OPERATING SYSTEM SUPPORT FOR HIGH-SPEED NETWORKING

(Ph.D. Dissertation)

Peter Druschel

94-24

August 1994

Department of Computer Science
The University of Arizona
Tucson, Arizona 85721

This research was supported by National Science Foundation Grant CCR-9102040, DARPA Contract DABT63-91-C-0030, and Sun Microsystems Inc.

OPERATING SYSTEM SUPPORT FOR HIGH-SPEED NETWORKING

by

Peter Druschel

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfillment of the Requirements

For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

1 9 9 4

OPERATING SYSTEM SUPPORT FOR HIGH-SPEED NETWORKING

Peter Druschel, Ph.D.
The University of Arizona, 1994

Director: Larry L. Peterson

The advent of high-speed networks may soon increase the network bandwidth available to workstation class computers by two orders of magnitude. Combined with the dramatic increase in microprocessor speed, these technological advances make possible new kinds of applications, such as multimedia and parallel computing on networks of workstations.

At the same time, the operating system, in its role as mediator and multiplexor of computing resources, is threatening to become a bottleneck. The underlying cause is that main memory performance has not kept up with the growth of CPU and I/O speed, thus opening a bandwidth gap between CPU and main memory, while closing the old gap between main memory and I/O. Current operating systems fail to properly take into account the performance characteristics of the memory subsystem. The trend towards server-based operating systems exacerbates this problem, since a modular OS structure tends to increase pressure on the memory system.

This dissertation is concerned with 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 behavior with the performance characteristics of modern computer hardware. Then, traditional approaches to supporting I/O in operating systems are re-evaluated in light of current hardware performance tradeoffs. This re-evaluation gives rise to a set of novel techniques that eliminate the I/O bottleneck.

STATEMENT BY AUTHOR

ACKNOWLEDGMENTS

First and foremost, I wish to thank Larry Peterson. He has been the ideal advisor, one who allowed me to pursue my own directions, while providing me with the optimism, experience and insight I often lacked. I owe thanks to Norm Hutchinson for his support and the many exciting debates we had during and after his time at Arizona. Thanks also to my other committee members, Rick Schlichting and Greg Andrews, for their support and encouragement.

The entire Computer Science department at Arizona deserves credit for creating and maintaining a superb environment in which to work and be productive. Special thanks to Cara Wallace and Margaret Newman for their patience, to Jana Zeff for getting me into the program six years ago, and to Wendy Swartz for helping me get out. I will miss the many enlightening discussions and Kippy's luncheons with Mark Abbott, Lawrence Brakmo, David Mosberger, Erich Nahum, Sean O'Malley, Michael Pagels, and Charlie Turner.

I am thankful to Bruce Davie and Bellcore for making the Osiris boards available to me, and for the opportunity to engage in a fruitful and pleasant research cooperation. Special thanks to Bruce for his support and all the fun we had.

As the real brain behind this work, my cat Nene deserves special consideration. His help with GigaMetrics was indispensable.

This work was supported by the National Science Foundation under Grant CCR-9102040, by ARPA under Contract DABT63-91-C-0030, and by Sun Microsystems, Inc.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

The advent of high-speed networks may soon increase the network bandwidth available to workstation class computers by two orders of magnitude. Combined with the dramatic increase in microprocessor speed, these technological advances make possible new kinds of applications, such as multimedia and parallel computing on networks of workstations.

At the same time, the operating system, in its role as mediator and multiplexor of computing resources, is threatening to become a bottleneck. The underlying cause is that main memory performance has not kept up with the growth of CPU and I/O speed, thus opening a bandwidth gap between CPU and main memory, while closing the old gap between main memory and I/O. Current operating systems fail to properly take into account the performance characteristics of the memory subsystem. The trend towards server-based operating systems exacerbates this problem, since a modular OS structure tends to increase pressure on the memory system.

This dissertation is concerned with 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 behavior with the performance characteristics of modern computer hardware. Then, traditional approaches to supporting I/O in operating systems are re-evaluated in light of current hardware performance tradeoffs. This re-evaluation gives rise to a set of novel techniques that eliminate the I/O bottleneck.

CHAPTER 1
Introduction

Emerging network technologies such as fiber-optic transmission facilities and Asynchronous Transfer Mode (ATM) hold the promise of delivering data rates approaching 1 Gb/s between individual workstations on local and wide-area networks[1] [IEE90, C^+93b]. This order-of-magnitude increase in network capacity, combined with the explosive growth in microprocessor performance, will enable a range of innovative new applications of distributed computing. 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 the bottleneck in delivering input/output (I/O) data streams to application programs at high rates [CT90, DWB^+93, ST93, Ram93]. 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 dissertation is concerned with 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 behavior with the performance characteristics of modern computer hardware. Then, traditional approaches to supporting I/O in operating systems are re-evaluated in light of current hardware performance tradeoffs. This re-evaluation gives rise to a set of novel techniques that eliminate the I/O bottleneck.

1.1 The Memory Bandwidth Bottleneck

The root cause of the OS I/O bottleneck is that speed improvements of main memory have lagged behind those of the central processing unit (CPU) and I/O devices during the past decade [CW92, HP90]. 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[2]. The previously

[1]Throughout this work, we use the convention that 1 Gb and 1 Mb equals 10^9 and 10^6 bits, respectively, and 1 MB = 2^{20} bytes.

[2]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

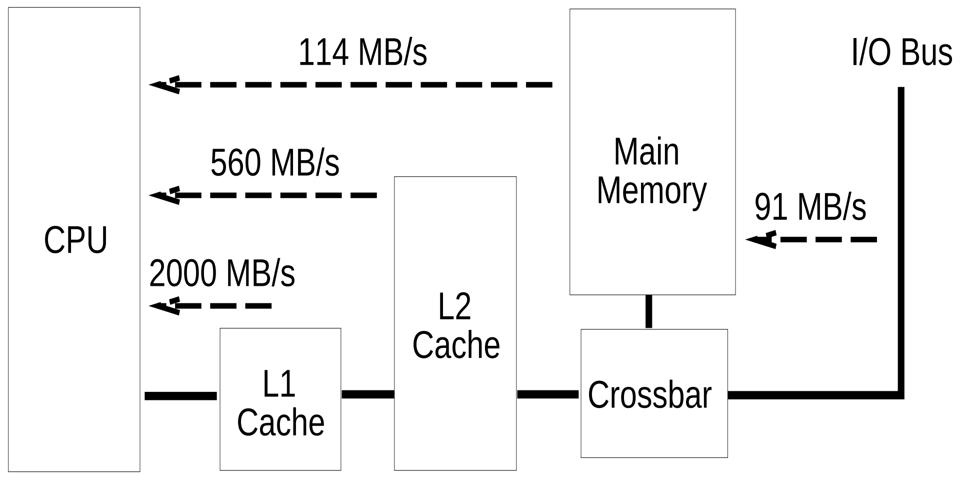

Figure 1.1: Block Diagram of a DEC 3000/600 AXP

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.

Figure 1.1 depicts the block diagram of a DEC 3000/600 AXP system. The performance characteristics of this machine are typical of currently available high-end desktop workstations. Observe that the available bandwidth on the path between CPU and main memory is less than twice the bandwidth of the I/O bus. Thus, in order to deliver the bandwidth of a high speed I/O device to an application that consumes the data, care must be taken to ensure that the data does not travel the data path between the cache and main memory more than once. Stated differently, a desirable scenario is one in which the data received from a fast input device is brought into the cache once and remains there until it is consumed. The cache system can achieve this scenario, as long as the set of CPU accesses to the data have sufficient locality, relative to the size and associativity of the cache.

Unfortunately, in most current systems the accesses to I/O data buffers generated by

devices.

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, i.e., throughput and latency. This poor locality is 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 causes poor access locality and thus increased memory traffic. Unnecessary 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 and distinct set of memory references. Thus, accesses to a particular I/O data unit are temporally separated, resulting in poor data access locality[3].

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 processor reallocation to 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 ineffective in avoiding the memory bottleneck in processing network I/O.

1.2 Server-Based Operating System Structure

The problem of poor memory access locality during I/O operations is aggravated by the recent trend in operating system design towards a modular structure, where different parts of the operating system reside in separate protection domains [ABB$^+$86, RAA$^+$88, Hil92]. In these systems, I/O requests and associated data may have to cross additional protection boundaries and scheduling points between application programs and I/O devices. Studies have shown that reduced memory access locality in theses systems has a negative impact on overall system performance [CB93], and can have a strong impact on networking performance [MB93b].

[3]An additional problem can occur on shared memory multiprocessors, when not all processing steps are scheduled to run on the same processor.

Thus, operating system designers are facing a dilemma. Growing concern for ease of distribution, validation, configuration, and maintenance in the face of increasingly complex operating systems strongly suggests more modular architectures, such as microkernel-based (server based) systems. On the other hand, server-based operating systems appear to make it even more difficult to achieve high-performance I/O than the traditional, monolithically structured systems.

1.3 Thesis Statement and Contributions

The hypothesis of this dissertation is that the I/O bottleneck in current operating systems is caused by the lack of a coordinated design of OS software and device adaptor hardware that minimizes main memory traffic. This coordinated design must encompass the entire I/O data path, from application programming interfaces (API), to OS architecture, OS facilities for buffer management and data transfer across protection boundaries, to I/O device adaptors.

The goal of this dissertation is to remove the I/O bottleneck, without sacrificing modularity in the structure of operating system and applications. Towards this goal, the contributions here are a set of novel techniques that are part of a coordinated design to minimize main memory traffic:

- A new structuring approach for operating systems that allows fine-grained modularity without the performance penalty of an equally fine-grained protection structure;
- a new OS facility for the management and transfer of I/O data buffers across protection domain boundaries;
- a new OS facility that allows applications to bypass the OS kernel in common I/O operations, while leaving control over the device in the hands of the operating system; and
- a set of techniques that reduce host processing overhead and achieve high performance in driving a high-speed network adaptor.

These new techniques result from a re-evaluation of traditional approaches to supporting I/O in operating systems. In particular, they are based on the following key observations.

Memory bandwidth is a scarce resource. The traditional approach to performance-oriented system design is to focus on minimizing the number of CPU instructions executed during an operation. The characteristics of modern computer systems with their fast CPUs, limited memory bandwidth, and fast I/O devices make it necessary to focus instead on minimizing main memory traffic.

Communication is a common operation. Current operating system designs require OS kernel involvement in each I/O operation. This approach is well suited for applications that spend most of their time operating on data stored in main memory, and that perform occasional, coarse-grained I/O operations. The kinds of distributed applications that are enabled by high-speed networking are likely to perform frequent, fine-grained I/O, making communication a common operation. Thus, operating systems should be optimized accordingly and should implement common I/O operations without requiring kernel involvement.

This dissertation focuses on the primary problem of preserving the bandwidth and latency of high-speed networks at the application level. The advent of high-speed networking and the applications that it enables also pose a number of secondary problems. For example, live audio and video applications require that the operating system schedules resources in such a way that the real-time constraints of continuous media are met. This issue is not addressed here, but is receiving attention by a host of other researchers [AH91, JH93, Jef92, RR93, LKG92].

1.4 Dissertation Overview

Chapter 2 reviews the relevant performance characteristics of modern computer systems (i.e., memory and cache behavior) in more detail. It describes the various causes of excess memory traffic in current systems and discusses techniques found in the literature to avoid these inefficiencies. We show that theses techniques, when applied naively, can fail to reduce memory traffic significantly. Instead, it is necessary to coordinate their application as part of an end-to-end design. The chapter concludes by proposing an abstract data type that encapsulates I/O data buffers along the entire data path through a host system. This abstract data type can serve as the centerpiece of a coordinated design for an efficient I/O data path.

We begin Chapter 3 by describing the experimental hardware and software platforms used in the research that led to this dissertation. Then, we report on a number of low-level performance problems that were encountered in driving a very high speed network adaptor connected to a modern workstation, and we present novel techniques to deal with the problems. The chapter concludes with a detailed performance study of network performance achieved between pairs of workstations connected by an experimental 622 Mb/s fiber-optic network.

Chapter 4 presents a new approach to structuring an operating systems that affords fine-grained modularity without the performance cost of an equally fine-grained protection structure. In particular, this approach avoids the tradeoff between a modular OS architecture and high-performance I/O, which governs conventional server-based operating systems. The implementation of a prototype kernel based on this approach is described, and a performance study is presented.

In Chapter 5 we present a novel OS facility, called *fast buffers* (fbufs), for the management and transfer of I/O data buffers across protection domains boundaries, both in monolithic and server-based operating systems. The design and implementation of this facility is described in detail. A performance study shows that fbufs allow the transfer of large network messages across multiple protection domain boundaries without a loss in throughput.

Chapter 6 introduces 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 in the hands of the operating system, thus allowing transparent sharing of the device among multiple, non-privileged application programs.

Finally, Chapter 7 summarizes the contributions, points out limitations, and identifies future directions of research.

CHAPTER 2
The Operating System I/O Bottleneck

This chapter presents an overview of the hardware and software issues relevant to I/O performance in operating systems. We show that the memory bandwidth of modern, general-purpose computer systems is within the same order of magnitude as the bandwidth of emerging high-speed networks. Furthermore, we present evidence that this situation will persist for the foreseeable future, because cost considerations prevent the use of dramatically faster main memory and interconnect technology in this class of machine. As a consequence, operating systems must be designed to minimize main memory traffic that results from network and other I/O activity. Specifically, the number of trips that network data takes across the CPU–memory data path must be minimized in order to avoid a bottleneck.

We proceed to survey several techniques from the literature that can be applied to this problem. An important lesson we learn from this survey is that naively applying these techniques is not sufficient for achieving good application-to-application throughput. Instead, it is necessary to integrate the entire data path, from a source device, through the operating system, application programs, and possibly to a sink device. The chapter concludes by outlining a fully integrated data path that optimizes end-to-end throughput.

Throughout this chapter, we include in our analysis the most general case of a micro-kernel-based system, where device drivers, network protocols, and application software all potentially reside in different protection domains. This consideration is well justified in light of the advantages of server-based systems (e.g., configurability, distributability, portability, and so on), as well as the recent trend towards such systems in commercial offerings.

2.1 Memory Performance

This section analyzes the performance of main memory in modern, general-purpose computer systems. We present measurements of main memory bandwidth in several commercially available desktop workstations, and discuss the constraints for designer of main memory subsystems. The conclusion is that main memory bandwidth is likely to remain within the same order of magnitude as I/O bandwidth for the foreseeable future.

Table 2.1 shows, for each of four desktop workstations, the results of several memory bandwidth measurements described below. The CPU/Memory bandwidth numbers reported were obtained using a simple benchmark that measures the sustained bandwidth during a series of read, write, and copy operations. The benchmark was coded in C. No attempt was made to tune the program for a particular machine. The C code was compiled in each case with two different compilers; the vendor's compiler, and the GNU project's `gcc` compiler [Sta93]. All appropriate compiler optimizations were turned on.

	MB/s, sustained		
	Copy	Read	Write
IBM RS/6000 340	51	81	73
Sun SS10/41	26	42	39
HP 9000/735	31	71	52
DEC 3000/600	52	109	111

Table 2.1: CPU–Memory Bandwidth of several Workstations

The read and write columns were measured by reading (writing) each element of an array. The copy column was measured using two methods: Element-wise assignment of two arrays, and invoking the `bcopy()` library function. The benchmark used arrays of type `int` (32bits) and `double` (64bits). The numbers reported are the best that could be achieved in each case; i.e., GNU compiler versus vendor's compiler, `int` versus `double` for the read and write columns, and `int` versus `double` versus `bcopy()` for the copy column. Finally, the size and alignment of the arrays was chosen to eliminate effects of the data cache.

The main result is that the measured memory bandwidths of current workstation is on the order of 100 MB/s. Note that somewhat better results could likely be obtained using machine-specific, hand-optimized machine code, since the compiler-generated machine code, and the vendor-provided implementation of the `bcopy()` function may be less than optimal for the intended task of the benchmark program. We have deliberately not tried to eliminate this factor, since it equally affects real programs. An inspection of the generated code for some of the machines convinced us that bandwidth improvements with hand-optimized costs would not exceed 50%.

An experimental ATM high-speed network adaptor described in Chapter 3 can simultaneously transmit and receive on a pair of 622 Mb/s fiber-optic links. Adjusting for ATM cell overhead, the adaptor can generate an aggregate I/O bandwidth of 134 MB/s. Clearly, the memory bandwidth of current workstation is within the same order of magnitude as the bandwidth of this network adaptor.

Network adaptors with such high bandwidth are not yet commercially available. One could conjecture that by the time such networks are on the market, main memory bandwidth will have improved to the point where it is no longer a bottleneck. In the following, we argue that main memory bandwidth in next generation workstations is not likely to enjoy order-of-magnitude increases in bandwidth.

Cost and density make Dynamic Random Access Memory (DRAM) the technology of choice for the main memory system in general-purpose computer systems. DRAM performance is mainly characterized by its access latency, i.e., the time it takes, once an address is presented to a DRAM chip, for the corresponding data to become available. While DRAM density has improved dramatically in recent years, access latency has improved at a rate of only 7% per year [HP90].

Data is transferred between cache and main memory using fixed sized cache blocks. The main memory bandwidth available to the CPU is bounded by the ratio of the cache block size and the time it takes to read or write a cache block. This time consists of the memory access latency, plus the transfer time for a cache block. No dramatic improvements in memory access latency can be expected in the near future, due to the slow speed improvements of DRAMs. Techniques exist to increase the transfer rate, but an increase in transfer rate alone cannot deliver major increases in memory bandwidth because latency would increasingly dominate the transfer time for a fixed sized cache block. Increases in data transfer rate must be combined with an increase in cache block size to be effective. Unfortunately, the cache block size cannot be increased arbitrarily without affecting the hit rate of the cache systems [Prz90].

Several recently announced components integrate some form of a cache with a dynamic RAM to reduce the average access latency [CW92]. These integrated second level caches use large cache lines and are connected to the DRAM using wide data paths. As for any cache, the hit rate of these components depends on locality of reference. As we shall discuss in Section 2.2, accesses to I/O data exhibit poor locality. Furthermore, it is unclear whether the industry will accept the higher cost of these components.

While substantial improvements in main memory bandwidth can be expected in general-purpose computer systems, it is unlikely that these improvements will out-pace the increases in I/O bandwidth by an order of magnitude. Thus, we conclude that main memory bandwidth in this class of machines will remain within the same order of magnitude as network bandwidth for the foreseeable future.

2.2 Effectiveness of Cache Memories

Modern computer systems employ caches to bridge the gap between CPU and main memory speeds. The idea is to keep a recently accessed portion of the main memory's data in a fast memory. Whenever a data item referenced by the CPU is found in the cache, no main memory access is necessary and the operation can complete at the speed of the cache memory. A cache increases system performance by reducing the average access latency for data and instructions. It also reduces contention for main memory access between processor(s) and DMA-based I/O devices. The effectiveness of a cache depends on the size and organization of the cache, and on locality in the CPU's memory accesses.

In general, an application program that processes I/O data causes the CPU to load from—and possibly store to—memory every word of a data unit, potentially multiple times. (The next section identifies reasons why these data accesses might occur.) This section argues that data caches are not effective in eliminating the corresponding main memory accesses. For the sake of this argument, we limit our attention to accesses of the actual I/O data, and ignore references to CPU instructions and other data that occur during an application's processing. For emerging high-bandwidth applications such as continuous media, accesses to I/O data arguably account for a significant portion of total execution time. In any case, we have already shown that unless care is taken, the I/O data accesses alone can cause a bottleneck. Consider the following factors, ordered roughly according to importance.

Processor Scheduling: CPU scheduling may cause the execution of other programs to be interleaved with the processing of a data unit. By the time processing resumes, cached portions of the data unit have most likely been replaced. On a multiprocessor, the processing may even be resumed by a different CPU with its own data cache. There are a number of situations where scheduling occurs during the processing of a data unit. When the data unit is passed on to another thread (i.e., queued), a processor must be scheduled to run that thread. Queuing typically occurs at the user/kernel boundary, in certain protocols, and between the device driver's interrupt handler and the top half of the driver. In the worst case, additional queuing may occur between protocol layers. Moreover, hardware interrupts and the events they signal trigger processor rescheduling.

Cache Size: Cache memories, particularly fast on-chip caches, are limited in size. In practice, their effective size is further reduced due to the limited associativity of direct-mapped and set-associative organizations [Prz90]. For data to remain cached during a data operation that involves loading every word and storing it in a different location, the cache must be at least twice the size of the data unit. In practice, cache size requirements are further increased by accesses to program variables during and between data manipulations.

Cache Organization: Caches that are virtually indexed and tagged do not require a virtual-to-physical address translation to access cached data. With this approach, cached data from virtually shared pages cannot remain valid across protection domain boundaries. In other words, data must be re-loaded after a switch into a different protection domain, even if the data buffers are shared between the domains. Physically tagged caches do not have this problem. They require, however, that a Translation Lookaside Buffer (TLB) entry be active for the page that contains the referenced data. Network data buffers tend to be scattered across memory pages, which increases the demand for TLB entries. The resulting contention for TLB entries can add substantial costs to the access of cached data [PDP93].

Cache Write Policy: Uniprocessor data caches often employ a write-through policy, meaning that every store operation requires a write to main memory. Write buffers are typically used with write-through caches to decouple the CPU from the timing of memory writes. However, many consecutive writes—such as would occur when reading and writing every word of a data unit—can cause the CPU to stall on store instructions.

To quantify the effectiveness of the data cache in handling network data, Pagels [PDP93] conducted experiments that measure the amount of network data resident in the cache after OS processing. This cache residency value is a measure of the potential benefit obtained by a user process due to the caching of network data, thereby avoiding CPU/memory transfers. It provides an upper bound on the effectiveness of the cache when network data is processed without copying. Both monolithic and microkernel-based operating systems are included in this study. The results obtained in these measurements confirm that caches are not effective in eliminating main memory traffic associated with the accesses of network data.

In summary, we have shown that the main memory bandwidth in modern, general-purpose computer systems is within the same order of magnitude as the bandwidth of emerging high-speed I/O devices such as ATM network adaptors. In order to sustain the bandwidth of these I/O devices, it is critical to minimize the main memory traffic caused by the processing of I/O data. Furthermore, the cache systems used in modern computers to bridge the gap between CPU and memory speed are not effective in eliminating main memory traffic caused by accesses to network data.

2.3 Avoiding Main Memory Traffic

The two previous sections document our argument that the CPU–memory bandwidth of workstations is within the same order of magnitude as the network bandwidth, and that data caches cannot eliminate main memory traffic caused by the processing of network I/O. It follows, therefore, that in order to preserve the bandwidth on the data path from a network device through the OS and application, and possibly to a sink device (e.g., display), multiple accesses of the data stored in main memory must be avoided.

