Operating System Support for High-Speed Communication

Techniques to eliminate processing bottlenecks in high-speed networking are presented.

EMERGING network technologies such as fiber-optic transmission facilities and Asynchronous Transfer Mode (ATM) hold the promise of delivering data rates on the order of gigabits per second between individual workstations on local- and wide-area networks [11]. This increase in network capacity, combined with the explosive growth in microprocessor performance, enables a range of innovative new distributed computing applications. Distributed multimedia, including real-time audio and video, and supercomputing on clusters of workstations are examples of such emerging applications. One important factor that could dramatically influence the success of these new technologies is the degree to which operating systems can make these networking resources available to application programs.

The role of an operating system (OS) is to mediate and multiplex the access of multiple application programs to the computing resources provided by the underlying hardware. Ideally, the operating system should not itself consume a significant share of these resources. Unfortunately, current operating systems are threatening to become bottlenecks in delivering input/output (I/O) data streams to application programs at high rates [5, 7, 21, 22]. In particular, data streams between applications on hosts connected by high-speed networks suffer bandwidth degradation and added latency due to the operating system running on the hosts.

This article looks at the I/O bottleneck in operating systems, with particular focus on high-speed networking. We start by identifying the causes of this bottleneck, which are rooted in a mismatch of operating system behaviors with the performance characteristics of modern computer hardware. Then traditional approaches to supporting I/O in operating systems are reevaluated in light of current hardware performance tradeoffs. This reevaluation gives rise to a set of novel techniques that eliminate the I/O bottleneck.

The root cause of the OS I/O bottleneck is that speed improvements of main memory have lagged

Peter Druschel

behind those of the central processing unit (CPU) and I/O devices during the past decade [6]. In state-of-the-art computer systems, the bandwidth of main memory is orders of magnitude lower than the bandwidth of the CPU, and the bandwidths of the fastest I/O devices approach that of main memory.[1] The previously existing gap between memory and I/O bandwidth has almost closed, and a wide gap has opened between CPU and memory bandwidth, leaving memory as a potential bottleneck.

To bridge the gap between CPU and memory speed in modern computers, system designers employ sophisticated *cache* systems. A cache exploits *locality of reference* in memory accesses to reduce main memory traffic. Locality of reference is the property of a sequence of memory accesses to reference preferentially those memory locations that were either accessed recently (temporal locality), or that are close to recently accessed locations in the address space (spatial locality). A cache is a high-speed memory that holds a recently accessed subset of the data stored in main memory. When the CPU accesses a main memory location for which the cache holds a copy, no main memory access is necessary, and the operation can complete at the speed of the cache memory. A cache reduces main memory traffic and lowers the average memory access latency experienced by the CPU. The effectiveness of the cache in bridging the CPU/memory speed gap depends on the degree of locality in the memory accesses of the executed program.

UNFORTUNATELY, in most current systems the accesses to I/O data buffers generated by operating system and applications *do not* have sufficient locality to allow the cache system to minimize memory traffic. As a result, excess memory traffic is causing a substantial drop in I/O performance, that is, throughput and latency. This poor locality is primarily caused by

- data movements (copying),
- inappropriate scheduling of the various I/O processing steps, and
- a system structure that requires OS kernel involvement in all I/O activity.

Moving or copying data from one main memory location to another has poor temporal locality, because the target locations of the copy are not likely to have been accessed prior to the copy, and the source locations will not likely be accessed after the copy. As a result, accesses to the target locations will cause many cache misses, and useful cache contents are replaced by useless cached copies of source locations, causing further cache misses after the copy. Both effects contribute substantially to memory traffic. Data copying occurs in current systems due to a lack of integration in the design of I/O adaptors, buffer management schemes, interfaces, and mechanisms for the transfer of data across protection domain boundaries.

Locality can suffer further because various I/O processing steps and their associated data accesses occur in the context of multiple, separately scheduled threads of control (e.g., interrupt handlers, kernel threads, application processes). In a multiprogrammed environment, these processing steps may not be scheduled to execute in strict succession. That is, the processing steps of a data unit may be interleaved with the execution of unrelated tasks, with their own distinct set of memory references. Thus, accesses to a particular I/O data unit are temporally separated, resulting in poor data access locality.[2]

Finally, current systems require that the operating system kernel be involved in each individual I/O operation that an application initiates. Thus, I/O requires a protection domain switch between application and operating system, and the transfer of data across the user/kernel protection boundary. Both entail a drop in memory access locality, which can limit I/O bandwidth and significantly contribute to I/O latency.

In summary, limited memory bandwidth in modern computer systems is a potential source of performance problems. Cache systems can hide the slow speed of main memory only when the memory accesses generated by a program have good locality of reference. Accesses to I/O data generated by operating systems and applications tend to have poor locality, rendering the cache unable to avoid the memory bottleneck in processing network I/O.

The goal of our work is to remove the I/O bottleneck, without sacrificing modularity in the structure of operating system and applications. This article describes two novel techniques that are part of a coordinated design to minimize main memory traffic. We present a novel OS facility, called fast buffers (fbufs), for the management and transfer of I/O data buffers across protection-domain boundaries, both in monolithic and server-based operating systems. Then we introduce an innovative OS facility that takes advantage of user-level network protocols and limited support from the network adaptor. It gives applications direct but controlled access to the network adaptor, which significantly reduces network message latency. This facility, called *application-device channels* (ADCs), leaves control of the network adaptor to the operating system, thus allowing transparent sharing of the device among multiple, non-privileged application programs.

