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.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
SIGOPS '93/12/93/N.C., USA
© 1993 ACM 0-89791-632-8/93/0012...$1.50

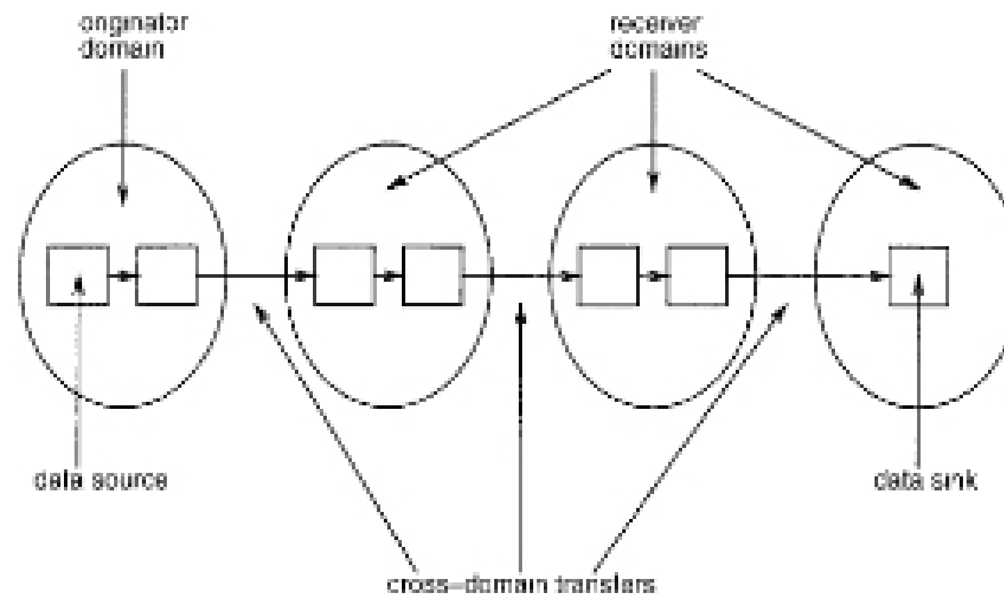

Figure 1: Layers Distributed over Multiple Protection Domains

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.