Each of the following subsections identifies a potential main memory access along this path, briefly describes one or more technique for efficiently handling or avoiding it, and discusses the assumptions and limitations of these techniques.

2.3.1 Device–Memory Transfers

Data must be moved between main memory and network/device adaptors. The techniques most commonly used are Direct Memory Access (DMA) and Programmed Input/Output (PIO). DMA allows an I/O adaptor to transfer data directly from/to main memory, without involving the CPU. PIO requires the processor to transfer individual data words (typically 32 bits) between main memory and I/O adaptor in a programmed loop.

With DMA, it is generally possible to transfer large blocks of data in a single bus transaction, thereby achieving transfer rates close to the limits of main memory and I/O bus. The data transfer can proceed concurrently with activity by the processor(s), although contention for main memory access may induce processor stalls during periods of heavy DMA traffic. On the downside, DMA requires some complexity in the device adaptors. In many machines, data caches and main memory may not be coherent with respect to DMA transfers. That is, after a DMA transfer from an I/O adaptor to main memory (DMA write), the cache may contain stale data. Consequently, the appropriate portions of the data cache must be invalidated by system software. Conversely, a write-back cache may have to be explicitly flushed prior to a a DMA read operation.

With PIO, the CPU is occupied during transfers from/to a device. Only a small fraction of the peak I/O bandwidth is often achieved, for the following reasons. The adaptor's control and data ports can either be mapped as cacheable or non-cacheable memory locations. In the non-cacheable case, each load/store instruction results in a short (i.e., one machine word) I/O bus transaction, resulting in poor sustained bandwidth. In the cacheable case, I/O bus transfers occur at cache line length. The resulting bandwidth is improved, but may still be far below the peak I/O bandwidth. Moreover, it is again necessary to flush the data cache to maintain consistency with the adaptor's ports.

Nevertheless, there are situations where PIO can be preferable over DMA. First, computations on the data that occur in the kernel, such as checksum calculations, can sometimes be integrated with the PIO data movement, saving a main memory access. Second, after a programmed data movement from an I/O adaptor to main memory, the data is in the cache. This can result in reduced memory traffic if the data is accessed again while it remains in the cache.

A scatter-gather capability in DMA-based I/O adaptors is important for reducing memory traffic. Scattering allows an incoming data unit to be placed into multiple, non-contiguous portions of main memory. Gathering allows an outgoing data unit to be collected from multiple, non-contiguous buffers. Scatter-gather allows DMA transfer from/to non-contiguous physical page frames, which greatly simplifies physical memory management in the operating system, and helps avoid copying data into contiguous storage. With PIO, scatter-gather can be trivially implemented in software.

Network adaptors may support packet demultiplexing prior to data transfer to main memory, allowing filtering and selective placement of data units in memory. In the simplest case, the adaptor allows the host to peek at a network packet's header. The host makes the demultiplexing decision and initiates the data transfer to the appropriate location in main memory, using DMA or PIO. More elaborate adaptors such as the ones described in Chapter 3 and in [KC88, BPP92] can be programmed by the host CPU to recognize network packets by their headers, and place them into appropriate memory locations using DMA without host intervention. This feature proves to be very useful in enabling OS facilities that can reduce main memory traffic (see Chapters 5 and 6).

2.3.2 Cross-Domain Data Transfers

Protection in operating systems necessitates the transfer of data across protection domain (address space) boundaries. In the simplest case, an I/O data stream is used by a single application process running on top of a conventional monolithic operating system kernel. Here, I/O data must cross a single user/kernel boundary. In general, additional user processes, such as window managers and multimedia servers, and the OS servers of microkernel-based operating systems, may introduce additional domain boundary crossings into the I/O data path.

Software data copying as a means of transferring data across domain boundaries exacerbates the memory bottleneck problem. A number of techniques exist that rely on the virtual memory system to provide copy-free cross-domain data transfer. *Virtual page remapping* [Che88, TA91] unmaps the pages containing data units from the sending domain and maps them into the receiving domain. With *Virtual copying* (copy on write) [FR86], the sending and receiving domain share a single copy of the transferred pages, and physical copying occurs only when one of the domains attempts to modify the shared data unit. *Shared virtual memory* [SB90] employs buffers that are statically shared among two or more domains to avoid data transfers.

Virtual 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. Virtual copying 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 low latency. 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. Measurements we performed on a DecStation 5000/200 suggest that page remapping is not fast enough to sustain the bandwidth of a high-speed network adaptor (see Section 5.3.1).

Another complication arises from the fact that both techniques work at the granularity of the VM page size. A mismatch between data unit size and VM page size implies that a portion of the last page overlapped by the data unit will remain unused. For reasons of data privacy, the kernel must clear (e.g., fill with zeroes) this portion of a newly allocated buffer page, which can incur a substantial cost, relative to the overall overhead of buffer management and cross-domain data transfer. A similar problem occurs if headers at the front of a received network packet contain sensitive data that must be hidden from user processes. The partial use of memory pages also requires more pages per amount of network data, resulting in increased physical memory consumption, page allocation and remap overhead, and demand for TLB entries.

Shared virtual memory avoids data transfer and its associated costs altogether. However, its use may compromise protection and security between the sharing protection domains. Since sharing is static—a particular page is always accessible in the same set of domains—a priori knowledge of all the recipients of a data unit is required. A novel technique, called *fbufs* (introduced in Chapter 5), combines page remapping with shared virtual memory, and exploits locality in network traffic to overcome some of the shortcomings of either technique.

2.3.3 Data Manipulations

Data manipulations are computations that inspect and possibly modify every word of data in a network data unit. Examples include encryption, compression, error detection/correction, presentation conversion, and application-specific computations. Data manipulations can be performed in hardware or software. Hardware support for data manipulations can reduce CPU load, and when properly integrated, reduce memory traffic. For certain manipulations like video (de)compression, hardware support may be necessary in the short term, due to the computational complexity of the task. To rely on hardware for all data manipulations, however, seems too constraining for innovative high-bandwidth applications.

Software data manipulations typically access the data independently from each other since they are generally part of distinct program modules that reside in separate protection domains (OS kernel, servers, application). The resulting memory references have poor locality, and thus require that data be loaded from (and possibly stored back to) main memory. The slow main memory accesses contribute significantly to the time required to complete the data manipulation, and thus limit the bandwidth.

Memory references resulting from data manipulations can be minimized through *Integrated Layer Processing* (ILP) [CT90, AP93]. ILP is a technique for implementing communication software that avoids repeated memory reference when several data manipulations are performed. The data manipulation steps from different protocol implementations are combined into a pipeline. A word of data is loaded into a register, then manipulated by multiple data manipulation stages while it remains in a register, then finally stored—all before the next word of data is processed. In this way, a combined series of data manipulations only references memory once, instead of potentially accessing memory once per distinct layer.

A detailed performance study demonstrates that integration can have a significant impact on performance [AP93]. On the other hand, a major limitation of ILP is that data manipulations performed by program modules in different protection domains cannot be

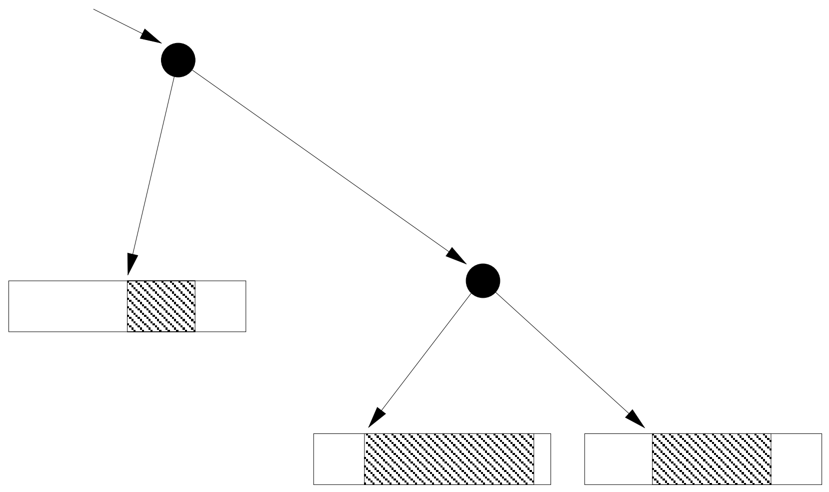

Figure 2.1: Implementation of Messages in the x-kernel

easily integrated.

2.3.4 Buffer Management

Buffer editing—which we distinguish from data manipulations that require the inspection and/or modification of each word of data—can be expressed as a combination of operations to *create*, *share*, *clip*, *split*, *concatenate*, and *destroy* buffers. When naively implemented, these operations may require physical copying of the buffers.

A buffer manager that employs lazy evaluation of buffers to implement the aforementioned primitives can facilitate copy-free buffer editing. The manager provides an abstract data type that represents the abstraction of a single, contiguous buffer. An instance of this abstract buffer type might be stored in memory as a sequence of not necessarily contiguous fragments. For example, the x-kernel [HP91] represents messages by a directed acyclic graph (DAG), where the leaves correspond to buffers, and the leftmost leaf can have information (e.g., headers) prepended to the front. Figure 2.1 depicts x-kernel messages.

Since buffer editing occurs frequently in network protocol implementations, buffer managers are used in the network subsystem of many operating systems. The scope of these managers is restricted to a single protection domain, typically the kernel. In most systems, a software copy into a contiguous buffer is necessary when a data unit crosses a protection domain boundary.

2.3.5 Application Programming Interface

The application programming interface (API) defines the services and operations that an operating system provides to application programs. In particular, it defines the syntax and semantics of the operations (system calls) exported by the OS. I/O data buffers appear in this definition as arguments to I/O operations. The argument passing semantics for the I/O operations defined by this interface can have a significant impact on the efficiency of I/O data transfers between OS kernel and applications.

Consider, for example, the UNIX `read()` and `write()` system calls [UNI89]. These operations specify a pointer argument that refers to the address of a contiguous buffer in the application's address space, and an integer argument that represents the size of the buffer. Application programs may choose an address anywhere within their populated address space, with no restrictions on the size and alignment of the buffer. The semantics of these operations specify that data is *copied* from and to the application buffer during the operation. That is, during a read system call, the application buffer is overwritten with the input data. After a write operation completes, it is assumed that output data has been copied from the application buffer, and the application is free to reuse (modify) the buffer.

The low-level representation of data buffers and the semantics of the read and write operations make it difficult for an implementation to avoid physical copying of data. First, all virtual memory (VM) based techniques for cross-domain data transfer operate at the granularity of a VM page. If the user buffer's first and last addresses are not page aligned, the system must copy portions of the first and last page overlapped by the buffer. Depending on the length of the user buffer, the portion of a buffer that must be copied can be significant. Second, the semantics of the write system call permit the user process to modify (reuse) the user buffer immediately after the call returns. If this happens, the system must either copy the affected page after all, or block the user process until the operating system is done with the output data. The latter approach may degrade the application's effective I/O bandwidth even though it avoids copying, because it prevents overlapping of I/O and application processing. Third, the read and write system calls specify a single, contiguous data buffer. Data that arrives from a network device is often scattered in main memory, due to the reassembly of fragmented network packets. If a read operation specifies an amount of data that spans several fragments, copying of data is unavoidable.

In summary, there are three problems with the UNIX read and write system calls, as they relate to avoiding data copying. They allow data buffers with arbitrary alignment and length, they require contiguous data buffers, and they have copy semantics. A proposal for an API that lends itself to efficient data transfer is presented in Section 2.5.

2.4 End-to-End Design

This section gives an overview of the design space for an I/O subsystem that minimizes main memory traffic along the entire data path from source to sink device. We focus on several representative sample points in the design space. For each sample point, we discuss tradeoffs, determine the optimal data path, and select appropriate implementation techniques to achieve the optimal data path.

Throughout this section, we refer to the block diagrams in Figure 2.2 to depict the data flow from source to sink device. In the architectural model underlying these diagrams, CPU and cache are connected to main memory using a dedicated memory bus. A bus converter ties the memory bus to a separate I/O bus, to which all devices are attached.

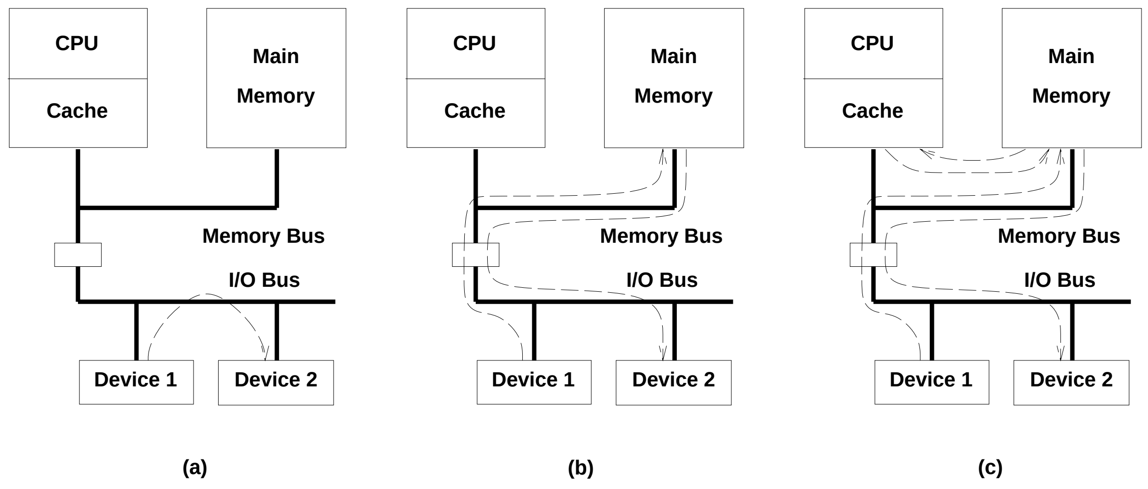

Figure 2.2: Various Forms of Data Streaming

2.4.1 Hardware Streaming

One approach to avoiding the main memory bottleneck is to remove both the CPU and main memory from the data path. The data path is set up and controlled by software, but the data itself is transferred directly from source to sink device, bypassing main memory (Figure 2.2, (a)). Both device adaptors and the I/O bus must support peer-to-peer transfers, and the adaptors must be able to perform demultiplexing and any necessary data format conversions.

This method can be characterized by a lack of integration with the host computer system. Since application programs have no access to the data, they are constrained by the functionality provided by the adaptor boards. The result is a tradeoff between complexity in the device adaptors on one hand, and flexibility and functionality on the other.

Adaptors that support a fixed set of capabilities offer little room for innovation to applications. An alternative is for the adaptor to provide components that can be programmed by applications. This can take the form of special-purpose processors, or programmable circuitry. Unfortunately, this approach has problems of its own. First, it is difficult for applications to use programmable device adaptors in a portable way. Second, at any given point in time, the technology used in I/O adaptor hardware tends to lag behind that of the main computer system, due to economics. Consequently, applications that rely on out-board processing may not be able to exploit performance gains resulting from an upgrade of the host computer system. An exception are computations whose performance is bound by the characteristics of the device, e.g., network link speed. Performing such computations in the adaptor seems appropriate, since they cannot take advantage of increased host processing speed [BPP92].

With hardware streaming, no specific techniques are required for data transfer in the I/O subsystem. However, many of the issues discussed in this chapter may apply to the *internal* design of the adaptor boards.

2.4.2 DMA–DMA Streaming

A second approach is for data to be transferred using DMA from a source device to buffers in main memory, where some buffer editing may occur under control of the operating system, and then, the data is transferred to a sink device using DMA (Figure 2.2, (b)). The CPU controls data transfers, may change the size of the data units and control information (headers), but remains removed from the data path. Unlike the previous approach, DMA–DMA streaming requires no special hardware other than DMA support on both the source and sink device adaptors. Consequently, generic devices (e.g., disks) can be used.

Two DMA I/O bus trips are required by this approach. It follows that the throughput is bounded by one half of the I/O bus bandwidth. In practice, the sustainable throughput is lower, since main memory accesses caused by concurrent CPU activity compete with the DMA transfers for main memory access, even if the I/O bus and memory bus are separated.

Two key techniques are required to keep the CPU removed from the data path in this approach: scatter-gather DMA support in the device adaptors, and an OS buffer manager that supports copy-free buffer editing. Support for gather DMA in the sink device adaptor is critical. Recall that a lazily evaluating buffer manager may cause a buffer to consist of multiple discontiguous fragments. In the absence of a gather capability, it is necessary to copy data units into contiguous space prior to a transfer to the sink device.

2.4.3 OS Kernel Streaming

Now consider systems where software data manipulations are performed, but all the manipulations are executed within the privileged kernel protection domain; no user-level programs are involved. Clearly, the data must pass through the CPU/data cache (Figure 2.2, (c)), implying that the achievable throughput is bounded by the system's copy bandwidth. The goal is to keep the resulting CPU–memory data traffic to the minimum, namely two transfers. The solution is to integrate data manipulations—if there is more than one—using, for example, ILP. Note that data manipulations may include data movement from and to devices via PIO.

If a device supports both DMA and PIO, it may be beneficial to use PIO in this type of system, since programmed data movement can be integrated with other data manipulations. That is, instead of first using DMA and then loading/storing data to/from main memory, the CPU could directly load data from the source I/O adaptor, and store data to the sink I/O adaptor, bypassing main memory. This approach saves two DMA bus trips, which would otherwise compete with the CPU for memory access. However, it also trades memory accesses for potentially much slower load and store operations across the I/O bus. Which approach results in a more efficient data path depends on the relative performance of memory accesses and DMA transfers, versus PIO on the target hardware.

Unlike the previous methods, OS kernel streaming offers full programmability of the data path. However, all data manipulations must be performed in the kernel of the operating system. Applications and user-level servers are restricted to the set of data manipulations implemented by the kernel.

2.4.4 User-Level Streaming

Finally, consider systems where data passes through the kernel, plus one or more user-level protection domains. These user-level domains could be part of a microkernel-based operating system, implement third-party servers for windowing, or be part of an application. In addition to the issues discussed in the previous subsections, the I/O subsystem designer is faced with protection domain boundaries. Protection boundaries are an obstacle to the integration of data manipulations. We conclude, therefore, that designers should make every effort to locate all data manipulation functions in the same domain. Protection also requires an efficient method for transferring data between domains.

Section 2.3.2 discussed several VM techniques for cross-domain data transfer. The use of such a facility is critical for data transfer between two user-level protection domains. Otherwise, two software copies are required for a user-to-user transfers on most systems—from the source user domain to a kernel buffer and from the kernel buffer to the target user domain. Chapter 5 introduces a novel facility for cross-domain data transfer that performs better than existing techniques, and achieves an efficient end-to-end data path, both in monolithic and server-based operating systems.

As a special case, it is possible to transfer data between a user domain and the operating system kernel without extra cost if the transfer is combined with the data movement between I/O adaptor and main memory. That is, data is transferred directly between the I/O adaptor and the user buffer [Jac90, DWB+93]. However, with this approach, a potentially huge amount of buffer memory is required in a high-speed network adaptor. On the receiving end, the data be buffered in the adaptor until the receiving application asks for the data. On the transmitting side, transmitted data must be buffered in the network adaptor for potential retransmission, since no intermediate copy of the data exists in main memory. Due to the large bandwidth-delay product of high-speed networks, large amounts of data must be buffered for retransmission. Finally, this approach does not generalize to I/O data streams that are accessed in more than one user-level protection domain.

2.4.5 Need for Integration

The API, cross-domain data transfer facility, and buffer manager must be integrated in a manner that takes into account their subtle interactions. Consider, for example, a system where the buffer manager is restricted to the kernel domain, a virtual copy facility is used for cross-domain data transfer, and the OS supports a UNIX-style API.

In this case, data units from the source device are placed in main memory buffers, and some buffer editing occurs as part of the in-kernel I/O processing; e.g., reassembly of network packets. When a data unit represented by a lazily evaluated buffer reaches the user/kernel boundary, it must be evaluated (copied into contiguous storage), despite the use of a virtual copy facility. The reason is that the interface defines data buffers to be contiguous. Since the API allows applications to specify an arbitrarily aligned buffer address and length, the buffer's first and last address may not be aligned with page boundaries. Consequently, the data transfer facility may be forced to copy the portion of the first and last page that is overlapped by the buffer.

Once in the application domain, more buffer editing may need to be performed. Since the buffer management tool is not available at user-level, the application must either perform data copying, or perform its own lazily evaluating buffer management. In the latter case, another copy is required when data crosses the next domain boundary. After data is transferred back to the kernel, the semantics of the API allow the user program to reuse the data buffer instantly, which is likely to force the virtual copy facility to copy parts of the buffer. We conclude that despite the use of copy-avoiding techniques, multiple copies occur along the data path and these copies are an artifact of the poor integration of these techniques.

One problem is that the implementation of the buffer manager is local to the kernel domain; a global implementation is necessary to maintain the lazily evaluated representation of data buffers along the entire data path. A global abstract buffer type has the additional benefit that all domains (including applications) can perform copy-free buffer editing. A second problem is the API, which commonly does not permit a non-contiguous representation of buffers, and as a consequence, stands in the way of efficient data transfer. A potential third problem is the cross-domain data transfer facility's inability to efficiently support the transfer of non-contiguous buffer fragments.

Figure 2.3 depicts some of the dependencies among the copy-avoiding techniques. The boxes represent situations where copies may occur along the data path, and the bubbles correspond to the appropriate copy-avoiding techniques. Edges indicate dependencies among the techniques that must be satisfied to achieve an efficient data path. For example, VM techniques for data transfer, the API's data passing semantics, and the buffer manager must agree on a common buffer representation. The dashed lines indicate that it is unclear how to effectively integrate ILP with API and data transfer facility, to permit integration of data manipulations across protection domain boundaries.

In conclusion, in the interest of minimizing main memory traffic, it is necessary to maintain a lazily evaluated representation of data buffers along the entire data path. This implies that all programs must deal with this representation of buffers. Consequently, a global buffer manager is needed that is integrated with the API and cross-domain transfer facility. The choice of a cross-domain data transfer method may further influence the design of a network adaptor. For example, shared virtual memory requires demultiplexing prior to the data transfer from adaptor to main memory.

2.5 A Buffer Abstraction for an Integrated Data Path

This section proposes an API that permits the integration of the API, buffer manager, and cross-domain data transfer facility to achieve an efficient end-to-end data path. The approach hinges on an abstract data type for data buffers called *IOData*. This section gives an overview of the IOData type design, discusses its use, and briefly sketches how it can be implemented using different buffer management schemes and cross-domain data transfer facilities.