Fast Buffers (fbufs)

In any operating system that supports protection and security, protection boundaries separate user process-

[1]We define CPU bandwidth as the maximal sustained rate, in bytes per second, at which the CPU can absorb data; memory bandwidth as the sustained rate at which the CPU can read data from main memory; and the I/O bandwidth as the sustained rate at which data can be transferred to and from I/O devices.
[2]An additional problem can occur on shared-memory multiprocessors, when not all processing steps are scheduled to run on the same processor.

es from each other, and from the OS kernel.[3] When an application and/or the operating system make use of user-level servers, many protection boundaries may occur in the I/O data path. Supporting high-speed I/O requires an efficient means of transferring I/O data across protection domain boundaries. Moreover, an appropriate transfer facility must be integrated with the application programming interface (API) and buffer manager to be effective [8].

This section describes a high-bandwidth cross-domain transfer and buffer management facility, 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 build on two well-known techniques for transferring data across protection domains—*page remapping* and *shared memory*—but overcome the disadvantages of either method. It is equally correct to view fbufs as using shared memory (where sharing is read-only and 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 retained for use by future transfers).

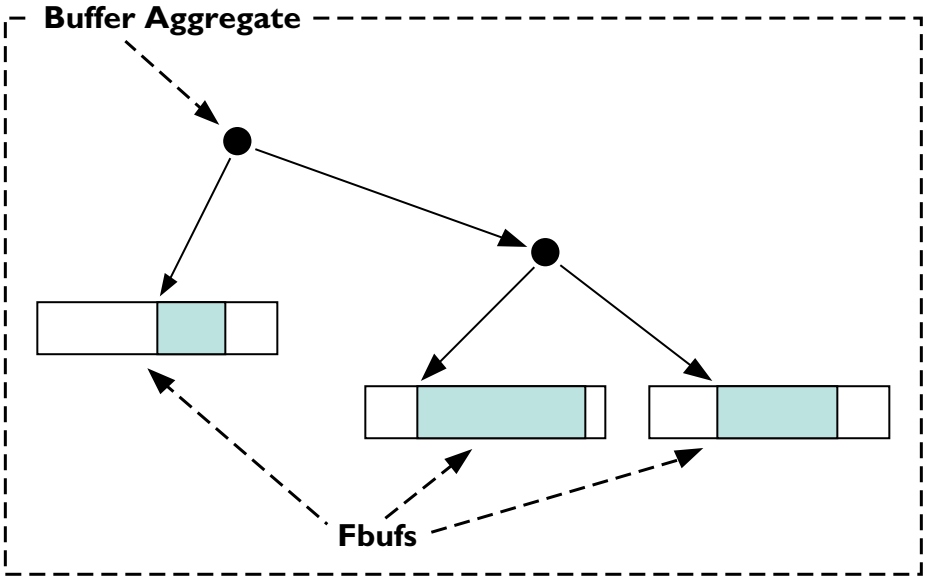

Figure 1. Buffer aggregate

A complete description of the design and implementation of fbufs, along with a detailed performance study, can be found in [10]. A number of research projects have recently adopted variations of the fbuf mechanism [16, 23]. Also, work is currently under way at Rice University to build an integrated, copy-free I/O and file caching system based on fbufs for a commercial Unix system.

I/O data is stored in memory buffers called *fbufs*, each of which consists of one or more contiguous virtual memory pages. A protection domain gains access to an fbuf either explicitly by allocating the fbuf, or implicitly by receiving the fbuf via inter-process communication (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 (ADT) is layered on top of fbufs to support *buffer aggregation*. This ADT can combine multiple (discontiguous) fbufs, and allows the manipulation of the resulting *buffer aggregate* as a single data object. This is important because data arrives from a network generally in the form of multiple independent fragment packets that are stored in separate fbufs. In the interest of avoiding data copying, this initial, discontiguous storage layout must be preserved even after the packet data is reassembled. Buffer aggregates support this in a convenient and efficient manner. Similar buffer aggregation facilities are widely used inside many operating system kernels. Examples include the BSD Unix *mbufs* [17], and *x*-kernel *messages* [14].

Operations supported by the buffer aggregate ADT include concatenation, splitting, truncation, and logical copying of buffer aggregates. Our implementation uses a directed acyclic graph (DAG) data structure (depicted in Figure 1) to keep track of the constituent fbufs of a buffer aggregate.

The ADT treats fbufs as *immutable* buffers, that is, they are created with initial data contents that may no be subsequently changed until the buffer is (logically) destroyed. This allows the ADT to implement logical copying and other operations efficiently by creating multiple references to the underlying fbufs and maintaining reference counts. It should be emphasized that the immutability of fbufs does not imply that buffer aggregates are immutable. It merely requires that the ADT implementation must store modified portions of an aggregate in a new fbuf.

Passing buffer aggregates across protection boundaries requires the transfer of constituent fbufs from one protection domain to another. The design of the fbuf transfer mechanism is best described by starting with a basic mechanism based on page remapping, and then evolving the design with a series of optimizations.[4]

We use a conventional remapping facility with

[3] In Unix and most other operating systems, a protection domain corresponds to a (heavyweight) process.
[4] Some of the optimizations can be applied independently, giving rise to a set of possible implementations with different restrictions and costs.

copy semantics (copy-on-write) as the baseline for our design. A virtual page remapping facility logically copies (or moves) a set of virtual memory pages among protection domains by modifying virtual memory mappings. The use of such a facility to create, destroy, and transfer a buffer aggregate involves the following steps. The complexity of each step is given in brackets.

