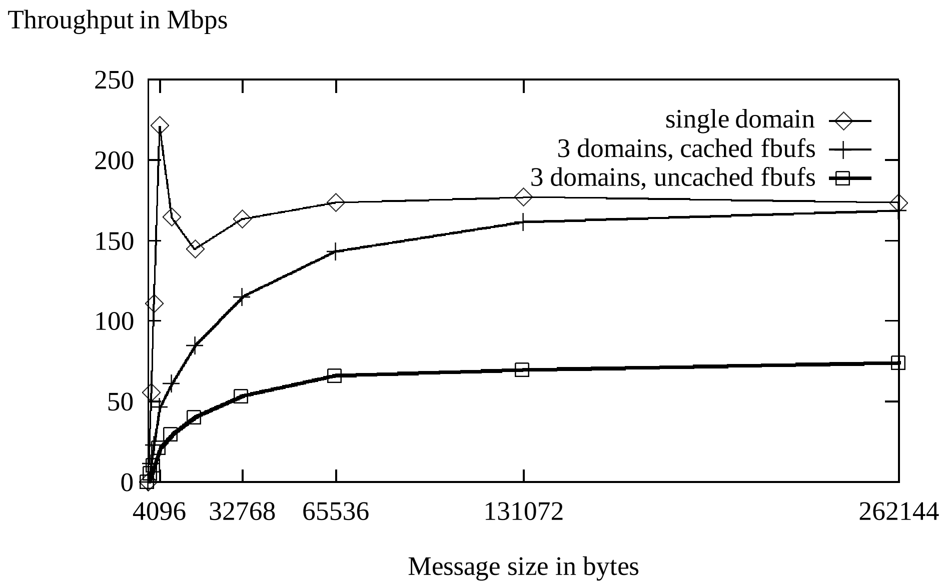

Figure 4: Throughput of a UDP/IP local loopback test

plications. Fourth, because the test is run in loopback mode, the throughput achieved is roughly half of what one would expect between two DecStations connected by an infinitely fast network.

A final experiment measures end-to-end throughput of UDP/IP between two DecStations using a null modem connection of the Osiris network interface boards. The protocol suite is identical to that used in the previous experiment, except that the local loopback protocol below IP is replaced with a driver for the Osiris board, and IP's PDU size was set to 16KBytes. The test protocol uses a sliding window to facilitate flow control.

Figure 5 shows the measured end-to-end throughput achieved with cached/volatile fbufs as a function of the message size. In the kernel-kernel case, the entire protocol stack, including the test protocol, is configured in the kernel. This case serves as a baseline for evaluating the impact of domain boundary crossings on throughput. The user-user case involves a kernel/user boundary crossing on each host. In the user-netserver-user case, UDP is configured in a separate user level server domain, necessitating both a user/user and a kernel/user boundary crossing as part of the data path on each host.

We make the following observations. First, the maximal throughput achieved is 285 Mbps, or 55% of the net bandwidth supported by the network link[4]. This limitation is due to the capacity of the DecStation's TurboChannel bus, not software overheads. The TurboChannel has a peak bandwidth of 800 Mbps, but DMA startup latencies reduce the effective throughput. The Osiris board currently initiates a DMA transfer for each ATM cell payload, limiting the maximal throughput to 367 Mbps. Bus contention due to CPU/memory traffic further reduces the attainable I/O throughput to 285 Mbps.

Second, domain crossings have virtually no effect on end-to-end throughput for large messages (> 256KB) when cached/volatile fbufs are used. For medium sized messages (8–64KB), Mach IPC latencies result in a significant throughput penalty per domain crossing. The throughput for small messages (< 8KB) is mainly limited by driver and protocol processing overheads.

For medium sized messages, the throughput penalty for a second domain crossing is much larger than the penalty for the first crossing. The difference is too large to be explained by the different latency of kernel/user and user/user crossings. We attribute this penalty to the exhaustion of cache and TLB when a third domain is added to the data path. Because our version of Mach/Unix does not support shared libraries, program text that implements the x-kernel infrastructure is duplicated in each domain. This duplication reduces instruction access locality and reduces hit rates in both TLB and instruction cache. The use of shared libraries should help mitigate this effect.