In Section 2.3.5 we argued that the UNIX API has three problems with regard to the efficient transfer of I/O data. Its read and write system calls allow applications to specify data buffers with arbitrary alignment and length, they require contiguous data buffers, and they have copy semantics. These problems can be solved by adding a set of new

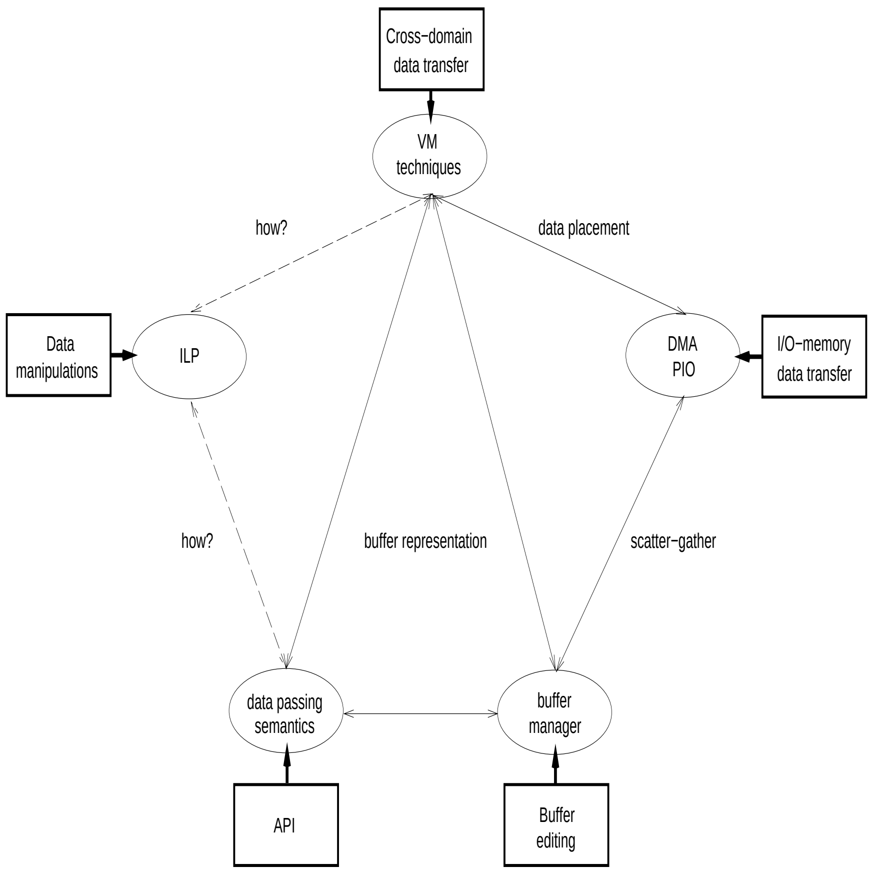

Figure 2.3: Dependencies among copy-avoiding techniques

I/O operations to the API. These I/O operations take as arguments an *abstract data type* (ADT) that represents data buffers (instead of pointers), and these arguments are passed with *handoff* rather than copy semantics. The standard read and write systems calls can be maintained for backward compatibility, and the new operations can be used by new applications that require high-performance I/O.

2.5.1 The IOData Abstract Data Type

An instance of the IOData abstract data type represents a data buffer of arbitrary length. It encapsulates one or more physical buffers that contain the data. At any given time, the physical buffers may not be contiguous, mapped in the current domain, or even in main memory. The IOData type is *immutable*, i.e., once an instance is created with an initial data content, the content cannot be subsequently changed. IOData instances can be manipulated using a well-defined set of operations. An implementation of this abstract type—i.e., code that implements its operations—is included as part of a library in each protection domain. The exact form and syntax of IOData's operations depends on the programming language used, which may vary from domain to domain.

The use of the IOData type for data buffers has important advantages. First, it ensures that a single buffer representation can be maintained along the entire data path, permitting lazy buffer evaluation. Second, it isolates applications, user-level servers, and large parts of the kernel from details of buffer management and cross-domain data transfer. This increases portability of both applications and operating system, and permits the use of the most efficient buffer management and data transfer techniques on different platforms. Third, the IOData type gives applications access to efficient buffer manipulation operations, and eliminates the need for separate application-level buffer management.

The IOData type supports the following operations. An *allocate* operation creates a new instance of the requested size and allocates an appropriate number of physical buffers. During an initialization phase, the client is provided with a list of pointers to these physical buffers, for the purpose of initialization. A *share* operation creates a logical copy of an IOData instance; it does not actually copy the physical buffers. *Clip*, *split*, and *concatenate* operations implement the necessary buffer editing operations. A *retrieve* operation generates a list of pointers to the physical data buffer fragments, thereby allowing the client to access the data. A *mutate* operation is a combination of retrieve and allocate. It allows a client to read the data from an IOData instance, and store the perhaps modified data into a new IOData instance. The operation generates a list of pointer pairs, one referring to a fragment of the source, the other pointing to a physical buffer of the target. Finally, a *deallocate* operation destroys an IOData instance, and deallocates the physical buffers if no logical copies of the data remain.

2.5.2 Implementation

Consider an implementation of the IOData type. One key feature is that the implementation has complete control over the size, location, and alignment of physical buffers. Consequently, a variety of buffer management schemes are feasible. All buffers may be part of a system-wide pool, allocated autonomously by each domain, located in a shared VM region, or they may reside outside of main memory in an I/O adaptor. Physical buffers can be of a fixed size to simplify and speed allocation. The other key feature of the IOData type is its immutability. It allows the transparent use of page remapping, shared virtual memory, and other VM techniques for the cross-domain transfer of IOData instances. Virtual copying can be used with increased efficiency since physical buffers are guaranteed not to be written after a transfer. The implementation of the IOData abstract data type can also be based on fbufs, the I/O buffer management and cross-domain data transfer facility described in Chapter 5.

It is possible to extend an existing API (such as that of UNIX) to include input/output operations based on the IOData type. New applications that depend on high bandwidth (such as multimedia) can use the new interface. The conventional interface can be maintained for backward compatibility, at the cost of copying the data into contiguous storage. Applications that use the new interface, and consume/modify input data—rather than merely perform buffer editing—must be able to deal with the non-contiguous storage of IOData instances. For the large class of applications that access input data sequentially, the additional program complexity and runtime cost are negligible. Typically, the array-indexing loop used to read data from a contiguous buffer must be nested in an additional loop that iterates over a list of buffers. Applications that require contiguous storage of data for efficiency must explicitly perform the copy (or use the old interface), thereby trading the copying cost for fast random access.

2.6 Concluding Remarks

This chapter shows that the bandwidth of the CPU–memory data path in modern, general-purpose computer systems is within the same order of magnitude as the bandwidth of emerging high-speed I/O devices, and that this state of affairs is likely to persist for the foreseeable future. This makes it essential to minimize main memory traffic during the processing of I/O operations, in order to avoid a bottleneck in delivering high-speed network traffic to application programs. Moreover, cache memories do not significantly reduce main memory traffic during I/O operations, due to the poor locality of the corresponding memory accesses in current operating systems.

We identify the main causes of main memory traffic during network I/O operations, namely data transfer between I/O device and main memory, data transfer across protection domain boundaries, data manipulations, buffer management, and interfaces. Techniques are presented that can help eliminate or reduce memory traffic in each case. From this discussion, we draw a number of conclusions.

First, protection domain boundaries are a major obstacle in delivering high-speed I/O to applications. Techniques for cross-domain data transfer described in the literature and in current use are not efficient enough to satisfy the needs of high-speed I/O. And, protection boundaries prevent the integration of data manipulations, which results in poor locality of access and increased main memory traffic. Lastly, the techniques for avoiding main memory traffic during I/O processing need to be carefully integrated to be effective.

The chapter concludes by proposing an abstract data type for I/O data buffers that permits an efficient and integrated design of application programming interface, buffer management, and cross-domain data transfer facility. The IOData abstract data type is complemented by the fbuf facility for buffer management and cross-domain data transfer introduced in Chapter 5. The IOData buffer abstraction implemented on top of the fbuf facility can eliminate main memory traffic caused by protection domain boundaries.

Chapters 4 and 6 introduce complementary techniques that can *remove* protection domain boundaries from the I/O data path. Removing protection boundaries has the additional benefit of reducing I/O latency, and of allowing the integration of data manipulations, which results in better memory access locality.

CHAPTER 3
OS Support for a High-Speed Network Adaptor

This chapter describes our experiences in providing low-level operating system support for a high-speed network adaptor. We begin by describing the experimental hardware and software platform used throughout this work. Then, we describe a number of problems that were encountered while attempting to integrate a high-speed network device efficiently into a standard operating system environment, and we present novel techniques to solve these problems. Next, we highlight certain features provided by the OSIRIS adaptor that we were able to exploit in the OS; features that facilitated new techniques (covered in detail in later chapters) for making the OS more effective in delivering network data to application programs. We close with a detailed study of the network performance achieved between a pair of commercial workstations connected by an experimental high-speed network link.

Note that this chapter does not address the problem of delivering high-speed networking services to *application* programs. Instead, it focuses on the efficient transport of data between the OS kernels running on hosts connected by a high-speed network. The issue of achieving application-to-application networking performance is treated in subsequent chapters.

3.1 Experimental Hardware and Software

With the emergence of high-speed network facilities, several research efforts are focusing on the design and implementation of network adaptors [C+91, BP93, B+93, Ram93, TS93]. We have used one of these adaptors in our research—the OSIRIS ATM board built for the AURORA Gigabit Testbed [C+93b, Dav93]. We consider this network adaptor from a software perspective, identifying the subtle interactions between the adaptor and the operating system software that drives it. The flexibility built into the OSIRIS board makes this interaction an especially interesting one to study.

The OSIRIS network adaptor was designed specifically to support software experimentation. Therefore, only the most critical, high-speed functions are implemented in hardware, and even these are primarily implemented in programmable logic. The architecture of the interface is depicted in Figure 3.1. It consists of two mostly independent halves—send and receive—each controlled by an Intel 80960 microprocessor.

The adaptor board attaches to a TURBOchannel option slot provided by DEC workstations. We used two generations of DEC workstations in this work: the DECstation 5000/200 (25 MHz MIPS R3000), and the DEC 3000/600 (175 MHz Alpha). From the host's perspective, the adaptor looks like a 128KB region of memory. A combination of host software and software executing in the on-board microprocessors determine the detailed structure of this memory. In general, the memory is used to pass buffer descriptors between the host and the adaptor. Network data is not normally buffered in the dual-port

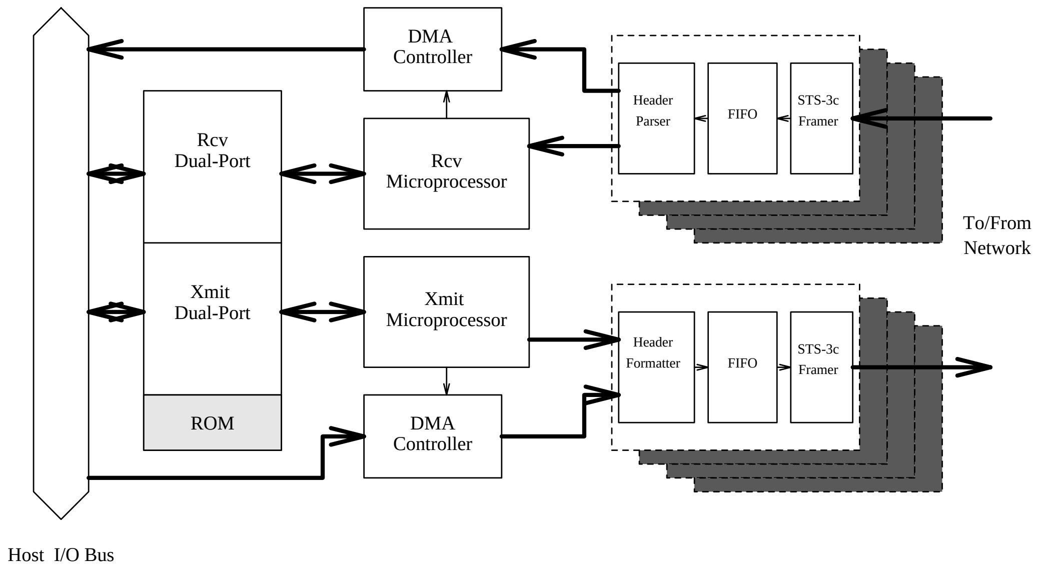

Figure 3.1: Architectural Overview of Osiris

memory; it is transferred directly from/to main memory buffers using DMA.

In the transmit direction, the software running on the microprocessor writes commands to a DMA controller and an ATM cell generator. The general paradigm is that the host passes buffer descriptors to the microprocessor through the dual-port RAM, and the microprocessor executes a segmentation algorithm to determine the order in which cells are sent. For example, the host could queue a number of packets and the microprocessor could transmit one cell from each in turn. The microprocessor has the capability to interrupt the host.

In the receive direction, the microprocessor reads from a FIFO the virtual circuit identifier (VCI) and ATM adaptation layer (AAL) information [Bel92] that is stripped from cells as they are received. By examining this information, and using other information from the host (such as a list of reassembly buffers), the microprocessor determines the appropriate host memory address at which the payload of each received cell is to be stored. It then issues commands to another DMA controller; typically one command is issued for each ATM cell received. As part of the reassembly algorithm, the microprocessor decides when it is necessary to interrupt the host.

The important point to understand from this brief description is that software running on the two 80960s controls the send/receive functionality of the adaptor, and perhaps just as importantly, this code effectively defines the *software interface* between the host and the adaptor. That is, the data structures, protocols, and events used for communication between host CPU and network adaptor are defined entirely by software.

The other relevant piece of software, of course, is the OS running on the host. On the DECstation 5000/200, we used the Mach 3.0 operating system [ABB+86], retrofitted with a network subsystem based on the *x*-kernel [DAPP93, HP91]. On the DEC 3000/600, we used a version of the *x*-kernel that runs as a stand-alone (native) operating system kernel. There are two relevant things to note about the OS. First, because the x-kernel supports arbitrary protocols, our approach is protocol-independent; it is not tailored to TCP/IP. Second, because Mach is a microkernel-based system and the x-kernel allows the protocol graph to span multiple protection domains, our approach has to allow for the possibility that network data traverses multiple protection domains; it is not restricted to kernel-resident protocols.

The following sections describe our experiences using the OSIRIS board, highlighting the problems it imposed on the software, and how we addressed them. For the most part, this discussion is limited to how we implemented the basic host-to-host functionality, both correctly, and with the highest possible performance; Section 3.8 describes how certain features of the board were exploited to implement novel OS techniques that turn this host-to-host performance into equally good user-to-user performance.

3.2 Host/Board Communication

We begin by defining the software interface between the host's device driver and the processors on the OSIRIS board. The host CPU communicates with the on-board processors through shared data structures in the dual-port memory. In addition, each on-board processor can assert an interrupt to alert the host CPU of asynchronous events. The design of the shared data structures and the discipline for using interrupts was guided by the goals of minimizing packet delivery latency and host CPU load, and of achieving host-to-host throughput close to the capacity of the network link. Particular attention was paid to (1) minimizing the number of load and store operations required to communicate with the on-board processors (accesses to the dual port-memory across the TURBOchannel are expensive), (2) avoiding delays due to lock contention while accessing shared data structures in the dual-port memory, and (3) minimizing the number of interrupts, which place a significant load on the host CPU.

3.2.1 Shared Data Structure

As with any shared data structure, measures must be taken to ensure consistency in the presence of concurrent accesses. The dual-port memory itself guarantees atomicity of individual 32 bit load and store operations only. Each half of the board provides a test-and-set register that can be used to implement a simple spin-lock. The intended use is to enforce mutually exclusive access to the dual-port memory by mandating that a processor must first acquire the corresponding lock. This approach allows arbitrarily complex shared data structures, but it restricts concurrency between host CPU and on-board processors. As a result, both packet delivery latency and CPU load can suffer due to lock contention.

To avoid this problem, we use simple lock-free data structures that rely only on the atomicity of load and store instructions. The basic data structure used in the dual-port memory is a simple, one-reader-one-writer FIFO queue used to pass buffers between the host and the adaptor. The queue consists of an array of buffer descriptors, a head pointer, and a tail pointer. The head pointer is only modified by the writer and the tail pointer is only modified by the reader. The processors determine the status of the queue by comparing the head and tail pointers, as follows:

$$head = tail \Leftrightarrow queue\ is\ empty$$

$$(head + 1) \bmod size = tail \Leftrightarrow queue\ is\ full$$

The simplicity of these lock-free queues maximizes concurrent access to the dual-port memory, and minimizes the number of load and store operations required to communicate.

A single queue is used for communication between the host CPU and the transmit processor. Each queue element describes a single buffer in main memory by its physical address and length. To queue a buffer for transmission, the host CPU performs the following actions (`xmitQueue[head]` refers to the buffer descriptor referred to by the head pointer).

- wait until the transmit queue is not full
- de-allocate any previous buffer described by `xmitQueue[head]`
- queue the buffer using `xmitQueue[head]`
- increment the head pointer (modulo array size)

The transmit processor continuously performs the following actions.

- wait until the transmit queue is not empty
- read the descriptor at `xmitQueue[tail]`
- transmit the buffer
- increment the tail pointer (modulo array size)

Two queues are required for communication between the host and the receive processor. The first queue is used to supply buffers to the receive processor for storage of incoming PDUs;[1] the second queue holds filled buffers waiting for processing by the host. Initially, the host fills the free buffer queue. When a PDU arrives, the receive processor removes a buffer from this queue, and stores incoming data into the buffer. When the buffer is filled, or the end of the incoming PDU is encountered, the processor adds the buffer to the receive queue. If the receive queue was previously empty, an interrupt is asserted to notify the host of the transition of the receive queue from the empty state to a non-empty state. The host's interrupt handler schedules a thread that repeatedly performs the following steps until the receive queue is found empty:

- remove a buffer from the receive queue
- add a free buffer to the free queue
- initiate processing of the received data

[1]We use the term *protocol data unit* (PDU) to denote a packet processed by a protocol, where the protocol in question is generally given by the context. In this case, the PDU corresponds to the unit of data sent between device drivers.

3.2.2 Interrupts

Handling a host interrupt asserted by the Osiris board takes approximately 75μs in Mach on a DecStation 5000/200. For comparison, the service time for a received UDP/IP PDU is 200μs; this number includes protocol processing and driver overhead, but not interrupt handling. Given this high cost, minimizing the number of host interrupts during network communication is important to overall system performance.

In our scheme, the completion of a PDU transmission, which is traditionally signalled to the host using an interrupt, is instead indicated by the advance of the transmit queue's tail pointer. The driver checks for this condition as part of other driver activity—for example, while queuing another PDU—and takes the appropriate action. Interrupts are used only in the relatively infrequent event of a full transmit queue. In this case, the host suspends its transmit activity, and the transmit processor asserts an interrupt as soon as the queue reaches the half empty state.

In the receiving direction, an interrupt is only asserted once for a *burst* of incoming PDUs. More specifically, whenever a buffer is queued before the host has dequeued the previous buffer, no interrupt is asserted. This approach achieves both low packet delivery latency for individually arriving packets, and high throughput for incoming packet trains. Note that in situations where high throughput is required (i.e. when packets arrive closely spaced), the number of interrupts is much lower than the traditional one-per-PDU.

3.3 Physical Buffer Fragmentation

The Osiris board relies on direct memory access (DMA) for the actual transfer of network data between main memory and network adaptor. The unit of data exchanged between host driver software and on-board processors is a physical buffer—a set of memory locations with contiguous physical addresses. The descriptors used in the transmit and receive queues contain the physical address and the length of a buffer. The on-board processors initiate DMA transfers based on the physical address of the buffers.

The per-PDU processing cost in the host driver increases with the number of physical buffers used to hold the PDU. Thus, one would like to minimize the number of physical buffers occupied by a single PDU. However, this is made difficult by the fact that the contiguous virtual memory pages used to store a PDU are generally not contiguous in the physical address space. The reason for this lies at the heart of any page-based virtual memory system—the ability to map non-contiguous physical pages to contiguous virtual memory addresses, in order to avoid main memory fragmentation.

Figure 3.2 depicts a PDU passed to the Osiris driver for transmission. The PDU consists of two parts—a header portion, which contains protocol headers, and the data portion. The header portion usually contributes one physical buffer. The data portion is typically not aligned with page boundaries, and may thus occupy $\lceil (\mathit{message\ data\ size} \Leftrightarrow 1)/\mathit{page\ size} \rceil + 1$ pages. When the physical pages occupied by the data portion are not contiguous, each page contributes a physical buffer. In practice, a PDU with a data portion of length n pages usually occupies $n + 2$ physical buffers.

Message fragmentation at the protocol level can aggravate this proliferation of physical buffers. The problem is that unless the fragment boundaries in the original message

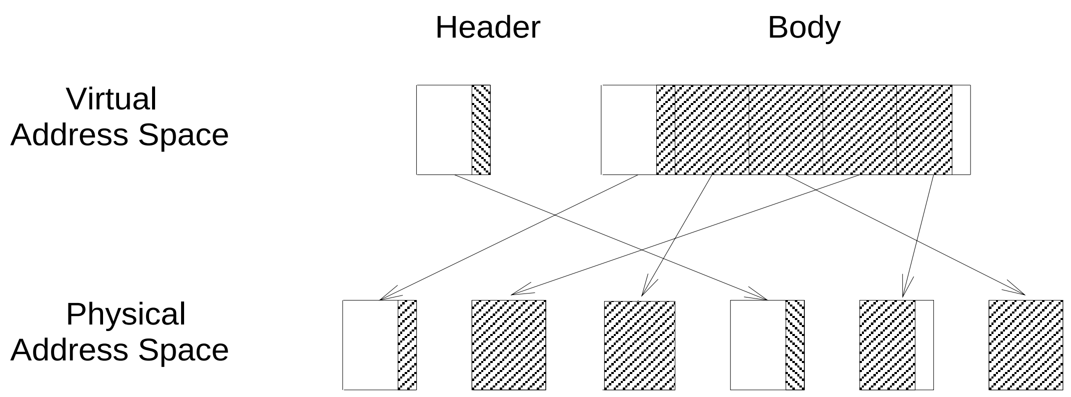

Figure 3.2: PDU Buffer Fragmentation

coincide with page boundaries, each fragment may generate excess physical buffers in the driver. As an example of an extreme case, assume that a contiguous 16KB application message is transmitted using UDP/IP with a maximal transfer unit (MTU) of 4KB,[2] which is also the system's page size. The inclusion of the IP header reduces the data space available in each fragment to slightly less than 4KB. Consequently, the data portions of most fragments are not page-aligned, and occupy two physical pages. In addition, the IP header attached to each fragment occupies a separate page. As a result, the transmission of a single, 16KB application message can result in the processing of up to 14 physical buffers in the driver. This compounding effect of IP fragmentation and buffer fragmentation can be avoided by ensuring page alignment of application messages, and by choosing an MTU size that is a multiple of the page size, plus the IP header size. This ensures that fragment boundaries align with page boundaries.