1. Allocate a Buffer Aggregate (Originator)
(a) Find and allocate a free virtual address range in originator's address space [O(#fbufs)]
(b) Allocate physical memory pages and clear contents [O(#pages x pagesize)]
(c) Enter mappings in originator's page table [O(#pages)]

2. Send Buffer Aggregate (Originator)
(a) Generate a list of fbufs from the buffer aggregate [O(#fbufs)]
(b) Raise protection of fbufs' address ranges in originator to read-only [O(#fbufs)]
(c) Update page table entries, ensure TLB/cache consistency [O(#pages)]

3. Receive Buffer Aggregate (Receiver)
(a) Find and reserve a free virtual address range in receiver's address space [O(#fbufs)]
(b) Enter shared mappings in receiver's page tables [O(#pages)]
(c) Construct a buffer aggregate from the received list of fbufs [O(#fbufs)]

4. Free a Buffer Aggregate (Originator, Receiver)
(a) Deallocate fbufs' virtual address ranges [O(#fbufs)]
(b) Remove mappings from page table, ensure TLB/cache consistency [O(#pages)]
(c) Free physical memory pages if there are no more references to the fbufs [O(#pages)]

Even in a careful implementation, these actions can result in substantial overhead. In modern virtual memory (VM) systems, modifying a mapping requires updates to at least three data structures: the machine-independent address space map, the machine-dependent virtual-to-physical translation table (page table), and the physical-to-virtual translation table. All of these tend to be large, pointer-based data structures with poor access locality. Moreover, page table modifications that increase protection for a page (i.e., reduce access permissions) require subsequent actions to ensure consistency of the translation lookaside buffer (TLB). If the machine uses a cache with virtual address tags, cache consistency operations may also be required. In multiprocessor machines, these consistency actions may need to be performed simultaneously on all CPUs, which can be very expensive.

For example, given an I/O data path with two domain crossings, six page-table updates are required for each page of I/O data, three of which may require TLB/cache consistency actions. Each allocated physical page may need to be cleared—that is, filled with zeroes—to ensure data privacy. In summary, page remapping suffers from significant per-page costs, even though it avoids data copying. Also, the relative cost of remapping is likely to increase as CPUs become faster, since it involves memory accesses with poor locality. The following set of optimizations are designed to eliminate virtual memory mapping changes and other per-page and per-fbuf costs associated with the baseline remapping mechanism.

The first optimization, called *restricted dynamic read sharing*, 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 reserved in all domains' address spaces. Note that domains do not have unrestricted access to the memory that is mapped into the fbuf region. 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. That is, an fbuf cannot be written by a receiver, and it can be written by its originator only if no receiver is referencing it. An fbuf is referenced by a receiver if it was passed to that receiver via IPC, and the receiver has not yet deallocated the fbuf.

The first restriction implies that an fbuf can be mapped at the same virtual address in the originator and all receivers. This eliminates the need for action (3a) during transfer. 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. Also, this form of sharing permits the transfer of pointer-based data structures in fbufs without pointer translation. The second restriction eliminates the need for a copy-on-write mechanism, since sharing of read-only buffers trivially ensures copy semantics. Eliminating copy-on-write simplifies the VM data structures needed to represent the fbuf region, and thus increases efficiency.

Restricted dynamic read sharing places two constraints on the use of fbufs. (1) A special allocator must be used for fbufs, since fbufs occupy a dedicated address range. (2) Fbufs are immutable, that is, they are allocated with an initial data content that may not be subsequently changed. Fortunately, these constraints are already satisfied due to the use of a buffer aggregate abstract data type on top of fbufs. Buffer aggregates already use special allocators, and they use immutable buffers to facilitate logical copying.

The next optimization, *fbuf caching*, takes advantage of locality in interprocess communication. Specifically, it exploits the fact that once a network packet has followed a certain data path—that is, visited a certain sequence of protection domains—more packets can be expected to travel the same path soon.

CONSIDER what happens when a packet arrives from the network. An fbuf is allocated in the kernel, filled with data, and then transferred one or more times until the data is consumed by the destination domain. At that 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 the 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 on the same data path, the fbuf can be reused. In this case, no clearing of the buffers is required to ensure data privacy (since all domains with access have already seen the fbuf's contents during its previous use), and the appropriate mappings in the receivers already exist.

point. This determination must be made *prior* to the transfer of the message body into main memory. A packet filter/classifier inspects the headers, determines the endpoint and associated data path, then the packet body is transferred into an appropriately allocated fbuf.

Recall that a buffer aggregate ADT is layered on top of fbufs. The transfer facility described so far transfers fbufs, not buffer aggregates, across protection boundaries. A buffer aggregate is 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 buffer aggregate is rebuilt on the receiving side (3c). Any internal data structures used in the implementation of buffer aggregates to link the constituent fbufs (e.g., interior DAG nodes) are stored in memory that is private to each domain.

