Effects of Data Passing Semantics and Operating System Structure on Network I/O Performance

José Carlos Brustoloni

September 1997
CMU-CS-97-176

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Thesis Committee:
Peter Steenkiste, *chair*
David B. Johnson
M. Satyanarayanan
Willy E. Zwaenepoel, *Rice University*

Copyright © 1997 José Carlos Brustoloni

This research was sponsored by the Defense Advanced Research Projects Agency under contracts DABT63-93-C-0054, F19628-92-C-0116, and N66001-96-C-8528.

The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Keywords: I/O, networking, copy avoidance, virtual memory, data passing semantics, APIs, operating system structure, network adapters, protocol processing

Abstract

Elimination of data and control passing overheads in I/O has been a long-sought goal. Researchers have often proposed changing the *semantics* of I/O data passing, so as to make copying unnecessary, or the *structure* of the operating system, so as to reduce or eliminate data and control passing. However, most such changes are incompatible with existing applications and therefore have not been adopted in conventional systems. My thesis is that, in network I/O, optimizations that preserve data passing semantics and system structure can give end-to-end improvements competitive with those of data and control passing optimizations that change semantics or structure. Moreover, current technological trends tend to reduce differences in such improvements.

To demonstrate the thesis, I introduce new models of I/O organization, optimization, and data passing, emphasizing structure and compatibility rather than implementation. I review previous network I/O optimizations and propose many new ones, including *emulated copy*, for data passing without copying but with copy semantics between application and system buffers, and *I/O-oriented IPC*, for efficient data passing to and from user-level server buffers. I examine in detail network adapter requirements for copy avoidance. I describe the implementation of the different optimizations in Genie, a new I/O framework.

Using Genie, I experimentally compare the optimizations on a variety of platforms and with different levels of hardware support. The experiments confirm the thesis, showing that: (1) Emulated copy performs competitively with data passing schemes with move or share semantics; (2) Emulated copy performs competitively with data and control passing optimizations enabled by extensible kernels; and (3) I/O-oriented IPC gives user-level I/O servers performance approaching that of kernel-level ones. In all tests, network I/O performance was determined primarily by limitations of the physical I/O subsystem and presence or absence of data copying, and not by semantics or structure of the operating system. Moreover, end-to-end differences among optimizations varied inversely to the processor's SPECint95 benchmark.

The experiments also demonstrate that emulated copy interoperates efficiently with mapped file I/O, allowing applications to pass data without copying between networks and file systems.

Acknowledgements

My advisor, Peter Steenkiste, contributed two early suggestions that were to become pivotal in this work: (1) seek a structured characterization of I/O data passing in end systems; and (2) investigate how the performance of data passing scales with processor, memory, and network speeds. Peter had been designing network interfaces before I started working with him and was able to offer me not only a valuable sounding board but also a rich experimental environment in which to test my ideas. I especially thank him for his non-dogmatism and the great deal of freedom I had to conduct my research.

Willy Zwaenepoel, another member of my thesis committee, was the "shepherd" of the OSDI paper where an important part of the results presented here first appeared in print. His thoughtful and experienced criticism (and also that of anonymous referees) helped me tremendously in clarifying and further developing my ideas. I feel greatly indebted to him.

The other members of my thesis committee were involved mostly in reviewing this dissertation. Satya's feedback caused me to more carefully delimit the range of applicability of my new techniques and examine experimentally how these techniques interoperate with solutions for other types of I/O. His input significantly strengthened my dissertation. David Johnson's editorial scrutiny and tireless attention to detail helped me make this document more visually pleasing and rhetorically consistent.

I developed my work in the context of the Credit Net and Darwin projects. I especially thank two members of those projects, Prashant Chandra and Todd Mummert, for their invaluable help in setting up equipment for my experiments. Kam Lee was another constant companion in the lab. I also thank Hui Zhang for his contagious enthusiasm for networking research.

Discussions, suggestions, and feedback from other faculty members and students at CMU helped in one way or another shape aspects of this work. I would like to thank in particular Adam Beguelin, Elmootazbellah Elnozahy,

Garth Gibson, Hugo Patterson, George Necula, Brian Noble, and Stefan Savage.

Before the final stretch of my dissertation, I participated in the CMU dinner coop and ballroom dance club. I will miss the many people I met there, including Erik Seligman, Karen Haigh, Bruce Maxwell, Lisa Rotunda, Daniel Tunkelang, Mark Lillibridge, Lino Santos, and João Albuquerque. I especially thank Mary Jo Flint for her companionship and support.

Last but not least, I am grateful to my parents, whose love of learning and devotion to teaching were and continue to be for me sources of motivation and inspiration in my own studies.

Contents

1 Introduction **1**
- 1.1 I/O organization . . . 2
  - 1.1.1 Operating system structures . . . 2
  - 1.1.2 Explicit I/O . . . 6
  - 1.1.3 Memory-mapped I/O . . . 12
- 1.2 Optimization approaches . . . 14
  - 1.2.1 Copy avoidance . . . 14
  - 1.2.2 IPC avoidance . . . 19
  - 1.2.3 OS avoidance . . . 20
  - 1.2.4 Data passing avoidance . . . 21
  - 1.2.5 Scheduling avoidance . . . 23
- 1.3 Thesis . . . 24
- 1.4 Outline . . . 25

2 Data Passing Model **29**
- 2.1 Semantics . . . 29
  - 2.1.1 Buffer allocation . . . 30
  - 2.1.2 Buffer integrity . . . 31
  - 2.1.3 Optimization conditions . . . 32
- 2.2 Protection . . . 34
- 2.3 Symmetry . . . 36
- 2.4 Summary . . . 37

3 Copy Avoidance **39**
- 3.1 System buffers . . . 40
  - 3.1.1 Unmapped access . . . 40
  - 3.1.2 Request eviction . . . 41
- 3.2 In-place data passing . . . 43

3.2.1 Page referencing 43
3.2.2 I/O-deferred page deallocation 44
3.2.3 Input-disabled pageout 44
3.2.4 Input-disabled copy-on-write 45
3.3 Migrant-mode data passing 45
3.3.1 Region caching 46
3.3.2 Region hiding 46
3.4 Summary 47

4 Emulated Copy **49**
4.1 Input alignment 50
4.2 Transient output copy-on-write (TCOW) 52
4.3 Output buffer reuse 54
4.4 Related work 56
4.5 Summary 57

5 I/O-oriented IPC **59**
5.1 Selective transient mapping 60
5.2 User-to-user copying and input alignment 62
5.3 Fragment subcontracting 63
5.4 Related work 63
5.5 Summary 65

6 Network Adapter Support **67**
6.1 Pooled in-host buffering 68
6.2 Header/data splitting 70
6.3 Header patching 71
6.4 Early demultiplexing 73
6.5 Buffer snap-off 75
6.6 Outboard buffering 79
6.7 Checksumming 80
6.8 Related work 80
6.9 Summary 82

7 Implementation in Genie **85**
7.1 Client interface 85
7.2 Server interface 87
7.3 Buffer representation 88

7.4 Native- and migrant-mode data passing schemes 90
7.4.1 Client output buffers 90
7.4.2 Client input buffers and customized system buffers . . 90
7.4.3 Client input buffers and overlaid in-host buffers 92
7.4.4 Client input buffers and overlaid outboard buffers . . . 92
7.5 Fragment data passing scheme 93
7.6 Summary 94

8 Evaluation of Emulated Copy 97
8.1 Experimental set-up 97
8.2 Single-packet end-to-end latency 99
8.2.1 Measurements 100
8.2.2 Analysis 105
8.3 Maximum throughput 108
8.4 End-to-end latency with checksumming 111
8.5 Multiple-packet end-to-end latency 112
8.6 Summary 115

9 Evaluation of I/O-oriented IPC 123
9.1 Experimental set-up 124
9.2 Measurements 125
9.3 Analysis 128
9.4 Summary 129

10 Interoperability 131
10.1 File access by mapped file I/O 132
10.2 File access by explicit I/O 134
10.3 Demonstration 135
10.3.1 Experimental design 135
10.3.2 Experimental set-up 137
10.3.3 Response time 137
10.3.4 I/O processing time 140
10.4 Related work 143
10.5 Summary 144

11 Data Passing and Scheduling Avoidance 147
11.1 Iolets 148
11.2 Limited hijacking 150

11.3 Related work 151
11.4 Summary 153

12 Evaluation of Iolets **155**
12.1 Experimental set-up 155
12.2 Device-to-device latency 156
12.2.1 Measurements 156
12.2.2 Analysis 159
12.3 Device-to-device throughput 160
12.4 Multicast latency and throughput 161
12.5 Summary 163

13 Conclusions **167**
13.1 Recommendations 170
13.2 Future work 172

List of Figures

1.1 Explicit I/O model. 6
1.2 I/O subcontracting. 7
1.3 Explicit data passing by copying in monolithic systems. 10
1.4 Explicit data passing by copying in microkernel systems. . . . 10
1.5 In-place data passing. 15
1.6 Data passing with move semantics. 16
1.7 Multiserver and outboard buffering in device-to-device data transfers. 22
1.8 Multiserver buffering in multicast. 23

2.1 Taxonomy of data passing schemes. 30

3.1 Request eviction. 42

4.1 Input alignment. 51

6.1 Copy avoidance with pooled in-host buffering and client-aligned buffering. 69
6.2 Header/data splitting. 70
6.3 Copy avoidance with header/data splitting and page-aligned client buffering. 71
6.4 Copy avoidance with header patching and pooled in-host buffering. 72
6.5 Early demultiplexing. 74
6.6 Copy avoidance with server-aligned buffering. 75
6.7 Data buffer representation alternatives. 76
6.8 Buffer snap-off 77

8.1 Single-packet end-to-end latency with early demultiplexing. . . 100

8.2 Single-packet end-to-end latency for short data, using early demultiplexing. 102
8.3 Single-packet end-to-end latency with client-aligned pooled in-host buffering. 103
8.4 Single-packet end-to-end latency with unaligned pooled in-host buffering. 104
8.5 I/O processing time ($t_{I/O}$) with early demultiplexing. 109
8.6 Single-packet end-to-end latency with or without checksumming. 111
8.7 Multiple-packet end-to-end latency with pooled in-host buffering. 113
8.8 Multiple-packet end-to-end latency with early demultiplexing. 114

9.1 End-to-end latency using kernel- or user-level protocol servers. 126
9.2 End-to-end latency for short datagrams using kernel- or user-level protocol servers. 127
9.3 End-to-end latency for short datagrams on different processors. 128

10.1 Fetch response time in the case of a hit. 137
10.2 Store response time in the case of a hit. 138
10.3 Store response time in the case of a miss. 139
10.4 Fetch I/O processing time in case of a hit. 140
10.5 Store I/O processing time in case of a hit. 141
10.6 Fetch I/O processing time in case of a miss. 142
10.7 Store I/O processing time in case of a miss. 143

11.1 Iolet for video capture, display, and transmission. 149

12.1 Total latency of data transfer from Cyclone board to ATM network. 156
12.2 Total latency of short data transfer from Cyclone board to ATM network on otherwise idle system. 157
12.3 Total latency of short data transfer from Cyclone board to ATM network with ten concurrent compute-bound processes. . 158
12.4 I/O processing time ($t_{I/O}$) of data transfer from Cyclone board to ATM network. 161
12.5 Total latency to output the same client data to the Cyclone board and to the ATM network. 162
12.6 I/O processing time ($t_{I/O}$) to output the same client data to Cyclone board and ATM network. 163

List of Tables

1.1 Approximate year of introduction and point-to-point bandwidth of several popular LANs. 11

6.1 Conditions for copy avoidance according to network adapter support. 83

7.1 Operations for data passing from client output buffer to system buffer. 91
7.2 Operations for data passing from customized system buffer to client input buffer. 95
7.3 Ready- and reply-time operations for data passing from overlaid in-host buffer to client input buffer. 96

8.1 Characteristics of the computers used in the evaluation of emulated copy. 98
8.2 Costs of primitive data passing operations on the Micron P166 computer at 155 Mbps. 116
8.3 Costs of primitive data passing operations on the Micron P166 computer at 512 Mbps. 117
8.4 Costs of primitive data passing operations on the Gateway P5-90 computer at 155 Mbps. 118
8.5 Costs of primitive data passing operations on the AlphaStation 255/233 computer at 155 Mbps. 119
8.6 Estimated and actual end-to-end latencies with early demultiplexing on Micron P166 computers. 120
8.7 Estimated and actual end-to-end latencies with pooled in-host buffering on Micron P166 computers. 121
8.8 Validation of scaling model on each platform. 121
8.9 Validation of scaling model across platforms. 122

9.1 Characteristics of the computers used in the evaluation of I/O-oriented IPC. 124
9.2 Throughput for single 60 KB datagrams using kernel- or user-level protocol servers. 125
9.3 Scaling of user-level server overhead according to processor speed. 129

10.1 Characteristics of the computers used in the demonstration of interoperability of copy avoidance techniques for network and file I/O. 136

12.1 Latency breakdown for device-to-device data transfers. 165
12.2 Estimated and actual total latencies for device-to-device data transfers 166
12.3 I/O processing time and throughput for CPU saturation in device-to-device data transfers. 166

Chapter 1

Introduction

To preserve protection, multiuser systems usually do not allow applications to access input/output (I/O) devices directly: Applications can perform I/O only indirectly, by explicit or implicit request to an authorized kernel- or user-level server or pager and ultimately to an authorized driver. However, requests and their respective replies may involve significant data and control passing overheads, such as copying and context switching.

A long line of research has aimed at alleviating I/O data and control passing overheads, often proposing:

1. Changing the *semantics* of data passing between applications and the operating system, so as to avoid data copying; or

2. Changing the *structure* of the operating system, so that data passing and control passing between applications and operating system can be reduced or eliminated.

Contrary to much previous work, this dissertation's thesis is that, in network I/O, optimizations that preserve data passing semantics and system structure can give end-to-end improvements competitive with those of data and control passing optimizations that change semantics or structure. Moreover, current technological trends tend to reduce differences in such improvements.

The rest of this chapter characterizes the problem in more detail, summarizes previous work and possible approaches, states the thesis more fully, and outlines its demonstration in the rest of the dissertation.

1.1 I/O organization

Direct application access to I/O devices can be unsafe. For example, unrestricted access to a disk controller or network adapter might allow an unauthorized application to access or modify data of other applications or shut the device down, compromising system protection or integrity.

To preserve protection and integrity, multiuser operating systems normally allow only authorized *drivers* to access I/O devices directly. Applications and drivers usually run in separate *protection domains*. A protection domain establishes what *privileges* code running in it has, that is, which objects and memory addresses such code can access. Unprivileged applications typically can perform I/O only indirectly, ultimately by request to a privileged driver. However, requests and their respective replies may introduce considerable overhead.

This section characterizes such overhead. Subsection 1.1.1 describes how operating systems of different structures implement protection domains. Subsections 1.1.2 and 1.1.3 then show how protection domains are used in the two most common I/O models, *explicit* and *memory-mapped*, respectively, and discuss how different system structures and I/O models may result in different data and control passing overheads.

1.1.1 Operating system structures

Drivers have to access device controller registers and other data structures that generally should not be accessible by applications. Therefore, the protection domains of drivers and applications usually must be different.

The assignment of separate protection domains for applications and drivers depends on the *structure* of the operating system. Such structure establishes how system implementation is decomposed into modules, how protection domains are implemented, and how protection domains are assigned to system modules and applications so as to preserve system protection and integrity.

To implement protection domains, most operating systems rely on two hardware-supported processor features: *virtual memory (VM)* management and *privilege modes*. VM hardware treats memory addresses issued by processes as *virtual* and automatically translates such addresses into *physical* ones. Physical addresses are used to access physical memory. In the most common scheme, physical memory is split into fixed-size blocks, called *pages*.

To translate virtual addresses into physical ones, VM hardware consults the current *page table*. If a process issues a virtual address for which no valid translation exists in the page table, control of the processor is automatically transferred to system's *VM fault handler* (this control transfer is called a VM fault *trap*). In systems such as Mach [67] and those derived from 4.4 BSD Unix, such as NetBSD, allocated pages belong to a *memory object*. Each memory object is backed by a *pager*. On a VM fault, the handler allocates a physical page and invokes the object's pager to retrieve the contents of the virtual page into the physical page. When the pager returns, the VM fault handler *maps* the physical page to the faulted process, that is, modifies the page table so that the faulted virtual address translates into the physical address of the page. The handler then makes the faulted process again runnable.

Because the number of pages in physical memory is limited, each page allocation necessitates, in general, a counterbalancing page deallocation. A kernel-level process, called the *pageout daemon*, scans and deallocates currently allocated pages when the number of free pages in the system is low. If the daemon selects for deallocation a page that was modified after being last retrieved from its pager, the daemon invokes the pager to save the page's contents. When the pager returns, the daemon unmaps the page and places it in a list of free pages. Pagers usually save and retrieve page contents to and from storage devices, but may also do so remotely, over a network. In Mach and related systems, applications can supply their own user-level pagers when allocating a *region*, that is, memory spanning a given range of virtual addresses. The correspondence between virtual addresses and pages in physical memory and backing storage devices is called an *address space*. Page allocation and retrieval is also called *paging in*, whereas page saving and deallocation is also called *paging out*.

The instruction to switch the current page table (like other instructions that can jeopardize system protection or integrity, such as enabling/disabling interrupts) is usually *privileged*, that is, can be executed only in the processor's *kernel* mode. By running each application in its own address space, in *user* mode, systems can prevent applications from gaining direct access to each other's or the system's data (including, for example, device controller registers). Code running in kernel or user mode is also called *kernel-level* or *user-level* code, respectively.

To switch into kernel mode, applications typically have to execute a special instruction, the *system call*, which jumps to a well-defined address. The

system installs its own code at such address, and sets up application address spaces so that applications cannot otherwise access (or corrupt) the memory occupied by system code. Most processors can address a wider range of virtual addresses in kernel mode than in user mode. The system address space, therefore, can be implemented as a complement to every application's address space, and no address space switching is necessary to cross the kernel/user protection boundary.

The assignment of protection domains to system modules varies according to the system structure. The two most common alternatives are:

1. *Monolithic*: All system modules (including drivers) run in kernel mode, in a single protection domain. Applications run in user mode, each in its own protection domain.

2. *Microkernel*: Only system modules that implement fundamental system abstractions, such as processes and inter-process communication (IPC), run in kernel mode. Other system modules, including drivers, run each in its own address space, in user mode, much like applications.

The microkernel structure has several advantages relative to a monolithic one, including: (1) greater maintainability, because user-level code can usually be modified and debugged more easily than kernel-level code, and (2) better fault isolation, because user-level system modules are each implemented in a separate protection domain. However, microkernel systems have typically had worse I/O performance than do monolithic systems. Consequently, few systems have a pure monolithic or microkernel structure. Unix [49], for example, is generally considered a monolithic system and integrates in the kernel all modules directly involved in processing most I/O requests. However, Unix also installs several auxiliary servers at user level (such servers are called *daemons* and include, for example, `inetd`, `named`, `routed`, and `ftpd`). Mach [37], on the other hand, is generally considered a microkernel system, but integrates in the kernel several non-fundamental modules, including many drivers. Windows NT [26] also has elements both of monolithic and microkernel structures: It currently integrates most I/O modules in the kernel, but employs user-level servers to emulate non-native application programming interfaces (APIs).

Operating system structure is an area of active research. Other alternatives currently being considered include *librarized* [74, 52, 34] and *extensible kernel* [9, 69] structures. In the librarized structure, some system modules

are linked as libraries with applications and execute at user level in the same protection domains as the respective applications. In the extensible kernel structure, both system and applications can run in kernel mode; protection domains are implemented in software, using a type-safe language [9] or software fault isolation [69]. These alternatives are further discussed in Sections 1.2.2 and 1.2.4.

The concept of operating system structure defined in this dissertation does not *per se* include or imply *extensibility*. Operating system extensibility is the ability of unprivileged applications to add or modify services provided by the system [9, 69, 34]. Those additions or modifications are called *extensions*. The requirements of extensibility include:

1. Protection domains for extensions, distinct from each other and from those of at least certain critical system modules.

2. A mechanism for transferring control and data from the system to extensions.

3. A policy interface that allows applications to specify when such transfers should occur.

4. A run-time library that allows extensions to request resources and services from the system.

5. One or more mechanisms to prevent extensions from hoarding resources, such as CPU and memory.

Both the policy interface and the run-time library need to check each call to make sure that it preserves system protection and integrity.

Operating system structure defines only the first of the above requirements. A system with extensible kernel structure is *extensible*, in the sense defined here, only if the system also satisfies the other requirements of extensibility. On the other hand, systems with other structures, e.g. librarized [45] or microkernel [51], can also be extensible.

This dissertation studies I/O data and control passing overheads and ways to reduce them, including changing the system structure. Consequently, this work concerns only parts of the first two of the above requirements. The remaining requirements for extensibility are beyond the scope of this dissertation.

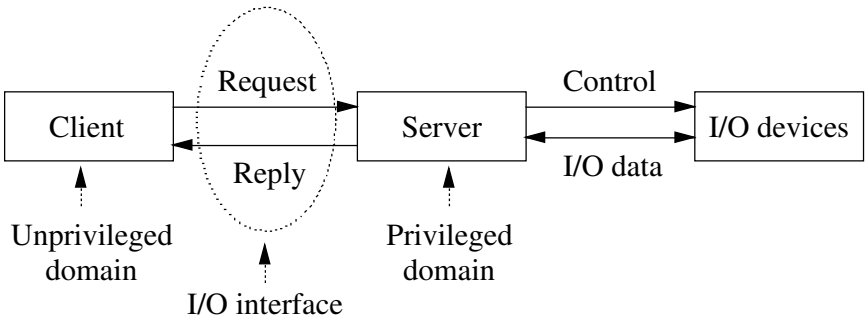


Figure 1.1: Explicit I/O model.

Regarding the benefits of extensibility, this dissertation examines only two optimizations that are closely related to I/O data and control passing. Those optimizations, *data passing avoidance* and *scheduling avoidance*, are defined in Sections 1.2.4 and 1.2.5, further discussed in Chapter 11, and evaluated in Chapter 12. There are many other potential uses of extensibility to improve I/O performance. An application might use extensibility, for example, to redefine disk layout or protocol implementation so as to better match the application's needs [45]. A broad characterization and evaluation of how applications might exploit and benefit from extensibility is, however, beyond the scope of this dissertation.

1.1.2 Explicit I/O

In multiuser systems, applications usually can perform I/O only according to either the *explicit* or the *memory-mapped* I/O model. In the explicit model, unprivileged applications perform I/O as *clients* that *request* each I/O service from a suitably privileged I/O *server* (client and server are also called the *parties* to a request). The server processes the request and returns a *reply* to its client, as shown in Figure 1.1. Requests and replies are mediated by an *I/O interface*. Clients or servers can execute at kernel or user level.

If processing of the client does not continue until the server replies, the request is called *synchronous*; otherwise, the request is called *asynchronous*. In the latter case, the server may return an *interim reply* indicating to the client that the request is *pending* and that the client should check completion later. Certain servers may return an *anticipated reply* before request processing completion, when the server can guarantee that it will carry out

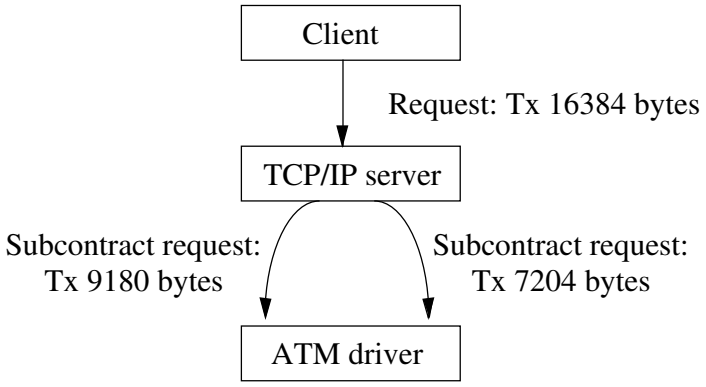


Figure 1.2: I/O subcontracting.

the request successfully. For example, a TCP/IP server may return a successful reply to an output request as soon as it has checked request parameters (e.g., connection) and has a reference to the request data, but well ahead of actually physically outputting the data. An anticipated reply suppresses generation of the respective reply at request processing completion time[1]. A client can make a synchronous *flush* request to obtain confirmation that processing of one or more previous requests has actually completed.

To process parts of an I/O request, I/O servers may *subcontract* other I/O servers. For example, a file system server will usually subcontract a disk driver, and a TCP/IP server will usually subcontract a network driver, as shown in Figure 1.2. *Drivers* are servers that access devices directly and that, therefore, typically do not have to subcontract other servers.

In a subcontract, a *contractor* (server acting as a client) requests a service from a *subcontractor* (another server), which processes the request and returns a reply to its client. The subcontract requests made by a contractor in order to process a given request are called the latter's *originated requests*. A request that is not originated of any other is called an *original request*, and the client of such request, usually an unprivileged application, is called an *original client*. A *subcontract graph* is a directed acyclic graph that links each request to its pending originated requests.

Requests and replies usually involve passing control and data between the parties. Both clients and servers hold I/O data in *buffers*. The type of buffer used by the server may affect data passing between client and server buffers.

[1]In such cases, the *reply time* is defined as the request processing completion time.

Servers normally use *in-host* buffers, allocated from host memory. Drivers, however, may also use *outboard* buffers and pass data between such buffers and in-host ones. Outboard buffers are allocated from outboard memory, e.g. that of a video card. In-host buffers can be *ephemeral* or *cached*. Ephemeral buffers are allocated at request time and deallocated at the corresponding reply time. Cached buffers, on the contrary, may remain allocated after reply time. Data passing between client and ephemeral server buffers can be optimized by exploiting the fact that ephemeral buffers remain allocated only until request processing completion. Similar optimizations may not be appropriate for cached server buffers because such buffers may remain allocated for an unbounded amount of time.

In BSD Unix [49], ephemeral buffers include those used by the socket interface, network protocols, network interface drivers, raw disk interface, raw and cooked tty interfaces, and character device drivers[2]. Cached buffers include those used by the file system, cooked disk interface, and block device drivers.