A similar problem exists on the receive side. Recall that the host driver allocates receive buffers, and queues these buffers for use by the receive processor. Most operating systems do not support the dynamic allocation of physically contiguous pages. In this case, the size of the receive buffers is restricted to the system's memory page size, since it represents the largest unit of physically contiguous memory that the driver can allocate. This limit on the size of receive buffers causes the fragmentation of all incoming network packets larger than the page size.

The proliferation of physical buffers is a potential source of performance loss in the Osiris driver. A general solution to this problem would require the use of physically contiguous memory for the storage of network data. In traditional operating systems, where network data is copied between application memory and kernel buffers, this can be achieved by statically allocating contiguous physical pages to the fixed set of kernel buffers. Unfortunately, this approach does not readily generalize to a copy-free data path (as outlined in Chapter 2, and detailed in Chapter 5), since applications generally cannot be allowed to hold buffers from a statically allocated pool.

Several modern workstations, such as the IBM RISC System/6000 and DEC 3000

[2]Keep in mind that the Osiris driver, not the hardware, defines the MTU. We are just using 4KB as an example.

AXP Systems provide support for virtual address DMA through the use of a hardware virtual-to-physical translation buffer (scatter/gather map) [IBM90, D$^+$92]. Host driver software must set up the map to contain appropriate mappings for all the fragments of a buffer before a DMA transfer. When data is transferred directly from and to application buffers, it may be necessary to update the map for each individual message. As a result, physical buffer fragmentation is a potential performance concern even when virtual DMA is available.

3.4 Cache Coherence

The cache subsystem of the host we were originally using—the DECstation 5000/200—does not guarantee a coherent view of memory contents after a DMA transfer into main memory. That is, CPU reads from cached main memory locations that were overwritten by a DMA transfer may return stale data. To avoid this problem, the operating system normally executes explicit instructions to invalidate any cached contents of memory locations that were just overwritten by a DMA transfer. Unfortunately, partial invalidations of the data cache take approximately one CPU cycle per memory word (32bits), plus the cost of subsequent cache misses caused by the invalidation of unrelated cached data.[3] This cost has a significant impact on the attainable host-to-host throughput, as quantified in Section 3.9 (Figure 3.3).

The key idea for avoiding this cost is to take a lazy approach to cache invalidation, and to rely on network transmission error handling mechanisms for detecting errors caused by stale cache data. When a data error is detected at some stage during the processing of a received message, the corresponding cache locations are invalidated, and the message is re-evaluated before it is considered in error. The feasibility of this approach depends on the following conditions.

1. The underlying network is not reliable, and therefore mechanisms for detecting or tolerating transmission errors are already in place.

2. The rate of errors introduced by stale cache data is low enough for the lazy approach to be effective.

3. Revealing stale data does not pose a security problem.

While the first condition is true for most networks, the second condition deserves some careful consideration. The Osiris driver employs a free buffer queue and a receive queue with a length of 64 buffers each, and a buffer size of 16KB. This implies that once a receive buffer is allocated and queued on the free buffer queue, normally 63 other buffers are processed by the host until that buffers re-appears at the top of the received buffer queue. In order to become stale, a cached data word from a particular buffer has to remain in the cache while 63 other receive buffers are being processed. During this time, the CPU typically reads the portion of the input buffers occupied by received data, as well as other

[3]The DECstation also supports a fast instruction that swaps the data and instruction cache, which amounts to an invalidation of the entire cache contents. However, the high cost of subsequent cache misses makes this an unattractive solution.

data relating to protocol processing, application processing and other activities unrelated to the reception of data. These accesses are likely to evict all previously cached data from the DECstation's 64KB data cache.

Experimentally, we have seen no evidence of stale data at all while running our test applications. This suggests that the error rate should be low enough for this optimization to be very effective. It should be noted that lazy cache invalidation is not likely to scale to machines with much larger caches. Fortunately, hardware designers have recognized the high cost of software cache invalidation, and tend to provide support for cache coherence on these machines. For example, the DEC 3000 AXP workstation data cache is updated during DMA transfers into main memory.

The third condition is satisfied whenever reliable protocols are used that detect data errors before the data is passed to an unprivileged application. However, with unreliable protocols, an application could access stale data from a previous use of the receive buffer, potentially violating the operating system's security policy. This problem can be solved by ensuring the reuse of receive buffers on the same data stream. In this way, stale data read by an application is guaranteed to originate from an earlier message received by that application, thus eliminating security problems. The reuse of receive buffers on the same data stream has other advantages, as described in Section 3.8.1.

3.5 Page Wiring

Whenever the address of a buffer is passed to the OSIRIS on-board processors for use in DMA transfers, the corresponding pages must be *wired*. Wiring, also referred to as *pinning*, refers to the marking of a page as being non-eligible for replacement by the operating system's paging daemon. Since changing the wiring status of a page occurs in the driver's critical path, the performance of this operation is of concern.

Our initial use of the Mach kernel's standard service for page wiring resulted in surprisingly high overhead. One problem is that Mach's implementation of page wiring provides stronger guarantees than are actually needed for DMA transfers. In particular, it prevents not only replacement of the page itself, but also of any pages containing page table entries that might be needed during an address translation for that page. We now use low-level functionality provided by the Mach kernel to prevent replacement of pages with acceptable performance.

3.6 DMA Length

The length of DMA transactions has a significant effect on performance. As mentioned above, DMA usually takes place one ATM cell at a time. This provides maximum flexibility in the transmit direction (e.g. to interleave several outgoing PDUs) and avoids the need for a reassembly buffer in the receive direction. The maximum data transfer speed that can be sustained with 44 byte transfers over the TURBOchannel on a DECstation 5000/200 is 367 Mbps in the transmit direction and 463 Mbps in the receive direction. These figures, which have been measured for brief periods on the actual hardware, can be derived simply by considering the minimal overhead for DMA transactions in each direction—8 cycles for writes, 13 cycles for reads. Thus, for example, the maximum throughput for transmission is $11/(11+13) \times 800 = 367$ Mbps.

Clearly, it would be advantageous to increase the length of DMA transfers. In the transmit direction, the only penalty for increasing DMA length is an increase in the granularity of multiplexing. It has been argued that fine-grained multiplexing is advantageous for latency and switch performance reasons [Dav91]. However, when the adaptor is used in a mode where the goal is to maximize throughput to a single application, neither of these reasons is relevant. It is therefore reasonable, and straightforward, to perform DMA transactions longer than one ATM cell. Note that with transfers of 88 bytes at a time, the maximum rate that data could be moved across the bus would be $22/(22+13) \times 800 = 503$ Mbps. This is close to the 516 Mbps data bandwidth available in a 622 Mbps SONET/ATM link [BC89] when 44 byte cell payloads are used.

In the receive direction, the primary advantage in doing single-cell DMAs is that it removes the need for a reassembly buffer on the adaptor; cells can be placed directly in host memory as they arrive. Not only does this reduce the hardware complexity of the interface, but it also reduces the likelihood that inadequate reassembly space is available.

In some circumstances, however, it is possible to preserve the advantages of not having a reassembly buffer on the adaptor while performing DMAs longer than one cell. The quantity that we really wish to optimize is the user-to-user throughput for a single application. In this case, as long as cells arrive in order, most successively received cells will contain data that is to be stored in contiguous regions of host memory, the only exception being at the end of a buffer. Since there is a small amount of FIFO buffering of cells on the adaptor, the microprocessor can look at two cell headers before deciding what to do with their associated payloads. If the header information suggests that the two payloads should be stored contiguously, then a single, 88-byte DMA can be initiated.

Note that the biggest gain is achieved just by going to double-cell DMAs, since we have already driven the overhead down from 42% to 26%. With any further increase in DMA length the returns diminish. The measured performance of doing 88-byte DMAs is reported in Section 3.9.

3.7 DMA versus PIO

One of the most lively debates in network adaptor design is over the relative merits of DMA and programmed I/O (PIO) for data movement between the host and the adaptor. Both the literature on the subject (e.g. [Ram93, BP93, DWB$^+$93]) and our own experience have led us to the conclusion that the preferable technique is highly machine-dependent. In the case of the DEC workstations we used, the low throughput achievable using PIO across the TURBOchannel ensures that, with well designed software (i.e. no unnecessary copies) DMA is preferable.

We argue that the best way to compare DMA performance versus PIO is to determine how fast an application program can access the data in each case. For example, when data is DMAed into memory on a DECstation 5000/200, it will not be in the cache; an additional read of the main memory is necessary when the application accesses the data. On the DECstation, reading data into the cache causes a dramatic decrease in throughput from the pure DMA results, but the throughput remains above that which can be achieved by PIO simply because of the high performance penalty for word-sized reads across the TURBOchannel. On DEC's Alpha-based machines, a greatly improved memory system with a crossbar switch that connects TURBOchannel, main memory and cache allows cache/memory transactions to occur concurrently with DMA transfers on the TURBOchannel. In addition, DMA writes to main memory update the second level cache. On these machines, applications are able to access the data at the rate of and concurrent with its DMA transfer into main memory (see Section 3.9).

In the PIO case, with carefully designed software, data can be read from the adaptor and written directly to the application's buffer in main memory, leaving the data in the cache [Jac90, DWB+93]. If the application reads the data soon after the PIO transfer, the data may still be in the cache. According to one study, the PIO transfer from adaptor to application buffer must be delayed until the application is scheduled for execution, in order to ensure sufficient proximity of data accesses for the data to remain cached under realistic system load conditions [PDP93]. Loading data into the cache too early is not only ineffective, but can actually decrease overall system performance by evicting live data from the cache. Unfortunately, delaying the transfer of data from adaptor to main memory until the receiving application is scheduled for execution requires a substantial amount of buffer space in the adaptor. With DMA, instead of using dedicated memory resources on the adaptor, incoming data can be buffered in main memory. Using main memory to buffer network data has the advantage that a single pool of memory resources is dynamically shared among applications, operating system, and network subsystem.

3.8 New OS Mechanisms

This section sketches two novel OS mechanisms—*fast buffers* (fbufs) and *application device channels* (ADCs)—and highlights those features of the OSIRIS board that facilitated these mechanisms. The mechanisms themselves are described in detail in later chapters.

3.8.1 Fast Buffers

One of the key problems faced by the operating system, especially a microkernel-based system in which device drivers, network protocols, and application software might all reside in different protection domains, is how to move data across domain boundaries without sacrificing the bandwidth delivered by the network. The fbuf mechanism is designed to address this problem—it is a high-bandwidth cross-domain buffer transfer and management facility.

The fbuf mechanism combines 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). Since fbufs are described in detail in Chapter 5, this section concentrates on the OSIRIS features that we were able to exploit.

The effectiveness of fbufs depends on the ability of the adaptor to make an early demultiplexing decision. That is, the "data path" through the system that the incoming packet is going to traverse must be determined by the adaptor so that it can be stored in an appropriate buffer; one that is mapped into the right set of domains. We say that an fbuf that is already mapped into a particular set of domains is *cached*. Being able to use a cached fbuf, as opposed to an uncached fbuf that is not mapped into any domains, can mean an order of magnitude difference in how fast the data can be transferred across a domain boundary.

In the case of the Osiris adaptor, the device driver employs the following strategy. It maintains queues of preallocated cached fbufs for the 16 most recently used data paths, plus a single queue of preallocated uncached fbufs. The adaptor performs reassembly of incoming packets by storing the ATM cell payloads into a buffer in main memory using DMA. When the adaptor needs a new reassembly buffer, it checks to see if there is a preallocated fbuf for the virtual circuit identifier (VCI) of the incoming packet. If not, it uses a buffer from the queue of uncached fbufs.

One of the interesting aspects of this scheme is how we use VCIs. The x-kernel provides a mechanism for establishing a path through the protocol graph, where a path is given by the sequence of sessions that will process incoming and outgoing messages on behalf of a particular application-level connection. Each path is is then bound to an unused VCI by the device driver. This means that we treat VCIs as a fairly abundant resource; each of the potentially hundreds of paths (connections) on a given host is bound to a VCI for the duration of the path (connection). This approach is not compatible with a regime that treats VCIs as a scarce resource, and in particular, a resource that the network charges for.

The early demultiplexing capability, which facilitates fbufs, has advantages beyond that of enabling efficient delivery of data to applications. It is also the basis for the appropriate processing of prioritized network traffic under high receiver load [Fel90]. The threads that de-queue buffers from the various receive queues may be assigned priorities corresponding to the traffic priorities of the network stream they handle. During phases of receiver overload, lower-priority receive queues will become full before higher priority ones, allowing the adaptor board to drop the lower priority packets before they have consumed any processing resources on the host.

3.8.2 Application Device Channels

Fbufs take advantage of the Osiris demultiplexing capability to avoid costs associated with the transfer of data across protection domain boundaries on the end host. These costs would otherwise limit the attainable application-to-application throughput. Application device channels (ADCs) take the on-board demultiplexing approach a significant step further. An ADC gives an application program restricted but direct access to the Osiris network adaptor, bypassing the operating system kernel. This approach removes protection domain boundaries from both the control and data path between application and network adaptor, resulting in minimal application-to-application message latencies.

While ADCs give application programs direct access to the network adaptor, several unprivileged applications can share the adaptor. The operating system remains in control of the network device, ensuring protection, security and fairness. The is accomplished by moving the protection boundaries effectively *into* the network adaptor. Application device channels are described in more detail in Chapter 6.

To support ADCs, the network adaptor must be able to support, in addition to packet demultiplexing, multiple transmit and receive buffer queues. Furthermore, it must be possible to map pairs of transmit/receive queues directly into application's address spaces. With OSIRIS, the transmit and receive processor each share 64 KB of dual-ported memory with the host. Assuming a 4 KB virtual memory page size, the adaptor can support up to 16 transmit/receive queues, which can be mapped into different address spaces, including applications. Other functionality required for ADCs, namely validation of transmit requests and application buffer addresses, was implemented by programing the OSIRIS on-board processors accordingly. In summary, the OSIRIS features that facilitated ADCs are its dual-ported memory that spans multiple VM pages, and its programmable on-board processors.

3.9 Performance

This section reports on several experiments designed to evaluate the network performance achieved with the OSIRIS board, and the impact of various optimizations described in earlier sections. All presented results refer to message exchanges between test programs linked into the OS kernel. Each data point is the average of 10 trials, where each trial consists of sending 100,000 messages, after an initial warm-up period. In all cases, the 90% confidence intervals where within 1% of the average. The workstations were running in single-user mode.

Machine	Protocol	Message size (bytes)			
DEC model		1	1024	2048	4096
5000/200	ATM	353	417	486	778
	UDP/IP	598	659	725	1011
3000/600	ATM	154	215	283	449
	UDP/IP	316	376	446	619

Table 3.1: Round-Trip Latencies (μs)

Throughout this section, we report results obtained on two generations of workstations: the DECStation 5000/200 (25Mhz MIPS R3000 CPU, 64/64 KB I/D caches, 16 MB of main memory), and the DEC 3000/600 (175MHz Alpha CPU, 8/8 KB I/D caches, 2 MB second level cache, 64 MB of main memory). Table 3.1 shows the round-trip latencies achieved between a pair of workstations connected by a pair of OSIRIS boards linked back-to-back. The rows labeled "ATM" refer to the round-trip latency of PDUs exchanged between test programs configured directly on top of the OSIRIS device driver. In the "UDP/IP" case, round-trip latency was measured between two test programs configured on top of the UDP/IP protocol stack[4]. IP was configured to use an MTU of 16KB, and

[4]Our otherwise standard implementations of IP and UDP were modified to support message sizes larger

UDP checksumming was turned off. The measured latency numbers for 1 byte messages are comparable to—and in fact, a bit better than—those obtained when using the machines' Ethernet adaptors under otherwise identical conditions. This is a reassuring result, since it demonstrates that the greater complexity of the OSIRIS adaptor did not degrade the latency of short messages.

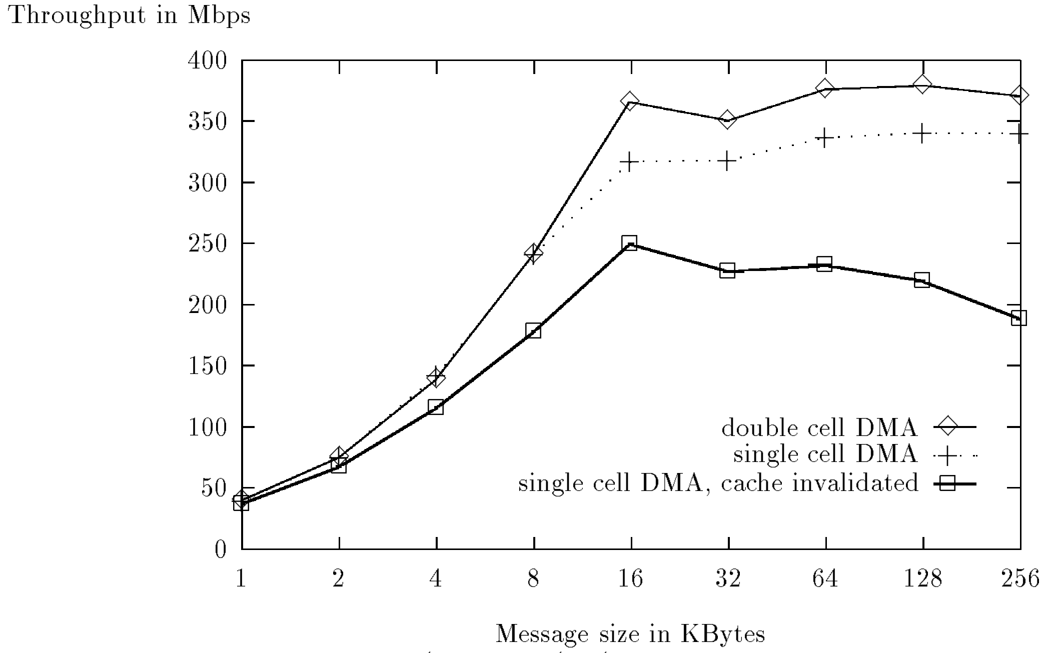

Figure 3.3: DEC 5000/200 UDP/IP/OSIRIS Receive Side Throughput

The next set of measurements was designed to evaluate the network performance of the receiving host in isolation. For this purpose, the receiver processor of the OSIRIS board was programmed to generate fictitious PDUs as fast as the receiving host could absorb them. Figure 3.3 shows the measured data throughput achieved on a DEC 5000/200 with the UDP/IP protocol stack, where the IP MTU was set to 16 KB. The graphs depict results measured with DMA transfer sizes of one and two ATM cell payloads, and with cache invalidation in the OSIRIS driver.

We make the following observations. First, the maximal throughput achieved is 379 Mbps with double cell DMA, 340 Mbps with single cell DMA, and 250 Mbps with single cell DMA when the data cache is pessimistically invalidated after each DMA transfer. The last number shows the significant impact of cache invalidations on throughput.

In the DECStation 5000/200, all memory transactions occupy the TURBOchannel and no part of a DMA transaction can overlap with the CPU accessing main memory. Thus, memory writes and cache fills that result from CPU activity reduce DMA performance. Conversely, DMA traffic increases the average memory access latency experienced by the CPU. The combined effect of DMA overhead and main memory contention result in a

than 64KB.

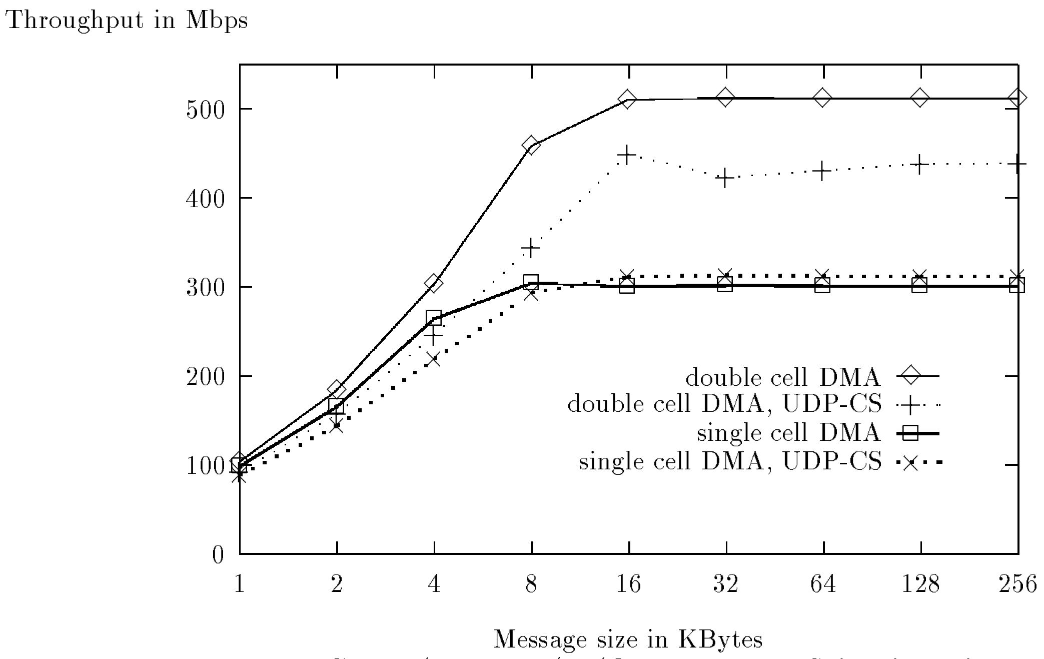

Figure 3.4: DEC 3000/600 UDP/IP/OSIRIS Receive Side Throughput

maximum throughput rate of 340 Mbps in the receive direction. Note that in this experiment, network data is never accessed by the CPU. In the case where the data is read by the CPU (e.g., to compute the UDP checksum), the maximal throughput decreases to 80 Mbps, due to the limited memory bandwidth on this machine.

Figure 3.4 shows the corresponding results obtained using DEC 3000/600 workstations. This machine has a greatly improved memory system. A buffered crossbar allows DMA transactions and cache fills/cache write-backs to proceed concurrently, and hardware ensures cache coherence with respect to DMA. The experiment was run with single and double DMA transfers, and with UDP checksumming turned on and off. With double cell length DMA, the throughput now approaches the full link bandwidth of 516 Mbps for message sizes of 16KB and larger. With UDP checksumming turned on, the throughput decreases slightly to 438 Mbps. This is an important result; it implies that the network data can be read and checksummed at close to 90% of the network link speed. Also, the throughput for small messages has improved greatly, thanks to the reduced per-packet software latencies on this faster machine.