Figure 6 shows the measured end-to-end throughput when uncached/non-volatile fbufs are used.[5] As

[4]The net bandwidth of 516 Mbps is derived from the link bandwidth (622 Mbps) minus ATM cell overhead.

[5]The use of non-volatile fbufs has a cost only in the transmitting host; this is because the kernel is the originator of all fbufs in the receiving host. For similar reasons, uncached fbufs incur

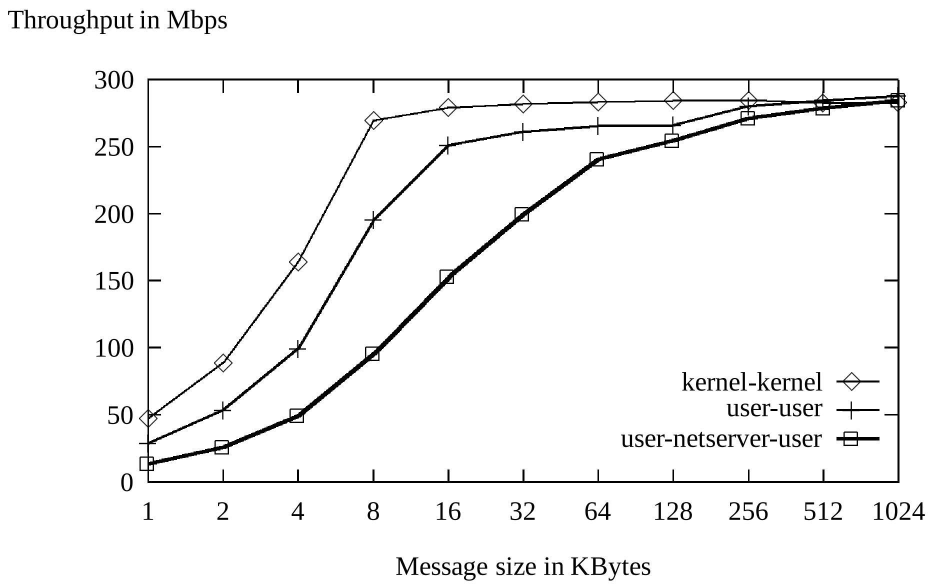

Figure 5: UDP/IP end-to-end throughput using cached, volatile fbufs

in the loop-back experiment, the significance of the uncached/non-volatile case is that it is comparable to the best one can achieve with page remapping. The kernel-kernel graph is once again included as a baseline for comparison. The maximal user-user throughput is 225 Mbps. Thus, the use of uncached fbufs leads to a throughput degradation of 21% when one boundary crossing occurs on each host. The throughput achieved in the user-netserver-user case is only marginally lower. The reason is that UDP—which resides in the netserver domain—does not access the message's body. Thus, there is no need to ever map the corresponding pages into the netserver domain. Consequently, the additional cost for using uncached fbufs in this case is small.

Note that the maximal throughput achieved with uncached fbufs is CPU bound, while throughput is I/O bound with cached fbufs. Thus, the throughput figure does not fully reflect the benefit of using cached fbufs. In our test, part of the benefit takes the form of a reduction of CPU load. Specifically, the CPU load on the receiving host during the reception of 1 MByte packets is 88% when cached fbufs are used, while the CPU is saturated when uncached fbufs are used[6].