Cached buffers are most often used to improve the performance of storage-related servers. To process a request, such servers always look for the requested data in their cache. If the data is found in the cache (an event that is called a cache *hit*), no physical I/O is performed. To maximize the hit ratio, storage servers often *prefetch* data, that is, bring into the cache data that is expected to be requested soon. If the hit ratio is high, cached buffers can approximate the performance of storage servers to that of memory rather than the typically much lower performance of disks and tapes. For an 8 MB cache, hit ratios of 80% or more have been reported in studies of file systems for microcomputers, minicomputers, and mainframes [40].

Using more memory for the cache often results in greater hit ratios and better performance. However, the amount of memory used for such purpose is necessarily limited. When a request does not hit in the cache, one or more cached buffers may have to be deallocated to make room for the request's data. A *cache policy* determines which buffers should be deallocated. Servers often use a *least-recently used* (LRU) policy. The adequacy of such policy depends on the client's *access pattern*, that is, sequence of requests. LRU is a reasonable policy, for example, for random accesses, but often not for sequen-

[2]Some character device drivers, e.g. those of graphical devices, may statically allocate in-host buffers and cache I/O data. More commonly, however, such drivers use a combination of outboard buffers and ephemeral in-host buffers.

tial accesses. However, explicit I/O interfaces usually do not allow clients to disclose their access patterns. Several recent studies have demonstrated that large performance improvements are possible when the explicit I/O interface is expanded so as to make such disclosure possible [19, 65]. The server can then fine-tune both its prefetch and cache policies.

The data passing problem

In explicit I/O, a party's buffers may be unsuitable for another party because, for example, the other party cannot access the buffers, or the buffers are pageable but the other party requires unpageable buffers. An *unpageable* buffer has all its pages always in physical memory, whereas a *pageable* one may have, at any given time, only some or none of its pages in physical memory. Normally, the buffers of unprivileged applications are pageable. On the contrary, servers, and especially drivers, often require unpageable buffers, because:

1. If the server faults on a page whose pager (possibly supplied by an untrusted party) subcontracts the server (whether directly or indirectly), the server may deadlock.

2. Even if the server is not required for paging, its throughput will suffer while it waits for paging.

In general, therefore, as part of a request, the explicit I/O interface must pass data from *client output buffers* to *server input buffers*, and, as part of a reply, the explicit I/O interface must pass data from *server output buffers* to *client input buffers*.

Explicit data passing often involves *system* (kernel-level) buffers. In monolithic systems, server buffers usually are system buffers and are accessible by multiple servers integrated in the kernel. Buffers of unprivileged applications normally are distinct from system buffers. Data passing between application and system buffers typically is by copying, as shown in Figure 1.3.

In microkernel systems, client, server, and system buffers often are all distinct. Explicit data passing between client and server buffers may be by copying, and often involves two data copies – once between each party's and system buffers, as shown in Figure 1.4.

Despite its popularity as a data passing technique, copying can impose significant performance penalties [63]. For example, among all personal computers and workstations used in the experiments reported in Chapter 8, the

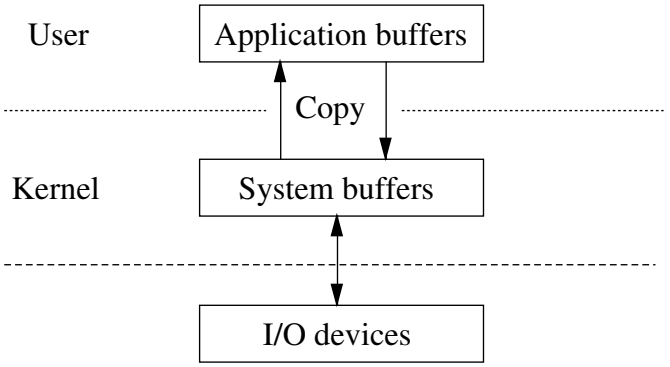


Figure 1.3: Explicit data passing by copying in monolithic systems.

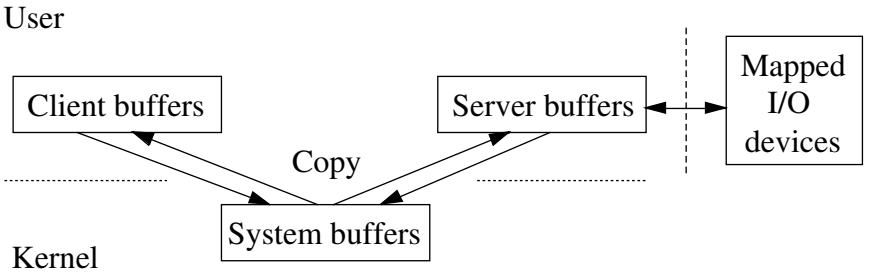


Figure 1.4: Explicit data passing by copying in microkernel systems.

maximum main memory copy bandwidth was 351 Mbps. If data passing is by copying and any such computer is connected to a fast network that transmits at 512 Mbps, the end-to-end bandwidth will be limited by data passing overheads to a value well below what the network would allow.

The relative cost of the memory accesses necessary for copying has been increasing dramatically. CPU performance has been improving over 50% per year [40], and local area network (LAN) point-to-point bandwidth, as shown in Table 1.1, has been increasing by roughly an order of magnitude each decade. In contrast, access times for DRAMs, the almost universal option for main memory, have been improving by roughly only 7% per year or twofold per decade [40]. These different rates of performance improvement over time may result in significant imbalances.

Fast page mode, a widely used technique, improves DRAM cycle times by about a factor of four [40]. Beyond that, memory bandwidth can be improved, for sequential accesses, by widening the memory bus (memory

LAN	Year introduced	Bandwidth (Mbps)
Token ring	1972	1, 4, or 16
Ethernet	1976	3 or 10
FDDI	1987	100
ATM	1989	155, 622, or 2488
HIPPI	1992	800 or 1600

Table 1.1: Approximate year of introduction and point-to-point bandwidth of several popular LANs.

bandwidth will be roughly proportional to width) or interleaving word-width memory banks (accessed in parallel and transferring one word per clock cycle).

Unfortunately, memory widening and interleaving become too costly or impractical beyond some factor. For a given total capacity, both techniques require a greater number of lower-density, possibly earlier-generation memory chips, which usually cost more because of the fast rate of growth in capacity per memory chip (about 60% per year [40]). Both techniques may also result in excessive minimum memory size and minimum memory expansion. Therefore, memory bandwidth has become relatively scarce in all but high-end machines.

The control passing problem

Request processing often involves passing control from client to server and, after completion, from server back to the client. Control passing can significantly add to I/O latency. Such overhead is most noticeable for short data. For long data, data passing overheads usually dominate.

Monolithic systems implement control passing of original requests as system calls. On the other hand, given that all I/O servers are integrated in the kernel, subcontract requests can be implemented as simple function calls. Therefore, the total control passing overhead introduced by a monolithic structure can be approximated by the cost of a null system call.

Microkernel systems normally implement all requests to and replies from user-level servers as IPC. IPC is typically much more expensive than a system call, because IPC usually requires, in addition to a system call, rescheduling the CPU and switching address spaces. Therefore, the total control passing overhead of a microkernel structure is normally much higher than that of a

monolithic structure on the same hardware.

1.1.3 Memory-mapped I/O

The memory-mapped I/O model uses VM techniques to enable clients to perform I/O without explicit requests. There are two main forms of memory-mapped I/O, *mapped device I/O* and *mapped file I/O*.

For mapped device I/O, the memory-mapped I/O interface maps device registers just like physical pages to a process's address space. The process can then access device registers as if they were in memory. Explicit I/O requests and replies are not necessary, and there are no data or control passing overheads. This is how drivers perform I/O. However, to preserve system protection and integrity, multiuser systems typically do not allow unprivileged applications to map devices such as network adapters and disk controllers.

Memory-mapped I/O interfaces also allow clients to map a *file* (or part of it) to a region in the client's address space. In the explicit I/O model, the request to map a file is equivalent to that of allocating a new region and inputting file data into it. Likewise, the request to unmap a file is equivalent to that of outputting the region's data to the file and deallocating the region[3].

In spite of the functional equivalence, the implementation of mapped file I/O can be quite different from that of explicit I/O involving the same files. On the first request to map a given file, the memory-mapped I/O interface may allocate a new memory object, move the file's cached buffers (if any) from the file system server to that memory object, and set up a pager that retrieves and stores page contents directly from or to the file, in the backing storage device. Additionally, for each request to map the file, the interface allocates a region backed by the file's memory object. On each request to unmap the file, the interface deallocates the corresponding region. On the last request to unmap the file, the memory-mapped I/O interface may simply move pages from the file's memory object to the cached buffers of the file system server, and deallocate the file's memory object and pager[4].

[3]This description corresponds to a file mapped in *shared* mode. If multiple clients map the file in *shared* mode, the region is shared among them, and the output of the region's data to the file occurs when the last such client unmaps the file. It is often also possible to map a file in *private* mode, in which case the region's data is not output back to the file.

[4]This description assumes that cached buffers and VM pages are allocated from the same pool. This is the case in many contemporary systems, but was not originally true

Using mapped files, clients make an *implicit* synchronous request when they access a page not currently in main memory. The VM fault handler converts the implicit request into an explicit one, and invokes the pager. The pager is simply a specialized server. Pagers can subcontract other servers or drivers, and can use explicit or memory-mapped I/O.

Mapped files allow data to be input *lazily*, when actually accessed by a client. Output may occur *eagerly* (and perhaps too much so), because the pageout daemon can select any of the file object's pages for pageout before the file is unmapped. To allow good performance, many memory-mapped I/O interfaces include calls that allow clients to disclose their expected access pattern. Unix's `madvise` call, for example, allows clients to specify whether they will access mapped regions randomly or sequentially and point to pages that no longer are needed or that will be needed in the near future. Such calls allow pagers to optimize their prefetch and cache policies. A recent study has shown that access disclosures can be generated automatically by a compiler, improving the execution time of several application by a factor of two or more [58].

Data and control passing

Data passing overheads can be much lower in mapped file I/O than in explicit I/O. In mapped file I/O, pages can passed to or from client buffers by mapping and unmapping. In explicit file I/O, on the contrary, it is normally necessary to copy data, because the server caches I/O data. Mapping and unmapping typically cost much less than copying pages.

Control passing overheads of mapped file I/O are often similar to those of explicit I/O. In a monolithic structure, the control passing overhead of mapped file I/O can be approximated by that of a VM fault trap, which costs about the same as a system call. In a microkernel structure (i.e., user-level pager), control passing overhead corresponds to a VM fault trap and address space switching. However, a pager will typically only be invoked to retrieve or save a page from or to a backing storage device. Given the latency of such devices, the overhead of passing control to the pager may not be significant.

in Unix [49]. When pools are separate, file mapping and unmapping may require copying data between pages from each pool.

1.2 Data and control passing optimization approaches

The previous section described the *explicit* and *memory-mapped* I/O models, both of which are available in most contemporary systems. As explained in that section, data and control passing overheads are much more significant in explicit I/O than in memory-mapped I/O. However, not all explicit I/O can be converted into memory-mapped I/O: In general, memory-mapped I/O can be used only by privileged clients (mapped device I/O) or when server buffers are cached (mapped file I/O).

For I/O involving unprivileged clients and ephemeral server buffers, most multiuser systems support only the explicit I/O model. In BSD Unix [49], for example, I/O with ephemeral server buffers can be performed only using explicit interfaces: sockets (e.g., for networking) or character device interfaces (e.g., for raw disk I/O, printing, or writing to high-speed graphics devices). This section broadly classifies the possible approaches for reduction of data and control passing overheads in such cases.

1.2.1 Copy avoidance

Data passing techniques or optimizations imply a certain data passing *semantics* that applications may rely on. For example, Unix [49] and many other systems pass explicit I/O data by copying and, therefore, are said to have an explicit I/O interface with *copy* semantics. In programming language terminology, this corresponds to passing data *by value*. Applications written for copy semantics, e.g., Unix's `ftp`, may reuse and modify output buffers while a previous output request is still being processed, without concern about corrupting the previous request's data, because the system is known to automatically copy data of application output buffers to system buffers at the time of an output request. Similarly, applications can access input buffers during or after processing of an input request without concern that the data there may be inconsistent or erroneous, because the system is known to copy data from system to application input buffers only at the time of a successful reply. If the data passing semantics change, such applications may no longer execute correctly.

Copy avoidance optimizations pass data without copying, reducing data passing overheads without changing operating system structure. As ex-

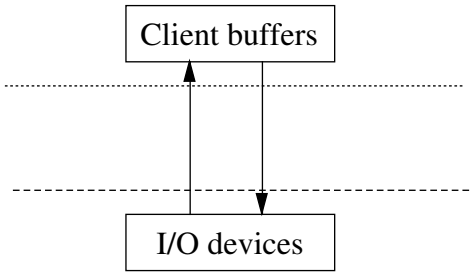


Figure 1.5: In-place data passing.

plained in the next subsections, many copy avoidance optimizations change data passing semantics, e.g. to *share* or *move* semantics, while others preserve copy semantics and, therefore, compatibility with the many applications written for such semantics.

Changing data passing semantics

Under *share* semantics, client buffers double as server buffers during the processing of a request. I/O occurs *in-place*, that is, directly to or from client buffers, without distinct intermediate server buffers, as shown in Figure 1.5. In programming language terminology, this corresponds to passing data *by reference*.

The explicit I/O interface *promotes* client buffers at request time and *demotes* them at reply time. Promotion may require VM manipulations, such as: (1) mapping client buffers to the server's address space, and (2) *wiring* the buffers, that is, making them unpageable. Inverse manipulations achieve demotion. For buffers larger than a certain minimum size, the cost of VM manipulations is usually equal to a small fraction of that of the avoided copying. Moreover, VM manipulations for in-place I/O can be eliminated by requiring client buffers to be located in a special region that is statically mapped to both client and server and, if necessary, unpageable.

Share semantics can offer the same explicit I/O interface as that of copy semantics, but does not guarantee the integrity of the contents of client buffers. Consequently, share semantics may be incompatible with applications, such as Unix's `ftp`, which reuse output buffers. Share semantics may also require special hardware support: The device controller must be able to input or output data directly to or from client buffers, with arbitrary location and length, as opposed to buffers from the device's own buffer pool.

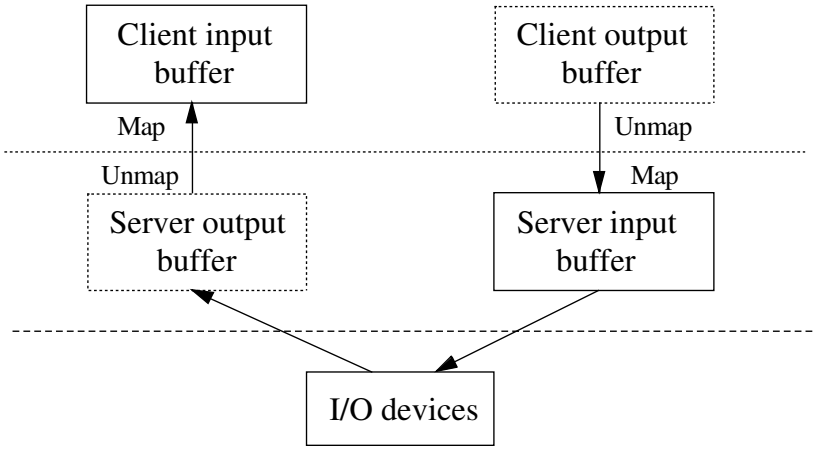


Figure 1.6: Data passing with move semantics.

Under *move* semantics, the explicit I/O interface passes data by *removing* a party's output buffers from that party's address space and *inserting* the buffers in freshly allocated regions in the other party's address space, as shown in Figure 1.6.

A party's output buffer *becomes* the other party's input buffer, and its pages carry the data without copying, being simply unmapped from one address space and mapped to another address space. These VM manipulations are usually more efficient than copying, but also imply that the owner of an output buffer cannot access the buffer after the I/O interface passes the data, and the owner of an input buffer cannot choose the buffer's location or layout. Consequently, move semantics requires an explicit I/O interface different from that of copy semantics and is incompatible with all applications written according to the latter.

Weak move semantics combines characteristics of move and share semantics. Weak move semantics passes data in-place, but also conveys ownership of the respective buffers from one party to the other. This conveyance is purely logical, unlike move semantics, which actually unmaps buffers from their previous owners. The explicit I/O interface is the same as that of move semantics, and the integrity guarantees and hardware requirements are the same as those of share semantics.

Chapter 2 develops a structured model of data passing semantics. Semantics changes can optimize both monolithic and microkernel systems. For example, to optimize data passing in the case of kernel-level servers, Firefly

RPC [68] uses share semantics, and the exposed buffering interface [12] uses both share and weak move semantics. Volatile cached fbuf output [29] uses share semantics, and both cached fbuf and volatile cached fbuf input use weak move semantics. LRPC [7] and URPC [8] copy data into and from statically shared regions for data passing between user-level clients and servers. Move semantics is used in several IPC mechanisms, including Tzou and Anderson's DASH [76], fbufs [29], and container shipping [64].

Preserving copy semantics

Data of length multiple of the page size in a page-aligned server output buffer can be passed to a client input buffer of matching alignment and length by *swapping* pages between the buffers. The I/O interface swaps each pair of pages at the same offset from the beginning of the respective buffer by invalidating all mappings of both pages, removing both pages from the respective memory object, inserting each page in the previous memory object of the other page, and mapping each page to the virtual address and address space where the other page was mapped. After swapping, the contents of the client input buffer is the same as if the data had been copied, but the contents of the server output buffer changes (it becomes equal to the contents of the client input buffer before swapping). Page swapping appears to have been used only in monolithic systems, such as IRIX, HP-UX, and Solaris [23].

Output buffer data can be passed in-place while preserving copy semantics by making the region that contains the data *copy-on-write* (COW). The I/O interface removes write permissions from all mappings of the pages in the region. A party's attempt to overwrite any such page will cause a VM fault. The system recovers from this fault by copying the page's data to a new page, swapping pages in the memory object of the faulted page, and mapping the new page to the virtual address of the faulted page in the party's address space, with writing enabled. Parties cannot overwrite output pages, which preserves integrity of the data, and copying only occurs if a party does attempt overwriting.

A page-level alternative to COW, reported to have better performance for network IPC [6], is *sleep-on-write*: The I/O interface removes write permissions from all mappings of pages in the region and marks the pages *busy* during the processing of the request. A party's attempt to overwrite any such page will cause the party to fault and stall until processing of the request completes. Therefore, this scheme can be used for data passing only from

client to server, and may deadlock if the server attempts to overwrite the region.

Another alternative to COW is *abort-on-write*, which requires buffers to be allocated and deallocated using a special interface. To pass data, the I/O interface removes write permissions from all mappings of the buffer pages. A party's attempt to overwrite the latter will cause a protection violation exception and normally will abort the party. The buffer remains read-only until both parties explicitly deallocate it. When that occurs, the buffer becomes eligible for reuse. Finally, when a party allocates a buffer, the previously deallocated buffer is again mapped to that party, with read and write permissions.

COW and abort-on-write can optimize both monolithic and microkernel systems. Mach IPC, for example, provides a selection of copy semantics (with or without COW) or move semantics [67]. Peregrine [43] uses COW for client output buffers, copying for client input buffers, and move semantics for server input and output buffers. Abort-on-write is used in cached fbuf output [29]. Sleep-on-write has been used only in user/kernel data passing [6].

Relationship with the memory-mapped I/O model

Mapped device I/O usually has share or weak move semantics.

File mapping can have copy, move, share, or weak move semantics. Relative to the client, mapping has *copy* semantics if the client specifies for mapping an appropriate region (usually between the end of the heap and start of the stack, with page-aligned start address and offset from the beginning of the file and length that are multiples of the page size); otherwise, mapping has *move* semantics. However, if multiple clients map the same file in shared mode, then mapping semantics relative to those clients becomes *share* or *weak move*, respectively. Relative to the server, if the file is not currently mapped in shared mode, mapping has *copy* semantics if the client selects mapping in private mode (which may be implemented by copying or COW), or *move* semantics if the client selects mapping in shared mode (the terminology may appear confusing; the file cache is passed from server to client with *move* semantics, but may be *shared* among multiple clients). After the file is mapped in shared mode, further mappings are simply VM operations, and do not cause data passing relative to the file server[5].

[5]This assumes that cached server buffers and VM pages are allocated from the same

Relative to the client, implicit requests on a mapped file, made by accessing a page that is not currently in physical memory, normally have *copy* semantics (i.e., input data exactly to the address accessed by the client). However, if the file is mapped by multiple clients in shared mode, the semantics becomes *share*. Relative to the pager, such requests always have move semantics, regardless of whether the file is mapped by multiple clients and whether the pager is implemented in kernel or user level.

File unmapping only causes data passing if the file was mapped in shared mode and no other shared mapping of the file exists. In that case, file unmapping has *move* semantics relative to both client and server.

In explicit I/O with cached server buffers (e.g., using a file server), input with move semantics or page swapping and I/O with share semantics may not be appropriate. Explicit input with move semantics or page swapping would deplete the server's cache, eliminating cache hits in future requests for the same data; explicit I/O with share semantics would allow clients to corrupt the server's cache. What makes file mapping in shared mode efficient is, more than its data passing semantics, the fact that the file cache is moved from server to client[6]. Explicit I/O, however, passes only the data, and not the cache, and therefore does not allow the same optimizations.

1.2.2 IPC avoidance

IPC avoidance reduces both control and data passing overheads in microkernel systems by changing the operating system structure so as to make IPC unnecessary.

IPC can be avoided by *server kernelization*, that is, moving user-level servers into the kernel. This optimization was employed, for example, in a recent release of the Windows NT GDI [18]. Kernelization usually lowers both data and control passing overheads, but at the cost of adopting a monolithic structure and forsaking benefits of a microkernel structure relative to a monolithic one.

Another alternative for IPC avoidance is to adopt a *librarized* operating system structure. In such structure, drivers are installed in the kernel and

pool, as is the case in many contemporary systems. If the pools are separate, mapping semantics relative to the server may always be copy.

[6] An interesting analogy is that moving a cache from server to client, over a network, is also a key ingredient in the performance and scalability of AFS [41], a distributed file system.

each remaining user-level server is decomposed into a *fastpath* component, which is linked as a library with user applications and processes common input and output requests, and a *slowpath* component, which remains a separate user-level server and ideally processes only exceptions. This goal has been achieved in the case of a TCP/IP server [52, 74], resulting in performance comparable to that of a kernel-level server.

However, librarized server organization and interfaces (and, consequently, design, implementation, debugging, and maintenance) are considerably more complex than those of comparable monolithic or microkernel servers.

Moreover, depending on the server, it can be quite hard to extract a fastpath component that does not depend on or modify global server state and is self-sufficient for input and output requests. For example, [53] has shown that such extraction was not possible for a file system. Some implementations allow libraries to access and modify global state, but thereby compromise system protection. In the library-based emulation of the Unix API in [45], for example, any application can corrupt certain system data structures of other applications. To prevent such corruption, the library would need to contact a server that manages or at least guards access to global state and is implemented in a separate protection domain. That, however, would reintroduce IPC, or at least a system call.

1.2.3 Operating system avoidance

I/O devices with special hardware support may allow mapped device I/O by unprivileged clients without compromising system protection or integrity [30, 33, 11, 17]. In the case of a network adapter with suitable hardware support, for example, the operating system can retain the processing of operations that affect the global operation of the device, but allow applications to open a *channel* to the device. A channel consists of control/status registers in the card and a buffer for data passing between application and card. The registers and buffer are mapped exclusively to the application that opened the channel; VM hardware maintains protection in all accesses. While the channel is open, the application can write directly into channel registers, without system call overheads, requests for input into or output from the channel buffer [30].

By avoiding indirection through the operating system, this scheme eliminates both data passing and control passing overheads in the common case. However, several difficulties make it hard to adopt this scheme in production

systems:

1. Data passing in channel buffers without copying does not preserve copy semantics and therefore causes incompatibilities with many existing applications, as explained in Section 1.2.1.

2. For applications that use multiple devices, it may be necessary to copy data between channel buffers of each device. Similarly, in an application that uses both mapped file I/O (for access to files) and mapped device I/O (for access to a network), it may be necessary to copy data between the file's mapped region and the network's channel buffer. Such copying would negate performance improvements of mapped device I/O[7].

3. Given that the application accesses the device directly, the functionality of any required servers and driver must be linked as a library to the application. As discussed in Section 1.2.2, library-based, decomposed servers can be considerably more complex than equivalent monolithic or microkernel servers and may not always be feasible or safe. For example, for quality of service guarantees, network drivers must schedule packets globally, which may be unsafe in code linked with user applications.

4. Few devices have channel registers. In devices that do have them, the number of channel registers is necessarily limited. Also, channel buffers must be unpageable, given that they are used by a device; unpageable memory is also a limited resource. It is not clear how the system would prevent the hoarding of channel registers and unpageable memory by applications.

1.2.4 Data passing avoidance

Explicit I/O requests and replies usually require data passing between client buffers and server buffers. *Data passing avoidance* consists in processing I/O requests without such data passing, or with only a reduced amount of it. Data passing avoidance involves changes in data passing semantics, and possibly also in operating system structure.

[7]This dissertation contributes a new technique, *user-directed page swapping*, that may help avoid such copying. User-directed page swapping is described in Section 10.1.

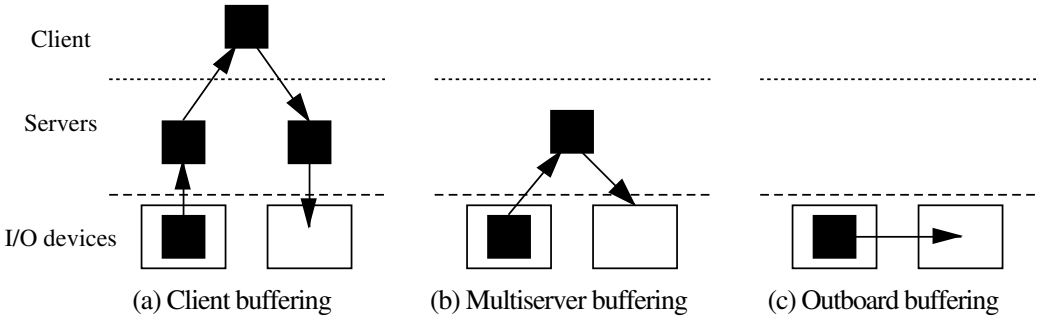


Figure 1.7: Data passing to and from client buffers (*a*) can be eliminated by using multiserver (*b*) or outboard (*c*) buffers.