The final set of measurements evaluates the network performance on the transmit side. The results for both the DEC 5000/200 and the 3000/600 are shown in Figure 3.5. The maximal throughput achieved on the transmit side is currently 325 Mbps. This number is limited entirely by TURBOchannel contention due to the high overhead of single ATM cell payload sized DMA transfers. A hardware change to allow longer DMA transfers in this direction is underway, but was not completed at the time of this writing. With double cell DMA transfers on the transmit side, the host-to-host throughput attained is expected to fall between the graphs for single cell DMA and that for double cell DMA on the receive

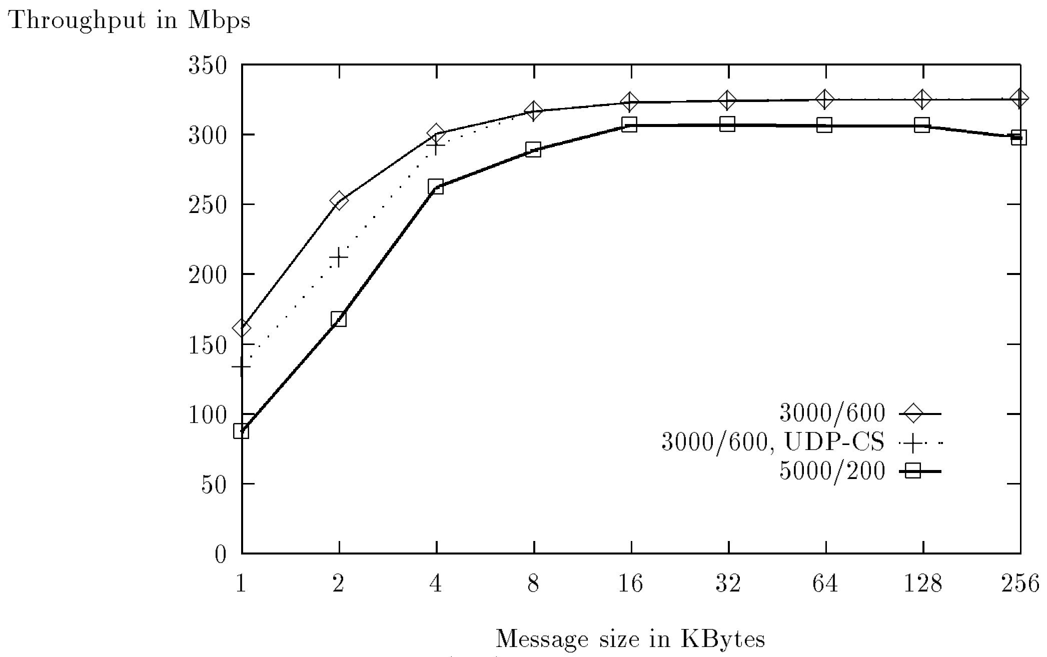

Figure 3.5: UDP/IP/Osiris Transmit Side Throughput

side (Figure 3.4).

3.10 Concluding Remarks

The material presented in this chapter demonstrates that high-speed networking traffic can be efficiently processed at the level of the operating system on current, general-purpose workstations. It identifies the main issues that must be addressed to support high-speed network I/O traffic in the OS. That is, minimizing host/adaptor communication costs, including interrupt overheads, and streamlining data transfer between main memory and adaptor, which includes issues such as cache coherence, buffer fragmentation, and virtual memory page wiring. The following chapters address the related issue of delivering high-speed networking traffic to application programs without loss of performance.

Finally, given that the Osiris adaptor was designed to provided maximal flexibility, it contains many more features than one would include in a production board. Based on our experience, we have found the following two features to be important, and would recommend that they be considered in future board designs.

- The ability to make an early demultiplexing decision; treating VCIs as an abundant resource that represents end-to-end connections is a reasonable way to do this on an ATM network. This is used by both the fbuf and ADC mechanisms.

- The ability to support multiple transmit and receive queues, and map each of them directly into user-level protection domains. It was this feature that facilitated the ADC mechanism.

CHAPTER 4
Decoupling Modularity and Protection

This is the first of a series of three chapters that present novel techniques that help deliver the performance of high-speed networks to *application* programs. In Chapter 2, we have seen that the protection domain boundaries that occur between operating system kernel and application programs present an obstacle for high-performance I/O operations. I/O data must be transferred across domain boundaries at a potential loss in bandwidth and added latency. Furthermore, protection boundaries destroy memory access locality, thus exposing the main memory bandwidth bottleneck.

The problem is a particularly pressing one in microkernel-based operating systems. In these systems, I/O data may have to cross multiple protection domain boundaries, and several server processes may have to be scheduled for execution during each I/O operation. This chapter presents a new approach for structuring an operating system that affords all the benefits of a modular OS structure, without the performance cost of additional protection boundaries and server processes.

4.1 Motivation

The benefits of a modular operating system design are well-known: The system is more easily configured, scaled, extended, ported, maintained, verified, and distributed across multiple processors. Growing support for modular OS design has recently popularized the idea of *microkernel-based* systems [ABB$^+$86, RAA$^+$88, Hil92]—a kernel that implements the most basic abstractions, and a small number of user-level *servers* that provide the functionality of a specific abstract machine. While microkernel-based systems are an improvement over monolithic operating systems like UNIX, they fall short of satisfying a number of demands placed on modern operating systems.

The problem with microkernel-based systems is that they tightly couple modularity and protection—servers are implemented in separate protection domains, and consequently, the communication mechanisms are designed for the cross-domain case. A number of limitations arise as a result of this coupling. First, the modularity supported by these systems is very coarse-grained. Since each module is implemented in a separate domain, concern for cross-domain communication costs prevents a fine-grained decomposition of the system[1]. For the same reason, it is difficult to extend the system vertically—i.e., through stackable services—because each layer adds communication overhead that degrades performance. Second, the partitioning of functionality into servers—and in particular determining what functionality should be provided by the kernel—is static and part of an early design decision. Consequently, it is difficult to reconfigure functionality among servers and between kernel and servers. Such reconfigurations are desirable to satisfy the

[1]Note that even light-weight RPC is an order of magnitude more costly than local procedure calls.

needs of applications, to match the characteristics of a variety of hardware architectures, and to integrate new technology as the system evolves.

The solution we propose is to provide architectural support for modules that is independent of protection domains. The key is a location transparent invocation mechanism that (1) has semantics similar to a local procedure call, (2) delivers performance close to that of an ordinary procedure call when the invocation is between modules in the same domain, and (3) allows the use of the most appropriate IPC mechanism in the cases of cross-domain and cross-machine invocations. Given such support, the decomposition of a system into modules can be guided by sound software engineering principles, with the modules distributed across protection domains and machines at configuration time based on criteria such as trust, security, fault-tolerance and performance. Put another way, protection and modularity are decoupled.

4.1.1 Why Separate Modularity and Protection?

The fundamental reasons for providing support for modularity that is independent of protection are that (1) it allows *modular decomposition without concern for cross-domain communication costs*, and (2) the *partitioning of functions into protection domains becomes a matter of configuration rather than design.* These two facts have several important consequences, which are explored below.

First, while configuring a given system, the granularity of protection can be adjusted from one resembling a capability-system [WCC$^+$74] (with each module in a separate domain), through the granularity found in microkernel-based systems, up to no protection at all (with all modules in a single domain). The partitioning of modules into domains can be adjusted according to their stage in the software life-cycle, and/or the requirements in a particular installation of the system. For instance, a subsystem consisting of a set of modules can be configured with each of its modules in a separate domain for ease of fault detection and debugging during its testing phase, and later, in the post-release phase, combined into a single domain (or even the kernel domain) for performance. In other words, during the validation phase, the chosen granularity of protection is such that encapsulation is enforced. Once in the post-release phase, the granularity of protection is reduced to the point where only modules with different levels of trust are separated by domain boundaries.

Second, determining the set of functions to be included in the kernel domain—the subject of an ongoing debate in the microkernel-based OS community—becomes a matter of configuration. The kernel module, which we call the *nugget*, can be reduced to include only functionality that *must* be provided in the privileged domain—management of hardware resources. All other services are implemented in separate modules which may or may not be configured into the kernel domain. This is important because new kinds of servers built on top of the kernel, new hardware platforms, and new OS technology will continue to shift the "right" set of functions to be included in the kernel domain.

Third, a system built in this manner can be extended vertically through layered services [GHM$^+$90, HP91] without imposing cross-domain invocation costs at each layer. Moreover, since interfaces are provided at module boundaries rather than domain boundaries, a layered service can be accessed at each level. That is, applications can access a layered system service at the level of abstraction most appropriate to their needs. This eliminates the need for unstructured, extensible interfaces like the UNIX `ioctl` call, which are used to allow the access of low-level services through an abstract, high-level interface provided at the domain boundary.

Note that most of the advantages of microkernel-based systems (when compared to monolithic systems) are a result of the modularity, the dynamic binding of clients and servers, and the location transparent communication mechanism. Most of the overhead associated with microkernel-based design, on the other hand, is a result of the more fine-grained protection. The proposed approach offers the advantages of fine-grained modularity, dynamic binding and location transparency without the cost of equally fine-grained protection. Decoupling modularity and protection turns the tradeoff between *modularity* and performance (which governs microkernel design) into a tradeoff between *protection* and performance.

4.1.2 How To Separate Modularity and Protection

An architecture that decouples modularity and protection must provide location transparency and dynamic binding between *modules* instead of merely between protection domains. Moreover, the module invocation mechanisms must be designed both for efficient intra-domain invocation, and cross-domain communication.

The requirements for a suitable location transparent invocation mechanism are somewhat different from those found in microkernel-based systems. The key difference is that many more interfaces exist (because the system is decomposed at a finer grain), and as a consequence, the most common case is an invocation between modules in the same domain. A suitable invocation mechanism should provide semantics similar to a local procedure call and deliver performance similar to a local procedure call when the invocation is local. This is necessary in order to encourage and permit a fine-grained decomposition of the system.

One key issue is how *object references* are implemented. Most location transparent invocation systems, including those found in microkernel-based systems, use global identifiers—such as ports or capabilities—to refer to all objects. With this approach, local invocations require the translation of a global identifier into a local object pointer. This design is pessimistic; it assumes that most object invocations are non-local. In the proposed architecture, most object references are to local objects. Consequently, an optimistic design that uses local pointers to refer to all objects is more appropriate, where the local pointer is translated in the case of a non-local invocation.

Another important issue is how to approximate the semantics of a local procedure call. The general approach is to shield the client and server from the complex issues of remote invocation. For example, binding and authentication are performed implicitly during object reference creation. Our design, which is discussed in detail in Section 4.3, employs the technique of *proxy objects* [Sha86] to provide local object references and to shield client and server from the complexity of remote invocation. Finally, an appropriate invocation facility must offer a set of parameter passing modes that provide the efficiency of pass-by-reference in the case of a local (same domain) invocation, yet can be implemented in a straightforward and efficient way in the case of non-local invocations.

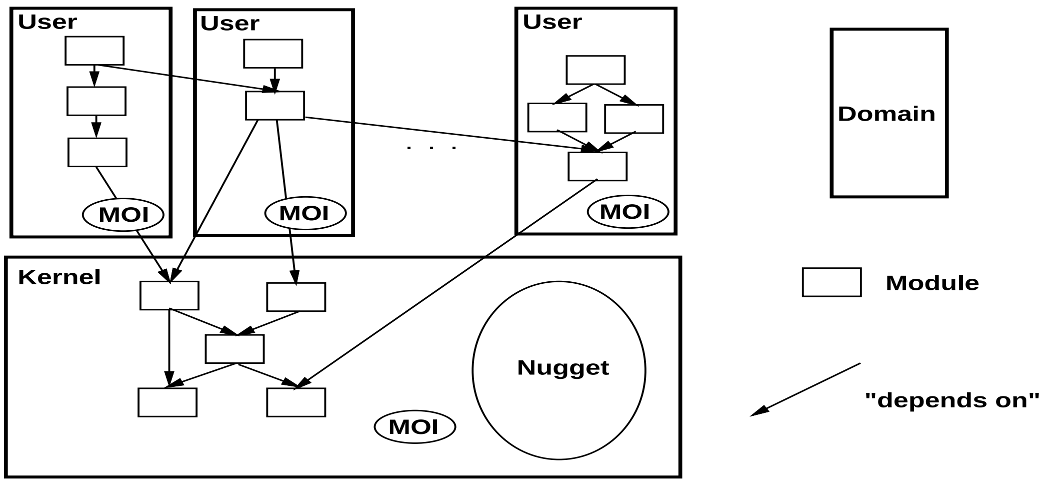

Figure 4.1: Lipto Architecture

4.2 Architectural Model

We have implemented and evaluated our proposed architecture in the context of an experimental operating system called Lipto. We focus here on those features of Lipto that pertain to the subject of this chapter. A more comprehensive description of Lipto's design and motivation can be found elsewhere [Dru93, DHP91, DPH91].

The components of Lipto's architecture are a fixed nugget, a module/object infrastructure, and a configurable collection of modules. The nugget consists of the lowest-level resource management, such as memory management, processor allocation and exception handling. The nugget is a truly minimal kernel; it includes functionality that *must* be executed with the machine in kernel mode, but not functionality that *may* be put into the kernel domain for performance.

The module/object infrastructure (MOI) provides location transparency and dynamic binding at the granularity of modules. All functionality not found in either the nugget or the module/object infrastructure is implemented in the form of configurable modules, which can be linked into any protection domain, including the privileged kernel domain of each machine. The basic Lipto architecture is illustrated in Figure 4.1.

A module provides the implementation for one or more types of *objects*. For example, a module that implements the top layer of a file system might provide two object types: a service object of type *file manager* and a resource object of type *file*. A module is the unit of configuration and distribution; it provides implementations of object types that represent a service or resource. Objects are passive run-time entities; they represent services and resources. An object's code is executed by independent threads of control upon invocation of one of the object's operations.

The architecture places three constraints on the implementation of objects: (1) An object must export a procedural interface consisting of a set of functions with well-defined signatures, (2) an object must not share data with any object implemented in a different module, and (3) the object invocation mechanism must conform to the architecture's specification. The module implementor is free to use any programming language and methodology, as long as these conditions are satisfied.

Encapsulation of objects with respect to objects implemented in other modules can be achieved in two ways. One is to use a programming language that enforces encapsulation through a safe type system. Unfortunately, such languages are currently not widely used in the implementation of operating system software. Without language support, encapsulation has to rely on convention. This has two consequences: first, each module should be configured into a separate protection domain during the testing phase to detect violations of encapsulation. Second, only modules that enjoy mutual trust can be configured into a common domain. Otherwise, an untrusted module could maliciously violate encapsulation to gain unauthorized access to information or resources.

The task of Lipto's module/object infrastructure is to provide location transparency. It consists of a service called the *system directory* that maps global object identifiers to object references, and a location transparent object invocation mechanism.

The system directory allows an authorized client object to obtain a *reference* to a server object using a global name for that object. This reference can be used to invoke the operations defined for the server object's interface. We omit a description of the mechanism for naming and locating objects, since it is not relevant to Lipto's architecture, which is the focus of this chapter. The design and implementation of the invocation mechanism is described in detail in the next section.

Lipto includes a proxy compiler that takes as input an interface definition for a class of modules that export the same interface. It produces target language specific interface definition files and source code that implements proxy objects. Our proxy compiler currently supports C and C++ as target languages for the implementation of modules.

4.3 Object Invocation Mechanism

This section describes in some detail the design and implementation of Lipto's object invocation mechanism. We start with the overall design issues and proceed in the following subsections with a description of individual components.

4.3.1 Design Issues

The main goals in the design of Lipto's invocation mechanism are: (1) to provide location transparency while closely resembling the semantics of a local procedure call, and (2) to retain the efficiency of a local procedure call in the intra-domain case. As stated earlier, this is essential for decomposing the system at a fine grain. The technique of proxy objects permits the representation of all object references as local pointers, and thus a very efficient implementation of local invocations. This technique is "optimized" for local invocation, which is the most common case. It is the task of the proxy objects to handle the remote case, hiding the associated complexity from both caller and callee.

Several problems arise in the attempt to provide location transparency while retaining local procedure call semantics: remote procedure calls can fail; it is difficult to provide pass-by-reference parameter passing semantics; binding a caller and callee involves complex naming, locating, protection, and authentication issues; and the performance of local, same-machine, and cross-machine invocations each differ by an order of magnitude. Most existing remote invocation systems do not attempt to hide the differences between local and remote calls, and require the caller to explicitly handle all the issues of a remote invocation, such as binding, authentication and failure handling. This approach conflicts with our goals since it exposes the full complexity of remote invocation to the caller, which would discourage a fine-grained decomposition.

The complexity of certain mechanisms used in the module/object infrastructure, and their associated costs, depend on the scale of the client-server distribution. Binding, authentication and failure detection are relatively simple between domains on a shared-memory machine or between nodes in a tightly coupled multiprocessor. They are more complex among machines connected by a LAN, and require elaborate protocols when the machines are connected by a WAN. Note that in a fine-grained modular system, many modules need only be accessed within a limited scope, e.g., within the same machine. Only modules that represent entire subsystems must be accessible on a larger scale. Consequently, the most common cases can be handled using simple and efficient mechanisms. A module's access scope is determined when it is registered with the system directory, and is transparent to the module implementor. We proceed with a discussion of the issues in location transparency from the perspective of the module implementor.

4.3.1.1 Binding and Authentication

Before invoking any operations on an object, a client has to obtain a reference for that object. This hides binding and authentication inside the mechanism for reference creation, thereby shielding the client from these issues. Further, the cost of binding, authentication and resource allocation is associated with reference creation, which allows a more efficient implementation of invocations because a "pseudo-connection" between client and server object is established. The implicit assumption is that a client that obtains a reference to an object will invoke several operations on that object before relinquishing the reference, so that the cost of reference creation is amortized over several calls. For cases where this is not a reasonable assumption, the proxy compiler can generate proxy object implementations for individual object types that delay connection establishment until the first invocation occurs.

4.3.1.2 Parameter Passing

Lipto's invocation mechanism provides the parameter passing methods `val`, `ref`, `in`, `out` and `inout`. `Val` is the default mode for data types. Since it always causes parameters to be copied, it is only used for simple data types. `Ref` is the method used for arguments of an object type. A reference for the object is passed and invocations of the object's operations through this reference are forwarded; the object is not moved nor copied. The only exception to this rule applies to the system-defined object type *IOData* (see Section 2.5). The IOData type encapsulates untyped, immutable data. When an instance of type IOData is passed as an argument/result, it is logically moved. Instances of type IOData are typically used as containers for raw data that is passed through the layers of the I/O subsystem.

In Lipto, we have found the parameter passing modes `val` (for simple data types) and `ref` (for object types) to be sufficient in almost all cases. This is because complex arguments are usually encapsulated in objects, and bulk data is passed in IOData objects. However, to provide efficient passing modes for special situations, and to ease the transition of non-object-oriented interfaces, the additional passing modes `in`, `out` and `inout` are provided, which can be applied to arbitrary non-object data types. In the non-local case, these modes cause parameters to be copied in the usual way, but in the local case, they are implemented by passing a pointer. This eliminates parameter copying costs in the case of a local invocation, which could otherwise impose a performance penalty in the local case due to the lack of a by-reference parameter passing mode.

In order to ensure location transparency, modules that use `in`, `out`, or `inout` passing modes must obey three restrictions. In particular, modules *must not* (1) pass aliases as actuals to `in`, `out`, or `inout` formal parameters, (2) reference an `out` formal parameter before it is set, or (3) modify an `in` formal parameter. In languages where these constraints can be expressed, the proxy compiler generates appropriate code as part of the interface definition. For modules implemented in languages such as C, the programmer is responsible for verifying these constraints. Alternatively, one could devise language specific type checkers that warn about potentially location dependent code.

4.3.1.3 Failure Handling

We next consider the issue of invocation failure handling. Our approach is to give the interface designer some flexibility. One choice is to reserve a special result value for each operation that indicates an invocation failure. The pitfall of this method is that the programmer has to include a test for failure after each invocation. Alternatively, a module interface specification can require each client object to support an *upcall* interface that includes an operation for failure notification. With this approach, clients can handle failures as exceptional conditions, and no explicit test is necessary after an invocation.

Finally, the interface designer is free to provide no failure notification at all in an interface. When an invocation fails, the system takes the default action of terminating the client's protection domain. This method effectively eliminates the potential for independent failure of client and server, and thus obviates the need for failure handling. Note that this may be a reasonable approach for many modules whose access scope is limited to the same physical machine. In general, location transparent invocation introduces complexity in failure handling, due to the potential for independent failure of modules. We believe that our design provides some flexibility, and minimizes the impact of this issue on the feasibility of fine-grained decomposition.

4.3.1.4 Performance

The final issue is that of non-local invocation performance. Our invocation mechanism relies on a configurable RPC service, which allows the dynamic substitution of the most appropriate RPC mechanism. As a consequence, RPC protocols can be used that take advantage of locality or special hardware support. In current hardware architectures, non-local invocations, even on the same machine, are an order of magnitude more costly than a procedure call. Thus, there is an inherent tradeoff between fine-grained protection and performance. Lipto's location transparent invocation system provides flexibility in dealing with this tradeoff. First, the dynamic configuration of RPC services allows the use of protocols that provide performance close to the architectural limits in each case. Second, proxy objects can employ caching mechanisms to reduce the performance hit of remote invocations. Third, by decoupling modularity and protection, the architecture allows the adjustment of the granularity of protection according to the needs of a particular installation and its users.

4.3.2 Object References and Local Invocation

Now consider the implementation of object references and local invocation. From the module/object implementor's perspective, all invocations are local; either the invocation is to a local server object, or to a proxy object. Thus, the implementation of object references and the conventions for local invocation entirely define the interface between module implementations and the module/object infrastructure.

An object reference is a pointer to a structure that represents the state of the referent object. This state structure is object implementation dependent. The architecture defines only the first member: A pointer to a table of $< functionpointer, offset >$ tuples, one for each operation that the object supports, indexed by the operation number. The $functionpointer$ refers to a function that implements the operation. The $offset$ is added to the value of the object reference, and the resulting value is passed as a hidden first argument to the function.

Object implementations are responsible for generating the operation table at compile time. To perform an invocation, the appropriate tuple is fetched from the table using the operation number as an index. Then, the offset is added to the value of the object reference and passed to the function, along with the operation's explicit arguments.