One can shift this effect by setting IP's PDU size to 32 KBytes, which cuts protocol processing overheads roughly in half, thereby freeing CPU resources. In this case, the test becomes I/O bound even when uncached fbufs are used, i.e., the uncached throughput approaches the cached throughput for large messages. However, the CPU is still saturated during the reception of 1 MByte messages with uncached fbufs, while CPU load is only 55% when cached fbufs are used. Here, the use of cached fbufs leads entirely to a reduction of CPU load. On the other hand, a hypothetical system with much higher I/O bandwidth would make throughput CPU bound in both the cached and uncached fbuf cases. The local loopback test (which simulates infinite I/O bandwidth) has demonstrated that the use of cached fbufs leads to a twofold improvement in throughput over uncached fbufs in this case. Thus, on the DecStation, the use of cached fbufs can reduce CPU load up to 45% or increase throughput by up to a factor of two, when compared to uncached fbufs in the case where a single user-kernel domain crossing occurs.

5 Discussion

5.1 How Many Domains?

An important question not yet answered is how many domains a data path might intersect in practice. One the one hand, there is a trend towards microkernel-based systems, the motivation being that systems structured in this way are easier to configure, extend, debug, and distribute. In a system with a user-level networking server, there are at least two domain crossings; a third-party window or multimedia server would add additional domain crossings. On the

additional cost only in the receiving host. In our test, the cost for non-volatile fbufs is hidden by the larger cost of uncached fbufs.

[6]CPU load was derived from the rate of a counter that is updated by a low-priority background thread.

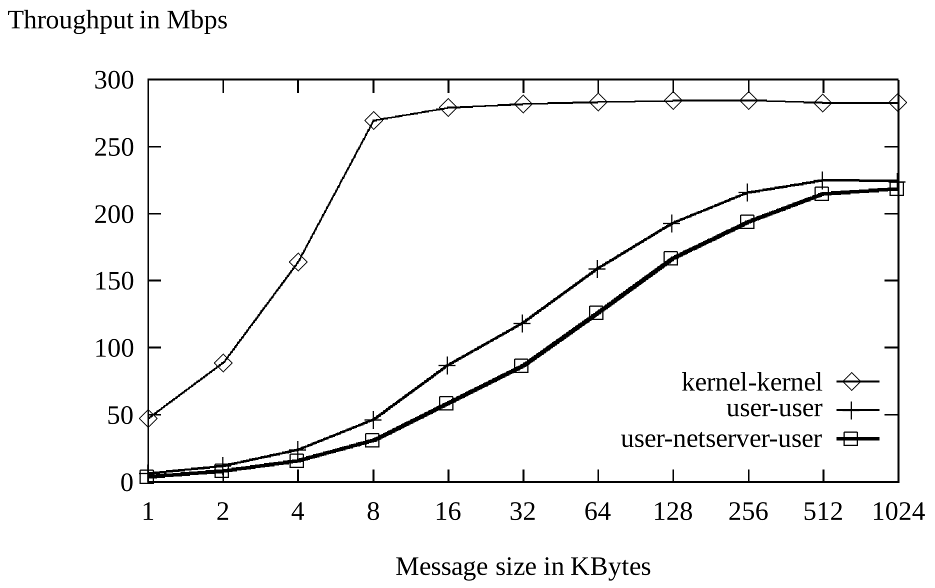

Figure 6: UDP/IP end-to-end throughput using uncached, non-volatile fbufs

other hand, even a microkernel-based system does not necessarily imply that multiple domain crossings are required. For example, recent work suggests that it is possible to implement the TCP/IP protocol suite using application libraries, thus requiring only a single user/kernel crossing in the common case [13, 18].

There are three responses to this question. First, server-based systems have undeniable advantages; it is a general technique that makes it possible to transparently add new services and entire OS personalities without requiring modification/rebuilding of the kernel and applications. It is not yet clear whether the application-library approach can achieve the same effect, or even generalize beyond TCP/IP. Second, our work shows how to avoid the negative impact of domain crossings on end-to-end throughput for large messages. This is significant because many applications that demand high throughput generate/consume large data units. Such applications include continuous media, data visualization, and scientific programs. For these applications, minimizing domain crossings may therefore not be as critical. Third, as demonstrated in the previous section, fbufs are also well suited for situations where only a single kernel/user domain crossing occurs in the data path.