Data passing avoidance uses special kinds of server buffers, *multiserver* and *outboard*. Multiserver buffers reside in host memory, are accessible by multiple servers, and can be used to pass data between servers without passing data to and from client buffers. Like other in-host server buffers, multiserver buffers can be ephemeral or cached. Outboard buffers reside in a device controller's outboard memory, can be allocated and deallocated by the device's driver, and can likewise be used to pass data directly between devices, without storage in host memory or data passing to and from client buffers.

A possible use for multiserver or outboard buffers would be, for example, transferring data directly from a video digitizer/compression card to a network driver for a networked multimedia application, as shown in Figure 1.7.

Multiserver buffers can also be used, as shown in Figure 1.8, for multicast clients, which output the same data to multiple servers. Data passing can be reduced by passing the data once from client to multiserver buffer, and then using that buffer for multiple output requests to servers that can access the buffer.

Because data passing avoidance reduces or eliminates data passing, rather than merely making the latter more efficient, data passing avoidance can be expected to provide greater I/O performance improvements than those of copy avoidance. However, data passing avoidance is also likely to require significant changes in the explicit I/O interface, and particularly in the server buffer allocation and naming schemes. Conventionally, explicit I/O requests and replies pass all I/O data between explicit client buffers and implicit server buffers. Each server buffer is automatically deallocated at the time of

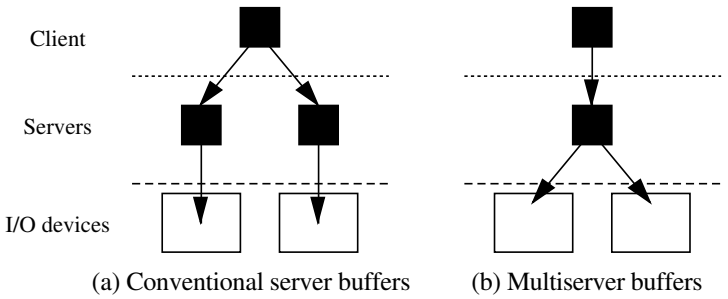


Figure 1.8: Multicast with a multiserver buffer reduces data passing between client and server buffers.

the respective reply (ephemeral buffer) or when another server buffer needs to be allocated (cached buffer). Multiserver and outboard buffers, however, must remain allocated for the processing of a certain number of I/O requests. Programming interfaces for data passing avoidance may:

1. Let the client explicitly allocate and deallocate server buffers, name them using capabilities in explicit I/O requests, and move data between client and server buffers. This is the solution used in the *container shipping* [64] facility, with kernel-level drivers.
2. Let the client specify, in an explicit I/O request, an input and an output operation using the same server buffer. This is the solution used in the *splice* [36] interface, with kernel-level servers.
3. Allow the client to run as an *extension* to the server, so that the client can directly access server buffers and use them in multiple explicit I/O requests. If the server is integrated in the kernel, this amounts to adopting an *extensible kernel* operating system structure, as implemented, for example, in SPIN [9], VINO [69], and facilities such as ASHs [34, 77].

1.2.5 Scheduling avoidance

Data passing avoidance, as explained in the previous subsection, makes it possible to use a given multiserver or outboard buffer to process multiple explicit I/O requests. An important question is how to schedule the processing of these requests. For example, after a multiserver or outboard buffer is filled with data from a video digitizer/compression card, that buffer may

be submitted to a network driver for data transmission. The transmission request to the network driver may occur in the context of:

1. a regularly scheduled process (e.g., the client itself), as in the *container shipping* interface [64];
2. a callout or software interrupt, as in the *splice* interface [36]; or
3. the interrupt that signals completion of the previous I/O request (e.g., video capture and compression), as in the ASHs facility [77].

The latter two alternatives are examples of *scheduling avoidance* optimizations, which, by eliminating context switching, reduce control passing overheads and may improve I/O performance. However, such unscheduled processing may also disrupt system scheduling, if, for example, the next requested server is integrated in the kernel and takes a long time processing.

1.3 Thesis

For storage-related and other I/O involving cached server buffers, mapped file I/O provides an interface that is widely adopted and offers low data and control passing overheads.

For network-related and other I/O involving ephemeral server buffers, however, no comparable alternative has been available. Conventional interfaces for such forms of I/O have high overheads. Most alternative interfaces, as discussed in Section 1.2, require changes in data passing semantics and/or operating system structure, and therefore are incompatible with many existing applications and systems. Consequently, they have not been widely adopted.

This dissertation's thesis is that, regarding network-related and other I/O with ephemeral server buffers:

1. Data and control passing optimizations that preserve data passing semantics and system structure can give end-to-end improvements competitive with those of data and control passing optimizations that change semantics or structure; and
2. Current technological trends tend to reduce differences in such improvements.

Specifically, this dissertation describes two novel copy avoidance schemes, *emulated copy* and *I/O-oriented IPC*, and shows that, in addition to preserving an explicit I/O interface with copy semantics and interoperating correctly and efficiently with mapped file I/O:

1. Emulated copy performs almost as well as or better than data passing schemes with non-copy semantics; and

2. I/O-oriented IPC gives user-level I/O servers performance approaching that of kernel-level ones.

The practical corollary of the thesis is that emulated copy and I/O-oriented IPC can significantly improve the performance of network-related and other I/O with ephemeral server buffers in conventional monolithic and microkernel systems, such as Unix and Windows NT. Coupled with mapped file I/O, emulated copy and I/O-oriented IPC provide a broadly applicable solution that preserves both the semantics of existing interfaces and the structure of existing systems and can result in I/O data and control passing overheads almost as low as those of optimal (but incompatible) modifications.

1.4 Outline of the dissertation

This Introduction presented models of I/O organization and approaches for reducing I/O data and control passing overheads. Chapter 2 expands this discussion by introducing a new data passing model that permits analyzing, in a structured way, data passing in systems of arbitrary structure. Analysis of new data passing optimizations is important for ensuring compatibility with existing I/O interfaces.

Chapters 3 and 4 present new copy avoidance optimizations for non-copy and copy semantics, respectively, for the case of kernel-level servers and ephemeral server buffers. New optimizations for in-place data passing are described (Section 3.2) that have broad applicability, enabling in-place implementations of data passing with move and weak move semantics (Section 3.3) and output with copy semantics (Section 4.2). *Emulated copy*, described in Chapter 4, can provide transparent copy avoidance in explicit I/O interfaces with copy semantics for network-related I/O. *Input alignment* enables emulated copy to input data by page swapping even when client buffers have arbitrary alignment and length (Section 4.1).

Chapter 5 describes *I/O-oriented IPC*, which expands the copy avoidance optimizations of the previous two chapters to the case of user-level servers. I/O-oriented IPC offers to user-level servers data passing semantics similar to that of kernel-level interfaces, enabling easy server migration between kernel and user level. I/O-oriented IPC exploits the asymmetric semantic requirements of clients and servers to pass input data to client buffers by page swapping. I/O-oriented IPC is possibly the first IPC facility to offer a client interface with copy semantics and avoid copying both on output and on input.

Chapter 6 discusses network adapter support for copy avoidance. For each level of hardware support, compensating software techniques for copy avoidance are discussed. A new feature, *buffer snap-off*, allows copy avoidance under very general conditions.

Chapter 7 describes an I/O framework, Genie, that implements the optimizations described in the previous chapters on the NetBSD operating system. Genie was used in the experiments of the following chapters.

Chapter 8 evaluates the performance of emulated copy in end-to-end communication over a fast network. Experiments show that emulated copy performs almost as well as or better than other copy avoidance schemes, including those with move or share semantics. Analysis of the results on multiple platforms and at different transmission rates suggests that current technological trends tend to increase the performance benefits of copy avoidance while also decreasing performance differences among copy avoidance schemes.

Chapter 9 evaluates I/O-oriented IPC. Experiments show that I/O-oriented IPC gives user-level servers performance approaching that of kernel-level ones. Results on different platforms indicate that, when I/O-oriented IPC is used, performance differences between user- and kernel-level servers are scaling roughly inversely to the processor's integer performance.

Chapter 10 shows that the new copy avoidance optimizations of Chapters 3 to 9, which are specific to ephemeral server buffers, interoperate efficiently with mapped file I/O. Mapped file I/O allows copy avoidance in the case of cached server buffers. Consequently, combined use of the latter and this dissertation's new copy avoidance optimizations allow data passing between networks and file systems without copying and preserving the semantics of existing interfaces and the structure of existing systems. This is experimentally demonstrated in that chapter.

Chapter 11 presents a new interface, *iolets*, that offers an alternative I/O model, with data passing avoidance, in systems with monolithic or micro-

kernel structure. Iolets enable *limited hijacking*, a safe form of scheduling avoidance.

Chapter 12 evaluates iolets and limited hijacking. Experiments show that, for device-to-device data transfers and for multicast applications, data passing avoidance can greatly reduce overheads when compared to conventional data passing by copying. However, the improvements are much less when compared to copy avoidance. In fact, for multicast applications, copy avoidance alone gives greater benefits than those of data passing avoidance alone. Analysis of the results suggests that, in many cases, performance differences between data passing avoidance and copy avoidance would not be observable because of application processing or saturation of the physical I/O subsystem. Performance improvements due to scheduling avoidance were modest.

Finally, Chapter 13 summarizes the dissertation's contributions, makes recommendations based on the dissertation's results, and points to future work.

Chapter 2

Data Passing Model

A good model of data passing can be a valuable tool for describing and analyzing data passing schemes and revealing sources of incompatibility. However, most previous models have not provided such tool. Some models have been descriptive only (e.g., [29]) and do not provide an analytical framework that can be generally applied. Other models have been analytical, but at a level excessively close to implementation techniques (e.g., [64]), and therefore do not clearly scrutinize or reveal incompatibility between schemes. Finally, certain models have described and analyzed compatibility, but only for systems of a particular structure (e.g., the model in [13] is applicable only to monolithic systems).

This chapter introduces a new data passing model that captures in a structured way those essential features of a data passing scheme that must be retained, for a desired level of compatibility, in proposed optimizations of that scheme — regardless of implementation. The new model can be used to describe and analyze data passing schemes encompassing a broad range of data passing semantics and system structures.

2.1 Semantics

For two data passing schemes to be compatible with each other, they must implement the same data passing semantics. As shown in Figure 2.1, data passing schemes can be analyzed according to three orthogonal characteristics: buffer allocation, buffer integrity, and optimization conditions. The following subsections discuss each dimension in turn.

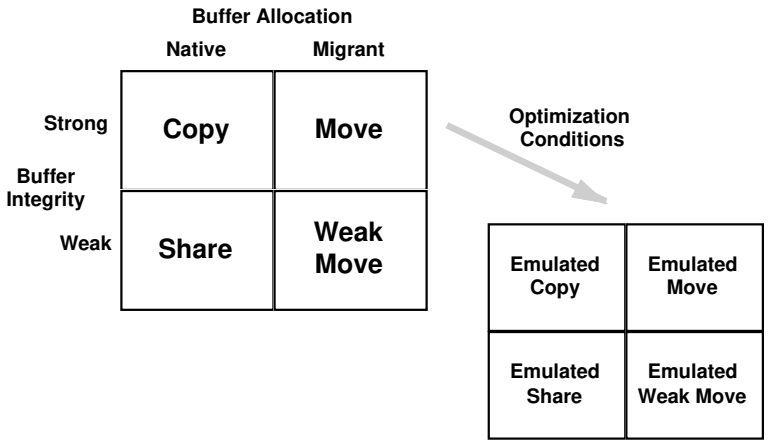


Figure 2.1: Taxonomy of data passing schemes. Buffer allocation and integrity define the *semantics* of a scheme. The *qualified semantics* also takes into account the scheme's optimization conditions.

2.1.1 Buffer allocation

The most fundamental requirement for compatibility between data passing schemes is that they have the same strategy for buffer allocation and deallocation. Data passing may or may not imply allocation and deallocation of the buffers that contain the data; each alternative determines a different programming interface.

Strategies for buffer allocation and deallocation establish a distinction between *migrant* and *native* buffers. *Migrant* buffers are those allocated and deallocated through an I/O interface. *Native* buffers are those otherwise allocated and deallocated. A given buffer can be simultaneously native with respect to one party and migrant with respect to another party. The owner of a migrant buffer can choose neither the location nor the layout of the buffer. The owner of a native buffer, on the contrary, generally can specify both. Buffer layout is how the data is distributed in memory, e.g. whether all contiguous or scattered, and can be represented by a vector of (offset, length) pairs, with each pair corresponding to one data segment of the buffer.

In *migrant-mode* data passing, the I/O interface automatically allocates input buffers and deallocates output buffers; the owner of an input buffer cannot choose the location or layout of the buffer, and the owner of an output buffer cannot or should not access the buffer after data passing. Such is the

case, for example, of the *move* semantics of Tzou and Anderson's DASH system [76].

In *native-mode* data passing, on the contrary, the I/O interface does not allocate or deallocate buffers; the owner of an input buffer determines its location and layout before data passing, and the the owner of an output buffer can still access the buffer after data passing. Such is the case, for example, of the *copy* semantics of the Unix explicit I/O interface [49].

Migrant-mode and native-mode data passing semantics necessitate different programming interfaces. The main difference regards input buffers: In migrant-mode interfaces, the location and layout of input buffers are *output* parameters, returned by the interface; in native-mode interfaces, the location and layout of input buffers are *input* parameters, passed to the interface.

Migrant-mode interfaces also include primitives for explicit migrant buffer allocation and deallocation. Parties with balanced amounts of input and output may be able to avoid explicit buffer allocation and deallocation by reusing input buffers as output buffers.

Migrant-mode interfaces should accept as output buffers only migrant buffers. This restriction prevents native regions that must be kept contiguous, such as the stack or the heap, from becoming discontiguous because a migrant-mode interface accepts part of the region as an output buffer and deallocates it, making the region discontiguous. Native-mode interfaces can accept as input or output buffers both native and migrant buffers.

Compared to native-mode data passing, migrant-mode data passing imposes more constraints on the parties and fewer constraints on the interface, which therefore can be more easily optimized. However, parties that require access to output buffers after data passing or that are sensitive to data layout, e.g., those using data structures such as arrays, may not be able to use migrant-mode interfaces without copying between migrant buffers and native buffers or data structures. This copying may defeat eventual performance advantages of a migrant-mode interface.

2.1.2 Buffer integrity

A second, more subtle requirement for compatibility between data passing schemes regards buffer integrity. For a data passing scheme s' to be compatible with programs written for data passing scheme s, s' must make buffer integrity guarantees at least as strong as those of s.

Buffer integrity guarantees can be *strong* or *weak*. *Strong-integrity* data

passing guarantees that: (1) the owner of an output buffer cannot, by overwriting the buffer after data passing, affect the contents of the other party's input buffer; and (2) the owner of an input buffer can access the buffer only in the states as of before an input request or after successful reply, but no intermediate, inconsistent, or erroneous state. *Weak-integrity* data passing makes no such guarantees.

Copy and move semantics provide strong integrity because each party cannot directly access the other party's buffers.

Weak integrity allows *in-place* data passing, that is, data passing using buffers that can be accessed by both parties. The client can access these buffers while its request is being processed and, consequently, can corrupt output data or observe input data in inconsistent states.

Native-mode weak-integrity data passing is called *share* semantics, whereas migrant-mode weak-integrity data passing is called *weak move* semantics. Under weak move semantics, an output buffer remains physically accessible to its previous owner after data passing, but this previous owner *should not* access the buffer because the other party logically becomes the owner of the buffer and may reuse it.

For weak-integrity, in-place input, requests have to be made *before* input physically occurs. If this condition is not met (e.g., when a packet is received unexpectedly from a network), input can be completed according to the strong-integrity semantics with the same buffer allocation scheme (i.e., share reverts to copy semantics, and weak move reverts to move semantics).

For correctness, clients should not access an input buffer during input request processing with weak-integrity semantics. Clients should also not overwrite an output buffer during output request processing with share semantics, or access an output buffer after making an output request with weak move semantics.

2.1.3 Optimization conditions

The dimensions *buffer allocation* and *buffer integrity* define the *semantics* of a data passing scheme. Therefore, there are four data passing semantics: copy, move, share, and weak move.

Each semantics may admit many different optimizations, some of which may depend on special conditions. The *qualified semantics* of a data passing scheme is defined by the scheme's semantics and special *optimization conditions*. Contrary to the other two dimensions, which each have two discrete

points, this dimension admits a spectrum of possibilities, including many not discussed here.

Optimization conditions may be as important as semantics for compatibility between data passing schemes. For a scheme s' to be compatible with applications written for scheme s, s' may have less conditions than those of s, but must not mandate any new conditions.

Some optimization conditions may be *spatial*, restricting, for example, buffer location, alignment, or length. Other optimization conditions may be *temporal*, restricting, for example, when requests should occur or when a party may access its buffers. The spatial restrictions of migrant-mode data passing, explained in Section 2.1.1, and the temporal restrictions of weak-integrity data passing, explained in Section 2.1.2, are intrinsic to the respective semantics and not special optimization conditions. Likewise, the spatial restrictions of memory-mapped I/O, noted in Section 1.1.3, are intrinsic to that I/O model.

The *restrictiveness* of an optimization is the likelihood that an application not aware of the optimization will not meet the optimization's special conditions. *Hard* conditions are those that are met by practically no application not aware of the optimization. Hard conditions usually require changes or additions to I/O programming interfaces and modifications in previously written applications. *Soft* conditions are those that are not hard. Soft conditions may involve additions to I/O programming interfaces, but do not generate incompatibility with previously written applications.

The *criticality* of an optimization is the degree to which non-conformance with the optimization's conditions causes performance to worsen relative to the base case against which the optimization is claimed. At one end of the criticality spectrum are *mandatory* conditions, those that must be met for data passing to occur or that impose heavy penalties if not met. At the other end of the spectrum are *advisory* conditions, which if not met do not cause substantial penalty.

For example, interfaces such as those of fbufs [29], exposed buffering [12], and operating system avoidance (Section 1.2.3) use optimizations that require client buffers to be located in special regions. Data passing fails if the data location does not conform to such requirement, which therefore is mandatory. Moreover, virtually no application not aware of such regions place their data in them, which makes such requirement hard.

Incompatibilities such as these can often be alleviated by a user-level library that copies I/O data between application buffers and buffers that

conform to the qualified semantics of the underlying data passing scheme. If incompatibility between schemes s' and s is due only to disagreements regarding implicit buffer allocation/deallocation and optimization conditions, the library can, by copying and explicitly allocating and deallocating buffers, convert s' into a new scheme s'' that *is* compatible with s. This would be necessary, for example, to run existing applications that expect copy semantics under a data passing facility, such as fbufs [29], that does not provide such semantics. However, given the library's copying overheads, s'' may not provide, for existing applications, any performance improvements relative to s. Moreover, incompatibility due to weaker buffer integrity guarantees may not be fully fixed by copying: Even with copying, misbehaved applications may still be able to corrupt I/O data.

Optimization conditions can also hurt copy avoidance in the interoperation of different I/O subsystems. For example, applications that use mapped files for file access and fbufs [29] or operating system avoidance (Section 1.2.3) for network access may have to copy data between mapped file regions and fbufs or channel buffers. Because of special conditions, the benefits of optimizations in each I/O subsystem may not be fully realized when the subsystems have to interoperate[1].

This dissertation describes and evaluates the implementation of one standard and one optimized explicit data passing scheme for each of the data passing semantics, as shown in Figure 2.1. All schemes other than *copy* avoid copying. The optimized schemes are called *emulated* because they have at most only soft, advisory optimization conditions and therefore do not require a "compatibility library" that may negate performance improvements. Moreover, they interoperate efficiently with each other and with mapped file I/O.

2.2 Protection

If a data passing scheme s' has qualified semantics compatible to that of s, programs written for s will run correctly under s'. However, data protection may be different under s and s'.

The *protection model* of a data passing scheme specifies what client data a server can access by virtue of a request, whether for reading and/or writing,

[1] This dissertation contributes a new technique, *user-directed page swapping*, that may help such interoperation. User-directed page swapping is described in Section 10.1.

and whether the data is pageable with backing by a client-supplied pager. No distinction is made regarding client access to server data, because no data passing scheme should allow clients to access *any* server data.

The prototypical *fully protected* data passing scheme is IPC between user-level clients and servers with data passing by copying: A server can strictly only read from the client's output buffers at the time of the request, can only write into the client's input buffers at the time of the reply, and the server's copy of client data is not backed by a client-supplied pager.

On the other hand, the prototypical *unprotected* (in this particular sense) data passing scheme is any scheme that passes data to or from a kernel-level server: The server can read or write any client data at any time, and such client data may be pageable and backed by a client-supplied pager.

IPC facilities with copy avoidance usually offer a protection model that is somewhat weaker than fully protected but stronger than unprotected. Copy avoidance can weaken IPC protection in the following ways:

1. *Page-sized granularity* — Copy avoidance techniques typically use VM manipulations to avoid copying. For that reason, protection granularity with copy avoidance often is page-sized, whereas with copying it is byte-sized. With copy avoidance, servers usually can access any other client data on the same pages as client buffers.

2. *Read access to client input buffers* — In facilities that pass data to client input buffers in-place or by page swapping (Sections 1.2.1, 4.1, 5.2), the server can often not only write into client input buffers but also read them.

3. *Paging by client-supplied pager* — If data passing is in-place (e.g., by COW) or migrant-mode and client data is passed in a buffer that is pageable, the server may take a VM fault on an access to the data. If the buffer is backed by an untrusted client-supplied pager, then the server may never recover from such fault.

Clients that do not trust a given user-level server may choose to rearrange the layout of their buffers or use IPC with data passing by copying. Untrusted client-supplied pagers, however, pose a more fundamental safety problem: In general, to protect their own integrity, servers would need to refuse IPC with copy avoidance on pageable client buffers.

2.3 Symmetry

A data passing facility is called *symmetrical* when it handles both output and input buffers of both client and server with the same qualified semantics. Symmetry makes programming interfaces simpler and more uniform. However, many facilities, especially optimized ones, are asymmetrical.

IPC facilities with copy, move, or weak move semantics often are symmetrical — e.g., Mach IPC (in the cases of move or copy semantics without COW) [37], DASH (move semantics) [76], and container shipping (move semantics) [64].

On the other hand, facilities with in-place native-mode data passing, including COW, COW variants, and share semantics usually are asymmetrical: To avoid copying, the facility migrates a client's in-place buffers to and from the server (i.e., the server does not choose the location or layout of its buffers and cannot access its output buffers after processing of the request completes) even if the client passes the data with native-mode semantics (i.e., the client retains access to its output buffers and determines the location and layout of its input buffers). Examples of asymmetrical facilities include Mach IPC (in the so-called *out-of-line* case of output with COW/input with move semantics) [37], Peregrine RPC (where client buffers have copy semantics and server buffers have move semantics) [43], and cached fbufs and volatile cached fbufs (where output is with abort-on-write and share semantics, respectively; input is with weak move semantics) [29].

Data passing between original clients and kernel-level servers in monolithic systems is normally asymmetric. The I/O interface often *copies* data between client and system buffers (e.g., mbufs in Unix [49]), but *migrates* system buffers to and from servers, with weak integrity (even if they shouldn't, servers can access system buffers before input or after output). Therefore, the semantics for clients is copy, but for servers, the semantics is weak move.

Data passing in subcontracts in monolithic systems usually can also be asymmetric. Output buffers can be handled with weak move semantics (e.g., in Unix, when a protocol layer does not gain an extra reference to an mbuf before passing it to another layer) or with share semantics (e.g., in Unix, when TCP allocates a reference to an mbuf before passing it to IP). Input buffers are handled with weak move semantics.

2.4 Summary

For a data passing scheme s' to be compatible with programs written for another scheme s, it is necessary that:

1. s' agree with s on whether data passing implies allocation and deallocation of the buffers that contain the data — either option determines a fundamentally different programming interface;

2. s' provide buffer integrity guarantees at least as strong as those of s — otherwise, I/O data may be corrupted; and

3. s' mandate no optimization conditions not also mandated by s.

Emulated copy (Chapter 4) is a new data passing scheme that preserves both the buffer allocation strategy and buffer integrity guarantees of copying. Emulated copy does have optimization conditions not mandated by copying, but such conditions are soft and advisory only. Consequently, emulated copy can optimize many unmodified applications, and does not significantly penalize non-conforming applications.

Data passing between user-level client buffers and kernel-level server buffers usually has asymmetric semantics. *I/O-oriented IPC* is a new IPC facility, described in Chapter 5, that exploits similar asymmetry for data passing between user-level client and server buffers, thereby achieving easy server migration and similar performance for kernel- and user-level servers.

IPC with pageable in-place client buffers can be unsafe because of untrusted client-supplied pagers. I/O-oriented IPC makes in-place client buffers unpageable during I/O requests, safeguarding servers from untrusted client-supplied pagers.

Chapter 3

Copy Avoidance

Copy avoidance can significantly improve I/O performance without changing the structure of the operating system. Mapped file I/O already provides copy avoidance in storage-related and other I/O involving cached server buffers. This and the next four chapters introduce new optimizations for network-related and other I/O involving ephemeral server buffers, using the explicit I/O model. The interoperation of these new optimizations with mapped file I/O is discussed in Chapter 10.

Copy avoidance may or may not change data passing semantics. Conventional explicit I/O interfaces, such as those of Unix [49] and Windows NT [26], usually have copy semantics. This chapter describes new techniques for data passing with non-copy semantics, while the next chapter describes emulated copy, a new scheme that preserves copy semantics.

The emphasis in this dissertation is in preserving existing interfaces and, consequently, copy semantics. The inclusion in this dissertation of new optimizations for non-copy semantics serves two purposes. First, these optimizations remove bias: It would be unfair to compare emulated copy, a highly optimized scheme with copy semantics, only with unoptimized implementations of non-copy semantics. Second, optimizations for in-place data passing (e.g., share semantics) can also be used for emulated copy output.