Consider now an optimization that incorporates

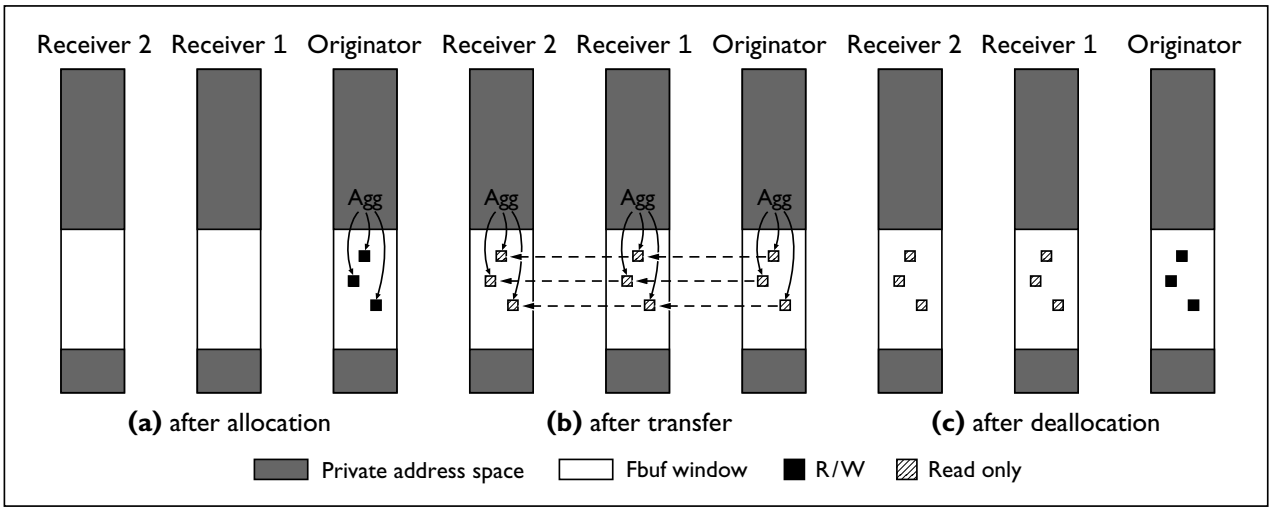

Figure 2. Fbuf caching

Fbuf caching eliminates actions (1a–c), (3a–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. Figure 2 depicts the operation of cached fbufs: A buffer aggregate is allocated by its originator (a); the aggregate is then transferred twice (b); finally, all aggregates have been deallocated, but the fbufs and their mappings are retained for future use (c).

The fbuf caching optimization requires that the originator be able to determine the I/O data path at the time of fbuf allocation. Application programs that generate output data can specify the I/O data path at the time of an fbuf allocation simply by referring to the communication endpoint that they intend to use for transmitting the data. The OS kernel allocates fbufs for incoming network packets. To take advantage of cached fbufs, the packet headers must be inspected to determine the data's destination end-

knowledge about the buffer aggregate ADT 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 buffer aggregate 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 buffer aggregate DAG 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 buffer aggregate. Steps (2a) and (3c) are eliminated under this optimization.

Under the previous optimizations, the transport of an fbuf from the originator to a receiver still requires two page table updates per page: one to remove write permissions 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 weakening the fbuf transfer semantics. We call the resulting fbufs *volatile*. A receiver must conservatively assume that the contents of a received volatile fbuf may change at any time unless the receiver explicitly requests that the fbuf be *secured*, that is, write permissions are removed from the originator—removing write permissions is unnecessary in many cases.

Consider the case where an fbuf is passed by its originator to a receiver domain. A correct and well-behaved originator will not attempt to write to the (immutable) fbuf after the transfer. If the originator is a trusted domain—that is, the OS kernel has allocated the buffer for an incoming packet—then the buffer's immutability clearly need not be enforced. That is, there is no need to remove write permissions from a trusted originator.

The situation is more interesting when the originator is an (untrusted) application that generated some data. In general, a receiver of an fbuf could fail while processing the buffer's contents if and when the fbuf is concurrently modified by a malicious or faulty originator. However, depending on the type of processing performed by the receiver, such an originator-induced "failure" may *not* violate the system's security and protection policies. For example, processing performed by layers of the I/O subsystem typically consists of data transformations such as adding redundancy (CRC), compression, encryption, or data presentation conversions. Almost all of these computations have the property that concurrent changes to the input data will result in garbled output data, without adverse effects to the state of the system and no violation of the system's protection and security policies. Thus, a faulty application merely interferes with its own output operation by modifying the buffer.

It should be emphasized that volatile fbufs, when properly used, *do not* weaken or compromise protection and security. Proper use requires that programmers determine if their code is vulnerable to concurrent modifications of received fbufs' contents.[5] If violations of the system's protection and security policies as a result of such modifications can be ruled out, then volatile fbufs can be used without change. Else, the receiver code must be modified to explicitly request that write permissions are removed from all received fbufs with untrusted originators. Our buffer aggregate ADT supports an appropriate operation that has this effect.

In practice, we have found that in most cases where a receiver cannot tolerate concurrent modifications, the originator is a trusted domain. Intuitively, this is because most input data originates from the trusted OS kernel. Output data, which often originates from untrusted application domains, is generally not interpreted but merely *transformed* by receivers.

When combined, the optimizations described eliminate all virtual memory mapping changes and other per-page and per-fbuf costs associated with cross-domain data transfer in the common case. This common case requires that the data path can be identified at fbuf allocation time, an appropriate fbuf is already cached, and removing write permissions from the originator is unnecessary. In the less common cases where one of these conditions is not satisfied, some VM mapping changes remain, but data copying can always be avoided.

In order to achieve a copy-free end-to-end data path (i.e., from source to sink device), applications should directly manipulate buffer aggregates. From the perspective of the programmer, this has two consequences: the data contained in an aggregate is not necessarily stored contiguously; and, the data cannot be modified in place, since fbufs are immutable. Applications that read I/O data mostly sequentially can easily be modified to work with buffer aggregates. They can use operations supported by the buffer ADT that generate pointers to the contiguous fbufs of the aggregate. Applications that read input data sequentially and write modified output data store the modified data into a new fbuf. If only a portion of the input data is modified, that portion is stored in a new fbuf, and then joined with the original fbufs to form a new aggregate, using operations provided by the buffer ADT. This is efficient, as long as there is not an excessive number of small, scattered modifications.

A large class of applications reads and writes I/O data mostly sequentially, and can be modified with little effort to use buffer aggregates, resulting in a copy-free I/O data path. Scientific applications often read large arrays of data from input devices. For such applications, the buffer aggregate representation may be inappropriate, since the program may depend on contiguous storage and in-place modifications for efficiency. In these cases, the cost of copying the data into a contiguous array can be recovered by the increased efficiency of subsequent accesses to the array.