Our implementation of object references and invocations is identical to the way many C++ compilers implement object references and invocations. This is not accidental: Since C++ is our primary implementation language for system modules, and we are using the GNU C++ compiler, it was convenient to choose our convention to conform to this compiler's implementation. This allows modules implemented in C++ to invoke objects implemented in other modules as if they were C++ objects.

4.3.3 Non-Local Invocation

Whenever a reference is created to a non-local object, a *client proxy* object is instantiated in the local domain, and a reference to the proxy object is returned to the client. This proxy object supports the appropriate interface, and is responsible for forwarding invocations to a *server proxy object* in the callee object's protection domain. Upon receiving an invocation, this server proxy in turn invokes the callee object and passes any results of the invoked operation back to the client proxy, which eventually returns control to the calling object. The invocations from caller to client proxy, and from server proxy to callee are handled as local invocations, as described above. The caller is unaware of the fact that the callee is remote, just as the callee is unaware of the fact that it is being invoked by a remote object. The pair of proxy objects forward invocations between client and server object using a

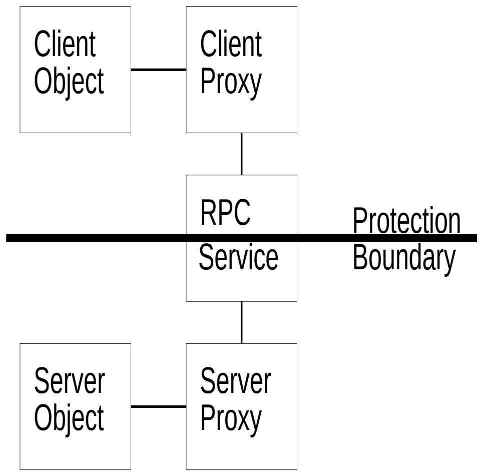

Figure 4.2: Proxy Objects and RPC

remote procedure call service. The specific RPC service used depends on the location of the server object with respect to the client object.

Figure 4.2 illustrates the basic components involved in a cross-domain call. Note that the "protection boundary" in this figure could correspond to a domain boundary as well as a machine boundary. The RPC service module may rely on a lower-level communication service to accomplish its task.

It is useful to isolate the orthogonal issues involved in location transparent object invocation. We have identified three such issues: (1) distribution hiding, (2) data transfer, and (3) control transfer. These three concerns are explicitly separated from one another in our implementation. First, distribution hiding is the responsibility of the proxy objects. Second, the data transfer mechanism is hidden inside the abstract data type IOData. The IOData data type encapsulates data as it is passed between the client proxy and server proxy. Finally, control transfer is handled by a set of highly specialized RPC services: a new protocol called *user-kernel RPC* (UKRPC) for the case of a user-level object invoking a kernel object; a new protocol called *kernel-user RPC* (KURPC) for the opposite case; LRPC and URPC protocols for invocations among user-level objects on the same machine; and Birrell-Nelson RPC for cross-machine invocations. The following subsections consider the three components of location transparent object invocation, in turn.

4.3.3.1 Distribution Hiding and Proxy Objects

The client proxy and server proxy hide distribution and communication issues from the client and server objects. The proxy pair must perform several tasks: (1) binding and authentication; (2) failure handling; (3) object reference management; (4) argument/result marshaling; (5) argument/result validation; (6) performance enhancement (caching); and (7) replication management.

The proxy object implementations generated from an interface definition by the proxy-compiler handle the first five tasks. Tasks (6) and (7) are related to performance and fault-tolerance; handling of these issues is optional. The compiler-generated proxy object implementations can be manually augmented to employ performance improving techniques such as caching, and reliability-related techniques such as transparent server replication based on an underlying group communication service.

Binding and authentication are performed during the instantiation of a pair of proxy objects; each proxy is passed an *address object* that refers to its peer. The address object encapsulates an RPC address and authentication credentials; it is used by the proxies to establish a connection.

Object reference management (3) is concerned with the mechanisms and policies used to translate an object reference passed as an argument/result, into a reference that is meaningful in the recipient's domain. There are subtle issues concerning references and object identity, the details of which are omitted for the sake of brevity; for a discussion of these issues, see [JLHB88]. Lipto avoids overheads associated with reference translation by allowing more than one proxy object (in the same domain) to represent a given remote object. This approach makes it harder to determine object identity, but our experience suggests that operations whose performance suffer from this are few and infrequently invoked. Alternatively, Lipto's proxy compiler could be extended to generate proxies that employ different policies for each reference type, according to information in the service class definition.

Tasks (4) and (5) require special attention because they do not have to be performed in all cases. For example, data presentation is not an issue when the invocation is between objects that reside on machines of the same architecture; argument/result validation can be omitted if they are supplied by an object in a trusted domain; and, argument/result marshaling can be simplified in the case of an invocation between objects on the same machine. Our compiler-generated proxy objects take advantage of these optimizations and perform these functions only when required.

4.3.3.2 Data Transfer and the IOData Type

The data transfer component of object invocation is handled in the implementation of the abstract data type IOData, which was introduced in Section 2.5. An IOData instance contains raw (untyped) data; its implementation avoids physical copying of data whenever possible. Abstractly, an IOData object is a string of untyped, immutable data. It supports a number of operations such as concatenate, fragment, truncate, prepend and append, to facilitate copy-free processing of the data by network protocols. Users of an IOData object cannot access its data directly; data must be explicitly embedded into and retrieved from the object. The implementation of the IOData type is integrated with the *fbuf* facility described in Chapter 5, which permits efficient data transfer across protection domain boundaries.

4.3.3.3 Control Transfer

We now turn our attention to the various RPC protocols used to transfer control across protection boundaries. UKRPC implements the case of an object in a user-level address space calling an object in the kernel address space. UKRPC implements a simple system call disguised as an RPC service. KURPC is responsible for handling calls from the kernel address space to an object in a user-level address space. Thus, it implements what is commonly called a user-space *upcall*, again disguised as an RPC service. What has become known as *light-weight* RPC (LRPC) [BALL90] is in our implementation simply a combination of UKRPC and KURPC.

For cross-machine invocations, we use a decomposed version of the conventional Birrell-Nelson RPC protocol [BN84, OP92]. This RPC service is implemented as three independent modules. Notice that this service, while it is used as a communication service for object invocation, could itself be configured with a domain boundary between any two component modules. In fact, the communication service used between two proxy objects can be arbitrarily complex. Consequently, the client and server proxy can transparently span heterogeneous systems and networks.

4.4 Performance

We ran a series of tests to evaluate the performance of our object invocation mechanisms. In each test, we measured the elapsed time for $1,000,000$ object invocations performed in a tight loop. From this number, we subtracted the elapsed time for $1,000,000$ iterations of an empty loop to compensate for the cost of the loop, and divided the result by $1,000,000$.

Most of the tests were run on a Sun 3/75, which uses a MC68020 microprocessor running at 16.67MHz. The processor has a 256 byte on-chip instruction cache. Our measurements suggested that in the test case for the invocation of an object within the same address space, all instructions of the test loop were held in the instruction cache (i-cache) after the first iteration. In all other test cases, the instructions of the test loop exceeded the capacity of the i-cache. In order to make the numbers comparable, we included code into the test loop that ensures that every iteration starts with a cold i-cache.

Although Lipto has not been ported to RISC-based machines, we have measured individual components of our invocation mechanisms on both a Sun SPARCstation 1 (SPARC processor at 20 MHz) and an IRIS 4D/25 (MIPS R2000 at 20MHz). We then calculated the expected performance of the simplest invocation cases from these measurements. Note that the measurements on the RISC machines were made with warm caches, which seems reasonable given their larger instruction caches. The following reports on four separate experiments.

First, we measured the round-trip cost of a local (intra-domain) object invocation, which corresponds to a C++ virtual function call. Table 4.1 quantifies the overhead imposed by Lipto's location transparent object invocation in the local case by comparing a local invocation to an ordinary function call. The numbers presented in this table are in microseconds, and were measured using the GNU C++ compiler, version 1.37.1.

The overhead is mainly due the indirection caused by the late binding, and the fact that the object pointer is passed as an argument. Note that our version of the GNU compiler does not use register windows on the SPARC. Using register windows, we suspect that the invocation cost on the SPARC would be within twice the cost of a function call, as is the case on the other processors. This cost—paid for location transparency and late

	MC68020	SPARC	MIPS R2000
function call	4	0.3	0.5
local invoc.	7	0.9	0.8

Table 4.1: Local Call Performance (μsec)

	MC68020	SPARC	MIPS R2000
System Call	26	15.2	9
No Arguments	57	22	15
One IOData Argument	51	19	13
Two `int` Arguments, `int` result	69	24	16

Table 4.2: UKRPC Performance (μsec)

binding—is small enough not to stand in the way of a fine-grained decomposition of the system.

Second, we measured the round-trip cost for an object in a user address space calling an object in the kernel address space (UKRPC). The results for various arguments and results are given in Table 4.2. The row labeled *system call* shows the cost of a kernel trap plus invocation of an empty C function in the kernel address space for each of the machines. It provides a reference value to determine the added overhead of a UKRPC invocation. Note that an invocation with one IOData argument is faster than a call with no arguments. This is because no argument message has to be created when one of the arguments is of type IOData; this IOData object is used as the argument. Also note that the cost of such an invocation is independent of the size of the IOData object. This is because no data is being copied.

We found that on the MC68020, a user-to-kernel object invocation costs approximately twice as much as a simple system call, but on a RISC processor, the additional penalty is on the order of only 33 to 50 percent.

Third, we measured the round-trip cost of a kernel-to-user object invocation with various arguments and results. The results are presented in Table 4.3. One observation is that upcalls are an order of magnitude more expensive than their counterpart downcalls. This is in part because of the lack of hardware support (i.e., a counterpart to a system trap) available on the Sun 3/75. Note that by combining UKRPC with KURPC, one gets a user-to-user RPC (LRPC) time of approximately 244 μsec in the null argument case, which is comparable to LRPC times reported elsewhere in the literature [BALL90].

Finally, we measured the round-trip performance of Lipto's network RPC service suite. This is interesting for two reasons. First, this suite implements one of the three basic RPC cases identified in Section 3. Second, our network RPC is in fact implemented by a stack of modules that invoke each other using Lipto's location transparent invocation mechanism. We have explicitly compared the performance of this service suite with a corresponding suite implemented in the x-kernel [HP91], which does not provide a location transparent invocation mechanism. As illustrated in Table 4.4, the performance of this suite in the

	MC68020
No Arguments	187
One IOData (128 bytes) Argument	240
Two `int` Arguments, `int` result	247

Table 4.3: KURPC Performance (μsec)

two systems is almost identical. Note that KURPC and UKRPC were used to cross the user/kernel boundary, which in this example was between an application object and the topmost module of the RPC protocol stack. The numbers reported are in milliseconds.

	Lipto RPC	x-Kernel
0 Bytes	2.6	2.6
128 Bytes	2.9	2.9
1k Bytes	5.2	5.1
2k Bytes	11.8	11.7
3k Bytes	15.0	14.8
4k Bytes	17.0	16.7
8k Bytes	29.2	29.2
15k Bytes	50.0	50.7

Table 4.4: NRPC Performance (msec)

In summary, Lipto's location transparent invocation mechanism is at most twice as expensive as a statically bound invocation mechanism. Moreover, a "macro" experiment comparing Lipto to the x-kernel suggests that this increased cost on the invocation mechanism has little effect on the performance of the system as a whole.

4.5 Discussion

The distinction between modularity and protection has often been blurred in operating system design. This implicit merging of modularity and protection, in turn, puts stress on how interfaces get designed, and how they evolve over time. This section looks at modularity and protection from a historical perspective, makes several comments about interface design, and relates our work to other object systems.

4.5.1 Modularity and Protection

A review of previous operating system architectures indicates that modularity and protection have always been implemented by a single mechanism, that is, they have always been tightly coupled. At one end of the spectrum, several systems have employed capabilities as their modularity/protection mechanism. Capabilities may either be implemented in hardware as in the Cambridge Cap [WN80, Yud83] and the Intel 432, or in software on top of standard paged virtual memory hardware as in Hydra on C.mmp [WCC$^+$74, WLH81]. Whether implemented by hardware or software, such systems offer fine grained modularity/protection, but limited flexibility since a single mechanism is used for both functions. Interfaces may be defined by the programmer, but every invocation through an interface requires crossing a protection boundary as well, resulting in poor performance.

At the other end of the spectrum, UNIX [RT74] is an example of a monolithic system. Both modularity and protection are supported only at a very coarse grain by the distinction of kernel and user contexts, where a single fixed invocation interface provides access to all system services. All services provided by the operating system are implemented in a single protection domain, and thus cannot be protected from each other. Moreover, the code contained in this single address space uses shared/global data, and thus provides almost no encapsulation. In such systems, both modularity and protection are sacrificed for speed.

The trend towards the use of UNIX as a standard operating system platform has stressed its design from two directions. First, new application domains have created a constant demand for new, specialized system services. Second, the increasing variety of hardware platforms to which the system is ported has made it necessary to modify and extend the system in major ways. This has caused a dramatic increase in the size and complexity of the systems, while at the same time its portability, reliability and efficiency have suffered.

In reaction to the maintenance problems caused by monolithic systems, and in order to provide more flexibility for implementing new kinds of services, several server-based systems are being implemented. These system are characterized by a small kernel or *microkernel* which provides only basic services, and multiple servers which provide all other services. Examples of such systems include Chorus, Topaz, and Version 3 of Mach [ABB$^+$86, RAA$^+$88, TSS88]. Server-based systems attempt to find a balance between modularity and performance by supporting coarse-grained protection and modularity, with the granularity at the level of the address space. Each service is allowed to define its own interfaces, which are implemented on top of a small, fixed set of interfaces to the kernel's primitive communication services. However, crossing an interface always requires crossing a protection boundary. As a result, the granularity of modules is rather coarse. While the system can be extended *horizontally*, i.e. by adding servers which rely on the kernel, it cannot readily be extended *vertically*, i.e. by adding servers which rely on the services provided by other servers, because communication costs are a concern. Finally, the kernel defines a fixed set of functions which are implemented in the kernel domain. Designers of microkernels have been debating for years in an effort to determine this set, striking a balance between concerns to keep the kernel small, and concerns for efficiency, which call for the inclusion of more functionality into the kernel domain. We believe that such a fixed set of functions which will satisfy all applications and allows the system to perform well on all hardware platforms does not exist. The problem is again that the idea of a module, the microkernel in this case, is associated with a protection domain, the kernel domain.

While modularity and protection have been coupled in previous operating system architectures, this coupling is not essential. From the modular decomposition of a system we hope to obtain software engineering advantages by limiting the knowledge that each module has of its surroundings. Such advantages include configurability, location transparence, and failure localization. In order to support modularity it is sufficient to be able to separate the interface of a module from its implementation, and to encapsulate modules so that they do not depend on shared data. Language support for this encapsulation is desirable, but not necessary. Encapsulation can rely on a weakly typed language, as is the case with Lipto which uses C++.[2] In fact, encapsulation can be based entirely upon programming conventions, as long as it is possible to reconfigure a given module into an isolated address space so that its encapsulation is enforced in hardware, if that is necessary.

Protection provides for failure isolation so that a failure in one component of a system cannot adversely affect the operation of another correct component. Such failure isolation is desirable whenever two components of a system enjoy different levels of trust, such as between two user components or between user and system components of a general-purpose operating system. The recent trend to server-based systems provides evidence that such failure isolation may also be desired between system components. Protection requires implementing domains (contexts, address spaces) that are separate from each other, as well as safe mechanisms for cross-domain communication.

A long held myth claims: "there is a fundamental tradeoff between modularity and performance." The programming language community has enjoyed some success in dispelling this myth with the introduction of languages that explicitly support modularity, either in the form of abstract data types as in CLU or Ada, or objects as in C++ or Eiffel. One key to the success of these languages is the observation that modularity can be enforced at compile time instead of at run time.

In operating systems, since modularity and protection have always been implemented using a single mechanism, the myth has been more difficult to dispel. As long as crossing a module boundary implies crossing a protection boundary, it will be true that modularity can be obtained only at the expense of performance. A major contribution of this work is the observation that the separation of modularity and protection allows their enforcement to be similarly separated.

4.5.2 Interface design

In traditional operating system architectures, all operating system services are accessed through a single, general, static system call interface. We believe that the traditional system call interface designs have three major flaws. First, they are only available at protection boundaries. This implies that it is difficult to layer new services atop existing ones. Second, it is difficult to modify the interface as the system evolves. Additions to the interface can be handled in a backwards compatible way, but modifications to the interface, or the introduction of alternate implementations of a service (e.g., `signal(2)` vs. `sigset(2)`) are problematic. The addition of new hardware to the system, which would most naturally be expressed by the addition of a new service interface, must be handled by less structured means such as the UNIX `ioctl` call. Third, it is difficult to provide direct access to lower level services. Naive applications require insulation from the

[2]By weakly typed, we mean that type safety is not strictly enforced in all cases.

details of the hardware on which they are executing, but sophisticated applications (such as databases and real-time audio and video) require low level access to the hardware in order to achieve acceptable performance.

An implementor of a database system, for example, does not want the underlying filesystem to perform buffering and prefetching, because these mechanisms are likely to interact negatively with the database's own caching mechanisms. Consequently, one would like the filesystem to be accessible through a low-level interface for use by sophisticated applications, in addition to the standard, high-level interface provided for naive applications. Similarly, databases often do not perform well on top of the virtual memory/process abstraction; they could benefit from a low-level interface to the memory and process management system that allows them to participate in page replacement and thread scheduling decisions [Duc89].

One of the innovations of the UNIX operating system is its uniform treatment of files and devices; a general, uniform interface (open, close, read, write, seek) is provided for such access. Together with the convention for the usage of the file descriptors stdin, stdout and stderr, it is the key to the composability of simple filter-like applications. This UNIX file interface is an example of what would be called a *service class* in Lipto. The problem, however, is that in UNIX access to lower level file system features and features that are specific to certain devices are only accessible through awkward, unstructured interfaces such as the `ioctl` and `fcntl` system calls.

The BSD UNIX *socket* interface, on the other hand, was designed as a general interface for access of network services at different levels. Its generality, however, not only makes it difficult to use, but also leads to paradoxical cases where the use of a lower level network service is more expensive as the use of a higher level service. As an example of this, in SunOS 4.0, the protocol suite UDP/IP/ETH performs better than the suite IP/ETH [HP91].

The separation of protection from modularity in Lipto encourages the modular decomposition of the system services, allowing and requiring the definition of many more interfaces. Each kind of service (File service, Network communication service), and each level of service of the same kind (Datagram, Stream, Remote procedure call) defines its own set of interfaces in the form of a service class. An application simply accesses a service at the desired level, using the appropriate service class. Thus, a naive application that has very general requirement for the type of services it is using, uses a general, high-level service class. A sophisticated application, on the other hand, accesses its services using lower-level, more specific service class.

It is not sufficient, however, to merely define and implement the mechanism that allows the addition of new interfaces. First, some conventions are required that prevent a proliferation of interfaces. Note that the software engineering benefits that come with a modular, composable system are maximized when the number of interfaces is relatively small, and the number of modules that support the same interface is large. A decision to define a new interface must be based on the determination that no existing interface can be used to access a new service in a natural way. This should only happen when the new service actually is of a new kind, or makes use of a new technology.

Second, the architecture must provide support for managing the interface space. Lipto allows the definition of a hierarchy of service classes. Although Lipto initially defines a flat set of service classes, our architecture is general enough to allow the definition of a hierarchy of service classes based on the subtyping principle. Thus, we can support the case where future technologies require the addition of new operations to the interfaces of an existing service class, while backward compatibility is required with respect to existing services and applications that use the service class.

4.5.3 Object Systems

Note that although Lipto's module/object infrastructure and currently almost all its system modules are implemented in C++, module implementors can in principle use any implementation language; i.e., Lipto is language independent from the module implementor's perspective. This is different from language-level object systems such as Emerald and Hermes [JLHB88, Str90], which support location-transparent objects in a single-language environment.

In the case where other object-oriented languages are used, our design does not impose an object model for language level objects. Moreover, Lipto does not provide an object management layer that supports the creation, migration and storage of such objects. Instead, it is our policy that distributed applications should be either implemented in a language that supports distributed and persistent objects, or use an object-support layer on top of the basic system services which provides these services. This is an important feature that makes Lipto different from other object-oriented systems, such as SOS [SGH$^+$89]. In other words, Lipto is an operating system that uses an object-oriented structuring technique internally, it is not an object-support system.

4.6 Related work

The FLEX project [C$^+$93a] provides support for decomposing existing operating system implementations using the approach of decoupling modularity and protection. The Spring operating system [HK93, HPM93] uses a location-transparent object model. However, object invocations are more complex and less efficient than Lipto's. Consequently, Spring's objects are more coarse-grained than Lipto's modules. Psyche [SLM90a, SLM$^+$90b] has separate abstractions for modularity and protection to facilitate parallel programming in multiple models.

The Chorus operating system allows its servers (actors) to be collocated with the microkernel in the same protection domain [RAA$^+$88, RAA$^+$92]. This results in increased system performance due to reduced communication costs. The same capability was added to a version of the Mach 3 kernel [LHFL93]. There are no provisions for collocating servers in user-level protection domains and the coarse-grained servers comprise entire subsystems, as is the case in conventional microkernel-based systems.

Software fault isolation [WLAG93] has been proposed as an alternative to hardware enforced protection between mutually trusting software modules. It provides fault isolation—but not data privacy—at a moderate performance cost. This technique is complementary to Lipto's decoupling of modularity and protection.

Clouds is a distributed operating system that is, like Lipto, based on the object-thread model [DLJAR91]. However, a Clouds object is persistent and resides in its own protection domain. Consequently, Clouds objects are "heavy-weight" and they do not support fine-grained decomposition. Choices is an object-oriented operating system [CRJ87]. It has a fine-grained, modular structure based on the encapsulation, subtyping, and inheritance mechanisms of its implementation language, C++. However, the entire system is contained in the kernel domain. The system's application interface is exported to user-level domains using proxy objects, but the internal interfaces are strictly local.

4.7 Concluding Remarks

This chapter makes three contributions. First, it articulates the principle of decoupling modularity and protection domains. We believe this principle is critical to the successful realization of modular operating systems. In particular, it allows increased modularity at a smaller cost, improves scalability, portability and distributability, and permits vertical extensions of the system through layering of services. Second, it describes how modularity and protection are decoupled in Lipto, an experimental operating system. Third, it describes in some detail the invocation mechanism used by Lipto. Our analysis of this mechanism suggests that efficient domain-independent invocations are possible.