The identification of copy avoidance techniques that have broad applicability, such as those for in-place I/O, is actually an important goal of this dissertation. Copy avoidance schemes are described in terms of the individual techniques used to implement them, so as to highlight similarities and differences among copy avoidance alternatives. Chapter 7 shows how the different techniques are composed to implement copy avoidance according to

the various semantics.

3.1 System buffers

This dissertation's optimizations assume that system buffers are unpageable. Some system buffers could, in principle, be pageable, while others, especially those used by drivers, must be unpageable. The option to make all system buffers unpageable is common in monolithic systems (e.g., Unix [49]); it affords simplicity and avoids the costs of building the VM data structures necessary for pageability and of dynamically wiring/unwiring buffers. In a microkernel system or any system where a server may execute in a context other than the client's (e.g., at interrupt level, in an extensible-kernel system with scheduling avoidance), the option of making system buffers (and, consequently, also promoted in-place or migrant client buffers) unpageable can safeguard servers or the system from client attacks with untrusted client-supplied pagers, as explained in Section 2.2.

The rest of this section describes how unpageable system buffers can be made efficient and safe without imposing special optimization conditions.

3.1.1 Unmapped access

In-place and migrant-mode data passing generally require mapping and unmapping to the server buffers that are, were, or will be owned by the client. To avoid such mapping and unmapping operations, many previous optimizations require client buffers to be located in special regions, e.g., fbuf regions [29] or exposed buffer areas [12], permanently mapped to both parties.

Unmapped access is an optimization that makes it unnecessary to map or unmap system buffers to or from the kernel address space, regardless of their location. In some processors (e.g., Alpha, MIPS), unmapped access is a hardware feature that allows kernel-mode accesses to physical memory using virtual addresses equal to physical addresses plus a fixed offset that bypasses the page table. In such cases, unmapped access may reduce TLB contention. In processors that do not provide this feature (e.g., Pentium), the I/O interface can emulate it by inserting, in the kernel's page table, *aliasing* entries mapping the entire physical memory and, in the machine-independent representation [67] (if any) of the kernel address space, a region serving as a placeholder for the corresponding virtual addresses. These aliasing entries are

not noted in any other VM data structures (e.g., physical-to-virtual tables), so that they are never invalidated. The placeholder region does not refer to any underlying memory objects.

Unmapped access eliminates server-side mapping and unmapping overheads for servers integrated in the kernel. Given that in-place buffers remain mapped to the client throughout processing of a request, the total mapping and unmapping overhead is the same as that of a special permanently co-mapped region, but without the location restrictions. As explained in Section 3.3, migrant-mode data passing can also be performed in-place and therefore benefit from this optimization.

3.1.2 Request eviction

Certain precautions are necessary to prevent user-level clients or servers from hogging physical memory. Clients can, for example, request input of packets that never arrive, and such requests may use in-place buffers. In-place buffers are promoted to system buffers and therefore are unpageable for the duration of the respective requests. Likewise, clients may make requests to user-level servers that never reply. As explained in Chapter 5, the I/O interface maps system buffers to user-level servers for processing. If the I/O interface held such requests indefinitely, the system could run out of physical memory, system buffers, or both.

The I/O interface, therefore:

1. Enforces per-process and global *limits* on the amount of physical memory occupied in pending I/O requests. If the interface determines that a given request would exceed one of these limits, the interface blocks the client and enqueues the request for later retrial, if the request is synchronous, or returns an indication of buffer exhaustion, if the request is asynchronous. Processes can request adjustment of their limit up to a hard limit imposed by the interface. This mechanism is similar to that of Unix's *socket* interface.

2. Enforces a configurable maximum *timeout* interval for every original request. If a given request times out, the interface *evicts* it, as explained below.

3. Maintains per-process lists of pending I/O requests. When a process terminates, the interface evicts all of the process's pending requests.

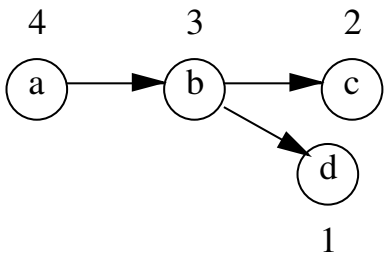


Figure 3.1: If request a originates request b, which in turn originates requests c and d, then c and d are evicted first, followed by b, and finally by a.

To evict a request, the interface aborts every request in its subcontract graph in *postorder* fashion (that is, for each request in the graph, the interface aborts the request *after* all its originated requests), as shown in Figure 3.1.

The interface aborts a request r by making an *abort* request r' to the server of r. Abort is a service that every server must provide. Abort requests are timed out. If request r' to abort request r times out, and the server of r executes at user level, the interface unilaterally unmaps the system buffers of r from that server's address space.

Eviction of the subcontract graph in postorder has the following advantages:

- It prevents race conditions between each request to abort a request r and completion of the originated requests of r. For example, in Figure 3.1, abortion of a before b could be premature, occurring exactly when b is completing successfully. In such case, abortion in postorder, of b before a, might allow the successful completion of b to propagate to a, which then might also complete successfully, rather than be aborted.

- It makes it unnecessary for the server of each aborted request r to make new subcontract requests to abort the originated requests of r or to reply to r.

- It ensures that all subcontract requests are aborted, including those whose contractor is an untrusted user-level server. This guarantees that the original request's buffers are actually freed.

3.2 In-place data passing

A major alternative for copy avoidance is to pass data with weak integrity guarantees (e.g., share semantics). Weak integrity guarantees allow data to be passed in-place, without copying.

In-place data passing may require, at request time, the wiring and mapping of client buffers to the server's address space, and, at reply time, the unwiring and unmapping of those buffers from the server's address space. To avoid wiring/unwiring and mapping/unmapping costs, several previous optimizations require client buffers to be located in special regions, e.g., exposed buffer areas [12], that are permanently wired and mapped to the address spaces of both client and server.

Special regions are not, however, necessary for optimization of in-place data passing. This section describes new techniques that make it possible to safely and efficiently promote and demote client buffers regardless of their location, for the case of kernel-level servers. Chapter 5 extends these techniques to the case of user-level servers.

3.2.1 Page referencing

In-place data passing schemes described in this dissertation all use *page referencing*. Page referencing consists in keeping counts of input and output references to each physical page in the system in pending I/O requests (i.e., how many times each page appears in a client's input or output buffer).

At request time, the I/O interface combines in page referencing the activities of verifying access rights, obtaining the physical address, and incrementing the input or output reference count of each page in in-place client buffers. Page referencing usually consults only the page table entry corresponding to each page. If an entry doesn't exist, is invalid, or provides insufficient access rights, page referencing invokes a VM fault, which may allocate a fresh new page, page in the data from the backing storage device, update the page table with respect to the machine-independent description of the client's address space, or return an error code [67]. At reply time, the I/O interface *unreferences* in-place pages by decrementing their reference counts.

3.2.2 I/O-deferred page deallocation

In the case of kernel-level servers, the mapping and unmapping overheads of in-place data passing can be avoided by using unmapped access (Section 3.1.1). However, unmapped access does not dynamically gain references to clients' memory objects. If a region in a client's address space holds the only reference to a memory object, the object's pages will be deallocated when the client terminates or explicitly deallocates the region, even if a server is, at the time, using the pages for in-place I/O. If the system then reallocated any such page with pending I/O to another party, there might be corruption of the other party's memory (when input occurs) or of output data (when the other party overwrites the page).

To make unmapped-access in-place I/O safe, the explicit I/O interface enforces *I/O-deferred page deallocation*. The system's page deallocation routine is changed to refrain from placing pages with nonzero input or output reference count in the list of free pages, whence they might be reallocated to other parties. When returning the reply to an I/O request, if a given page no longer has any input or output references, the I/O interface verifies whether the page is still allocated to a memory object; if not, the I/O interface assumes that the page was deallocated during I/O, and enqueues it in the list of free pages for reutilization.

In this dissertation, I/O-deferred page deallocation is used for all in-place I/O.

3.2.3 Input-disabled pageout

The wiring and unwiring overheads of in-place data passing can be avoided by using *input-disabled pageout*. Input-disabled pageout consists in changing the system's pageout daemon so that it refrains from paging out pages with nonzero input reference count. Such pages are poor candidates for pageout, because:

1. Pending input would modify these pages *after* pageout, making paged out data inconsistent; and

2. The owners of the input buffers containing these pages are likely to access them after completion of the corresponding requests.

The daemon is allowed to page out pages with zero input count normally, regardless of their output reference count.

This optimization adds no overhead to page referencing and makes it unnecessary to wire input and output buffers, in the usual sense of removing their pages from lists where the pageout daemon might find them.

In this dissertation, input-disabled pageout is used for all optimized in-place data passing schemes. It is not used for the unoptimized share and weak move schemes, which rely on region wiring and unwiring.

3.2.4 Input-disabled copy-on-write

COW is frequently used to optimize IPC or memory inheritance with copy semantics [67]. However, it may not implement copy semantics correctly if there is a pending in-place input operation in the region. Indeed, if the input is by DMA, the input will modify memory without generating any write faults, even though the pages are mapped read-only. Consequently, changes may be observed by processes other than the one that issued the input, and COW in this case actually implements share rather than copy semantics.

Input-disabled COW maintains COW correctness by monitoring the total number of input references to pages of each memory object in pending I/O requests. The explicit I/O interface updates these counts at page referencing and unreferencing. The system's COW set-up routine is modified to perform a physical copy, instead of setting up COW, if any of the region's backing memory objects has nonzero input count.

The reverse case, when a region is marked COW before in-place input, does not require special handling, because input page referencing verifies write access rights, which will automatically fault-in a private, writable copy of the data.

In this dissertation, input-disabled COW is used for all in-place input.

3.3 Migrant-mode data passing

Another major alternative for copy avoidance is to pass data with migrant (move or weak move) semantics. Migrant buffers accompany the data they contain, and therefore do not require copying.

Migrant buffers can be represented as regions that are marked *immigrant* when owned by the client, whereas all other regions are marked *native* when allocated. The migrant-mode interface accepts only immigrant regions as output buffers or for explicit deallocation. As explained in Section 2.1.3, such

restrictions are intrinsic to migrant semantics, and not special optimization conditions.

The prototypical migrant-mode data passing scheme, move, passes data by unmapping buffers from one party and mapping them to the other party. This section describes new techniques that implement migrant-mode data passing *in-place*, without mapping/unmapping overheads, and with weak or strong integrity.

3.3.1 Region caching

Region caching implements weak-integrity migrant-mode data passing (i.e., weak move semantics) in-place as follows: The explicit I/O interface marks immigrant regions that correspond to output buffers or are explicitly deallocated *weak emigrant* and enqueues them in the corresponding list, per client, where the I/O interface can find them for later reuse. To allocate a weak-integrity migrant buffer for a client, the explicit I/O interface dequeues a weak emigrant region from the client's list and marks the region again *immigrant*.

In this dissertation, region caching is used in the *weak move* and *emulated weak move* data passing schemes.

3.3.2 Region hiding

Region hiding implements strong-integrity migrant-mode data passing (i.e., move semantics) in-place as follows: The explicit I/O interface invalidates all mappings of pages of immigrant regions that correspond to output buffers or are explicitly deallocated, marks such regions *emigrant*, and enqueues them in the corresponding list, per client, where the I/O interface can find them for later reuse. The system's VM fault handler is modified to recover from faults only in *native* or *immigrant* regions. Attempts by a client to access an output buffer after data passing will therefore cause unrecoverable VM faults, as would be the case if the region had actually been removed, but the region and its pages remain allocated to the client's address space. To allocate a strong-integrity migrant buffer for a client, the explicit I/O interface dequeues an emigrant region from the client's list. The interface then makes the region again accessible by reinstating read and write access permissions to the region's pages and marking the region *immigrant*.

In this dissertation, region hiding is used in the *emulated move* data passing scheme.

3.4 Summary

This chapter describes new optimizations for explicit I/O with share, move, and weak move semantics, for the case of kernel-level servers and ephemeral server buffers. None of these optimizations require special conditions. The next chapter describes new optimizations for explicit I/O with copy semantics. Chapter 5 extends these techniques to the case of user-level servers.

This dissertation's new optimizations assume unpageable system buffers, which simplify or eliminate server-side mapping/unmapping overheads. Depletion of system buffers is avoided by: (1) per-process and global limits in the amount of physical memory in pending I/O requests, and (2) request eviction in cases of timeout or process termination.

Data can be passed in-place (e.g., with share semantics), without mapping/unmapping and wiring/unwiring overheads, by using page referencing, I/O-deferred page deallocation, input-disabled pageout, and input-disabled COW.

Migrant buffers can also be passed in-place, without mapping/unmapping and wiring/unwiring overheads, by using region hiding (move semantics) or region caching (weak move semantics) in addition to the techniques for in-place data passing.

Chapter 4

Emulated Copy

Copy avoidance does not imply non-copy semantics. This chapter describes a new scheme, *emulated copy*, that passes data to or from client buffers without copying but with copy semantics. Emulated copy is specifically designed for network-related and other explicit I/O with ephemeral server buffers (i.e., server buffers that are allocated at request time and deallocated at the corresponding reply time). Because emulated copy preserves copy semantics, it can be used to optimize the explicit I/O interfaces of systems such as Unix [49] and Windows NT [26], which also have copy semantics.

Emulated copy uses a new technique, *input alignment*, for input by page swapping even when the client buffer is not page-aligned or is of length that is not multiple of the page size (Section 4.1). Another new optimization, *transient output copy-on-write* (TCOW), allows emulated copy to output data in-place and with strong integrity guarantees (Section 4.2).

The analysis in Section 4.3 shows that the optimization conditions of input alignment and TCOW are soft and advisory only. Consequently, emulated copy can improve the performance of unmodified existing applications and does not significantly penalize non-conforming applications.

The network adapter support necessary for emulated copy is discussed in Chapter 6. Experiments in Chapter 8 demonstrate that emulated copy performs almost as well as or better than data passing schemes with non-copy semantics, including move and share.

4.1 Input alignment

In-place data passing to client input buffers may corrupt the latter and therefore cannot be used to implement copy semantics. For example, if network packet data is received directly into a client buffer, and the packet CRC is determined to be wrong, the client buffer ends up corrupted with incorrect data. To guarantee client buffer integrity, it is necessary to input data into a separate server buffer and, only after successful input completion, pass the data to the client buffer.

Integrity guarantees on client buffers do not, however, mandate copying. If client and server buffers are distinct but start at the same page offset, it is possible to pass data from server to client buffer by swapping pages between them. After swapping, the contents of the client buffer are the same as if data had been copied. The contents of the server buffer, however, are modified as a side effect. Because the buffers of kernel-level servers are usually migrant (Section 2.3), the side effect of page swapping is normally inconsequential for such servers: Buffers are modified at reply time, when they are also being implicitly deallocated. Chapter 5 shows that user-level servers can also use migrant buffers and therefore benefit from data passing by input alignment and page swapping.

Page swapping has been used before for data passing from kernel-level server buffers to user-level client buffers (Section 1.2.1). However, contrary to what has been commonly assumed [23], it is not necessary that client buffers be page-aligned or of length multiple of the page size. Pages partially occupied by buffer data can be handled as follows. Let t_i be a configurable threshold for emulated copy input and l be the length of buffer data on the page. If l is shorter than t_i, it is more efficient to copyout the data from server to client page, as shown in item 1 of Figure 4.1. If l is longer than t_i, however, it costs less to complete the server page with the complementary data of the corresponding client page, using *reverse copyout*, that is, copying from client to server page, and then swap pages between the buffers, as illustrated in items 3 and 4 of Figure 4.1.

If the cost of page swapping is s, the cost of copying is cl (where l is the length of the data copied), the data lengths in corresponding client and server pages are l_c and l_s, respectively, $l_d = \min(l_c, l_s)$, and p is the page size, copying data from server to client page costs less than swapping the two pages if:

$$cl_d < s + c(p - l_d) + c(l_s - l_d)$$

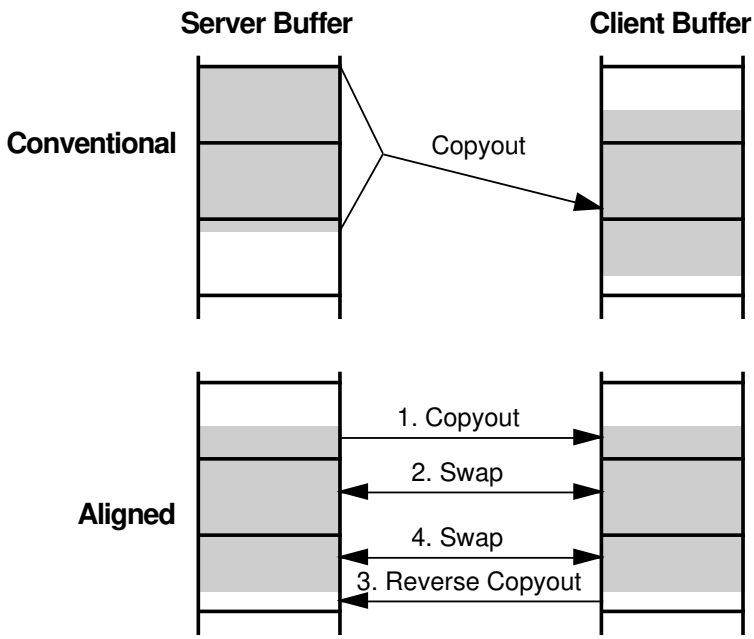


Figure 4.1: Conventionally, both client and server buffers are allocated without concern for alignment, and all data needs to be copied. Input alignment enables page swapping.

where $(p - l_d)$ is the length of data that needs to be copied from client to server page, by reverse copyout, before swapping, and $(l_s - l_d)$ is the length of server data, beyond the end of client data, that the server needs to save before reverse copyout. If $l_c = l_s$, this relation reduces to:

$$l_c < t_i = \frac{\frac{s}{c} + p}{2}$$

where t_i is usually only slightly greater than $p/2$ (half the page size).

Alignment between server and client buffers can be achieved in one of two ways:

1. *Server-aligned buffering* – The server allocates buffers starting at the same page offset and having the same length as the corresponding client buffers. This requires that:

   (a) The client inform the server about the alignment and length of client input buffers *before* input occurs (possibly using a preposted,

asynchronous input request) or, in the case of networking, the sender inform input buffer location in packet headers [17, 73] or implicitly by the connection used [59]; and

(b) Any directly or indirectly subcontracted servers and underlying device controllers can customize input buffers for any selected alignment and length.

2. *Client-aligned buffering* – The client queries the server (e.g., protocol stack) about the *preferred alignment and length* of client input buffers and lays out client buffers accordingly. The preferred alignment may be nonzero, for example, because of unstripped packet headers, and the preferred length may not be a multiple of the page size, for example, because of network maximum transmission units.

In the terminology of Section 2.1.3, server-aligned buffering imposes a temporal condition, while client-aligned buffering imposes a spatial condition. Depending on the particular workload, compared to client-aligned buffering, server-aligned buffering may be less restrictive (because client buffers have arbitrary alignment and length) or more restrictive (because input requests do not occur before physical input). Existing APIs typically do not have calls for querying the preferred alignment and length. However, examination of existing Unix programs (especially system libraries and utilities) shows that client buffers often are allocated via `malloc`, have length multiple of the page size, and therefore are page-aligned. Several commercially available copy avoidance schemes (e.g., [23]) optimize assuming such client-aligned buffering. Both server-aligned and client-aligned buffering impose only advisory conditions: With a properly tuned t_i, the cost of data passing is never greater than that of copying.

4.2 Transient output copy-on-write (TCOW)

Systems derived from 4.4 BSD Unix, such as NetBSD, have VM system similar to that of Mach [67]. In such systems, conventional COW can be too expensive for copy avoidance in I/O data passing [6]. Page referencing (Section 3.2.1), however, enables a specialized form of COW, transient output copy-on-write (TCOW), that is highly efficient for such purpose.

TCOW differs in two ways from Mach's conventional COW. First, TCOW is transient — after having been set, Mach COW persists indefinitely, while

TCOW only lasts during processing of an I/O request, which is when it is actually useful. Second, TCOW operates at page level instead of at region level. This allows TCOW to prevent the proliferation of regions on each I/O request and reduce the number of VM data structures that it needs to manipulate.

TCOW optimizes output as follows. Let t_o be a configurable threshold for emulated copy output. At request time, for each page in the client output buffer, if the data length is less than t_o, TCOW allocates a system page and copies the data to it; otherwise, TCOW references (i.e., increases the output reference count) and removes write permissions from all mappings of the client page. The system routine that maps a page to a virtual address also has to be modified as follows: If the page has nonzero output reference count, then the routine maps the page without write access permissions. Likewise, the system routine that changes access permissions to a page has to be modified to refrain from allowing write access permission to a page that has nonzero output reference count.

Client pages with nonzero output reference count serve as system pages during request processing; the result is the same as if data had been copied because any attempt to overwrite such pages causes a VM fault. In systems that have Mach's VM organization [67], such as NetBSD, the VM fault handler is modified as follows, for the case of write faults in regions for which the faulted party has write permissions:

- If the page is found in the top memory object, which is directly referenced by the region (i.e., the region itself is not COW, and the page is in physical memory):
  - If the output count of the page is nonzero, the system recovers from the fault by invalidating all mappings of the page, copying the contents of the page to a new page, swapping pages in the memory object, and mapping the new page to the same virtual address, in the party's address space, with writing enabled.
  - Otherwise, the system recovers by simply reenabling writing on the page (no copying). A zero output count implies that all output requests that referenced the page have already completed.
- Otherwise, the fault is a conventional COW fault and is handled normally.

In systems with VM organization different from that of Mach, similar operations would achieve the same purpose.

Whether a system page is especially allocated or is actually an in-place client page, it can be used as an ephemeral server buffer by kernel-level servers. Chapter 5 describes techniques that make that possible also for user-level servers.

At reply time, for each page in the client output buffer, if the data length is less than t_o, TCOW deallocates the respective system page; otherwise, TCOW unreferences the client page. For data longer than t_o, TCOW therefore adds to page referencing only the cost of removing page write permissions, which is arguably the minimum necessary overhead for strong-integrity, safe in-place output.

In cases of asynchronous requests or servers that return anticipated replies, TCOW imposes a temporal, soft optimization condition: It is more efficient not to overwrite a client output buffer until request processing completion. The next section analyzes the impact of this condition on TCOW's criticality and restrictiveness.

4.3 Output buffer reuse

The criticality and restrictiveness of TCOW depend on whether and how clients overwrite their output buffers during output request processing, as well as on the setting of t_o.

Examination of existing Unix applications reveals that often output buffers are overwritten not by the client itself, but by a server processing an input request on behalf of the client. For example, many applications iteratively input data, perhaps process the data, and then output data always using the same circular buffer.

A simple analysis shows that TCOW and input alignment interact synergistically to eliminate copying in such cases. For the part of a client buffer that is page-aligned and has length multiple of the page size, input alignment and page swapping will cause pages with outstanding output to be simply swapped out of the client buffer, with deallocation deferred until completion of the output request. No copying at all occurs for data output or input.

Analysis for pages only partially occupied by the client buffer is more complicated but provides valuable insight. If l is the data length, p is the page size, the cost of copying is cl, and the cost of swapping pages is s, then

the cost of output with TCOW is: for $l < t_o$, cl, and for $l >= t_o$, ct_o. The cost of input with input alignment is: for $l < t_i$, cl, and for $l >= t_i$, $c(p-l)+s$. Therefore, the combined cost of output and input with TCOW and input alignment is:

1. $l < t_o$ and $l < t_i$: $2cl$
2. $l < t_o$ and $l \geq t_i$: $cp + s$
3. $l \geq t_o$ and $l < t_i$: $c(t_o + p + l) + s$
4. $l \geq t_o$ and $l \geq t_i$: $c(t_o + p - l) + s$

Substituting above the definition of t_i, $t_i = \frac{\frac{s}{c}+p}{2}$, and considering that the cost of input and output by copying is $2cl$, it can be verified that TCOW and input alignment, compared with copying, break even in the first condition, give lower cost in the second and fourth conditions, but give higher cost in the third condition ($t_o \leq l < t_i$: output makes page read-only and input by copyout causes a write fault). The third condition can easily be avoided by requiring that t_o be tuned with $t_o \geq t_i$.

Alternatively, it is also clear from the above equations that when output is performed with copy avoidance (third and fourth conditions), the total cost is always less when input is effectively by reverse copyout and page swapping (fourth condition). Therefore, input alignment should be refined as follows: If the client page has an outstanding output reference, then input should be by reverse copyout and page swapping and never by copyout, regardless of the data length. With this modified input alignment rule, TCOW and input alignment will give higher cost than that of copying only for $t_o \leq l < \frac{t_o+2t_i}{3}$.

Therefore, in the common cases where clients do not overwrite output buffers during output or do so by reusing them as input buffers, TCOW has low criticality.

In the remaining case, clients themselves (and not input servers on their behalf) overwrite buffers with outstanding output. In such case, compared to copying, TCOW with $t_o = p$ gives output data passing costs that are the same for pages only partially occupied by client buffers, and that are greater by s for fully occupied pages. If $s \ll cp$, as is the case, for example, of all computers used in the experiments reported in Chapter 8, then TCOW has low criticality even in this case.

If copying is expensive, however, it may be desirable to more aggressively optimize, setting $t_o < p$. Two alternative additional conditions can make

TCOW's temporal condition still advisory even with such tuning, but at the cost of making TCOW more restrictive. The first condition is to require that a client, before overwriting an output buffer, make a synchronous *flush* request to the server, so as to ensure that processing of the previous output request actually completed. The second, alternative condition is to have clients use a circular buffer, overwriting and synchronously outputting, successively at each time, only a fraction of size f of the buffer. The client sets its limit on amount of physical memory in pending I/O requests (as explained in Section 3.1.2) to a value less than the total size of the circular buffers by at least f. In that case, the fraction that is being overwritten at any given time is sure not to have pending output — the client would block on an output before it would have the opportunity to overwrite parts of the buffer with pending output.

4.4 Related work

TCOW and sleep-on-write [6] are both page-level techniques and perform very similarly for clients that do not overwrite their output buffers during processing of their I/O requests. Both schemes add to the cost of removing write permissions only that of the same number of updates to fields (output reference count or busy bit, respectively) of the page data structure. TCOW offers the added benefits of supporting multiple concurrent output references to a given page and not stalling user-level clients that do overwrite output buffers during output.