5.2 Characteristics of Network I/O Revisited

Fbufs gain efficiency partly by placing certain restrictions on the use of I/O buffers, as described in Section 2. Nevertheless, fbufs can be transparently integrated with network subsystems that are written to an immutable buffer abstraction, as demonstrated by our x-kernel based implementation. Necessary modifications are restricted to software modules that allocate buffers based on cached fbufs, and modules that interpret I/O data.

In the interest of preserving user-level throughput, it is necessary to transfer buffers between application programs and operating system as efficiently as between modules of the operating system. Unfortunately, the semantics of the UNIX read/write interface make it difficult to use fbufs (or any other VM based technique). This is because the UNIX interface has copy semantics, and it allows the application to specify an unaligned buffer address anywhere in the its address space. We therefore propose the addition of an interface for high-bandwidth I/O that uses immutable buffer aggregates [7]. New high-bandwidth applications can use this interface; existing applications can continue to use the old interface, which requires copying.

The use of such an interface requires applications to use an abstract data type that encapsulates buffer aggregates. This implies that an application that reads input data must be prepared to deal with the potentially non-contiguous storage of buffers, unless it is willing to pay the performance penalty of copying the data into contiguous storage. To minimize inconvenience to application programmers, our proposed interface supports a generator-like operation that retrieves data from a buffer aggregate at the granularity of an application-defined data unit, such as a structure or a line of text. Copying only occurs when a data unit crosses a buffer fragment boundary.

Since fbufs are immutable, data modifications require the use of a new buffer. Within the network subsystem, this does not incur a performance penalty, since data manipulations are either applied to the entire data (presentation conversions, encryption), or they are localized to the header/trailer. In the latter case, the buffer editing functions—e.g., join, split, clip—on the aggregate object can be used to logically concatenate a new header with the remaining, unchanged buffer. The same is true for application data manipulations, as long as manipulations on part of the data are localized enough to warrant the small overhead of buffer editing. We cannot imagine an application where this is a problem.

Cached fbufs require that the I/O data path of an fbuf be identified at the time the fbuf is allocated. In those cases where the I/O data path cannot be determined, a default allocator is used. This allocator returns uncached fbufs, and as a consequence, VM map manipulations are necessary for each domain transfer. The driver for the Osiris network interface used in our experiments employs the following strategy. The driver maintains queues of preallocated cached fbufs for the 16 most recently used data paths, plus a single queue of preallocated uncached fbufs. The adapter board performs reassembly of incoming PDUs from ATM cells by storing the cell payloads into a buffer in main memory using DMA. When the adapter board needs a new reassembly buffer, it checks to see if there is a preallocated fbuf for the virtual circuit identifier (VCI) of the incoming PDU. If not, it uses a buffer from the queue of uncached fbufs.

Note that the use of cached fbufs requires a demultiplexing capability in the network adapter, or it must at least permit the host CPU to inspect the packet header prior to the transfer of data into main memory. While most low-bandwidth (Ethernet) network adapters do not have this capability, network adapters for high-speed networks are still the subject of research. Two prototypes of such interfaces we are familiar with (the Osiris board, and the HP Afterburner board [5]) do have adequate support.

5.3 Architectural Considerations

As observed in Section 5, the performance of cached fbufs for large messages is limited by TLB miss handling overhead.[7] In many modern architectures (including the MIPS), TLB entries are tagged with a domain identifier. This organization penalizes sharing in a global address space, since a separate TLB entry is required in each domain for a particular shared page, even if the address mapping and protection information are identical.

Several modern processors permit the use of a single TLB entry for the mapping of several physical pages (HP-PA, MIPS R4000). This facility can be used to reduce the TLB overhead for large fbufs. However, the physical pages mapped with a single TLB entry must be contiguous in physical address. This requires a form of physical memory management currently not present in many operating systems.