At a more detailed level, the cross-domain invocation mechanism is optimistic, i.e., it is tailored to local (intra-domain) invocation, which is the most common case. Consequently, it uses local object references and closely resembles procedure call semantics. The mechanism is partitioned into three distinct concerns: distribution hiding, data transfer, and control transfer. Proxy objects implement distribution hiding, independent of the underlying mechanisms for data and control transfer. Control transfer is handled by an appropriate RPC protocol. The data transfer mechanism is hidden inside the IOData abstract data type, which is used to pass invocation arguments and results between client proxy and server proxy. The appropriate invocation mechanism for each situation can be dynamically configured by choosing an appropriate RPC protocol for control transfer.

CHAPTER 5
High-Bandwidth Cross-Domain Data Transfer

The previous chapter introduced a novel operating system structuring approach that achieves fine-grained modularity without the cost of additional protection boundaries and server processes. However, in any operating system that supports protection and security, some protection boundaries necessarily remain. In current microkernel-based operating systems, many protection boundaries may occur in the I/O data path. As outlined in Chapter 2, 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 API and buffer manager to be effective. This chapter presents a novel OS facility for the efficient management and transfer of I/O buffers across protection domain boundaries.

5.1 Motivation

Optimizing operations that cross protection domain boundaries has received a great deal of attention recently [Kar89, BALL90, BALL91]. 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 work 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.

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. As outlined in Chapter 2, 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.

This chapter 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

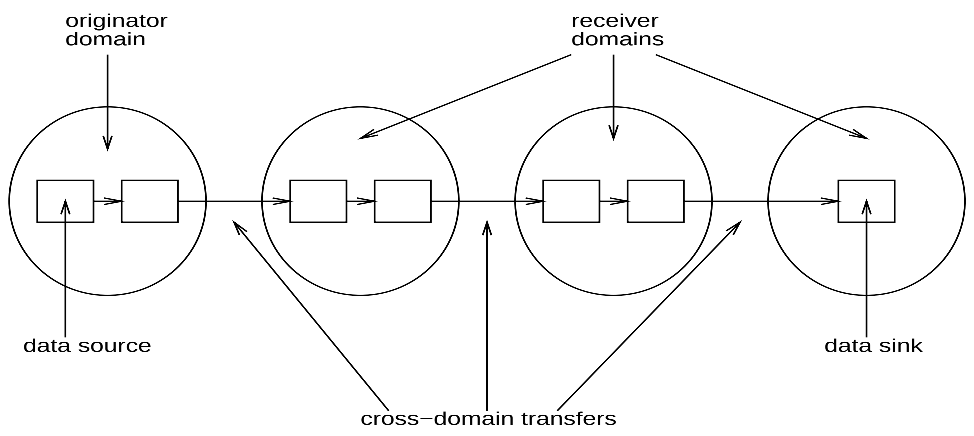

Figure 5.1: Layers Distributed over Multiple Protection Domains

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).

5.2 Characterizing Network I/O

This section outlines the requirements for a buffer management and cross-domain data transfer facility by examining the relevant characteristics of network I/O. The discussion appeals to the reader's intuitive notion of a data buffer; Section 5.4 defines a specific representation.

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 5.1 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 5.6 shows how different transfer mechanisms perform in the two domain case, and Section 5.7 discusses the larger issue of how many domains one might expect in practice.

5.2.1 Networks and Buffers

On the input side, the network adaptor 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

[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.

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 application 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.

5.2.2 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 5.1 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 consequence, 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) 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 [Mog92] 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.

5.2.3 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 with 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.

5.2.4 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 5.4 gives the design of a cross-domain transfer facility that supports (exploits) these requirements (restrictions).

5.3 Related Work

Before we proceed to describing a facility that achieves the goals outlined in the previous Section, we first review related work on the subject of cross-domain data transfer.

5.3.1 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 [Che88, TA91] support *page remapping*, while Accent and Mach support *copy-on-write* (COW) [FR86, ABB+86]. 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 [TA91]. 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 [JZ93] 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.

5.3.2 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 [SB90]. 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 [BALL90] 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 [Jac90, DWB$^+$93]. 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.

5.4 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.

5.4.1 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 is typically 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 the IOData type introduced in Section 2.5, the x-kernel *messages* [HP91], and BSD Unix *mbufs* [LMKQ89]. 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 5.2.

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

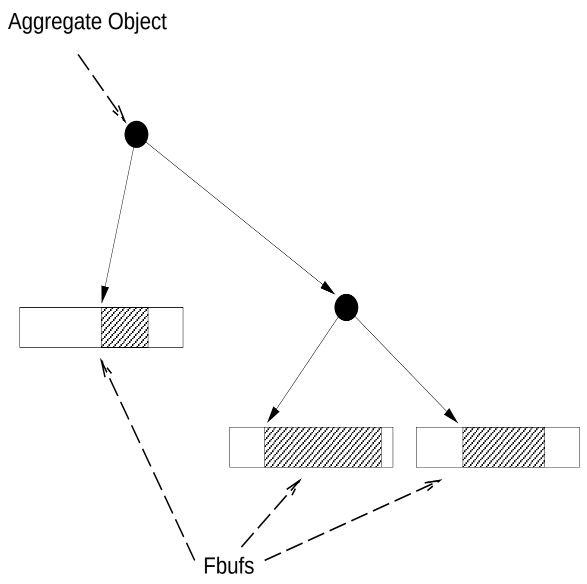

Figure 5.2: Aggregate Object

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.

5.4.2 Optimizations

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

5.4.2.1 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.

5.4.2.2 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), (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. The optimization requires that the originator is able to determine the I/O data path at the time of fbuf allocation.

5.4.2.3 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.

5.4.2.4 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 5.2.3, 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.

5.4.2.5 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 [BALL91], and other highly optimized IPC mechanisms such as MMS [GA91].

5.5 Implementation

We have implemented and evaluated fbufs using an experimental platform consisting of CMU's Mach 3.0 microkernel [ABB$^+$86], augmented with a network subsystem based on the University of Arizona's x-kernel [HP91]. Mach is a portable, multi-processor capable microkernel that provides virtual memory, local message passing facilities, and device access. The x-kernel is a framework for the implementation of modular, yet efficient communication protocols.

The x-kernel based network subsystem consists of a *protocol graph* that can span multiple protection domains, including the microkernel. Proxy objects are used in the x-kernel to forward cross-domain invocations using the Mach IPC facilities. The x-kernel supports an abstract data type that employs lazily evaluated buffer management; it uses a DAG-based representation similar to the one illustrated in Figure 5.2. The IOData abstract data type described in Section 2.5 is an extended version of the x-kernel's buffer ADT. All applications, network protocols and device drivers deal with network data in terms of the IOData abstraction.

Incorporating fbufs into this software environment required the following changes: the Mach microkernel was modified to provide virtual memory support for fbufs, the x-kernel buffer ADT was replaced with the IOData abstraction, implemented on top of fbufs, the proxies were modified to use the fbuf data transfer facility, and minor modifications were made to device drivers and other software modules that allocate IOData buffer instances. These changes resulted in a fully functional system.

5.5.1 Allocation

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. User domains initially have no access permissions to the fbuf region, while the kernel has permanent unrestricted access to the entire region. Upon request, the kernel hands out ownership of fixed sized chunks of the fbuf region to user-level protection domains, and grants read/write access for the chunk to the owning 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. There is a generic allocator that allocates uncached fbufs, and per-datapath allocators that allocate cached fbufs. 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. Consequently, most fbuf allocations can be satisfied without kernel involvement.

Since fbufs are pageable, the amount of physical memory allocated to fbufs depends on the level of I/O traffic relative to other system activity. Similarly, the amount of physical memory allocated to a particular datapath'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, which minimizes paging activity. 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.

The IOData abstract data type is implemented using a DAG structure, with interior nodes and buffer fragments placed into fbufs obtained from the appropriate fbuf allocator. The nodes and buffer fragments of a single IOData instance are clustered onto fbufs to reduce the number of pages occupied by a single DAG.

5.5.2 Transfer

The fbuf transfer mechanism comes into play when an IOData buffer is passed as an argument in a cross-domain invocation. The proxy in the calling domain inspects the buffer's fbuf allocator. If the allocator produces cached fbufs and fbufs from this allocator have been passed to the destination domain before, then no virtual memory mapping changes are necessary to transfer the fbuf DAG that represents the buffer. In this case, a reference to the root node is sent in a simple Mach IPC message. Otherwise, the Mach IPC message is specially tagged as carrying a fbuf DAG reference.

The Mach kernel's IPC mechanism intercepts such tagged messages and invokes fbuf-specific virtual memory extensions to perform the appropriate mapping changes. If the IOData's fbufs are cached, then the mappings are changed for *all* of the allocator's fbufs. This ensures that all fbufs maintained by a cached fbuf allocator always have the same mappings. Consequently, inspecting an fbuf's allocator suffices to determine whether mapping changes are required to transfer the fbuf. For uncached fbufs, mappings are only changed for the fbufs reachable in the transferred fbuf DAG.

The proxy in the receiving domain creates a new instance of the IOData buffer type. This instance is initialized to refer to the fbuf DAG received from the calling domain.

5.5.3 Deallocation

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

	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 5.1: Incremental per-page costs

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. Deallocation notices that refer to uncached fbufs are tagged, since mappings need to be removed by the VM system as part of the deallocation.

5.5.4 Domain Termination

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.

5.5.5 Exceptional Cases

In response to a read access to a location of the fbuf region for which the offending domain has no access permission, the virtual memory system's fault handler maps a page that contains an empty message leaf node at the appropriate address. This is necessary to protect receivers of volatile fbufs from asynchronous modifications of an fbuf DAG by the originator, as discussed in Section 5.4.2.4. Upon illegal write access within the fbuf region, an access violation exception is raised in the offending domain.

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 fbuf lots that can be allocated to any per-datapath fbuf allocator.

5.6 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) [ABB$^+$86], augmented with a network subsystem based on the University of Arizona's *x*-kernel (Version 3.2) [HP91]. The hardware platform consists of a pair of DecStation 5000/200 workstations (25MHz MIPS R3000 CPU, 64/64 KB I/D caches, 16 MB of main memory), each of which was attached to a prototype ATM network interface board, called Osiris, designed by Bellcore for the Aurora Gigabit testbed [Dav91]. The Osiris boards were connected by a null modem, and they support a link speed of 622Mbps.

For all results reported in this section, each data point is the average of 10 trials, where each trial consists of sending 100,000 messages, after an initial warm-up period. In all cases, the 90% confidence intervals where within 5% of the average. The workstations were running in single-user mode.

The first experiment quantifies the performance of fbuf transfers across a single protection boundary. A test protocol in the originator domain repeatedly allocates an IOData buffer, 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 buffer, deallocates the buffer, and returns. Table 5.1 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 5.3. Unlike Table 5.1, 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 IOData buffer,

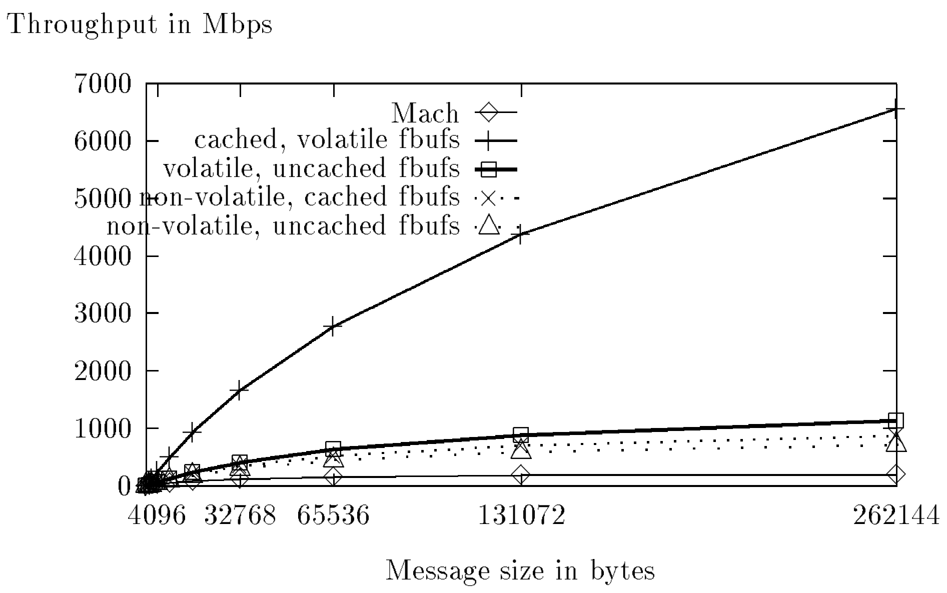

Figure 5.3: Throughput of a single domain boundary crossing

and sends it using a UDP/IP protocol stack that resides in a network server domain.[2] 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 5.4 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

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

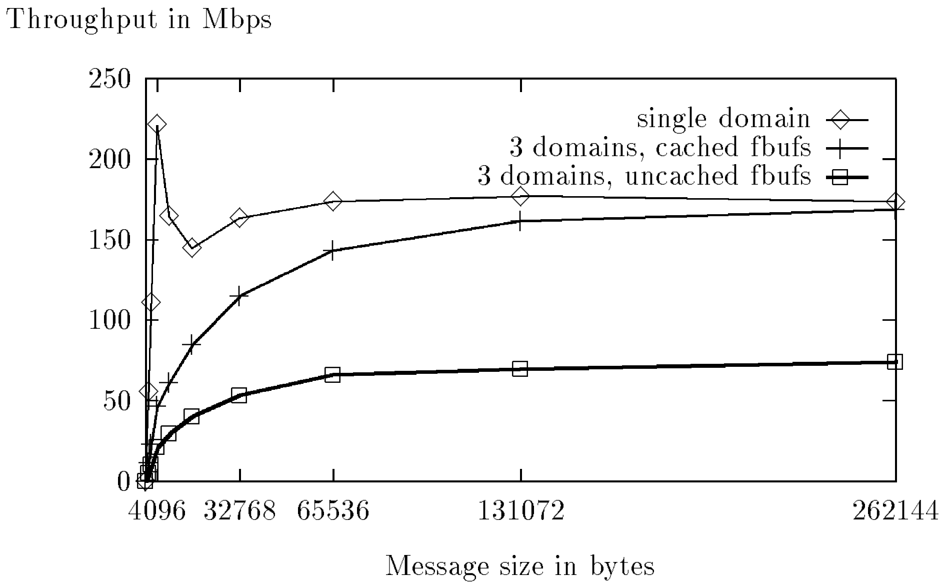

Figure 5.4: Throughput of a UDP/IP local loopback test

crossings. This is of practical importance since large messages are common with high-bandwidth applications. 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 5.5 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[3]. 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.

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

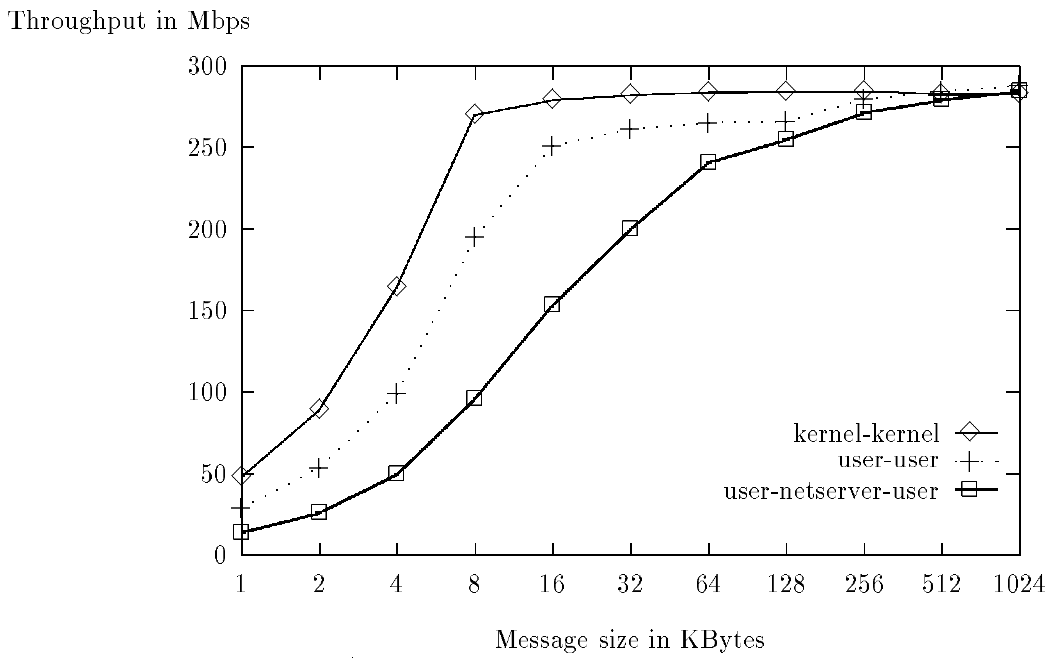

Figure 5.5: UDP/IP end-to-end throughput using cached, volatile fbufs

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 5.6 shows the measured end-to-end throughput when uncached/non-volatile fbufs are used.[4] As 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

[4]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 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.

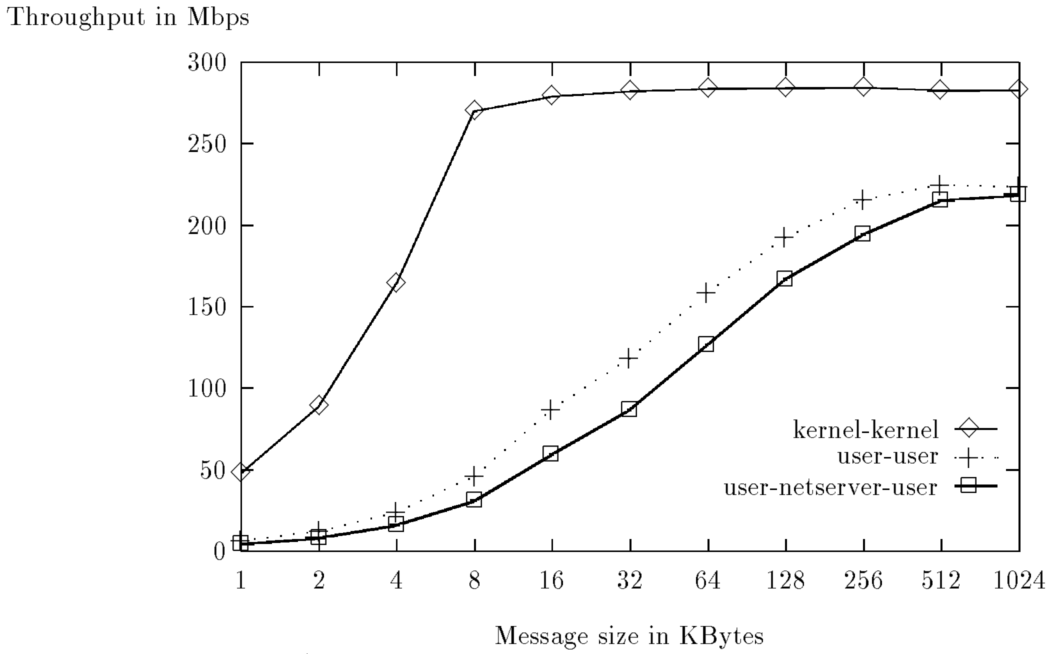

Figure 5.6: UDP/IP end-to-end throughput using uncached, non-volatile fbufs

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[5].

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

5.7 Discussion

In this section, we discuss several issues relating to the applicability and effectiveness of fbuf in various circumstances.

5.7.1 How Many Domains?

An important question not yet answered is how many domains a data path might intersect in practice. On 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 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 [MB93a, TNML93].

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.

5.7.2 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 5.2. 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 (see Section 2.5). 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 adaptor board performs reassembly of incoming PDUs from ATM cells by storing the cell payloads into a buffer in main memory using DMA. When the adaptor 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 adaptor, 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 adaptors do not have this capability, network adaptors 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 [DWB+93]) do have adequate support.

5.7.3 Architectural Considerations

As observed in Section 5.6, the performance of cached fbufs for large messages is limited by TLB miss handling overhead.[6] 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

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

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.

5.7.4 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 5.4.2.4) 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.

5.8 Concluding Remarks

This chapter 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 worst 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.

CHAPTER 6
Application Device Channels

The previous chapter described a technique that can 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 processor reallocation, scheduling, validation, and the resulting drop in memory access locality. As shown in Section 5.6, fbufs cannot eliminate the added latency caused by protection domain boundaries. In this chapter, we briefly 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 all protection boundaries from the network I/O data path.

6.1 Motivation

Consider how an application program generally interacts with the operating system and the underlying hardware. Conceptually, the OS is layered between the application program and the hardware. This layering, however, is not strictly preserved in the implementation. For efficiency reasons, application programs are allowed to directly access main memory and to directly execute instructions on the CPU and the floating-point co-processor (FPU). The operating system is only involved in relatively infrequent, "coarse-grained" operations such as resource allocation, scheduling, and I/O (See Figure 6.1).

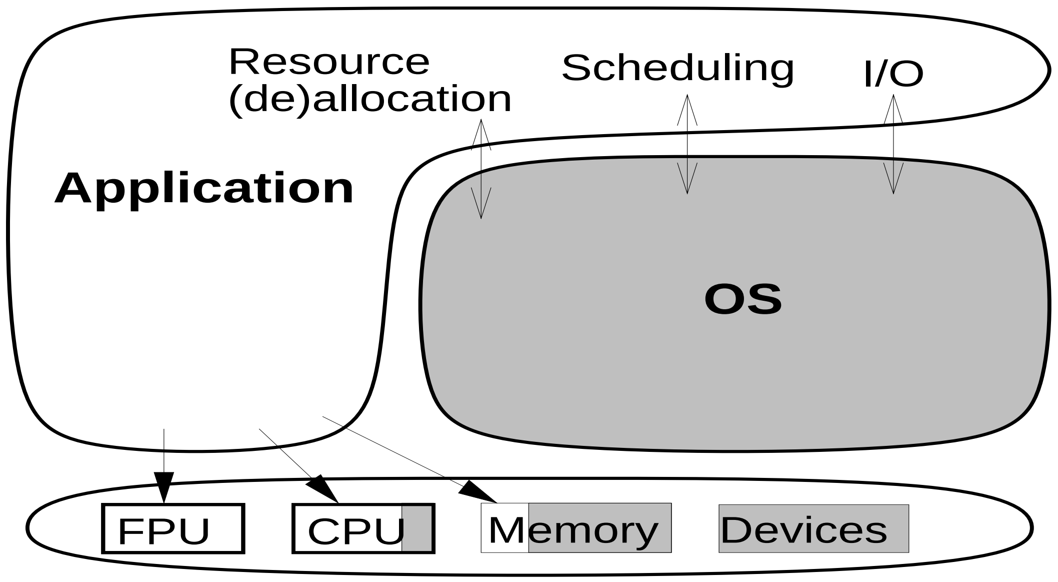

Figure 6.1: Application–OS–Hardware Interaction

Even though applications have direct access to hardware operations, a protected operating system maintains control of the machine in order to ensure a fair allocation of resources and to isolate application software faults. This is accomplished by limiting the set of memory locations that an application can access, by disallowing user access to devices, and by restricting the CPU instruction set available to applications. This is depicted by the shaded areas in Figure 6.1. For example, applications are not allowed to mask interrupts, since doing so could prevent the OS from re-gaining control of the machine when a user's time-slice has expired. By restricting an application's capabilities through hardware assist, the OS can maintain protection, safety, and fairness, while still allowing applications to perform the most common operations (i.e., computations in main memory) directly and without OS intervention. This strategy reflects a sound system design that isolates, and optimizes for, the common case.