Another page-level COW scheme is that of Solaris, but it differs in important aspects from TCOW. According to the description in [23], the Solaris scheme eagerly reinstates write permissions on completion of the I/O request. Eager reinstatement can be incorrect if the status of the region containing the page changes after the I/O request. The client may, for example, change region protection to read-only, deallocate the region, or fork another process that inherits the region by copy-on-write. Additionally, if more than one I/O request references the same page, requests other than the first one either require data copying or may output corrupted data after the first request completes, as the client may then overwrite the page. Finally, [23] does not describe whether or how the Solaris scheme prevents the page from being reallocated during output if the client explicitly deallocates the page or terminates. Apparently, the Solaris scheme does not have an I/O-

deferred page deallocation scheme that would allow its page swapping and COW mechanisms to combine synergistically, as they do in emulated copy, without program modification: To avoid COW faults, test programs in [23] had to be modified to use a circular buffer and limit socket window sizes.

4.5 Summary

Emulated copy passes data between client and ephemeral server buffers with copy avoidance, preserving, however, copy semantics for client buffers.

Emulated copy uses input alignment and page swapping to pass data into client input buffers. Client buffers do not have to be page-aligned or of length multiple of the page size for page swapping. A new optimization, reverse copyout, guarantees that no more than about half a page at each end of the client buffer needs to be copied; the rest of the data can be passed by swapping. Input alignment can be achieved by either client- or server-aligned buffering: Either the client or the server aligns its buffers with respect to those of the other party. Input alignment imposes only soft, advisory optimization conditions.

Emulated copy uses TCOW to pass data from client output buffers in-place. TCOW keeps references to client pages during request processing, so as to prevent them from being deallocated before request processing completion. TCOW also removes write permissions from client output pages, so that attempts to overwrite such pages cause VM faults. On a fault, TCOW copies and replaces the faulted page only if there still are output references to the page; otherwise, TCOW simply reenables writing on the faulted page, and no copying occurs.

The performance of TCOW depends on whether and how clients overwrite output buffers before request processing completion. If an output buffer is overwritten by being reused as an input buffer, no copying happens at all: input alignment causes client pages to be simply swapped out of the buffer, with deallocation deferred until completion of the last respective output request. With an appropriate configuration, TCOW has only soft, advisory optimization conditions even in the worst case, where the client itself overwrites output buffers with pending output.

The optimization conditions of emulated copy are special only relative to copy semantics. They are less restrictive and/or critical than conditions that are intrinsic to non-copy semantics. TCOW's temporal condition, for

example, is as restrictive as and less critical than that of output with share semantics, where overwriting during output request processing causes errors. TCOW's temporal condition is also both less restrictive and less critical than that of migrant semantics, where overwriting at any time after the output request causes an error. Likewise, the temporal condition of server-aligned buffering is as restrictive as and less critical than that of input with share semantics, where, if the request does not happed before input, no input can occur unless converted to copy semantics. The spatial condition of client-aligned buffering is as restrictive as and less critical than that of input with migrant semantics, where, if the client must specify alignment and length of input buffers, no input can occur unless copying is also performed.

Chapter 5

I/O-oriented IPC

The previous two chapters introduced new optimizations for copy avoidance with each data passing semantics, in the case of kernel-level servers and ephemeral server buffers. However, in systems with microkernel structure, servers execute at user level, not at kernel level. User-level servers are usually much easier to debug and maintain than kernel-level ones. This chapter describes *I/O-oriented IPC*, a new copy avoidance scheme that extends the optimizations of the previous two chapters to also support user-level servers.

I/O-oriented IPC offers different explicit I/O interfaces for, respectively, user-level clients and servers. The client interface supports all four data passing semantics and their optimizations discussed in the previous two chapters. In particular, the client interface supports emulated copy, allowing transparent emulation of existing APIs, such as Unix's *sockets*, without copying.

The server interface, on the contrary, offers data passing semantics similar to that of the *kernel*-level socket interface. As discussed in Section 2.3, data passing between user-level clients and kernel-level servers is usually asymmetric. To allow easy server migration between kernel and user level, it is important to preserve compatibility with existing kernel-level interfaces, not user-level ones.

I/O-oriented IPC uses *selective transient mapping* to make system buffers accessible by user-level servers. Relative to servers, such data passing has move semantics. Before mapping, to preserve protection, I/O-oriented IPC zero-completes system buffers not filled with client data (e.g., on input with the emulated copy scheme). To avoid zero-complete overheads, I/O-oriented IPC allows *user-to-user input alignment* and page swapping directly between address spaces of client and server. User-level servers can pass fragments of

a request's system buffers to other user- or kernel-level servers, using *fragment subcontracting*. Relative to contractors, such data passing has share semantics.

The greatest novelty in I/O-oriented IPC is to show that the asymmetric semantic requirements of clients and servers can be exploited to pass input data by page swapping. This allows bi-directional copy avoidance even though the client interface has copy semantics and preserves compatibility with existing applications. Experiments in Chapter 9 demonstrate that I/O-oriented IPC gives user-level servers performance approaching that of kernel-level ones.

5.1 Selective transient mapping

When delivering a request to a user-level server, the IPC facility maps the request's system buffers to the server, and, when the server notifies request processing completion, the IPC facility unmaps those buffers from the server. Consequently, relative to the server, such data passing has move semantics. Mapping is *transient*: a user-level server can access a request's system buffers only while processing the request.

Mapping is also *selective*. When a user installs a user-level server, the user also specifies *read*, *write*, and *physical* mapping flags for each service provided by the server. The IPC facility uses these flags as follows, when delivering to a user-level server a request for a given service:

- Services that do not require access to I/O data and that are implemented by subcontracting may have no mapping flags set. In such cases, the IPC facility passes only the length of each buffer to the server.

- If the physical flag is set, the IPC facility passes the physical addresses of each buffer to the server. This option may be sufficient for user-level drivers of devices that transfer data by DMA, in cases where the driver itself does not need to access I/O data.

- If the read or write flag is set, the IPC facility maps the buffers and passes their virtual addresses to the server. Output buffers are mapped read-only. Input buffers are mapped with access permissions according to the read and write flags.

The read flag is necessary for any service that requires direct access to I/O data. The write flag is usually required only in input services of drivers of programmed-I/O devices (since DMA devices typically can write into memory regardless of access permissions). Other servers normally modify only headers or trailers. Headers and trailers can be freshly allocated and prepended or appended to fragments of buffers received in requests, without modifying the latter, as explained in Section 5.3.

The IPC facility maps system buffers to *transient mapping regions* which, to minimize virtual address calculations, start at the same virtual address and have the same length in all user-level servers. The length is equal to the global limit on the amount of physical memory in pending I/O requests (Section 3.1.2).

The IPC facility maintains a stack of *transient virtual addresses* (TVAs), each corresponding to an unassigned page-aligned address in the transient mapping region. When first mapping a system buffer page to a user-level server, the IPC facility pops a TVA and assigns it to the page. In subsequent subcontracts, all transient mappings of the page use the same virtual address. The IPC facility pushes the TVA back when unmapping the page from the initial user-level server.

Because system buffers are unpageable, the IPC facility can reduce mapping and unmapping overheads by updating only the server's page table, as opposed to also updating a machine-independent representation of the server's address space. The system's VM fault handler is modified to treat faults in the transient mapping region as unrecoverable.

To preserve protection, before mapping to a user-level server a system page that is not an in-place client page, the IPC facility completes with zero those parts of the page not yet filled with zero or with data of the request's client. System pages that are not in-place client pages occur when data passing between client and system buffers is by copying (output or input) or with the emulated copy or move schemes (input only). In general, such system pages may be allocated from the VM pool of free pages and contain data of clients who do not trust the particular server to which the page needs to be mapped.

Because zero completion can be expensive, the IPC facility may pass data to and from user-level servers by copying, instead of mapping and unmapping. Let t_m be a configurable threshold for selective transient mapping. The IPC facility maintains separate pools of free pages mapped to each user-level server. The IPC facility may, when delivering a request to a server s, instead

of zero-completing and mapping a system page p to s, assign to p a page p' allocated from the pool of s. Such cases occur when the data length in p is less than t_m; the IPC facility then passes to s the virtual address of p', rather than the mapped address of p. If the requested service has *read* flag set, the IPC facility copies data from p to p' when delivering the request. If the requested service has *write* flag set, the IPC facility copies data from p' to p when s notifies request processing completion. The IPC facility also deallocates p' at request processing completion.

5.2 User-to-user copying and input alignment

To avoid both zero-completion and copying costs, the IPC facility allows clients to specify, for each client buffer that is not in-place, the *lender* of the corresponding system buffer. If a lender is not specified, the IPC facility assumes the lender to be the server of the client's request. The IPC facility allocates system buffers, if the lender is a user-level server, from the lender's pool[1]; otherwise, from the VM pool of free pages. Data passing has weak move semantics relative to the lender. The IPC facility does not zero-complete, map, or copy to or from a server a page lent by that server.

In requests to a user-level server that subcontracts a kernel-level driver, the default specification of the user-level server as the lender avoids all zero-complete costs. In requests to a user-level server that subcontracts a user-level driver, zero-complete costs can be minimized by specifying as the lender the driver (input or output) or the server (output only).

If the lender is a user-level server, the IPC facility copies data or swaps pages directly between the address spaces of client and lender. Before page swapping, reverse copyout guarantees that server pages are filled with client data, and therefore no zero completion is necessary to preserve protection.

In the terminology of Section 2.1.3, indication of a lender is a soft condition: The default lender is often one that reduces data passing costs. With a properly tuned t_m, relative to copying, lender specification is also advisory only.

[1]Except in input with the move scheme, where system buffers become net memory gains for the client, and therefore allocation must be from the VM pool of free pages. Allocation from the lender's pool would cause a net loss for the lender.

5.3 Fragment subcontracting

In a request, neither the client nor the server interface allows a party to specify by virtual address the location of a buffer in the transient mapping region. Such location can be specified only by reference to the request that passed the buffer to the party. Let r be a request that passes a buffer b to a server s. In a subcontract request r', s specifies the location of a fragment b' of b by a triple containing the identifier of r, the index of b within r, and the offset of b' from the start of b. The identifier of r is an integer passed by the IPC facility to s when delivering the request. s uses this same identifier to notify request processing completion. To preserve protection, the interface checks that the caller (s) is indeed the server of r. Indication of location by such triple offers the following advantages:

1. It indicates that b' already has corresponding system buffers and that no further data passing may be required to pass data between b' and system buffers. Relative to s, b' is passed with share semantics.

2. It allows also services whose read and write flags are not set, and that therefore are not passed mapped system buffers, to be subcontracted to other servers or drivers.

3. It provides the information necessary for updating the subcontract graph of r, linking r and r'. If r needs to be evicted (Section 3.1.2), this link indicates that r' should also be evicted, so as to eliminate the references of r' to the system buffers of r.

s may include in r', in addition to b', other buffer elements, possibly not located in the transient mapping region. If s is a TCP/IP server, for example, it may prepend to b' a buffer element h native to s, to hold the packet header. The data in h can be passed to and from system buffers using, for example, the emulated copy scheme.

5.4 Related work

Most previous IPC facilities with copy avoidance provide interfaces with non-copy semantics (e.g., Tzou and Anderson's DASH [76], fbufs [29], container shipping [64]) and therefore are incompatible with the many applications written according to that semantics.

I/O-oriented IPC is possibly the first IPC facility to offer a client interface with copy semantics (using emulated copy) and pass data with copy avoidance both on output and on input. Previous IPC facilities that have an interface with copy semantics typically pass data by copying (e.g., Unix-domain sockets [49]) or by COW (e.g., Mach out-of-line IPC [37]). Copying provides poor performance. In general, COW can avoid copying, while preserving copy semantics, only on output. The main novelty in I/O-oriented IPC is that it passes input data by page swapping. There does not appear to be a precedent to the latter in IPC facilities. Local Peregrine RPC [43], for example, comes very close to the I/O-oriented IPC solution, passing data out of client buffers by COW and passing data into and out of server buffers by mapping and unmapping. However, unlike I/O-oriented IPC, Peregrine inputs data into client buffers by copying. This dissertation's careful analysis of the semantics of data passing between user-level clients and kernel-level servers provides the missing conceptual link for the use of page swapping also in IPC. The modification of the data in the server buffer, which occurs as a side effect of page swapping, has no consequence in I/O-oriented IPC, because the server buffer is migrant: The buffer is modified when it is also being implicitly deallocated, at reply time. Page swapping allows a client interface that has both copy semantics and bi-directional copy avoidance.

Fbufs [29] are passed to or from servers by mapping and unmapping, like system buffers in I/O-oriented IPC. However, the fbuf facility updates also a machine-independent description of the server's address space, incurring greater overhead than that of selective transient mapping. On the other hand, cached fbufs and volatile cached fbufs are passed to or from servers without mapping and unmapping, but require client buffers to be located in a special *fbuf region*, permanently mapped to the server. That is a hard optimization condition, which, as explained in Section 2.1.3, causes incompatibility with previously written applications.

Fbufs are pageable, unlike the buffers used in I/O-oriented IPC. Although [29] does not make this point clear, the safety of fbufs against attacks with untrusted client-provided pagers (Section 2.2) probably hinges on the hard condition that client buffers be in the fbuf region: The fbuf facility can impose a trusted pager for that region. The Mach IPC facility [37] also uses pageable buffers but, contrary to fbufs, allows client buffers to be arbitrarily located. However, Mach IPC with COW is unsafe relative to untrusted pager attacks. Indeed, it appears that all IPC facilities that use pageable buffers have been either unsafe, like Mach IPC with COW, or have had hard

optimization conditions and attendant incompatibility, like fbufs, or have required copying. I/O-oriented IPC uses unpageable buffers (with request eviction, as explained in Section 3.1.2) so that it can be safe, can avoid hard optimization conditions, and can avoid copying while maintaining compatibility with applications that use copy semantics.

Container shipping [64] allows something similar to a service with no mapping flag set, that is, a subcontracted service whose contractor passes to a subcontractor a buffer that the contractor itself cannot access. Like the I/O-oriented IPC facility, fbufs map most buffers read-only (in fbufs, except in the case of *producers* – driver on input, client on output). DASH [76] and fbufs also use a region that starts at the same virtual address in every address space, so as to reduce virtual memory calculations.

IPC facilities that pass data by copying usually copy the data twice, once between each party's and system buffers. User-to-user copying reduces the number of copies to one. It does so by unmapped access (Section 3.1.1) from the client address space to the unpageable, ephemeral server buffers. LRPC [7] and URPC [8] also reduce the number of copies to one, but require a statically shared region for each pair of client and server. L4 [50, 51] reduces the number of copies to one using a *communication window* that temporarily maps one party's region to the other party's address space. Relative to data passing[2] with user-to-user copying, that with LRPC or URPC has greater space overhead, while that with L4 has greater time overhead. Neither LRPC, nor URPC, nor L4 have anything analogous to user-to-user input alignment.

5.5 Summary

An asymmetric IPC facility can provide bi-directional copy avoidance while offering to clients an interface with copy semantics and to servers an interface with semantics similar to that of kernel-level interfaces. Such asymmetry extends to IPC (and microkernel systems) data passing optimizations previously available only in system calls (and monolithic systems). In particular, judicious exploitation of semantic asymmetry allows input data passing by page swapping. Page swapping provides the missing link for copy avoidance in IPC facilities that have client interface with copy semantics.

[2]Note that the comparison is not about context switch time, which has not been optimized in the implementation of I/O-oriented IPC described here.

Chapter 6

Network Adapter Support for Copy Avoidance

The ability to gather data, now common in most network adapters, is sufficient to support in-place output with copy or share semantics, without copying. However, the converse is not true: The ability to scatter data, now also common, does not by itself support, in the general case, input with copy avoidance and copy or share semantics. In that general case, client input buffers can have any alignment and length. For copy avoidance to be possible, the network adapter must be able to customize its input buffers for each input request. In the case of copy semantics, for server-aligned buffering, adapter buffers have to start at the same page offset and have the same length as the corresponding client buffers. In the case of share semantics, for in-place input, the adapter must be able to use, as adapter buffers, promoted client buffers. However, packets may arrive in an order different from that in which clients make their requests. If the adapter has a single scatter list, packets and buffers may be mismatched.

Moreover, adapters receive data fragmented into packets. Fragmentation may make it hard to correctly and directly input client data into server-aligned or in-place buffers because data is in general received in packets of arbitrary length, each containing header and trailer that contain extraneous data and that therefore need to be stripped.

Lack of checksumming support may also significantly reduce the performance advantage of copy avoidance. If data passing is by copying, checksumming can be integrated with it at little extra cost [24]. On the contrary, if data passing is performed with copy avoidance, checksumming cannot be

integrated. If the adapter does not checksum I/O data, the host processor has to read all the data to checksum it, which can be expensive if memory bandwidth is low.

This chapter describes different levels of network adapter support for copy avoidance, and highlights, for each case, compensating software techniques. In particular, two new optimizations are contributed: *header patching*, a software technique that allows stripping headers of arbitrary length even with only the lowest level of adapter support, and *buffer snap-off*, a new adapter feature that enables copy avoidance under very general conditions (including headers of arbitrary length).

6.1 Pooled in-host buffering

Network adapters with *pooled* in-host buffering receive packets into buffers allocated from a single scatter list shared among multiple *reception ports*. The elements of the scatter list are resident in host memory and are called *buffer segments*. Each segment is specified by a (pointer, length) pair. The host enqueues segments at the tail of the scatter list when preparing for reception, and the adapter dequeues segments from the head of the scatter list when receiving packets. Reception ports are the objects from which clients receive data; they are identified in packet headers and can be, for example, ATM virtual connections or TCP ports.

In hosts with physically addressed DMA, the length of each segment is such that the segment does not cross page boundaries unless pages are physically contiguous; commonly, each segment is a page. In hosts with virtually addressed DMA, the length of each segment is normally equal to the network maximum transmission unit (MTU); commonly, all segments are page-aligned. In either case, at the end of a packet the adapter drops the remainder of the current segment, so that storage of the next packet starts at the next segment.

Because it uses a single, shared scatter list, pooled in-host buffering cannot support server-aligned or in-place buffering. Input alignment is possible, however, if clients use client-aligned buffering. Specifically, if:

1. Packet headers have fixed lengths;

2. Data lengths of all packets in a data transfer have a known length L ($\leq$ MTU minus header and trailer lengths), except possibly for the last

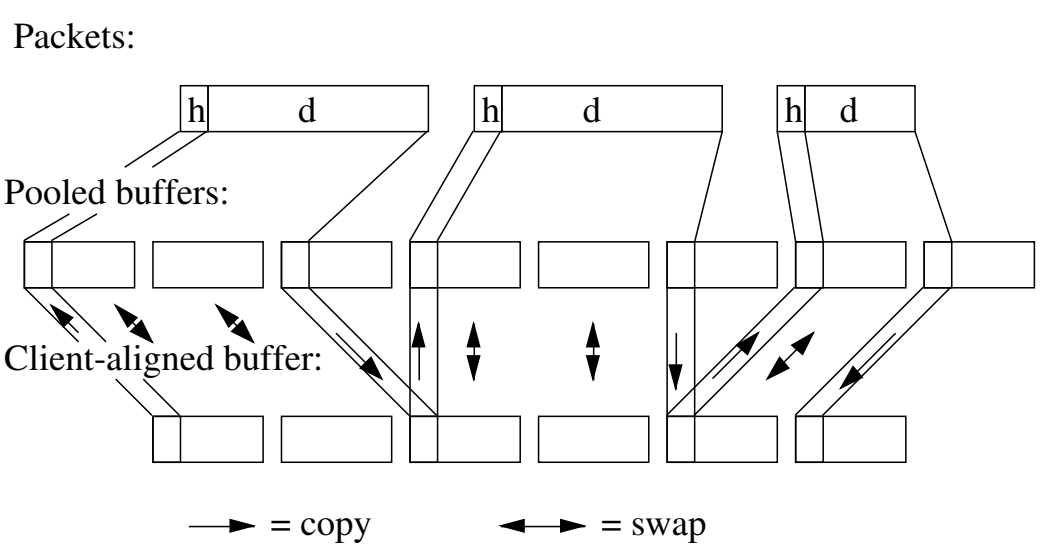


Figure 6.1: Copy avoidance with pooled in-host buffering and client-aligned buffering.

packet, which may have data length of at most L; and

3. Either:

   (a) For each data transfer, the length of client input buffering matches that of the data transfer; or

   (b) All data transfers and client input buffers have lengths that are integral multiples of L, although not necessarily equal;

then copying can be avoided by setting the preferred alignment equal to the header length and the preferred length equal to L.

Figure 6.1 shows how data would be passed to a client-aligned buffer in a data transfer of length five times the page size, when L is twice the page size. The data of the first packet, by itself, would occupy two pages. However, because of the header, the packet is received into three pooled pages. The data in the first pooled page is offset from the start of the page by the header length. Given that the client buffer also starts with the same offset, the data can be passed by reverse copyout, from the start of the client page to the area containing the header in the pooled page, and then swapped. The second page is simply swapped. The third page contains only data of length equal to

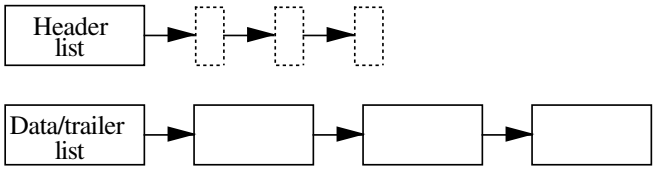


Figure 6.2: Separate header and data/trailer scatter lists.

the header, and therefore is simply copied out. Data passing for the following packets is analogous. Because L is an integral multiple of the page size, the data ends up virtually contiguous in the client's address space.

With pooled in-host buffering, copy avoidance can also be achieved by migrant-mode data passing: The I/O interface can simply remove a filled adapter buffer from the adapter's buffer pool, insert the buffer without alignment constraints in the client's address space, pass the resulting buffer location and layout to the client, and replenish the adapter's buffer pool with the same number of newly allocated pages.

6.2 Header/data splitting

A refinement of pooled in-host buffering is to have the adapter use separate *header* and *data/trailer* scatter lists, shared among multiple reception ports, as depicted in Figure 6.2. The initial portion of a packet is stored in a segment allocated from the header list, while the remainder of the packet is stored in segments allocated from the data/trailer list [47]. If segments in the header list have length equal to that of packet headers, this scheme normalizes the preferred buffer alignment to page boundaries.

Solaris zero-copy TCP [23] uses this technique with L set to the largest integral multiple of the page size not greater than the MTU minus TCP/IP header length. Input copying is thereby avoided if packet headers do not contain options (so that TCP/IP headers have fixed length), client input buffers are page-aligned, and both client input buffers and data transfers have lengths that are integral multiples of the page size.

Figure 6.3 shows how data would be passed to a page-aligned client buffer in a data transfer of length five times the page size, when L is twice the page size. Because the header is received into a segment from the header list, the data of the first packet is received into exactly two pages from the

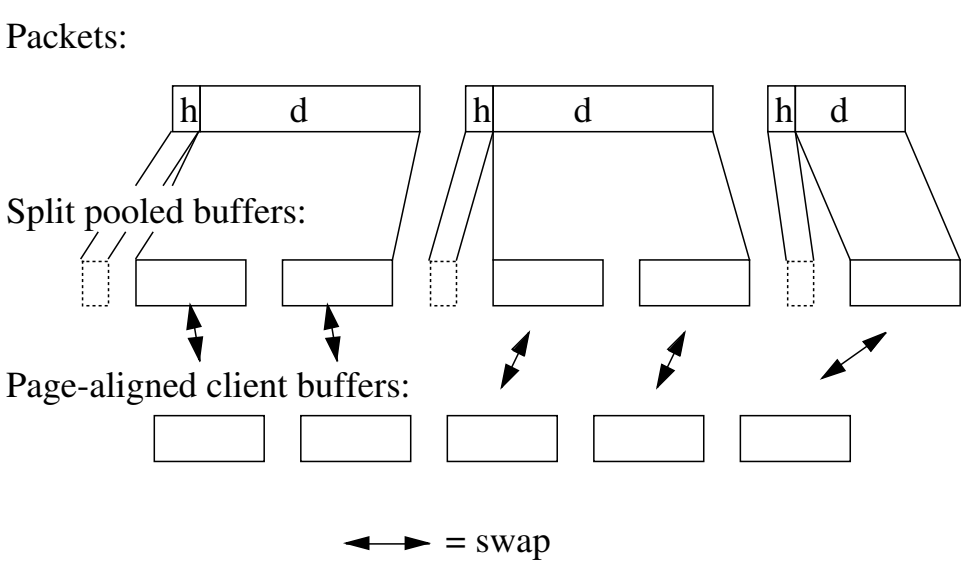


Figure 6.3: Copy avoidance with header/data splitting and page-aligned client buffering.

data/trailer list. Given that the client buffer is page-aligned, all data can be passed simply by swapping, with no copying. Data passing for the following packets is analogous. Because L is multiple of the page size, the data ends up virtually contiguous in the client's address space.

6.3 Header patching

An adapter cannot by itself provide page-aligned input data if it supports only pooled in-host buffering. Even if the adapter supports header/data splitting, it may still not be able to provide page-aligned input data if headers have variable length or data is preceded by *application*-level headers not stripped by the adapter. However, many clients do require data to be received in page-aligned buffers. For example, VM-based distributed shared memory systems [1] transfer page-sized data blocks between nodes connected by a network. For efficient data passing to a client, by mapping instead of copying, data blocks have to be received in page-aligned buffers.