The original prototype implementation of fbufs was done in the context of CMU's Mach 3.0 microkernel (MK74), augmented with a network subsystem based on the University of Arizona's *x*-kernel (Version 3.2) [14]. The hardware platform consists of a pair of DecStation 5000/200 workstations, each of which was attached to a prototype Osiris ATM network interface board, designed by Bellcore for the Aurora Gigabit testbed [9]. The Osiris boards were connected by a null modem, and they support a link speed of 622Mbps (OC-12).

Micro-benchmarks of a single protection boundary crossing show that fbufs perform an order of magnitude better than the baseline page-remapping scheme described previously. This page-remapping facility is our reimplementation of one of the more highly tuned implementations of page remapping

[5]For example, it can be shown that any program that reads each input buffer location only once can be safely used with volatile fbufs.

described in the literature [25]. Moreover, we found that fbufs consistently outperform Mach's native IPC facility over all message sizes. In a second experiment, application-to-application throughput is measured using the UDP/IP protocol suite in a local loopback test. The throughput achieved with fbufs exceeds that of page remapping by a factor of two in this experiment. Finally, the throughput of a simple sliding window protocol on top of UDP/IP was measured between two DecStations connected by an OC-12 ATM link. Results show that fbufs can increase throughput by up to a factor of two, or reduce CPU load by up to 45%, when compared with the baseline remap scheme. Detailed performance results can be found in [10].

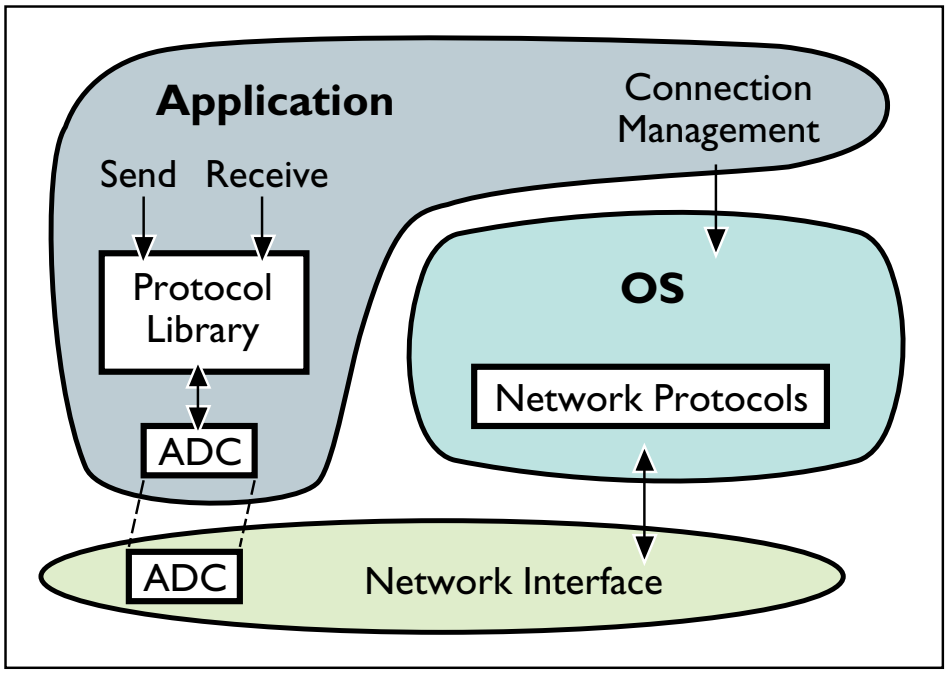

Figure 3. Application-device channel

Application-Device Channels

The work on fbufs shows how to avoid bandwidth degradation due to protection boundary crossings along the I/O data path. Protection boundaries also have the effect of adding latency to I/O operations, due to the necessary argument validation, protection domain switch, and the resulting drop in memory access locality. Fbufs cannot eliminate the added latency caused by protection domain boundaries. In this section, we describe a new approach that gives applications direct access to a network device for common I/O operations, thus bypassing the OS kernel and removing protection boundaries from simple application-network data paths.

In the past, I/O operations had inherently long latencies due to slow hardware devices (disks, slow network links). Additional software latencies due to OS kernel involvement were insignificant. With the low delays of high-speed local-area networks, on the other hand, OS software latencies can easily dominate the end-to-end communication delays experienced by applications. For example, parallel programming systems implemented on workstation clusters are very communication-intensive. The performance and scalability of such systems can suffer from added communication latencies caused by the lack of direct access to the network hardware.

The design of application-device channels (ADCs) recognizes communication as a fine-grained, performance-critical operation, and allows applications to bypass the operating system kernel during network send and receive operations. The OS is normally only involved in the establishment and termination of network connections. Protection, safety, and fairness are maintained, because the network adaptor validates send and receive requests from application programs based on information provided by the OS during connection establishment. Unlike other systems that support user-level network access using special-purpose, dedicated network interfaces, [3, 4, 12, 19], ADCs can be used with many commercial, general-purpose network adaptors.

The basic approach taken in designing ADCs is depicted in Figure 3. First, instead of centralizing the network communication software inside the operating system, a copy of this software is placed into each user domain as part of the standard library that is linked with application programs. This user-level network software supports the standard application programming interface. Thus, the use of ADCs is transparent to application programs, except for performance.

Second, the user-level network software is granted direct access to a restricted set of functions provided by the network adaptor. This set of functions is sufficient to support common network send and receive operations without involving the OS kernel. As a result, the OS kernel is removed from the critical network send/receive path. An application process communicates with the network adaptor through an application device channel, which consists of a set of data structures that is shared between network adaptor and the user-level network software. These data structures include queues of buffer descriptors for the transmission and reception of network messages.