Traditionally, I/O operations had inherently long latencies due to slow hardware devices (disks, slow network links). Additional software latencies due to OS kernel involvement where therefore acceptable. With 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 [KDCZ94, TMP94]. 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 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 received requests from application programs based on information provided by the OS during connection establishment.

6.2 Design

The basic approach taken in designing ADCs is depicted in Figure 6.2. 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 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 domain 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 network software linked with the application. These data structures include queues of data buffer descriptors for transmission and reception of network messages.

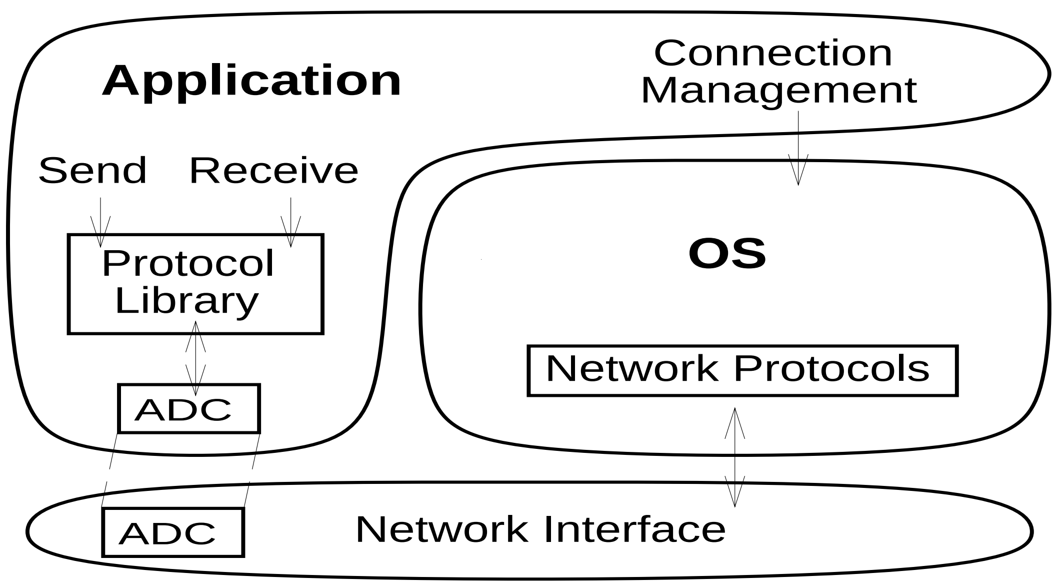

Figure 6.2: Application Device Channel

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 domain 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.

The use of user-level network software in place of a centralized, trusted network subsystem inside the OS kernel poses several problems. These problems and their solutions are described elsewhere in the literature [TNML93, MB93a]. The main issue is that many networks protocols have state that must be maintained on a per-host basis, such as the TCP/UDP port space. Fortunately, maintaining this state inside the OS is acceptable, since its access is not required as part of common send and receive operations.

There are two main advantages to the use of application device channels. First, network send and receive operations bypass the OS kernel. This eliminates protection domain boundary crossing, which would otherwise entail data transfer operations, processor reallocation, scheduling, 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.

6.3 Implementation

We have implemented ADCs in a Mach 3.0/x-kernel environment, using the OSIRIS ATM network adaptor described in detail in Chapter 3. Here, the OSIRIS transmit dual-port memory is divided into sixteen 4KB pages, each of which contains a separate transmit queue. The receive dual-port memory is similarly partitioned so that each page contains a separate free buffer queue and receive queue. One transmit queue, and one pair of free/receive queues are used by the operating system in the usual way. The remaining pages are grouped in pairs of one transmit and one receive page.

When an application opens a network connection, the operating system maps one pair of pages 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, application can only receive and transmit messages on connection 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 an 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, otherwise, an access violation exception is raised in the offending application process.

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 (see Section 3.2.2).

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 assigns 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.

6.4 Related Work

At first glance, ADCs may appear similar to the mapped devices used in Mach [RH91] and other microkernel-based systems. In these systems, the user-level UNIX server is granted direct access to, and control of, the network device. However, application device channels are different from mapped device drivers in two important ways. First, the OS kernel remains in control of the device in the case of ADCs; only certain kinds of access are granted to the application domain. Second, the device can be fairly shared among and directly accessed by a number of untrusted applications; the device is not mapped into—and therefore controlled by—a single domain, as is the case with mapped devices. With ADCs, the device can be shared by multiple application domains, rather than a single network server domain.

The network interface in the SHRIMP multicomputer [BLA+94] allows application access without sacrificing protection, as do ADCs. The complex, custom-designed interface supports communication based on remote virtual memory mapping, and it connects to a dedicated multicomputer interconnection network. An ADC, on the other hand, is a software mechanism implemented with minimal hardware assist from a general-purpose network adaptor (Section 3.8.2 discusses the requirements for a network adaptor to support ADCs). As such, it can support general TCP/IP internetwork access along with highly efficient message-passing traffic for parallel computing on workstation clusters.

The TMC CM-5 Connection Machine supports application-level communication through a memory-mapped network interface [L+92]. Unlike ADCs, the CM-5 maintains protection only between computing nodes in separate partitions, not within a partition (a partition is a subset of compute nodes configured to communicate).

6.5 Performance

To assess the performance of application device channels, we have repeated the experiments described in Section 3.9, except that the test programs were user-level applications, and the Mach kernel was modified to use ADCs. All results we obtained were within the error margins of the results reported in that section for the case of kernel-to-kernel message exchanges. That is, the latency and bandwidth of messages exchanged between application programs on separate hosts using ADCs was identical to that obtained when messages were exchanged between the OS kernels running on the hosts. This is significant, since it shows that ADCs can eliminate the performance penalty normally associated with crossing the user-kernel protection domain boundary.

6.6 Summary

The design of application device channels recognizes communication as a performance-critical operation for emerging distributed applications. ADCs allow applications to bypass the OS kernel during network send and receive operations. Operating system intervention is normally limited to connection establishment and termination. ADCs deliver network performance between application programs that is otherwise achievable only between OS kernels. They permit further performance improvements due to the use of application-specific, customized network communications software.

CHAPTER 7
Conclusion

The objective of the work presented in this dissertation is to avoid the I/O bottleneck in operating systems. This bottleneck results from the lack of main memory bandwidth in modern computer systems, combined with the poor memory access locality that current operating systems display during I/O activity. This dissertation introduces a set of new techniques that facilitate high-performance I/O without sacrificing the advantages of a modular OS structure.

7.1 Contributions and Limitations

The contribution of this dissertation is a set of set of novel techniques that facilitate high-performance I/O in both modular and monolithic operating systems.

- A new structuring approach for operating systems that allows fine-grained modularity without the performance penalty of an equally fine-grained protection structure;
- the fbuf facility for the management and transfer of I/O data buffers across protection domain boundaries;
- application device channels, which allow applications to bypass the OS kernel in common I/O operations, while leaving control over the device in the hands of the operating system; and
- a set of techniques that reduce host processing overhead and achieve high performance in driving a high-speed network adaptor.

These new techniques result from a re-evaluation of traditional approaches to supporting I/O in operating systems. In particular, they are based on the following key observations.

Memory bandwidth is a scarce resource. The traditional approach to performance oriented system design is to focus on minimizing the number of CPU instructions executed during an operation. The characteristics of modern computer systems with their fast CPUs, limited memory bandwidth and fast I/O devices make it necessary to focus instead on minimizing main memory traffic.

Communication is a common operation. Current operating system designs require OS kernel involvement in each I/O operation. This approach is well suited for applications that spend most of their time operating on data stored in main memory, and perform occasional, coarse-grained I/O operations. The kinds of distributed applications that are enabled by high-speed networking are likely to perform frequent, fine-grained I/O, making communication a common operation. Thus, operating systems should be optimized accordingly, and implement common I/O operations without requiring kernel involvement.

This dissertation focuses on the primary problem of preserving the bandwidth and latency of high-speed networks at the application level. The advent of high-speed networking, and the applications that it enables also pose a number of secondary problems. For example, live audio and video applications require that the operating system schedules resources in such a way that the real-time constraints of continuous media are met. This work does not directly address these issues, but complements efforts in that area [AH91, JH93, Jef92, RR93, LKG92].

The experiments used to evaluate the techniques developed in this work use network communication as the only form of I/O. Network traffic is arguably the most challenging form of I/O, since incoming data arrives at network devices asynchronously. Thus, we believe that with the exception of application device channels, the presented techniques readily generalize to other forms of I/O, such as disk traffic.

7.2 Future Directions

The techniques introduced in this dissertation resulted from a re-evaluation of traditional approaches to supporting I/O in operating system. This re-evaluation was necessary in light of technological changes in computer and network hardware. The most important of these technological advances—the advent of high-speed networks—is likely to enable an exiting new class of I/O intensive distributed applications. For these applications, communication is as common as computations on data in main memory. This differs significantly from traditional applications, where I/O operations are interspersed with long periods of computation on data in main memory.

This change in the nature of computing, from compute-centric, centralized applications to communication-centric, distributed applications suggest a fundamental re-thinking of the design and implementation of operating systems. Our long-term research agenda is to re-examine all the basic abstractions, services, and mechanisms provided by operating systems in light of a communication-centric view of computation.

REFERENCES

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

[AH91] David P. Anderson and George Homsey. A continuous media I/O server and its synchronization mechanism. *IEEE Computer*, 24(10):51–57, October 1991.

[AP93] Mark B. Abbott and Larry L. Peterson. Increasing network throughput by integrating protocol layers. *IEEE/ACM Transactions on Networking*, 1(5):600–610, October 1993.

[B$^+$93] G. Blair et al. A network interface unit to support continuous media. *IEEE Journal on Selected Areas in Communications*, 11(2):264–275, February 1993.

[BALL90] Brian Bershad, Tom Anderson, Ed Lazowska, and Hank Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[BALL91] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Transactions on Computer Systems*, 9(2):175–198, May 1991.

[BC89] R. Ballart and Y. C. Ching. SONET: Now it's standard optical network. *IEEE Communications Mag.*, 29(3):8–15, March 1989.

[Bel92] Bellcore, Morristown, NJ. *Asynchronous Transfer Mode (ATM) and ATM Adaptation layer (AAL) Protocols Generic Requirements, Technical Advisory TA-NWT-001113*, August 1992.

[BLA$^+$94] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathan Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 142–153, April 1994.

[BN84] Andrew Birrell and Bruce Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BP93] D. Banks and M. Prudence. A high-performance network architecture for a PA-RISC workstation. *IEEE Journal on Selected Areas in Communications*, 11(2):191–202, February 1993.

[BPP92] Mary L. Bailey, Michael A. Pagels, and Larry L. Peterson. The x-chip: An experiment in hardware multiplexing. In *IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, February 1992.

[C+91] Eric Cooper et al. Host interface design for ATM LANs. In *Proc. 16th Conf. on Local Computer Networks*, Minneapolis, MN, October 1991.

[C+93a] John B. Carter et al. FLEX: A tool for building efficient and flexible systems. In *Fourth Workshop on Workstation Operating Systems*, pages 198–202, Napa, CA, October 1993.

[C+93b] D. D. Clark et al. The AURORA gigabit testbed. *Computer Networks and ISDN Systems*, 25(6):599–621, January 1993.

[CB93] J. B. Chen and Brian Bershad. The impact of operating system structure on memory system performance. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, December 1993.

[Che88] David R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[CRJ87] Roy H. Campbell, Vincent Russo, and Gary Johnston. Choices: The design of a multiprocessor operating system. In *Proceedings of the USENIC C++ Workshop*, pages 109–123, November 1987.

[CT90] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the SIGCOMM '90 Symposium*, pages 200–208, September 1990.

[CW92] Richard Comerford and George F. Watson. Memory catches up. *IEEE Spectrum*, 29(10):34–57, October 1992.

[D+92] Todd A. Dutton et al. The design of the DEC 3000 AXP systems, two high-performance workstations. *Digital Technical Journal*, 4(4):66–81, 1992.

[DAPP93] Peter Druschel, Mark B. Abbott, Michael Pagels, and Larry L. Peterson. Network subsystem design. *IEEE Network (Special Issue on End-System Support for High Speed Networks)*, 7(4), July 1993.

[Dav91] Bruce S. Davie. A host-network interface architecture for ATM. In *Proceedings of the SIGCOMM '91 Conference*, pages 307–315, Zuerich, Switzerland, September 1991.

[Dav93] B. S. Davie. The architecture and implementation of a high-speed host interface. *IEEE Journal on Selected Areas in Communications*, 11(2):228–239, February 1993.

[DHP91] Peter Druschel, Norman C. Hutchinson, and Larry L. Peterson. Service composition in Lipto. In *Proceedings of the 1991 International Workshop on Object-Orientation in Operating Systems*, pages 108–111, October 1991.

[DLJAR91] Partha Dasgupta, Richard J. LeBlanc Jr., Mustaque Ahamad, and Umakishore Ramachandran. The Clouds distributed operating system. *IEEE Computer*, 24(11):34–, November 1991.

[DPH91] Peter Druschel, Larry L. Peterson, and Norman C. Hutchinson. Lipto: A dynamically configurable object-oriented kernel. In *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems and Application Environments*, volume 5, pages 11–16. IEEE, 1991.

[Dru93] Peter Druschel. Efficient support for incremental customization of OS services. In *Proceedings of the Third International Workshop on Object Orientation in Operating Systems*, pages 186–190, Asheville, NC, December 1993.

[Duc89] Dan Duchamp. Analysis of transaction management performance. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 177–190, December 1989.

[DWB$^+$93] Chris Dalton, Greg Watson, David Banks, Costas Calamvokis, Aled Edwards, and John Lumley. Afterburner. *IEEE Network*, 7(4):36–43, July 1993.

[Fel90] David C. Feldmeier. Multiplexing issues in communication system design. In *Proc. ACM SIGCOMM '90*, pages 209–219, Philadelphia, PA, Spetember 1990.

[FR86] Robert Fitzgerald and Richard F. Rashid. The integration of virtual memory management and interprocess communication in Accent. *ACM Transactions on Computer Systems*, 4(2):147–177, May 1986.

[GA91] Ramesh Govindan and David P. Anderson. Scheduling and IPC mechanisms for continuous media. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 68–80, October 1991.

[GHM$^+$90] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Gerald J. Popek, and Dieter Rothmeier. Implementation of the ficus replicated file system. In *Proceedings of the USENIX Summer '90 Conference*, pages 63–71, June 1990.

[Hil92] Dan Hildebrand. An architectural overview of QNX. In *Proceedings of the Usenix Workshop on Micro-Kernels and Other Kernel Architectures*, April 1992.

[HK93] Graham Hamilton and Panos Kougiouris. The Spring nucleus: A microkernel for objects. In *Proc. of the 1993 Summer Usenix Conference*, pages 147–159, Cincinatti, OH, June 1993.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., Palo Alto, California, 1990.

[HP91] Norman C. Hutchinson and Larry L. Peterson. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[HPM93] Graham Hamilton, Michael L. Powell, and James J. Mitchell. Subcontract: A flexible base for distributed programming. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 69–79, 1993.

[IBM90] IBM Corporation. *IBM RISC System/6000 POWERstation and POWERserver: Hardware Technical Reference, General Information Manual, IBM Order Number SA23-2643-00*, 1990.

[IEE90] Gigabit network testbeds. *IEEE Computer*, pages 77–80, September 1990.

[Jac90] Van Jacobson. Efficient protocol implementation. In *ACM SIGCOMM '90 tutorial*, September 1990.

[Jef92] Kevin Jeffay. On Kernel Support for Real-Time Muiltimedia Applications. In *Proceedings Third IEEE Workshop on Workstation Operating Systems*, pages 39–46, Key Biscayne, FL, April 1992.

[JH93] Alan Jones and Andrew Hopper. Handling audio and video streams in a distributed environment. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 231–243, 1993.

[JLHB88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[JZ93] David B. Johnson and Willy Zwaenepoel. The Peregrine high-performance RPC system. *Software—Practice and Experience*, 23(2):201–221, February 1993.

[Kar89] Paul A. Karger. Using registers to optimize cross-domain call performance. In *Third Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 194–204, Boston, Massachusetts (USA), April 1989. ACM.

[KC88] Hemant Kanakia and David R. Cheriton. The VMP network adapter board (NAB): High-performance network communication for multiprocessors. In *Proceedings of the SIGCOMM '88 Symposium*, pages 175–187, August 1988.

[KDCZ94] Peter Keleher, Sandhya Dwarkadas, Alan Cox, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, January 1994.

[L$^+$92] Charles E. Leiserson et al. The network architecture of the Connection Machine CM-5. In *Proceedings of the 4th ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, June 1992.

[LHFL93] Jay Lepreau, Mike Hibler, Bryan Ford, and Jeffrey Law. In-kernel servers on Mach 3.0: Implementation and performance. In *Proc. of the Third Usenix Mach Symposium*, pages 39–55, April 1993.

[LKG92] Li Li, A. Karmouch, and N.D. Georganas. Real-time Synchronization Control in Multimedia Distributed Systems. *ACM Computer Communications Review*, 22(3):79–86, 1992.

[LMKQ89] Samuel J. Leffler, Marshall K. McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley Publishing Company, 1989.

[MB93a] Chris Maeda and Brian Bershad. Protocol service decomposition for high-performance networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, December 1993.

[MB93b] Chris Maeda and Brian N. Bershad. Networking performance for microkernels. In *Proceedings of the Third International Workshop on Object Orientation in Operating Systems*, pages 154–159, Asheville, NC, December 1993.

[Mog92] Jeffrey C. Mogul. Network locality at the scale of processes. *ACM Transactions on Computer Systems*, 10(2):81–109, May 1992.

[OP92] Sean W. O'Malley and Larry L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[PDP93] Michael A. Pagels, Peter Druschel, and Larry L. Peterson. Cache and TLB effectiveness in the processing of network data. Technical Report TR 93-4, Department of Computer Science, University of Arizona, Tucson, Ariz., 1993.

[Prz90] Steven A. Przybylski. *Cache and Memory Hierarchy Design: A Performance-Directed Approach.* Morgan Kaufmann, San Mateo, CA, 1990. TK7895.M4P79.

[RAA$^+$88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Chorus distributed operating systems. *Computing Systems Journal*, 1(4):305–370, December 1988.

[RAA$^+$92] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the Chorus distributed operating system. In *Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–70, Seattle WA (USA), April 1992. Usenix.

[Ram93] K. K. Ramakrishnan. Performance considerations in designing network interfaces. *IEEE Journal on Selected Areas in Communications*, 11(2):203–219, February 1993.

[RH91] F. Reynolds and J. Heller. Kernel support for network protocol servers. In *Proceedings of the USENIX Mach Symposium*, pages 149–162, Monterey, Calif., November 1991.

[RR93] S. Ramanathan and P. V. Rangan. Adaptive feedback techniques for synchronized multimedia retrieval over integrated networks. *IEEE/ACM trans. on Networking*, 1(2), April 1993.

[RT74] Dennis M. Ritchie and Ken Thompson. The Unix time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[SB90] Michael D. Schroeder and Michael Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[SGH$^+$89] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Celine Valot. Sos: An object-oriented operating system—assessment and perspectives. *Computer Systems*, 2(4):287–338, December 1989.

[Sha86] Marc Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *Proceedings of the Sixth International Conference on Distributed Computing Systems*, pages 198–204, Boston, Mass., May 1986.

[SLM90a] Michael L. Scott, Thomas J. LeBlanc, and Brian D. Marsh. Multi-model parallel programming in Psyche. In *Proc. 2nd Annual ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, page 7078, Seattle, WA (USA), March 1990. ACM.

[SLM$^+$90b] Michael L. Scott, Thomas J. LeBlanc, Brian D. Marsh, Timothy G. Becker, Dubnicki Cezary, and Evangelos P. Markatos. Implementation issues for the Psyche multiprocessor operating system. *Computing Systems*, 3(1):101–137, 1990.

[ST93] Jonathan M. Smith and C. Brendan S. Traw. Giving applications access to Gb/s networking. *IEEE Network*, 7(4):44–52, July 1993.

[Sta93] Richard Stallman. *Using and porting GNU CC.* Free Software Foundation, Cambridge, MA, June 1993.

[Str90] Robert E. Strom. Hermes: An integrated language and system for distributed programming. In *1990 Workshop on Experimental Distributed Systems*, October 1990.

[TA91] Shin-Yuan Tzou and David P. Anderson. The performance of message-passing using restricted virtual memory remapping. *Software—Practice and Experience*, 21:251–267, March 1991.

[TMP94] Charles J. Turner, David Mosberger, and Larry L. Peterson. Cluster-C*: Understanding the performance limits. In *Proceedings of the 1994 Scalable High Performance Computing Conference*. IEEE, May 1994.

[TNML93] C. Thekkath, T. Nguyen, E. Moy, and E. Lazowska. Implementing network protocols at user level. In *Proceedings of the SIGCOMM '93 Symposium*, September 1993.

[TS93] C. B. S. Traw and J. M. Smith. Hardware/software organization of a high-performance ATM host interface. *IEEE Journal on Selected Areas in Communications*, 11(2):240–253, February 1993.

[TSS88] C. P. Thacker, L. C. Stewart, and E.. H. Satterthwaite. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[UNI89] *UNIX System V Release 3.2—Programmer's Reference Manual.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[WCC$^+$74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, 1993.

[WLH81] William A. Wulf, Roy Levin, and Samuel P. Harbison. Hydra/c.mmp: An experimental computer system, 1981.

[WN80] Maurice V. Wilkes and Roger M. Needham. The Cambridge model distributed system. *OSR*, 14(1):21–29, January 1980.

[Yud83] Mark Yudkin. Resource management in a distributed system. *Proceedings of the Eighth Data Communication Symposium*, pages 221–226, October 1983.