Header patching is a new software technique that allows stripping headers of arbitrary length with minimal copying, resulting in page-aligned data even if the adapter has only pooled in-host input buffering. Header patching does

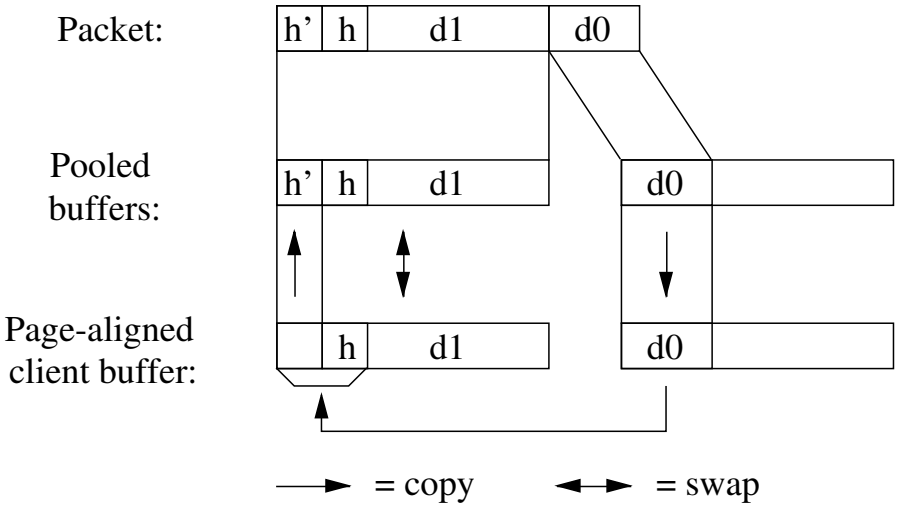


Figure 6.4: Header patching allows stripping headers of arbitrary length even if the adapter supports only pooled in-host input buffering.

require, however, end-to-end agreement on its use.

Let h' be the preferred alignment for input from the network (usually equal to the length of any unstripped protocol headers below the application level), h be the length of the application-level header, and l be the data length (less than or equal to the preferred length minus h). h' must be fixed and known by both sender and receiver. On the contrary, h and l can vary from packet to packet. Header patching requires transmission of the application-level header followed by the data d_1 that corresponds to offset $h'+h$ from the beginning of the application data and has length $l - h' - h$, followed by the data d_0 that corresponds to the beginning of the application data and has length $h' + h$ (to achieve this out-of-order transmission, the sender may use, e.g., Unix's `writev` call with a gather list). After decoding the application-level header (if any), the receiving client copies the data at offset l from the beginning of the input buffer and of length $h' + h$ to the beginning of the input buffer, thereby patching data d_0 over application- and lower-level headers. After patching, the input buffer starts at a page boundary with the beginning of the application data and runs uninterrupted for length l with the remainder of the application data in correct order.

Figure 6.4 shows how data of length equal to the page size p would be received into a page-aligned buffer. The client requests input of the first

$p - h'$ bytes starting at an offset h' (the preferred alignment) from the final page-aligned destination of the data, followed by $h' + h$ bytes to an unrelated client input buffer (the client may use, e.g., Unix's `readv` call with a scatter list). The first $p - h'$ bytes are passed to the client by reverse copyout and page swapping. The following $h' + h$ bytes are simply copied out. The client decodes the application-level header at offset h' from the final destination of the data, and then copies the initial data d_0, which has length $h' + h$, onto the final destination of the data. Receiving d_0 in a separate, unrelated client input buffer preserves the client data that may be adjacent to the page being received (otherwise, such data would be "spilled over"). Alternatively, the client may receive d_0 directly to its final destination if: (1) $h = 0$ (no application-level header), or (2) the client *peeks* at the application-level header (using, e.g., Unix's `recv` with `MSG_PEEK` flag) and decodes it before inputting the packet.

6.4 Early demultiplexing

Network adapters with *early demultiplexed* input buffering maintain a separate scatter list for each reception port. When receiving the beginning of a new packet, the adapter demultiplexes the packet header to determine which scatter list to use for reception of the packet [4], as shown in Figure 6.5[1]. If the reception port's scatter list is empty (e.g., because the client did not request input before packet arrival), the adapter uses instead a pooled scatter list (i.e., early demultiplexing degenerates into pooled in-host buffering).

Early demultiplexing supports server-aligned buffering: The I/O interface can enqueue, in the scatter list of each reception port, segments with alignment and length matching those of the respective client buffers. Each segment may have different alignment or length, and consequently, it must be possible for the host to *reclaim* (dequeue) specific buffer segments in case of exceptions such as timeout of the corresponding I/O request. Early demultiplexing analogously supports in-place buffering. Copy avoidance with client-aligned buffering or migrant-mode data passing are also possible.

If packet header, data, and trailer lengths are predictable by the receiving

[1]ATM adapters that reassemble packets from ATM cells must implement at least a primitive form of early demultiplexing, keeping track of the different segments being reassembled per reception port, but possibly allocating segments from a pooled (rather than per-port) list.

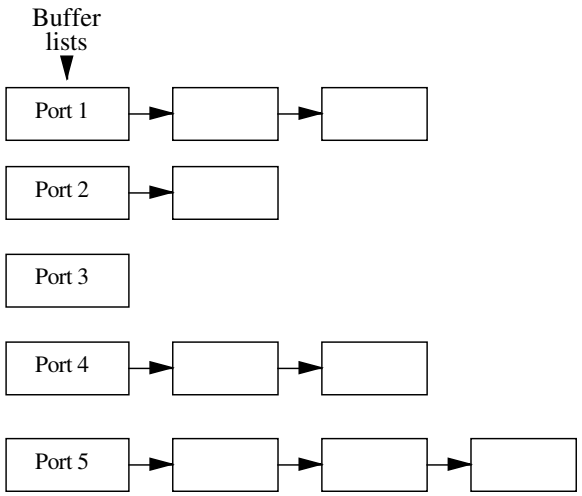


Figure 6.5: Early demultiplexing.

system (not necessarily fixed), proper alignment of input data can be maintained by interspersing header and trailer segments at the appropriate points in the scatter list of the respective reception port. Header and trailer segments allow the remaining data segments to end up concatenated in virtual memory in spite of data packetization.

The length of data transfers may be known by previous arrangement between the end clients. Packet header, data, and trailer lengths are then predictable if the header and trailer lengths are fixed or easily computable and the protocol used follows some simple deterministic rule, such as sending all packets except possibly the last one with length equal to the MTU. Knowledge of the total data transfer length is necessary to avoid enqueueing more buffer segments than necessary to hold the data. The MTU can be estimated by *MTU tracking*, that is, monitoring the maximum length of packets received at the given reception port, or may be determined by path MTU discovery [55]. The MTU may be also known because it is fixed (for example, in a LAN serving a cluster of workstations) or configurable (for example, using the TCP MSS option [66]).

Figure 6.6 shows how data would be passed to a client buffer of arbitrary page offset using server-aligned buffering, when the client requests input before it physically happens, in a data transfer of length five times the page size, with MTU equal to 9180 bytes. Headers and data are scattered to different segments. The data of the first packet is received into three pages,

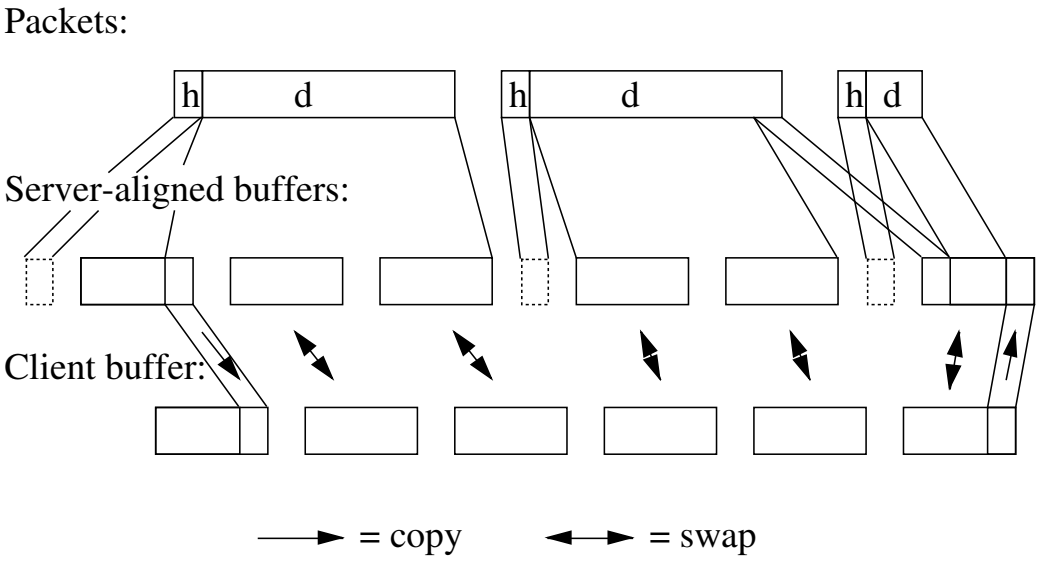


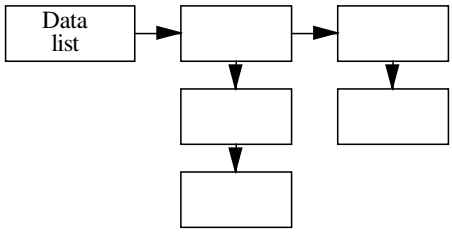
Figure 6.6: Copy avoidance with server-aligned buffering.

beginning at a page offset that is the same as that of the start of the client buffer. Likewise, headers of the second and third packets are scattered away from the pages where the data is received, so that the data of both packets is concatenated. In this example, the client buffer starts close to the end of a page, and therefore the data of the first page is simply copied out. Other pages, except the last, are simply swapped. The last page is completed, by reverse copyout, with data from the last page of the client buffer, and then swapped.

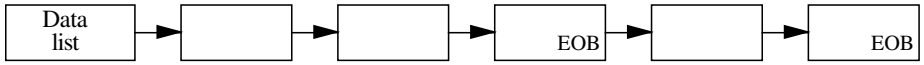
6.5 Buffer snap-off

Early demultiplexing does not by itself support buffer boundaries. Consequently, it enables server-aligned and in-place buffering only if the length of each data transfer over the network matches that of the corresponding input request buffering. A novel adapter feature, *buffer snap-off*, can be used to remove this restriction.

An adapter with buffer snap-off executes input requests that include specifications of reception port (p), length of packet headers (h) and trailers (t), and data buffering. If p corresponds to an adapter-level connection (e.g.,



(a) Hierarchical list



(b) Flat list

Figure 6.7: Data buffer representation alternatives.

ATM virtual connection), it can be specified by a constant. Likewise, if h and t are fixed, they can be specified by constants. In general, however, p, h, and t have to be specified by a program that processes packet headers and computes, in addition to p (as in early demultiplexing), also h and t. Such programs can be written in a non-Turing-complete language similar to that of a packet filter [54, 4] and be executed by adapter hardware when packets are received (as in [4]). After processing an input request, the adapter signals completion to the host and indicates the length of the data actually received and a list of headers and trailers received.

The adapter maintains separate header, data, and trailer scatter lists. The first h bytes of each packet are stored in a segment allocated from the header list, the last t bytes are stored in another segment from the trailer list, and the remaining data is stored in segments allocated from the data scatter list (*header/data/trailer splitting*). The header and trailer lists can be shared among multiple reception ports. If h is fixed, the segments of the header list have length h, and likewise for t and the trailer list. If h and t have variable length, the header and trailer lists can be merged and have segments with length greater than or equal to the largest allowed h and t (in which case header and trailer segments may be incompletely filled).

The adapter maintains separate data scatter lists per reception port. Each input request's data buffering is appended to the data scatter list of the re-

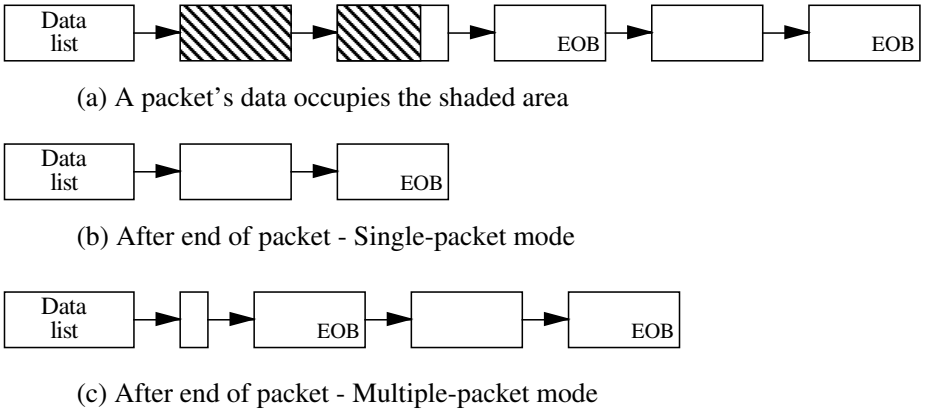


Figure 6.8: Buffer snap-off. Data from an incoming packet occupies the shaded area in (a). The rest of the buffer is then dequeued (b) or snapped off (c).

spective reception port. If, when a packet is received, the respective reception port's data scatter list is empty (e.g., because the client did not request input before packet arrival), the adapter allocates data segments from a pooled data scatter list (i.e., buffer snap-off degenerates into a more sophisticated form of header/data splitting). Each port's data scatter list can be represented by a hierarchical list of buffer segments, or more simply by a list of buffer segments, where each segment contains an end-of-buffering (EOB) bit set only at the last segment of the buffering of an input request. These alternatives are shown in Figure 6.7.

There are two snap-off modes, selectable per reception port: *single-packet* and *multiple-packet*. Single-packet mode is suitable for datagram service, whereas multiple-packet is especially suitable for byte-stream service, as shown in Figure 6.8.

In single-packet mode, on end-of-packet, the adapter should dequeue and deallocate the remaining segments of the current data buffering and signal completion of the input request. Conversely, if EOB is reached before end-of-packet, the adapter should allocate additional segments from the pooled data scatter list and, on end-of-packet, signal completion of the input request. The additional segments are in general necessary to check CRC and/or checksum. Many interfaces (e.g., Unix's sockets [49]) would, however, truncate the buffering, dropping the additional segments and passing to the client only the data that fit in the input request buffering.

On the contrary, in multiple-packet mode, on end-of-packet, the adapter should *snap off* the remainder of the current data buffer segment and place it at the head of the data scatter list of the current reception port. This way, the data of successive packets for the same reception port will be concatenated in the adapter input buffer, as is required for byte streams, even if packet data lengths are not integral multiples of the page size (as, for example, in maximally-sized Ethernet, FDDI, or ATM AAL5 packets). In this mode, each input request remains pending until the respective buffering is completely filled. Packets may span the buffering of multiple input requests; end of the buffering of an input request (EOB) simply marks the completion of the request, deferred until successful end of the last packet using it.

Consequently, in multiple-packet mode, on end-of-packet, the adapter should signal completion of all requests whose buffering was completely used, and partial completion of the request whose buffering was snapped off (if any). The case of partial completion of the last outstanding request r in an adapter's reception port can be handled as follows, if the buffering of r is server-aligned (i.e., emulated copy input). Let r' be a request that directly or indirectly originated r, r'' be the request that originated r', s be the server of r'' (e.g., Unix's socket layer), and c be the client of r'' (e.g., a user-level application). Partial completion with nonzero data length is an interim reply that propagates from each subcontractor to its contractor, and that therefore reaches s. s may then reply successful completion of r'' with the data length already received. In so doing, s converts r' into an original (and speculative) request for the remainder of the data[2]. If c then requests input of the remainder of the data (common case), s does not make another subcontract request (presumably, such input may already be occurring). However, if c requests input to buffering of different alignment or length, s *amends* its request r'. The amendment reaches the adapter, which computes the length l_s of the data already received in the reception port (if any), dequeues and deallocates the remaining segments of the port's data scatter list, snaps off and deallocates l_s bytes from the beginning of the amendment's data buffering, places the resulting buffering in the port's data scatter list, and replies that the amendment is pending, with l_s bytes already received with the previous buffering. If l_s is nonzero, copying of l_s bytes will be necessary.

Out-of-order reception (not possible in ATM) in multiple-packet mode

[2] s can disable speculation by aborting r' before replying to r''.

is dealt with as an exception, in software, by copying. In case of server-aligned buffering (emulated copy input), adapter buffers and client buffers are distinct, and therefore out-of-order data in adapter buffers can be copied to client buffers directly. That is not possible, however, with in-place buffering (e.g., share or emulated share input). If, for example, d_0 and d_1 are received in-place out of order (where the length of d_1 is greater than or equal to that of d_0), it is necessary to copy d_1 to an area d', copy d_0 into the area previously of d_1, and finally copy d' into the area previously of d_0. Therefore, in out-of-order reception, compared with conventional data passing, emulated copy requires the same amount of copying, but share and emulated share require more copying.

Existing adapters do not implement buffer snap-off and instead simply drop the remainder of the current buffer segment at the end of a packet. This behavior jeopardizes copy avoidance, because:

1. In single-packet mode, if the current segment at the end-of-packet is not the last segment of the current request's buffering, storage for the next packet will incorrectly start in a segment that is not the first segment of the next request's buffering; and

2. In multiple-packet mode, the unfilled remainder of the current segment creates a gap in adapter buffers with respect to client buffers, causing data in subsequent segments to become misaligned and therefore require copying.

If the adapter supports early demultiplexing with header/data splitting per reception port, h is fixed, and t is null, host software may be able to approximate snap-off when processing a reply by dequeuing remaining segments of the request's buffering (single-packet mode) or enqueuing the remainder segment at the head of the data scatter list (multiple-packet mode). This software approximation does not work, however, if packets are received back-to-back, which doesn't give the host opportunity to fix the data scatter list before arrival of the next packet.

6.6 Outboard buffering

Network adapters with outboard buffering allocate input buffers from a pool in outboard memory. Data in outboard buffers can be transferred directly

into client buffers after successful input completion, providing strong integrity regardless of the semantics of data passing between client and system buffers, client buffer alignment or length, packet lengths, and whether the input request happens before packet arrival.

However, outboard buffering adds complexity and cost to the adapter. Unlike pooled in-host buffering, early demultiplexing, and buffer snap-off, which are examples of "cut-through" architectures, outboard buffering has a "store-and-forward" architecture, which typically imposes higher latency.

6.7 Checksumming

Network adapter hardware may checksum data while it is transferred (by DMA or other block move mechanism) into or out of host memory, thereby relieving the host processor (and memory) of this task. In the case of the emulated copy scheme, the adapter inputs data not into client buffers, but rather into server-aligned buffers. Consequently, if the checksum is incorrect, the previous contents of client buffers can be left intact, thus fully preserving copy semantics.

If checksums are transmitted in packet headers, as is the case in TCP/IP, hardware output checksumming requires packets to be staged in outboard buffers. Such staging allows the checksum field in the header to be inserted after data transfer from host memory and before transmission to the network. The amount of outboard memory necessary for staging all output may be non-trivial. A low-cost alternative may be to use hardware checksumming only for *input*, which does not require outboard memory, and for output use TCOW and checksumming by the host processor.

6.8 Related work

Carter and Zwaenepoel [20] show how driver software can approximate early demultiplexing for *blast* protocols on adapters with pooled in-host buffering. Blast protocols transmit long data in *blasts*, that is, multiple back-to-back packets, the last of which is acknowledged by the receiver. In [20], the driver, on receiving the first packet in a blast of more than two packets, replaces data segments at the head of the (single) scatter list by segments that point directly to the receiving client's memory beyond the data of the first two

packets in the blast. Data of the first two packets is copied, while the rest of the data is input directly from the network into the corresponding client memory, if the blast is not interrupted by packets destined to other reception ports. Interruptions by an unrelated packet require copying the data of that and the remaining blast packets to their correct destinations. [20] gives experimental evidence that such interruptions are very rare in practice; however, they constitute a potential protection leak. Additionally, the scheme supports in-place buffering but not emulated copy. The protection leak could be eliminated and emulated copy could be supported by making replacement segments point to server-aligned buffers and swapping pages only after input completion.

O'Malley et al. [62] modify Carter and Zwaenepoel's scheme so that each blast is preceded by a *control packet* describing the blast, and a 150 μs delay. The delay allows the receiving host to insert, at the head of the scatter list, appropriate header and data segments for all packets in the blast, so that blast data ends up concatenated in an adapter buffer. After receiving the last packet of the blast, the adapter buffer is mapped to a new region in the client's address space. This scheme eliminates the protection leak but does not support emulated copy; it implements move semantics.

The considerable effort to avoid copying demonstrated in [62] for an interface with move semantics reveals a little-appreciated fact about migrant-mode data passing: It makes copy avoidance easy only if the I/O interface can set both the location *and layout* of client input buffers. If clients can impose the layout constraint of, for example, input buffers being virtually contiguous, then the copy avoidance problem, in the multiple-packet case, can become as hard as in native-mode data passing.

The Nectar protocol processor [25] and, more recently, the Charisma [60] adapter allow clients to directly map outboard buffers, in a truly zero-copy arrangement that may be beneficial if the host processor can access outboard memory as efficiently as it can access host memory. Unfortunately, adapters normally connect to the host via an I/O bus. In most current architectures, this makes outboard memory uncacheable and imposes relatively high arbitration overheads for word-sized accesses by the host. Consequently, clients that actually access the I/O data (not the case in [60]) may very well run more slowly with the data outboard than in a more conventional arrangement, where I/O data is transferred to and from host memory. Compatibility may also be a problem because the zero-copy arrangement does not provide copy semantics. Additionally, outboard memory is normally not pageable,

making the scheme particularly susceptible to resource management problems when confronted with malicious or buggy clients that hog outboard buffers.

Depending on when data is checksummed, outboard buffering may result in semantics other than copy and introduce considerable jitter in transport-level acknowledgements. The Medusa [5] and Afterburner [27] adapters, for example, checksum while transferring data from outboard buffers to client buffers. Consequently, the latter may be corrupted with incorrect data, making the semantics share (weak integrity) rather than copy (strong integrity). Additionally, those adapters require packet acknowledgements to be delayed until the receiving client is scheduled to run *and* the client decides to receive the data. This can introduce considerable variability in round-trip times (RTT), which may interfere with transport-level algorithms that depend on precise RTT estimates. By checksumming while receiving data from the network into outboard buffers, the Gigabit Nectar WCAB [46] adapter is able to both guarantee copy semantics and maintain the usual decoupling between transport-level acknowledgements and client processing.

6.9 Summary

Table 6.1 summarizes the optimization conditions for native-mode input copy avoidance with each different type of input buffering in the adapter. Each successively higher level of network adapter support reduces native-mode copy avoidance restrictiveness.

Condition	Pooled in-host	Early demux	Buffer snap-off	Outboard
Input request before physical input		x	x	
Client buffers of preferred alignment and length	x			
Fixed-length headers	x			
All packets except last one have data length L	x			
Client buffers and transfers multiple of L	y			
Known transfer lengths	y	x		
Header, data and trailer lengths predictable given transfer length		x		

Table 6.1: Optimization conditions for native-mode input copy avoidance according to adapter input buffering (x = is required; y = either is required).

Chapter 7

Copy Avoidance Implementation in the Genie I/O Framework

Genie is a new I/O framework that implements the copy avoidance optimizations discussed in Chapters 3 to 6. This chapter describes Genie's interfaces and how Genie composes the different optimizations to implement each copy avoidance scheme.

This dissertation's experiments use an implementation of Genie on the NetBSD operating system. The implementation of Genie modifies the VM system and adds several system calls to NetBSD. However, Genie is not specific to that system; similar modifications could be made in other systems.

As an experimental testbed, Genie contains features that may not be suitable for production use. The intention is that optimizations and features found to be valuable in Genie may be transferred to conventional I/O frameworks, such as *sockets* [49].

7.1 Client interface

To allow direct comparisons among the many data passing schemes, Genie defines a client interface that is much more general than would be necessary if a single scheme were used. In production, the client interface could be a conventional explicit I/O interface, such as that of sockets. The emulated copy scheme is compatible with such interfaces, and therefore could be used

by default.

The Genie call for making a request is:

```
io_request(server, service,
           bufc, bufv,
           timeout, handlep)
```

where `server` specifies the server of the request (for example, a protocol stack or device driver), `service` specifies the requested service (for example, to input or output data or to open or close a connection), `bufc` is the number of buffers for the request, `bufv` is a vector of pointers to the descriptors of the buffers (explained in Section 7.3), `timeout` is the maximum time allowed for completion of the request, and `handlep` points to a request identifier returned by the interface. The client can use that identifier to synchronize with request completion or to abort the request. A null `handlep` makes the request synchronous. `io_request` returns the result of the request (for example, whether *success* or *pending*).

Clients can synchronize with completion of an asynchronous request by calling:

```
io_sync(handle, delivery, bufc, bufv, timeout)
```

where `delivery` can be *normal*, *peeked*, or *redirected*, `bufc` and `bufv` have the same meanings as in `io_request`, and `timeout` specifies the maximum time allowable for request completion. It `timeout` is null, `io_sync` polls the status of the request and does not block. If `delivery` is *normal*, `bufc` and `bufv` are ignored. If `delivery` is *peeked* and the request completed successfully, the interface copies data from each input buffer of the request to the corresponding peeking buffer specified by `bufc` and `bufv` (the amount of data copied is determined by the shortest of the two buffers); however, the request remains pending. If `delivery` is *redirected*, scatter-gather elements of client buffers specified in the original request are replaced by those specified by `bufc` and `bufv`. Only elements originally of input buffers and with copy or emulated copy data passing schemes are replaced.

Clients can abort an asynchronous request by calling:

```
io_abort(handle)
```

The client interface also includes calls to allocate and deallocate migrant buffers.

7.2 Server interface

The server interface allows any application to install itself as a user-level server, using the following call:

```
io_server_register(device, number_services,
                   service_mappingv, pool_size,
                   private_pool_size, private_poolp,
                   serverp)
```

where `device` is the device of which the server will be the driver (if any); `number_services` is the number of services offered by the server; `service_mappingv` is a vector with one element per service, defining the *read*, *write*, and *physical* mapping flags for each service; `pool_size` is the number of physical pages in the server's pool (used for data passing by copying, lending, or by the server itself); `private_pool_size` is the number of pages of the server's pool that should be *private*, that is, not used for copying or lending; `private_poolp` points to a vector with the virtual and physical addresses of each page in the server's private pool (returned by the interface); and `serverp` points to the identifier of the server (returned by the interface). If, before the call, the application points `serverp` to a nonzero value, the interface will attempt to assign that value (if it is still unassigned) as the server's identifier.