When an application opens a network connection,

the operating system informs the network adaptor about the mapping between network connection and ADC, creates the associated shared data structures, and grants the application process access to these data structures. The network adaptor passes subsequently arriving network messages to the appropriate ADC, and transmits outgoing messages queued on an ADC by an application using the appropriate network connection. An application cannot gain access to network messages destined for another application, nor can it transmit messages other than through network connections opened on its behalf by the OS.

There are a number of compelling advantages to the use of application device channels. First, network send and receive operations bypass the OS kernel. This eliminates protection domain boundary crossings, which would otherwise entail data transfer operations, domain switching, and the associated drop in memory access locality. In traditional implementations, these costs account for a significant portion of the application-to-application latency experienced on a high-speed local-area network.

Second, since application device channels give application domains low-level network access, it is possible to use customized network protocols and software. This flexibility can lead to further performance improvements, since it allows the use of application-specific knowledge to optimize communications performance. For example, a parallel programming system implemented on a workstation cluster can gain efficiency by using specialized message-passing protocols and buffering strategies instead of generic TCP/IP network connections.

Finally, with application device channels, all processing and resources necessary for network communication are associated with an application process. This eliminates kernel resource constraints and scheduling anomalies that plague traditional network implementations.

There are three components to the implementation of ADCs: first, a user-level implementation of the networking software, including device driver, network protocols, and communications API; second, the actual ADC mechanism, which provides a shared-memory communication channel between application process and network adaptor; and third, network adaptor support for ADC-based networking. A general discussion of possible approaches for each component and a description of our prototype implementation follows.

> The goal of our work is to remove the I/O bottleneck without sacrificing modularity in the structure of operating system and applications.

User-level implementations of TCP/IP network software have been described in [18, 24]. The general approach is to separate and decentralize common-case operations and link the resulting code with user applications. A complete implementation of the network software remains in the kernel. The in-kernel software handles exceptional cases and tasks that require centralized control.

Several approaches are possible for the implementation of the ADCs, depending on the capabilities of the network adaptor and the I/O architecture of the host. If the network adaptor contains memory that is accessible on the host's I/O bus, a portion of this memory can be divided into VM page-sized portions and mapped individually into user processes. These pages then serve as shared-memory channels between applications and network interface. We call this approach *network interface-based ADCs*. Alternatively, ordinary main memory pages can be used for the same purpose. The user process accesses the ADC through its mapping of a main memory page. The network interface accesses the ADC using direct memory access (DMA) of the corresponding physical page. This approach is called *main memory-based ADCs*.

A second distinction concerns the type of data exchanged through ADCs. One approach is to only pass control/status information through the ADC. Here, the ADC pages contain essentially queues of buffer descriptors for transmission and reception. The actual network (payload) data is stored in ordinary pageable application memory and accessed by the network adaptor using DMA. This solution allows copy-free (zero-copy) I/O. Hardware limitations on a given host machine may preclude this approach. The host I/O bus may support DMA to only a subset of main memory, and/or the network adaptor may not support DMA at all. In this case, payload data can be passed directly through the ADC.

The final component of an ADC-based network architecture is ADC support in the network adaptor. The network adaptor must be able to demultiplex incoming network packets to the correct ADC, handle transmission requests on ADCs while exercising appropriate network access control, interact with the OS kernel to establish/relinquish ADCs, and handle exceptional cases. We shall refer to this set of tasks collectively as *ADC multiplexing*. Naturally, appropriate support is not likely to be found in off-the-shelf net-

work interfaces. Fortunately, the necessary support can be implemented relatively easily by modifying the firmware that controls the on-board CPU found in many commercial high-performance network adaptors.

Even with primitive network adaptors that do not contain a programmable CPU or otherwise lack the necessary support, it is still possible to implement ADCs. The approach in this case is to perform the ADC multiplexing task inside the OS kernel. ADCs are implemented as shared memory channels between user processes and the OS kernel. A kernel thread monitors ADCs and multiplexes traffic from and to the network device. On a single-processor machine, a context switch to this kernel thread is necessary during send and receive operations, which comes at some cost in latency. However, the advantages of customizable network software and improved resource control can still be retained. An ADC implementation using this approach on a uniprocessor machine is similar to the user-level protocol architectures described in the literature [18, 24].

The implementation of ADCs with uncooperative network adaptors can be as efficient as one with appropriate NI support when executed on a shared-memory multiprocessor machine. Here, one of the host CPUs is scheduled to execute the in-kernel ADC multiplexing thread. The remaining CPUs execute user applications and perform network I/O through their ADCs, bypassing the kernel. The system behaves similar to one with a smart network interface, except that a host CPU takes care of the ADC multiplexing. Note that it is not necessary to permanently dedicate a CPU to the task of ADC multiplexing. A CPU can be scheduled for this task when it is otherwise idle. If all CPUs are busy executing applications, then network communication again requires a context switch to the ADC multiplexing thread, at some cost in latency.

We next describe our prototype implementation of ADCs. The system was realized in a Mach 3.0/*x*-kernel environment, using the Osiris ATM network adaptor. Here, the Osiris on-board memory is divided into 16 4KB pages, each of which contains a free buffer queue, a send and a receive queue. One set of queues is used by the operating system in the usual way. The remaining 15 sets are available as application device channels.

When an application opens a network connection, the operating system maps a set of queues into the application's address space to form an application device channel. Linked with the application is an ADC channel driver, which performs essentially the same functions as the in-kernel Osiris device driver. Also linked with the application is a replicated, user-level implementation of the network protocol software.