Choosing the size of the fbuf region involves a tradeoff. The region must be large enough to accommodate the I/O buffering needs of both the kernel and all user domains. On the other hand, a large window reduces the size of the private address spaces of kernel and user domains. The trend towards machines with 64-bit wide virtual addresses should make this less of an issue.

[7] Our implementation already clusters DAG nodes to reduce the number of pages occupied by a single fbuf aggregate.

5.4 Relationship to Other VM Systems

This paper describes an integrated implementation of fbufs based on modifying/extending the Mach kernel. We now briefly discuss ways to layer fbufs on top of existing VM systems. Note that in each case, kernel modifications are still required to give in-kernel software modules (e.g., device drivers) access to the fbuf facility.

Several modern VM systems—e.g., those provided by the Mach and Chorus microkernels—export an external pager interface. This interface allows a user process to determine the semantics of a virtual memory object that can be mapped into other protection domains. We have designed an fbuf implementation that uses a Mach external pager and does not require modifications of the VM system. In this implementation, an fbuf transfer that requires VM mapping changes must involve the external pager, which requires communication between the sender of the fbuf and the pager, and subsequently between the pager and the kernel. Consequently, the penalty for using uncached and/or non-volatile fbufs is expected to be quite high.

A shared memory facility such that provided by System V UNIX could presumably be used to implement fbufs. However, it is unclear how the semantics for read accesses to protected locations in the fbuf region (Section 3.2.4) can be achieved. Also, many implementations of System V shared memory have rather severe limitations with regard to the number and size of shared memory segments.

6 Conclusions

This paper presents an integrated buffer management/transfer mechanism that is optimized for high-bandwidth I/O. The mechanism, called fbufs, exploits locality in I/O traffic to achieve high throughput without compromising protection, security, or modularity. Fbufs combine the page remapping technique with dynamically mapped, group-wise shared virtual memory. In the worse case, it performs as well as the fastest page remapping facilities described in the literature. Moreover, it offers the even better performance of shared memory in the common case where the data path of a buffer is know at the time of allocation. Fbufs do not compromise protection and security. This is achieved through a combination of group-wise sharing, read-only sharing, and by weakening the semantics of buffer transfers (volatile buffers).

A micro experiment shows that fbufs offer an order of magnitude better throughput than page remapping for a single domain crossing. Macro experiments involving the UDP/IP protocol stack show when cached/volatile fbufs are used, domain crossings have virtually no impact on end-to-end throughput for large messages.

Acknowledgements

Special thanks to Bellcore for making the Osiris boards available, and to Bruce Davie for the time he invested in supporting our work.

References

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, June 1986.

[2] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, Feb. 1990.

[3] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Transactions on Computer Systems*, 9(2):175–198, May 1991.

[4] D. R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, Mar. 1988.

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

[6] B. S. Davie. A host-network interface architecture for ATM. In *Proceedings of the SIGCOMM '91 Conference*, pages 307–315, Zuerich, Switzerland, Sept. 1991.

[7] P. Druschel, M. B. Abbott, M. A. Pagels, and L. L. Peterson. Network subsystem design. *IEEE Network*, 7(4):8–17, July 1993.

[8] 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.

[9] 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. Association for Computing Machinery SIGOPS, October 1991.

[10] 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.

[11] D. B. Johnson and W. Zwaenepoel. The Peregrine high-performance RPC system. *Software—Practice and Experience*, 23(2):201–221, Feb. 1993.

[12] 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.

[13] 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.

[14] J. C. Mogul. Network locality at the scale of processes. *ACM Transactions on Computer Systems*, 10(2):81–109, May 1992.

[15] J. K. Ousterhout. Why Aren't Operating Systems Getting Faster As Fast as Hardware? In *Usenix 1990 Summer Conference*, pages 247–256, June 1990.

[16] M. D. Schroeder and M. Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, Feb. 1990.

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

[18] C. Thekkath, T. Nguyen, E. Moy, and E. Lazowska. Implementing network protocols at user level. In *Proceedings of the SIGCOMM '93 Symposium*, Sept. 1993.

[19] 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.