In Unix, `device` is the file descriptor of the corresponding device, obtained by opening the device. For protection, devices such as network adapters and disk controllers usually can be opened only by privileged users. A driver may use a private pool (returned in `private_poolp`) to implement, for example, pooled in-host buffering (Section 6.1).

A given application can install itself as one or more servers. A server can unregister itself implicitly, by exiting, or explicitly, by calling:

```
io_server_unregister(server)
```

While a server is registered, it sends and receives messages using the call:

```
io_server(server, out_msg, in_msg, blocking)
```

where `server` is the identifier of the server, `out_msg` points to an output message (if any), `in_msg` points to an input message (returned by the interface, if any), and `blocking` indicates whether the server would like to block if there

is no input message. The interface checks that the caller indeed is registered as the indicated `server`. `io_server` returns the status code resulting from processing the output message.

The output message is processed before the input message. An output message can contain any of the calls available to clients, including `io_request`, `io_sync`, and `io_abort`; or the reply to a request processed by the server. In the latter case, the interface checks that the caller indeed is the request's server.

An input message can be an interrupt notification (in case the server is registered as the driver of a device); a request to the server; or an asynchronous reply to a previous `io_request` made by the server using an output message. Interrupt notifications of a given device are coalesced and delivered before any other messages. Other input messages are delivered in FIFO order.

7.3 Buffer representation

A client buffer is described by a header and a scatter-gather list. Buffer headers are structures of the following type:

```
struct io_buffer_header {out_arg_length, out_arg,
                         in_arg_length, in_arg,
                         first_sg, direction,
                         actual_lengthp}
```

where `out_arg` points to an argument of length `out_arg_length` that is set by the client and should be passed to the server, `in_arg` points to an argument of length `in_arg_length` that is set by the server and should be passed to the client, `first_sg` points to the first scatter-gather element in the buffer, `direction` specifies the direction in which data specified by the scatter-gather list should be passed to or from the client (*in* or *out*), and `actual_lengthp` points to the data length actually input or output (returned by the interface).

Each element in the scatter-gather list is a structure of type:

```
struct io_scatter_gather {scheme,
                          location,
                          length,
                          next_sg}
```

where `scheme`, `location`, and `length` specify the data passing scheme, location and length of the data, and `next_sg` points to the next element in the list. The data passing scheme can be:

1. *Native-mode* (copy, emulated copy, share, or emulated share), in which case `location` is set by the client and gives: (1) the virtual address where data starts, and (2) the lender of corresponding system buffers (if element is not passed in-place);

2. *Migrant-mode* (move, emulated move, weak move, emulated weak move), in which case `location` points to a structure containing the number of virtually contiguous data segments in the element, set by the client, followed by pairs specifying the virtual address where data starts and the data length of each successive data segment. The data segments are set by the client, when buffer direction is *out*, or by the interface, when buffer direction is *in*.

3. *Fragment*, in which case `location` specifies the identifier of a request r, the index of a buffer b of r, and an offset from the start of b. The interface verifies that the caller indeed is the server of r.

Elements on the same scatter-gather list may have different semantics.

The interface links a system buffer to each scatter-gather element of a client buffer. The only cases in which scatter-gather element and corresponding system buffer refer to different physical memory are those where data passing is by copying or, on input, by the emulated copy scheme. System buffers are denoted by lists of structures of type:

```
struct io_seg_v {last_seg,
                 vec[SEG_V_SIZE],
                 next_sv}
```

where `last_seg` is the index of the last occupied element of `vec`, `next_sv` points to the next list element, and `vec` is a vector of structures of type:

```
struct io_seg {length, pa, va}
```

where `pa` and `va` are physical and virtual addresses of data in the system buffer, and `length` is the data length, guaranteed not to cross page boundaries.

System buffers become server buffers when they are passed to servers. In the case of kernel-level servers, the virtual addresses `va` are unmapped, as explained in Section 3.1.1. In the case of user-level servers, each virtual address `va` corresponds to the transient virtual address (TVA) assigned to the given page, as explained in Section 5.1.

7.4 Native- and migrant-mode data passing schemes

This section describes, in terms of primitive operations, how Genie implements each native-mode and migrant-mode data passing scheme. The breakdown into primitive operations is used in the analysis of experimental results in Chapter 8.

In Genie, data passing involves operations at *request* and *reply* times. For client input buffers, some request-time system buffer allocation operations may be deferred until *ready* time, which is when a server actually needs those buffers.

Genie takes advantage of the fact that copying usually is very efficient for short data. If data is shorter than configurable thresholds, Genie automatically converts the data passing scheme from emulated copy or emulated share to copy.

7.4.1 Client output buffers

The operations for passing data from client output buffers to system buffers are summarized in Table 7.1, where "read-only" means "remove write permissions", and "invalidate" means "remove all access permissions" from all mappings (page table entries) corresponding to a given physical page.

Genie does not remove a migrant region until reply time in order to guarantee that the corresponding virtual addresses will not be reassigned during request processing, thus allowing graceful recovery in case of error.

7.4.2 Client input buffers and customized system buffers

The system buffers that Genie links to client input buffers at request or ready time are called *customized system buffers*. Such buffers reside in host

	Request	Reply
Copy	Allocate system buffer. Copyin data.	Deallocate system buffer.
Emulated copy	Reference client pages. Read-only client pages.	Unreference client pages.
Share	Reference client pages. Wire region.	Unwire region. Unreference client pages.
Emulated share	Reference client pages.	Unreference client pages.
Move	Reference client pages. Wire region. Mark region *emigrating*. Invalidate client pages.	Unwire region. Unreference client pages. Remove region.
Emulated move	Reference client pages. Mark region *emigrating*. Invalidate client pages.	Unreference client pages. Mark region *emigrant* and enqueue.
Weak move	Reference client pages. Wire region. Mark region *emigrating*.	Unwire region. Unreference client pages. Mark region *weak emigrant* and enqueue.
Emulated weak move	Reference client pages. Mark region *emigrating*.	Unreference client pages. Mark region *weak emigrant* and enqueue.

Table 7.1: Operations for data passing from client output buffer to system buffer.

memory and are customized to each request so as to allow streamlined data passing to client input buffers. When the data passing scheme is emulated copy, customized buffers are server-aligned; when the data passing scheme is share, emulated share, emulated move, weak move, or emulated weak move, customized buffers are in-place.

For a device controller to be able to use customized buffers, special hardware features may be required. Network adapters, for example, must have early demultiplexing or buffer snap-off, as explained in Chapter 6. In addition, customized system buffers require that the input request occur before physical input of the data. Table 7.2 summarizes the data passing operations

under such conditions.

To maintain protection, the move data passing scheme has to complete with zero the unused portions of a system buffer before mapping it to a user-level client. If the data passing scheme is emulated move, weak move, or emulated weak move and at request time no suitable cached region can be found in the appropriate queue, Genie allocates a new region and marks it *immigrating*. For the same three schemes, Genie checks that the cached region that was linked as a system buffer at request time and was used for input is still present in the client address space at reply time. If it was removed (perhaps inadvertently) by the client, Genie maps the corresponding pages to a new region, guaranteeing that the location information returned to the client correctly points to the input data.

7.4.3 Client input buffers and overlaid in-host buffers

If a device controller does not support customized system buffers (for example, in the case of a network adapter with pooled in-host or outboard buffering) or the request does not happen before physical input, then the controller inputs data into a buffer allocated from the device's own buffer pool. At ready time, the device's driver *overlays* such a buffer on the client input buffer, and, if the pool is outboard, data passing continues as discussed in the next subsection. Otherwise, at reply time, Genie passes data from the overlay buffer to the client input buffer, makes an auxiliary request to the driver to deallocate the overlay buffer, and the driver returns the overlay buffer to the device's buffer pool for reuse, as shown in Figure 7.3.

In the move data passing scheme, Genie migrates overlay pages to the client address space and therefore needs to refill the overlay buffer with the same number of newly-allocated pages to avoid depletion of the device's buffer pool.

7.4.4 Client input buffers and overlaid outboard buffers

If the device controller allocates input buffering in outboard memory, Genie alters the operations in Table 7.2 as follows: For all data passing schemes other than emulated copy, at ready time, after the operations in Table 7.2, make an auxiliary request to the driver to DMA the data from outboard buffer to host memory and deallocate the outboard buffer. For emulated copy data passing, Genie does not allocate a system buffer at ready time

and, at reply time, references the client pages, makes an auxiliary request to the device's driver to DMA the data from the outboard buffer to the client pages and deallocate the outboard memory, and, when the driver replies, unreferences the client pages. Consequently, with outboard buffering, emulated copy is implemented much as emulated share.

7.5 Fragment data passing scheme

The client, in the fragment scheme, is a contractor making a request r' that is originated of some other request r. The fragment is a part b' of a buffer b of r. Genie links and uses as system buffers for b' a part of the system buffers of b.

On output, no data passing is necessary, because the system buffers for b already contain the data of b'. On input, however, some operations may be necessary at reply time to pass the system buffers of b' to the contractor or to b:

1. If: (1) b' had system buffers whose allocation had been deferred at the request time of r, and (2) these buffers were allocated during the processing of r', then Genie passes these buffers to the contractor and to b.

2. If the subcontractor overlaid an in-host buffer b_o on b', Genie: (1) overlays b_o on b on behalf of the contractor, and (2) passes b_o to the contractor.

3. If the subcontractor overlaid an outboard buffer b_o on b', then: If the service requested in r does not have a mapping flag set, Genie overlays b_o on b on behalf of the contractor; otherwise, Genie: (1) allocates any system buffers in b' whose allocation had been deferred, (2) makes an auxiliary request to the driver that overlaid b_o to DMA data from b_o to b' and deallocate b_o, and (3) passes b' to the contractor.

Genie passes buffers to the contractor using selective transient mapping. According to the mapping flags of the service requested in r, selective transient mapping may result in mapping, copying, passing physical addresses, or just passing the length of buffers to the contractor, as explained in Section 5.1.

7.6 Summary

Genie is a new I/O framework that provides different explicit I/O interfaces for, respectively, clients and servers. The main novelty in the client interface is the buffer representation, which allows data passing according to various schemes. This feature is used in this dissertation's experiments for direct comparisons between different schemes. In production, however, a conventional client interface, such as sockets, could be used. The server interface allows receiving interrupt notifications and requests, sending replies, making subcontract requests, and receiving replies to subcontract requests.

In addition to Genie's explicit I/O interfaces, this chapter describes how Genie composes the optimizations described in the previous chapters to implement native- and migrant-mode data passing schemes (used in original requests) and the fragment scheme (used in subcontract requests). Genie's input data passing is affected by whether the input request occurs before physical input and by the level of hardware support provided by the device.

Genie also supports the optimizations discussed in Sections 1.2.4 and 1.2.5, data passing avoidance and scheduling avoidance, respectively. Chapter 11 describes how Genie implements such optimizations.

	Request	Ready	Reply
Copy		Allocate system buffer.	Copyout data. Deallocate system buffer.
Emulated copy		Allocate aligned buffer.	Swap pages. Deallocate aligned buffer.
Share	Reference client pages. Wire region.		Unwire region. Unreference client pages.
Emulated share	Reference client pages.		Unreference client pages.
Move		Allocate system buffer.	Create region. Zero-complete system pages and fill region. Map region and mark *immigrant*.
Emulated move	Dequeue *emigrant* region, mark region *immigrating*, and reference client pages.		Check region, unreference client pages, reinstate page accesses, and mark region *immigrant*.
Weak move	Dequeue *weak emigrant* region, mark region *immigrating*, and reference client pages. Wire region.		Check region. Unwire region. Unreference client pages and mark region *immigrant*.
Emulated weak move	Dequeue *weak emigrant* region, mark region *immigrating*, and reference client pages.		Check region, unreference client pages, and mark region *immigrant*.

Table 7.2: Operations for data passing from customized system buffer to client input buffer.

	Ready	Reply
Copy	Allocate overlay buffer. Overlay buffer.	Copyout data. Deallocate overlay buffer.
Emulated copy	Allocate overlay buffer. Overlay buffer.	If aligned, swap pages, else copy out. Deallocate overlay buffer.
Share	Allocate overlay buffer. Overlay buffer.	Unwire region. Unreference client pages. If aligned, swap pages, else copy out. Deallocate overlay buffer.
Emulated share	Allocate overlay buffer. Overlay buffer.	Unreference client pages. If aligned, swap pages, else copy out. Deallocate overlay buffer.
Move	Allocate overlay buffer. Overlay buffer.	Create region. Zero-complete overlay pages, fill region and refill overlay buffer. Map region and mark *immigrant*. Deallocate overlay buffer.
Emulated move Emulated weak move	Allocate overlay buffer. Overlay buffer.	Check region. Unreference client pages. Swap pages. Mark region *immigrant*. Deallocate overlay buffer.
Weak move	Allocate overlay buffer. Overlay buffer.	Check region. Unwire region. Unreference client pages. Swap pages. Mark region *immigrant*. Deallocate overlay buffer.

Table 7.3: Ready- and reply-time operations for data passing from overlaid in-host buffer to client input buffer.

Chapter 8

Evaluation of Emulated Copy

This chapter evaluates the performance of emulated copy in end-to-end communication over a fast network. For such evaluation, experiments were conducted with different data passing schemes and levels of network adapter support. Experiments were repeated using different platforms and network transmission rates. In all experiments, user-level clients communicated by making requests directly to a kernel-level network driver.

The results show that emulated copy greatly improves performance relative to that of the copy scheme. Performance differences were small between emulated copy and other copy avoidance schemes, which have non-copy semantics. Analysis of the performance on various platforms at different network transmission rates suggests that current technological trends tend to increase the performance advantages of copy avoidance. Moreover, current trends tend to reduce performance differences among copy avoidance schemes.

8.1 Experimental set-up

The experiments were performed on computers of the types shown in Table 8.1. The cache and memory copy bandwidths reported are the peak values observed in *bcopy* benchmarks at user level. The integer rating taken as upper bound for the Gateway P5-90 is the listed SPECint95 of the Dell XPS 90, which has a bigger and faster L2-cache. The rating taken as upper bound for the AlphaStation is its listed SPECint_base95 because the version of NetBSD used on it could not be compiled with optimizations.

Model	Gateway P5-90	Micron P166
CPU	Pentium 90 MHz	Pentium 166 MHz
SPECint95	< 2.88 (Dell XPS 90)	4.52 (Dell XPS 166s)
L1-cache	8 KBI + 8 KBD, 1910 Mbps	8 KBI + 8 KBD, 3560 Mbps
L2-cache	256 KB, 244 Mbps	256 KB, 486 Mbps
Memory	32 MB, 4 KB page, 146 Mbps	32 MB, 4 KB page, 351 Mbps

Model	DEC AlphaStation 255/233
CPU	21064A 233 MHz
SPECint95	< 3.48 (255/233 base)
L1-cache	16 KBI + 16KBD, 2860 Mbps
L2-cache	1 MB, 1366 Mbps
Memory	64 MB, 8 KB page, 350 Mbps

Table 8.1: Characteristics of the computers used in the experiments. The integer rating used for each model is the listed SPECint95 of the related system indicated in parenthesis, which has the same CPU.

All computers ran the NetBSD 1.1 operating system with the VM modifications described in Chapters 3, 4, and 5. Conditional compilation directives were also inserted in NetBSD's *pmap* (physical VM) module for the Intel family of processors (including Pentium). These directives cause the pmap module to invalidate individual TLB entries instead of invalidating the whole TLB, when the CPU is i486 or later (the i386 lacks such instruction). This optimization significantly reduces the overhead of Genie's VM manipulations. New system calls were also added in order to support Genie's interfaces, which are summarized in Chapter 7. Clients used the Genie interfaces to request services of drivers implemented at kernel level.

In all experiments, clients requested data input and output through a Credit Net ATM adapter [47]. Two versions of the adapters were used, supporting transmission rates of 155 Mbps or 512 Mbps, respectively. Credit Net adapters transfer data between main memory and the physical medium by burst-mode DMA over the PCI I/O bus and support both pooled in-host buffering and early demultiplexing. The adapters perform ATM AAL5 segmentation and reassembly and automatically generate and check CRCs, but do not provide IP checksumming. The 155 Mbps cards automatically *depad* (i.e., remove the AAL5 trailer of) packets of length l such that (l mod 48) ≤ 40. The 512 Mbps cards do not have automatic depadding, causing,

in the case of early demultiplexing, some additional CPU utilization in the receiving host.

All experiments involved measuring latencies by capturing the value of the CPU on-chip cycle counter at appropriate points while running the code. Unless otherwise noted, all measurements were on otherwise idle computers and network. Each measurement run consisted of a "warm up" measurement, discarded, followed by measurements for data of increasing lengths. The data for the warm up measurement had length equal to the page size except in experiments for short data, where that length was 256 bytes. Reported measurements are the averages of five runs.

End-to-end latency was measured directly, while *I/O processing time* was estimated. The latter is the time spent by the CPU processing I/O, which is roughly inversely proportional to the maximum I/O throughput that the CPU is able to support before the CPU saturates. To estimate I/O processing times, the idle loop of the system's scheduler was instrumented to measure idle time during the measurements of end-to-end latency. I/O processing times were estimated by subtracting from end-to-end latency the time the CPU was idle. This yields only an estimate because systems were in multiuser mode and were subject to asynchronous activity, such as timer interrupts. The coefficient of variation (COV) was less than 3% for all end-to-end latency measurements and less than 20% for all I/O processing time measurements (and, respectively, less than 1.5% and 12.5% for 90% of the measurements).

All figures correspond to measurements on Micron P166 PCs and, unless otherwise stated, at 512 Mbps[1].

8.2 Single-packet end-to-end latency

This section examines the impact of the data passing scheme and input buffering on single-packet end-to-end latencies. Models of how latencies depend on the costs of primitive data passing operations and scale with processor, memory, and network speeds are presented and validated.

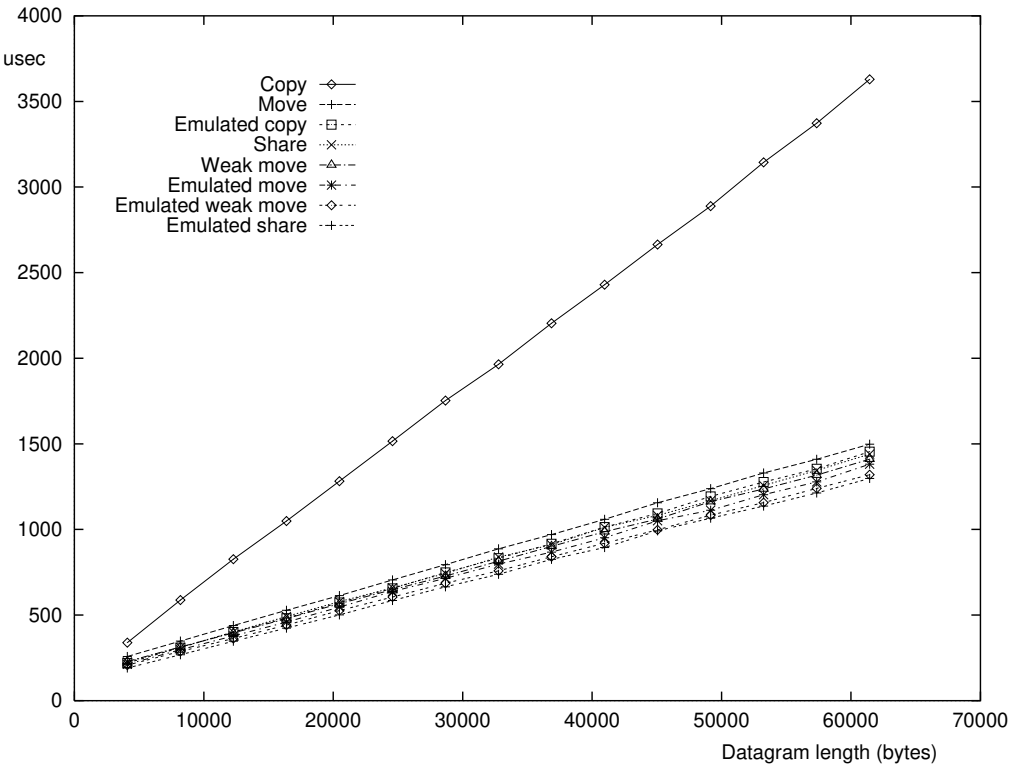


Figure 8.1: Single-packet end-to-end latency with early demultiplexing. Performance differences among schemes other than copy were small.

8.2.1 Measurements

Figure 8.1[2] shows single-packet end-to-end latencies, using early demultiplexing, for page-aligned client buffers and data lengths equal to a multiple of the page size. Data lengths varied up to 60 KB, the largest such multiple that fits in a single ATM AAL5 packet. For these data lengths, copy gave much higher latency than did any of the other schemes, which pass data using VM manipulations instead of copying. The differences among schemes other than copy were small. The most striking difference is that between the copy and emulated copy schemes, which implement the same data passing semantics. Using TCOW and input alignment, emulated copy reduced la-

[1]For corresponding measurements at 155 Mbps, please refer to [13].

[2]In all figures in this chapter, the legend lists data passing schemes in the same order as the respective curves. For fine discrimination among curves that appear cluttered in the figure, please refer to the fourth column of Table 8.6.

tencies for 60 KB datagrams by 60% relative to copy. For all data lengths, emulated copy also resulted in a latency lower than that of the move scheme and very close to that of the share scheme. Emulated copy benefits from its use of input-disabled pageout, which makes it unnecessary to wire and unwire regions, as is done in the move and share schemes. Emulated move gave slightly lower latencies than those of emulated copy because it simply invalidates and reinstates page table entries instead of fully swapping pages, which also requires updating the respective memory object. Latency was still lower, but only slightly, with the emulated weak move and emulated share schemes, which do not require page table updates. In this experiment, the equivalent throughput for single 60 KB datagrams was 135 Mbps for copy, 328 Mbps for move, 338 Mbps for emulated copy, 341 Mbps for share, 348 Mbps for weak move, 356 Mbps for emulated move, 372 Mbps for emulated weak move, and 379 Mbps for the emulated share scheme[3].

Figure 8.2 shows single-packet end-to-end latencies for short page-aligned data, using early demultiplexing. The move scheme gave by far the highest latency for short datagrams because it has to complete with zero the part of the page not occupied by client data. The emulated move scheme gave much lower latencies because it performs I/O in place, using region hiding, and therefore does not need to complete with zero the remainder of the page. The copy scheme gave close to the the lowest (120 μsec, practically tied with emulated copy and emulated share) but also the most rapidly rising latency because of the high incremental cost of copying.

In this experiment, output thresholds were set so that Genie automatically converted output of data shorter than 2048 bytes with the emulated copy scheme (t_o) or 280 bytes with the emulated share scheme to the copy scheme. The reverse copyout threshold (t_i) was set at 2304 bytes. (Performance is only moderately sensitive to these settings; these values were empirically determined to give good results.) With these settings, emulated copy had about the same latency as that of copy for data up to half a page long; above that, reverse copyout and swapping significantly reduced the latency of emulated copy relative to that of copy. Emulated share had, for all data lengths, the lowest latency, because its data passing overhead consists solely of page referencing and unreferencing. The difference between latencies with the emulated copy and emulated share schemes was maximal

[3]By pipelining data passing between client and server and network transmission, byte-stream protocols may achieve higher throughputs than those for single packets.

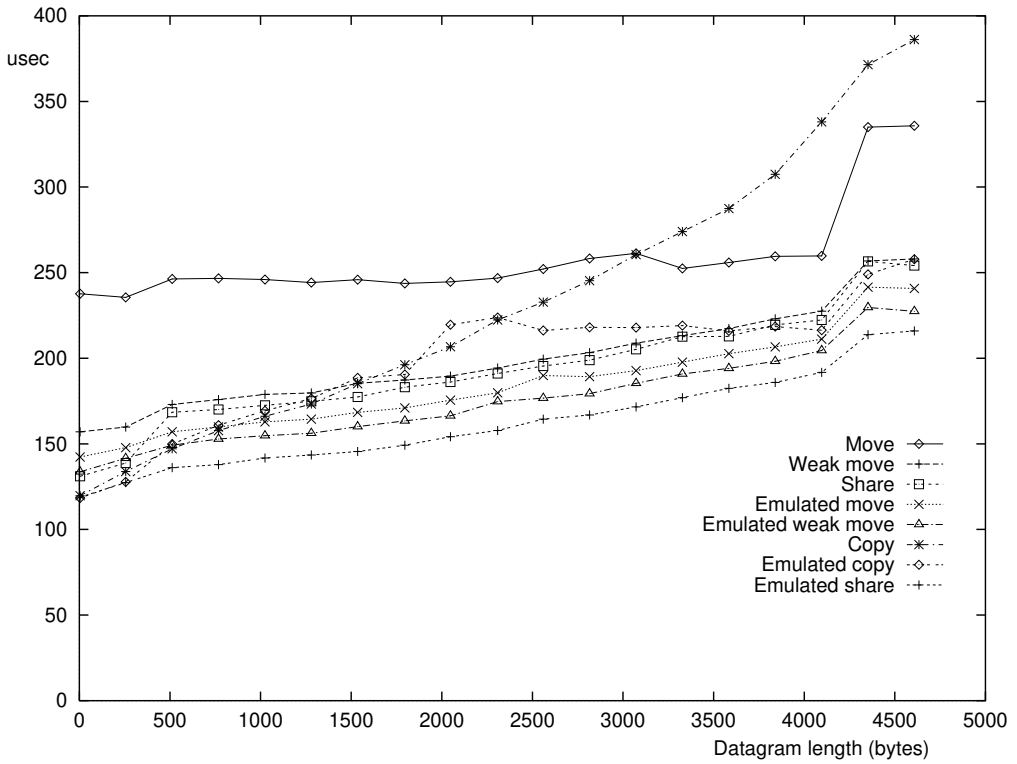


Figure 8.2: Single-packet end-to-end latency for short data, using early demultiplexing. Using reverse copyout, emulated copy avoids copying more than about half a page.

at half page size: 220 vs. 154 μsec. Weak move and emulated weak move gave slightly higher latencies than those of share and emulated share, respectively, because of region caching costs avoided in native-mode schemes. The slightly higher latency of the emulated move scheme relative to that of emulated weak move is due to region hiding. The higher latencies of share and weak move relative to their emulated counterparts are due to region wiring and unwiring, which are unnecessary in the emulated schemes because of the input-disabled pageout optimization.