As part of the connection establishment, the operating system assigns a set of ATM virtual circuit identifiers (VCIs) and a priority to the ADC. The Osiris receive processor queues incoming messages on the receive queue of an ADC if the VCI of the message is in the set of VCIs assigned to that ADC. Applications can send messages through an ADC using the VCIs assigned to that ADC. Therefore, applications can only receive and transmit messages on connections that they have opened with the authorization of the operating system. The ADC priority is used by the Osiris transmit processor to determine the order of transmissions from the various ADCs' transmit queues. Using these priorities, the OS can enforce network resource allocation policies.

For each ADC, the Osiris on-board processors maintain a virtual-to-physical address translation table. This table provides address translation for the application's virtual buffer addresses (needed for DMA) and ensures memory access protection. When an application queues a buffer with an address not contained in this table, the on-board processor asserts a host interrupt. The operating system's interrupt handler in turn looks up the address in the application's page table. If a valid mapping exists, the kernel provides the appropriate translation information to the network adaptor, paging in the page if necessary; otherwise, an access violation exception is raised in the offending application process. When the kernel selects a page for replacement, it asks the network interface to flush the corresponding mapping from its translation table.

All host interrupts are fielded by the OS kernel's interrupt handler. If the interrupt indicates an event affecting an ADC, such as the transition of an ADC's receive queue from the empty to a non-empty state, the interrupt handler directly signals a thread in the ADC channel driver linked with the application. A full context switch to an OS kernel thread does not occur in this case. Note also that due to the use of interrupt-reducing techniques, the average number of host interrupts per network message is less than one [9]. For example, when a message is added to a non-empty receiver queue, no host interrupt is necessary, because the host has already been notified of a non-empty receiver queue when the first message arrived.

The number of application processes with open network connections can exceed the maximal number of ADCs (15 in our implementation). In this case, only a subset of the processes can use ADCs, and the remaining processes must use the normal I/O path through the OS kernel. For best performance, the OS tries to assign the available ADCs to the processes with the most network traffic. An ADC can be reassigned from one application to another. To do this, the OS deactivates the ADC channel driver in one application, causing subsequent I/O requests to follow the normal path through the kernel. Then, another ADC channel driver is activated, causing further network traffic to use the ADC. ADC reassignment is transparent to application programs, except for performance.

Performance results of our prototype ADC imple-

Achieving a high-performance I/O system requires an integrated design of application programming interface, cross-domain data transfer, buffer management, and network interface.

mentation were obtained on DEC 3000/600 (175MHz Alpha) workstations connected by a pair of OSIRIS boards, linked back-to-back. The latency figures include interrupt latency, that is the receiver does not poll the network device. A short message (1 byte) round-trip latency of 154μsecs was measured between test programs configured directly on top of the user-level OSIRIS device driver. This number is significant since it is a lower bound for the latency an application can achieve using customized network protocols on top of ADCs. With a user-level implementation of the UDP/IP protocol suite, the short message round-trip latency is 316μsecs. For comparison, the short message round-trip latency for the standard UDP/IP implementation in DEC OSF/1 V3.0 on the same hardware is 550μsecs. In other words, ADCs reduce the UDP/IP round-trip latency by 42% on our platform. We suspect that this reduction could be even more substantial if our user-level UDP/IP code were as highly tuned as the production-strength DEC OSF/1 implementation. Further results show that on a pair of 3000/600s, applications can saturate the OC-12 network link for message sizes of 16KB and above. More comprehensive performance results are presented in [9].

Summary

Recall that the OS I/O bottleneck is rooted in poor memory-access locality during I/O processing. This poor locality is primarily caused by (1) data copying, (2) inappropriate scheduling of I/O processing steps, and (3) OS kernel involvement in all I/O activity. Application device channels allow common case send and receive operations to bypass the OS kernel, eliminating one source of poor memory access locality (3). When combined with an appropriate application programming interface, ADCs allow copy-free I/O, thus eliminating data copying (1). Finally, since ADCs concentrate all processing and resources associated with I/O in the application process, it is possible to schedule I/O activities for improved memory-access locality (2).

ADCs eliminate protection domain crossings—and the resulting overheads—from I/O data paths that involve a single application process. Unfortunately, many applications require I/O data paths that intersect several user processes. For example, audio and video playback using a WWW browser typically involves the browser process, an audio/video server, and the GUI server. Additional domain crossings occur when the operating system relies on user-level servers. Fbufs can substantially reduce the cost of transferring data across multiple protection domain boundaries, and thus complement ADCs.

Related Work

Many operating systems provide some form of virtual memory (VM) support for transferring data from one domain to another. For example, DASH [25] supports *page remapping*, while Mach supports *copy-on-write* (COW) [1]. Container Shipping [20] is a Unix I/O system that uses page remapping. As mentioned previously, we found that fbufs consistently outperform our reimplementation of the DASH page-remapping facility, and also Mach's native copy-on-write facility on the DecStation 5000/200.

ANOTHER approach is to statically share virtual memory among two or more domains, and to use this memory to transfer data. 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 read/write 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 [7, 15]. This approach works when data is accessed only in a single application domain. A substantial amount of memory may be required in the network adaptor 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.

A number of specialized network interfaces exist that support user-level network access, for example SHRIMP [3], Memory Channel [12], Hamlyn [4], Telegraphos [19], and MAGIC [13]. These interfaces are specialized to support a shared-memory abstraction on loosely coupled multicomputers, and they attach to dedicated networks. An ADC, on the other

hand, is a software mechanism implemented with minimal assist from a general-purpose network adaptor. As such, it can support general TCP/IP internetwork access along with highly efficient message-passing traffic.

The U-Net project has recently built a system similar to ADCs, using the commercial Fore SBA-200 ATM network adaptor [2]. In our terminology, U-Net uses main memory based ADCs, where the network data is passed directly through the ADCs (the Sun SBUS does not support DMA to arbitrary main memory locations.) The SBA-200 firmware was modified to add support for the necessary ADC multiplexing.

Recommendations for Network Adaptor Designers

We conclude this article with a recommendation for designers of general-purpose, high-performance network adaptors. Achieving a high-performance I/O system requires an integrated design of application programming interface, cross-domain data transfer, buffer management, and network interface. As demonstrated in this article, small amounts of support by the network adaptor can facilitate large performance gains in the I/O system. In particular, fbufs require that incoming packets can be classified (demultiplexed) prior to their transfer into main memory. Application device channels additionally benefit from a network adaptor's ability to multiplex among several sets of transmit/receive queues associated with open network connections.

Neither of these facilities require additional hardware support. They merely require an appropriate interface between operating system and NI, plus a small amount of processing on an already existing onboard CPU. On the other hand, it seems infeasible for network interface designers to anticipate the needs of all operating systems and new innovative OS mechanisms. Thus, our recommendation is to (1) avoid design decisions that preclude facilities such as fbufs and ADCs, and (2) allow operating systems to download OS-specific firmware into the network adaptor, so that OS designers have the flexibility to tailor and optimize the NI–OS interface. ◻

References

1. Accetta, M., Baron, R., Bolosky, W., et al. Mach: A new kernel foundation for Unix development. In *Proceedings of the USENIX Summer '86 Conference* (July 1986).
2. Bas, A., Buch, V., Vogels, W., et al. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating System Principles* (1995).
3. Blumrich, M.A., Li, K., Alpert, R., et al. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture* (Apr. 1994), pp. 142–153.
4. Buzzard, G., Jacobson, D., Marovich, S., et al. Hamlyn: A high-performance network interface with sender-based memory management. In *Proceedings of the Hot Interconnects III Symposium* (Palo Alto, CA, Aug. 1995).
5. Clark, D.D. and Tennenhouse, D.L. Architectural considerations for a new generation of protocols. In *Proceedings of the SIGCOMM '90 Symposium* (Sept. 1990), pp. 200–208.
6. Comerford, R. and Watson, G.F. Memory catches up. *IEEE Spectrum 29*, 10 (Oct. 1992), 34–57.
7. Dalton, C., Watson, G., Banks, D., et al. Afterburner. *IEEE Network 7*, 4 (July 1993), 36–43.
8. Druschel, P., Abbott, M.B., Pagels, M. Network subsystem design. *IEEE Network 7*, 4 (July 1988), 8–17.
9. Druschel, P., Davie, B.S., and Peterson, L.L. Experiences with a high-speed network adaptor: A software perspective. In *Proceedings of the SIGCOMM '94 Conference* (London, UK, Aug. 1994), pp. 2–13.
10. Druschel, P. and Peterson, L.L. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, (Dec. 1993).
11. Gigabit network testbeds. *IEEE Computer* (Sept. 1990), 77–80.
12. Gillett, R.B. Memory channel network for PCI. *IEEE Micro 16*, 2 (Feb. 1996), 12–18. Also in *Proceeding of the Hot Interconnects III Symposium (*August 1995).
13. Heinlein, J., Charachorloo, K., Dresser, S., et al. Integration of message passing and shared memory in the Stanford flash multiprocessor. In *Proceedings of the 6th Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, Oct. 1994), pp. 38–50.
14. Hutchinson, N.C. and Peterson, L.L. The x-kernel: An architecture for implementing network protocols. *IEEE Trans. Softw. Engineering 17*, 1 (Jan. 1991), 64–76.
15. Jacobson, V. Efficient protocol implementation. In *ACM SIGCOMM '90 tutorial* (Sept. 1990).
16. Jones, M.B, Leach, P.J., Draves, R.P., et al. Modular real-time resource management in the Rialto operating system. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HotOS-V)*, (Orcas Island, WA, May 1995).
17. Leffler, S.J., McKusick, M.K., Karels, M.J., et al. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, Reading, Mass., 1989.
18. Maeda, C. and Bershad, B. Protocol service decomposition for high-performance networking. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (Dec. 1993).
19. Markatos, E.P. and Katevenis, M.G. Telegraphos: High-performance networking for parallel processing on workstation clusters. In *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture* (San Jose, CA, Feb. 1996), pp. 144–153.
20. Pasquale, J., Anderson, E., and Muller, P.K. Container Shipping: Operating system support for I/O-intensive applications. *IEEE Comput. 7*, 3 (Mar. 1994), 84–93.
21. Ramakrishnan, K.K. Performance considerations in designing network interfaces. *IEEE Journal on Selected Areas in Communications 11*, 2 (Feb. 1993), 203–219.
22. Smith, J.M. and Traw, C.B.S. Giving applications access to Gb/s networking. *IEEE Network 7*, 4 (July 1993), 44–52.
23. Thadani, M.N. and Khalidi, Y.A. *An efficient zero-copy I/O framework for UNIX.* Tech. Rep. SMLI TR-95-39, Sun Microsystems Laboratories, Inc., May 1995.
24. Thekkath, C., Nguyen, T., Moy, E., et al. Implementing network protocols at user level. In *Proceedings of the SIGCOMM '93 Symposium* (Sept. 1993).
25. Tzou, S.-Y. and Anderson, D.P. The performance of message-passing using restricted virtual memory remapping. *Software—Practice and Experience 21* (Mar. 1991), 251–267.

PETER DRUSCHEL is Assistant Professor of Computer Science at Rice University; email: druschel@cs.rice.edu