Figure 8.3[4] shows single-packet end-to-end latencies with pooled in-host buffering and client-aligned input buffers. Copy and emulated copy had latencies only very slightly higher than the respective latencies with early demultiplexing, corresponding to the same operations plus buffer overlay

[4]For fine discrimination among curves, please refer to the third column of Table 8.7.

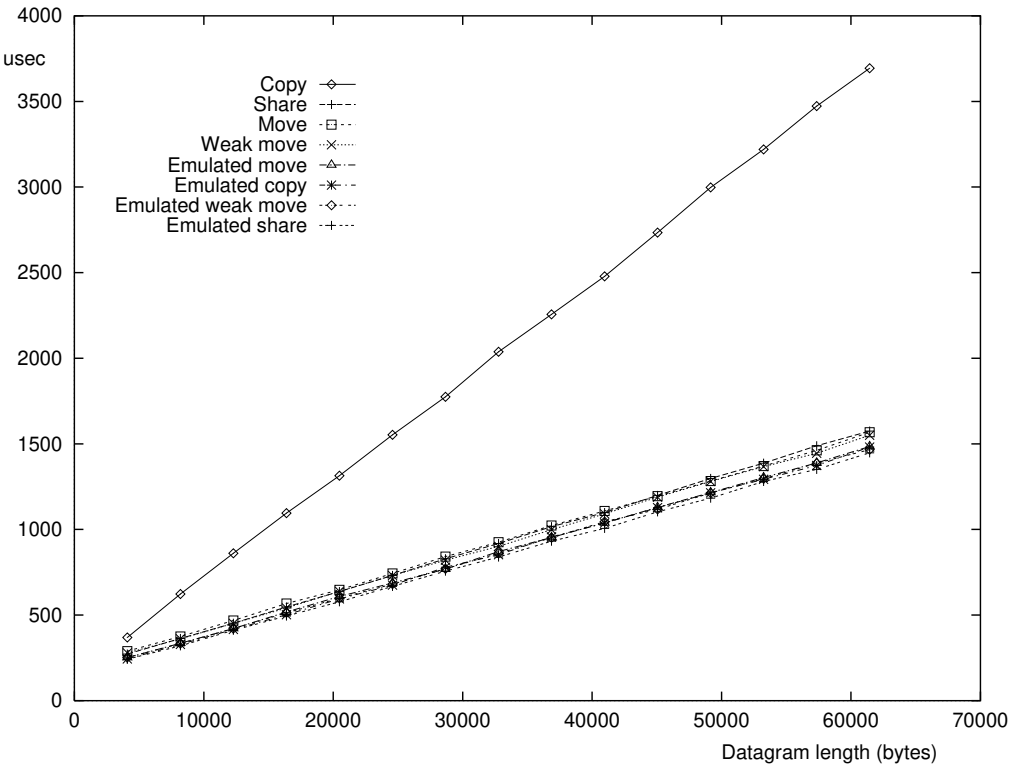


Figure 8.3: Single-packet end-to-end latency with client-aligned pooled in-host buffering. If there is alignment, native-mode schemes other than copy give performances similar to those of migrant-mode schemes.

overhead. The share, move, and weak move schemes had higher latencies than those of emulated copy because of region wiring and unwiring overheads. All other schemes had latencies very close to that of emulated copy. In this experiment, the equivalent throughput for single 60 KB datagrams was 133 Mbps for copy; 312 Mbps for share; 313 Mbps for move; 317 Mbps for weak move; 331 Mbps for emulated move; 332 Mbps for emulated copy; 335 Mbps for emulated weak move; and 339 Mbps for emulated share.

Figure 8.4[5] shows single-packet end-to-end latencies with pooled in-host buffering and unaligned client input buffers. Without alignment, emulated copy, share, and emulated share have to copy data to pass it to client input buffers, whereas the copy and migrant-mode schemes are unaffected. This figure shows the impact of data copying, splitting the data passing

[5]For fine discrimination among curves, please refer to the fourth column of Table 8.7.

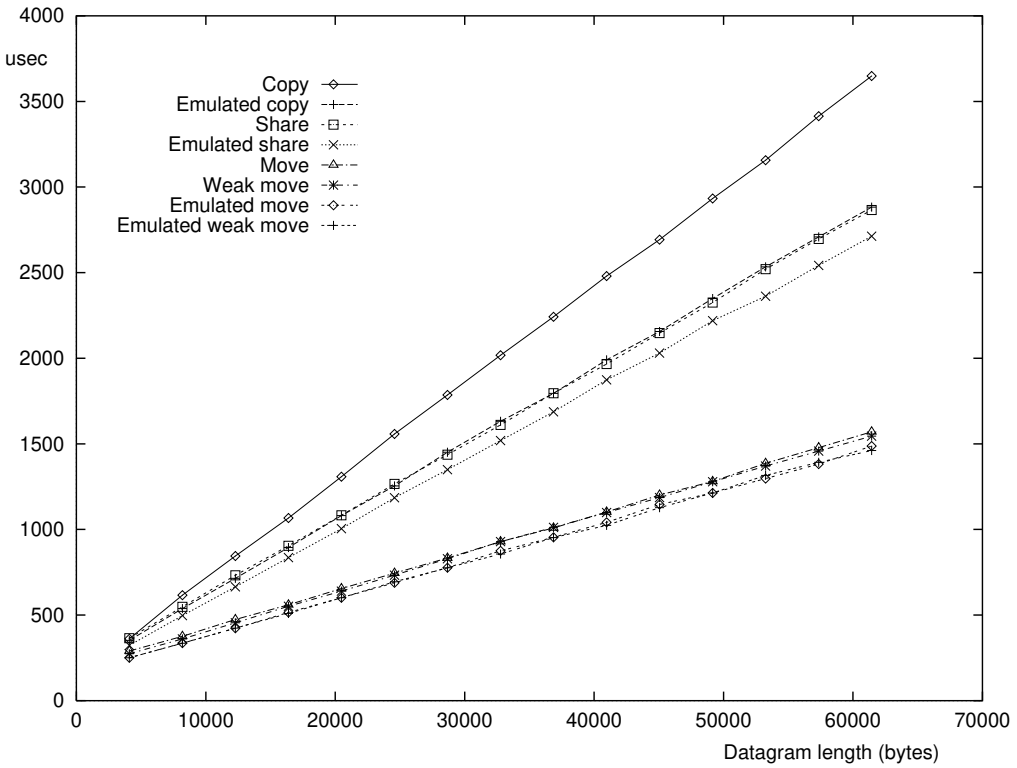


Figure 8.4: Single-packet end-to-end latency with unaligned pooled in-host buffering. Without alignment, native-mode schemes require copying to client input buffers.

schemes into a group with no copies (migrant-mode schemes), another with two copies (copy scheme, with one copy from client output buffers and another to client input buffers), and the remaining group, between the other two, with one copy (to client input buffers only). In this experiment, the equivalent throughput for single 60 KB datagrams was 135 Mbps for copy, 170 Mbps for emulated copy, 172 Mbps for share, 181 Mbps for emulated share, 313 Mbps for move, 318 Mbps for weak move, 331 Mbps for emulated move, and 336 Mbps for emulated weak move.

Figure 8.4 may give the impression that migrant-mode data passing is intrinsically more efficient than native-mode data passing if the device used supports only pooled in-host buffering. However, if a client is insensitive to data layout enough to use migrant buffers, then in principle that client can also use client-aligned buffers, and then emulated copy, emulated share, and

migrant-mode schemes give very similar performance, as shown in Figure 8.3. If, on the contrary, a client *is* sensitive to data layout, it would require client-level copies between migrant buffers and client data structures. The total number of copies (possibly one copy on output, if the client needs to retain access to the same or other data on the same pages, plus one copy on input) is then at best the same as if emulated copy or emulated share were used (one copy on input only). In those cases, migrant-mode data passing may actually give worse end-to-end performance than that of native-mode data passing.

In a fair comparison, therefore, emulated share comes out as the scheme with the best performance, and emulated copy is almost as good. Both emulated copy and emulated share can offer the same programming interface as that of copy and thus less incompatibility with applications written for copy semantics than do migrant-mode schemes.

8.2.2 Analysis

Single-packet end-to-end latencies can be broken down into the sum of a *base* latency and *data passing* latencies at the sender and receiver. The base latency captures end-to-end costs that are independent of the particular data passing scheme or input buffering used, such as crossing the user-kernel boundary and incurring driver, device, network, and interrupt latencies. Data passing latencies, on the contrary, depend on the data passing scheme and input buffering used. The base latency can be approximated by the end-to-end latency of the emulated share scheme with early demultiplexing, reduced by the costs of referencing and unreferencing client buffers.

Only *request*-time data passing operations at the sender contribute to end-to-end latency, because reply-time operations overlap with network latencies and latencies at the receiver. Conversely, with early demultiplexing, request- and ready-time operations at the receiver overlap with latencies at the sender and in the network, and only the *reply*-time operations at the receiver contribute to end-to-end latency. With pooled in-host or outboard buffering, only *ready*-time and *reply*-time operations at the receiver contribute to end-to-end latency.

The time intervals for each data passing operation and datagram length when performing experiments of the types shown in Figures 8.1, 8.3, and 8.4 were captured and averaged over five runs for different platforms and network transmission rates. For each platform and transmission rate, least-squares

linear fit on each operation latency versus data length provided excellent correlation, except in cases of constant or very small latencies. The fitted equations of each operation latency were averaged over the data passing schemes and type of input buffering where the operation is used, for each platform and transmission rate. Tables 8.2 and 8.3 show the results for Micron P166 PCs at 155 and 512 Mbps, respectively, while Tables 8.4 and 8.5 show the results for Gateway P5-90 PCs and AlphaStation 255/233 workstations at 155 Mbps.

On all three platforms, *copyin* cost less than *copyout* because the experiments were performed on warm caches. On output (*copyin*), data can be read from the cache, while on input (*copyout*) it has to be read from memory. The *copyin* cost is actually nonlinear because the L1-cache has, on all three platforms, much higher bandwidth than that of the L2-cache. This may cause a negative fixed term in the corresponding linear fit.

For the Micron P166 PCs at 155 Mbps and each data passing scheme, the base latency, the costs of the respective output request-time operations indicated in Table 7.1, and the costs of the respective input reply-time operations indicated in Table 7.2 were added, taking values from Table 8.2, so as to estimate the respective end-to-end latency with early demultiplexing. The third column of Table 8.6 shows these estimates, along with the least-squares linear fit of the actual end-to-end latencies. Likewise, for the Micron P166 PCs at 512 Mbps, base latency, costs of output request-time operations, and costs of input ready-time and reply-time operations indicated in Table 7.3 were added, for each scheme, so as to estimate the respective end-to-end latencies with pooled in-host buffering and client-aligned or unaligned client input buffers. Table 8.7 shows these estimates along with the least squares linear fits of the actual end-to-end latencies from Figures 8.3 and 8.4. The good fit between estimated and actual latencies suggests that this breakdown model is accurate for the data lengths considered, which are multiples of the page size (additional terms would increase accuracy for intermediate lengths but would also make the model more complicated).

Using the breakdown model, single-packet end-to-end latency when sender and receiver use different data passing schemes can be expected to be equal to the sum of the base latency plus sender-side latencies of the scheme used by the sender plus receiver-side latencies of the scheme used by the receiver.

The breakdown model can be extended into a scaling model that takes into account CPU, memory, and network speeds. To a rough approximation:

1. The multiplicative factor of the base latency is *network-dominated* and equal to the inverse of the effective network transmission rate (net of overheads for medium access control and physical encoding), subject to adapter and I/O bus bandwidth limitations;

2. The fixed term of the base latency is equal to the sum of I/O bus, device, and network latencies, plus a term corresponding to fixed operating system overhead, which scales inversely to CPU speed;

3. The *copyout* multiplicative factor is *memory-dominated* and equal to the inverse of the main memory copy bandwidth, and the associated fixed term can be ignored;

4. The *copyin* multiplicative factor is *cache-dominated* and may vary between the inverse of the copy bandwidth of the L2-cache and the inverse of the copy bandwidth of main memory, depending on data and cache sizes and cache associativity and locality. The fixed term can be ignored;

5. All other parameters are *CPU-dominated* and scale inversely to to CPU speed, as estimated by SPECint95 or some other integer benchmark[6] that is based on the performance of large programs with sparse references to memory (as is the case of an operating system).

This scaling model can be verified with data from Tables 8.1, 8.2, 8.3, 8.4, and 8.5. A comparison between costs for Micron P166 PCs at 155 and 512 Mbps (Tables 8.2 and 8.3) shows that differences other than that of the base cost are minimal, which is consistent with the model. Items (1) and (2) of the model are verified by comparing respectively the multiplicative factor and fixed term of the base cost in each case. The fourth column of Table 8.6 shows estimates for end-to-end performance with Micron P166 PCs at 512 Mbps based on measurements of costs on the same computers at 155 Mbps (Table 8.2). For simplicity, the multiplicative factor of the base latency was assumed, according to item (1) of the model, to vary inversely to transmission rate, but all other terms and factors were assumed to be

[6]The cost of page table updates may scale otherwise between processors of different architecture, causing the cost of the *read-only*, *invalidate*, *swap*, *region map*, and *reinstate* operations to diverge from this model. Page table updates are particularly costly in multiprocessors, where the data passing schemes that do not use these operations – weak-integrity (with customized system buffers) and copy – may have some advantage.

constant. This causes, according to (2), a somewhat inflated estimate of the fixed cost of the base case. However, even with this simplifying assumption, the predicted latencies are very consistent with the least squares linear fit of the actual end-to-end latencies from Figure 8.1.

Table 8.8 shows the verification of (1), (3), and (4) for each platform at 155 Mbps. Table 8.9 shows the verification of (3), (4), and (5) across platforms, also at 155 Mbps. Agreement between estimated and actual scaling is quite good for the Gateway P5-90 PC, which has a Pentium CPU, the same (albeit at a slower clock) as that of the Micron P166 PC (which is the base case). In the AlphaStation, CPU-dominated ratios have geometric means consistent with the model but varied much more widely (between 0.47 and 3.77) than those of the Gateway P5-90 (which varied between 1.53 and 2.59). This could be expected, given the architectural differences: The AlphaStation has an Alpha CPU, instead of a Pentium.

A comparison between Tables 8.3, 8.6, and 8.7 using the scaling model explains the clustering in Figures 8.1, 8.3, and 8.4. Network-dominated costs strongly dominated CPU-dominated costs (making performance differences among schemes other than copy relatively minor), but not memory and cache-dominated costs (penalizing the copy scheme).

Extrapolating based on the scaling model, if CPU speeds continue to increase faster than transmission rates, as is the current trend, the performance differences among schemes other than copy will tend to decrease, and if CPU speeds continue to increase faster than main memory bandwidth, the performance difference between copy and other schemes will increase.

8.3 Maximum throughput

This section estimates the maximum throughput θ that each data passing scheme can sustain.

To simplify analysis, the system is assumed to run a single client that handles multiple *work units* concurrently. Each unit corresponds to data of length L and requires total CPU time t_C for client processing and $t_{I/O}$ for I/O processing. The physical I/O subsystem is assumed to support a maximum throughput θ_{PHYS}, considering device, controller, and I/O bus capacity.

If the system has no idle CPU time (i.e., there are enough concurrent work units to eliminate CPU idle time), then $\theta = L/(t_C + t_{I/O})$. Conversely, if the physical I/O subsystem is saturated, then $\theta = \theta_{PHYS}$. In general, the

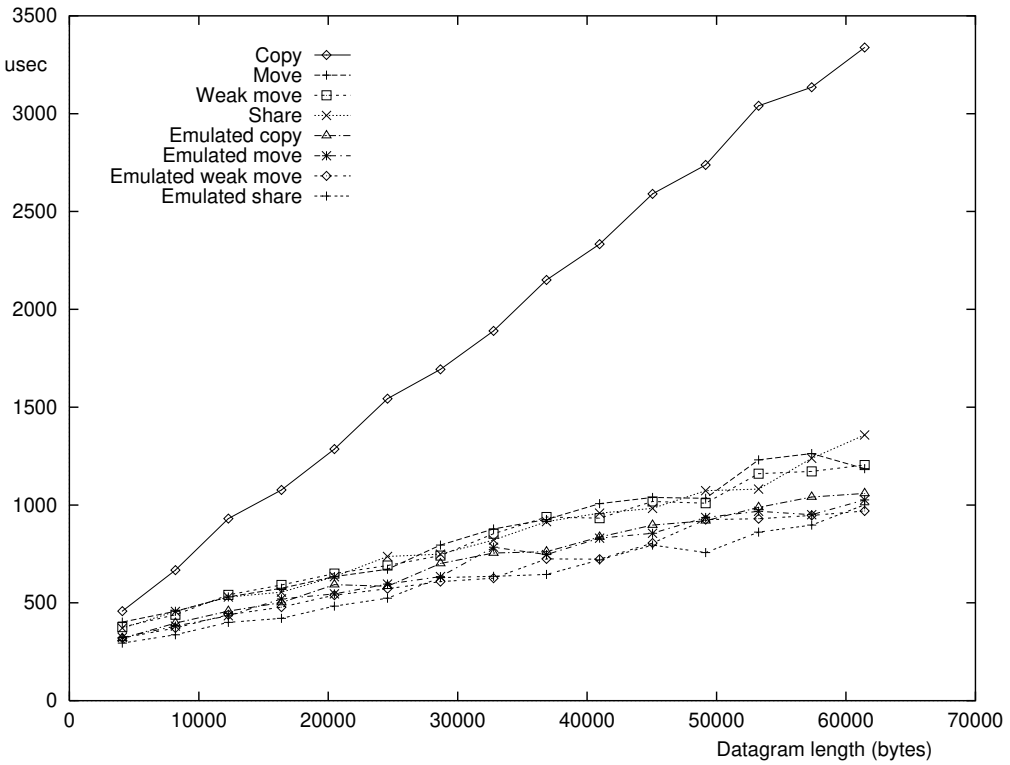


Figure 8.5: I/O processing time ($t_{I/O}$) with early demultiplexing. I/O with the copy scheme leaves much less CPU time available for client processing.

system can support a maximum throughput[7]:

$$\theta = \min(\theta_{PHYS}, \frac{L}{t_C + t_{I/O}})$$

$t_{I/O}$ can be estimated by subtracting CPU idle time from end-to-end latency. In the case of early demultiplexing, reported in Figure 8.1, the $t_{I/O}$ obtained is shown in Figure 8.5. $t_{I/O}$ was much higher for the copy scheme than for any other scheme: The I/O processing for an exchange of 60 KB datagrams occupied the CPU during 3338 μs with copy, 1358 μs with share, 1204 μs with weak move, 1186 μs with move, 1059 μs with emulated copy, 1026 μs with emulated move, 1001 μs with emulated share,

[7]The throughput for single datagrams quoted in other sections does not involve multiple concurrent work units, may saturate neither CPU nor physical I/O subsystem, and therefore may be less than the maximum throughput derived here.

and 970 μs with emulated weak move. Given the coefficients of variation of these measurements, between 5% and 10%, the differences among emulated schemes are not statistically significant.

Substituting back into the equation for θ:

1. If $t_C \gg t_{I/O}$ (client performs much more computation than I/O), then θ is essentially independent of the data passing scheme.

2. Conversely, if $t_C \ll t_{I/O}$ (I/O-intensive client, e.g., network file server), then:

   (a) If $t_{I/O} \leq L/\theta_{PHYS}$ (saturated physical I/O subsystem), then θ is also independent of the data passing scheme; or

   (b) If $t_{I/O} > L/\theta_{PHYS}$ (saturated CPU), then $\theta = L/t_{I/O}$. In the case of early demultiplexing, using the data from Figure 8.5, one can estimate that, for $L = 60$ KB, emulated copy and emulated share can increase maximum throughput with respect to that of copy by roughly 215%. (The improvement will be less if the the CPU saturates with copy but not with the other schemes.)

To a rough approximation, t_C scales inversely to CPU speed; $t_{I/O}$ scales inversely to the memory copy bandwidth for the copy scheme, or inversely to CPU speed for other schemes; and θ_{PHYS} scales proportionally to device speed. Therefore, the effects of current trends in CPU, memory, and device speeds can be projected as follows:

1. For the copy scheme, the relationship between t_C and $t_{I/O}$, for clients with good cache locality, may tend to $t_C \ll t_{I/O}$ (I/O-intensive client) because CPU speeds are increasing faster than memory bandwidth is. For clients with poor locality, t_C may scale similarly to $t_{I/O}$. The relationship between $t_{I/O}$ and L/θ_{PHYS} tends to $t_{I/O} > L/\theta_{PHYS}$ (saturated CPU) for devices, such as high-speed networks, whose bandwidth is increasing faster than that of memory.

2. For schemes other than copy, t_C tends to scale similarly to $t_{I/O}$. The relationship between $t_{I/O}$ and L/θ_{PHYS} tends to $t_{I/O} \leq L/\theta_{PHYS}$ (saturated physical I/O subsystem) because CPU speed is improving faster than device bandwidth is.

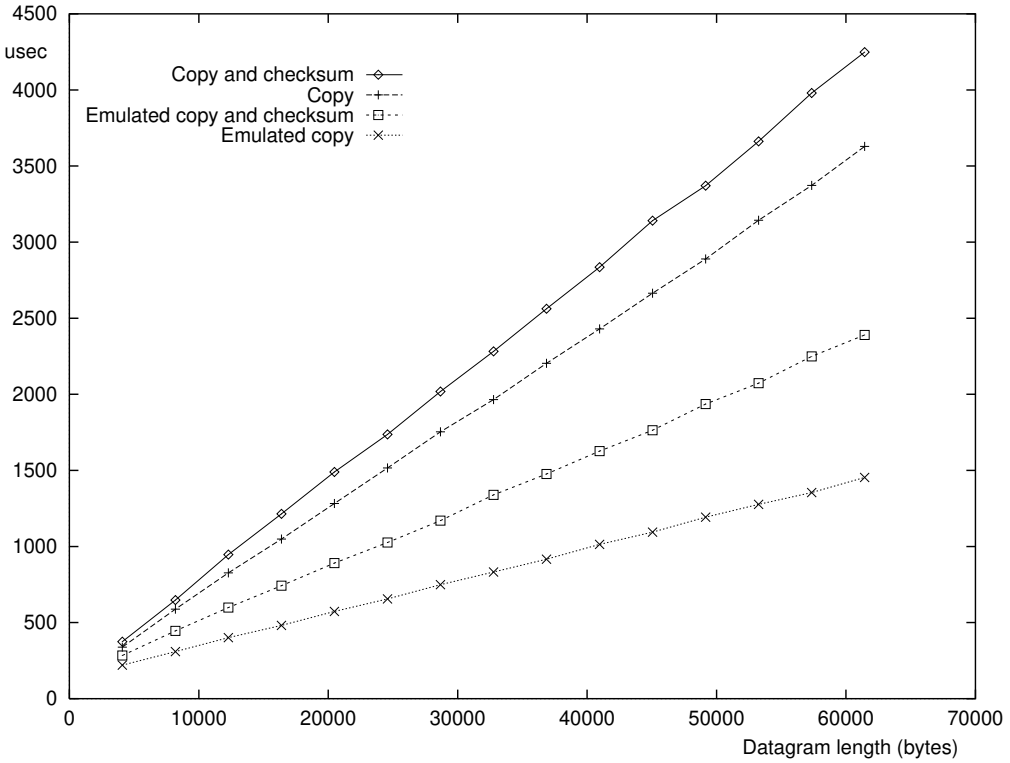


Figure 8.6: Single-packet end-to-end latency with or without checksumming. Emulated copy is cheaper than copy even if the host processor also has to read the data for checksumming.

Consequently, either because of compute-intensive clients or saturated physical I/O subsystem, current trends tend to reduce maximum throughput differences among schemes other than copy. For I/O-intensive clients, the tendency is to increase performance differences between copy and other schemes, because of saturation of CPU/memory in the former and of the physical I/O subsystem in the latter.

8.4 End-to-end latency with checksumming

This section examines the impact of checksumming on the performance advantages of data passing schemes that avoid copying.

Figure 8.6 shows single-packet end-to-end latencies with early demultiplexing, using the copy or emulated copy data passing schemes, and with or

without IP checksumming by the host processor. Client buffers were page-aligned. In this experiment, the equivalent throughput for single 60 KB datagrams with or without checksumming was, using copy, 116 or 135 Mbps, and using emulated copy, 206 or 338 Mbps, respectively. The penalty of checksumming was lower when using copy because copy brings data into the cache, which emulated copy doesn't. The penalty of checksumming could be further reduced in the copy scheme, on output, by integrating the checksum and copy in a single step [24], so that each data word is read and written only once (in the experiment of Figure 8.6, copying and checksumming were performed separately, so that each data word was read twice and written once). The same optimization would be incorrect for input because the client buffer would be overwritten with erroneous data when the checksum fails, violating the integrity guarantees of copy semantics. The curves for "copy" and "copy and checksum" in the figure can be taken, respectively, as the lower and upper bound of the latency using integrated copy and checksum. The figure shows that the cost of emulated copy plus checksumming was substantially less than that lower bound. This is because VM manipulations are very cheap and checksumming only requires one memory access, to read data, while copy requires two memory accesses, to read and write data.

8.5 Multiple-packet end-to-end latency

This section examines the impact of data fragmentation into packets on the performance advantages of data passing schemes that avoid copying.

Figure 8.7 shows multiple-packet end-to-end latencies at 155 Mbps with pooled in-host buffering, eight-byte packet headers, and no transport-level flow control. All curves other than the top two used emulated copy. The bottom two curves indicate that, when the client input buffer was aligned according to the preferred offset and the MTU was equal to 8 KB (two pages), the throughput for 60 KB with or without header/data splitting was 122 or 119 Mbps, respectively. Performance was slightly worse without header/data splitting because of the overhead of reverse copyout at the beginning and copyout at the end of each packet. The throughput for 60 KB decreased to 93 Mbps when the MTU used was 9180 bytes (the default MTU for IP over ATM [3]) and the client input buffer started with the preferred offset but was contiguous. Only the data in the first two pages ended up aligned, and the rest had to be copied. (The first two pages of *each* packet could

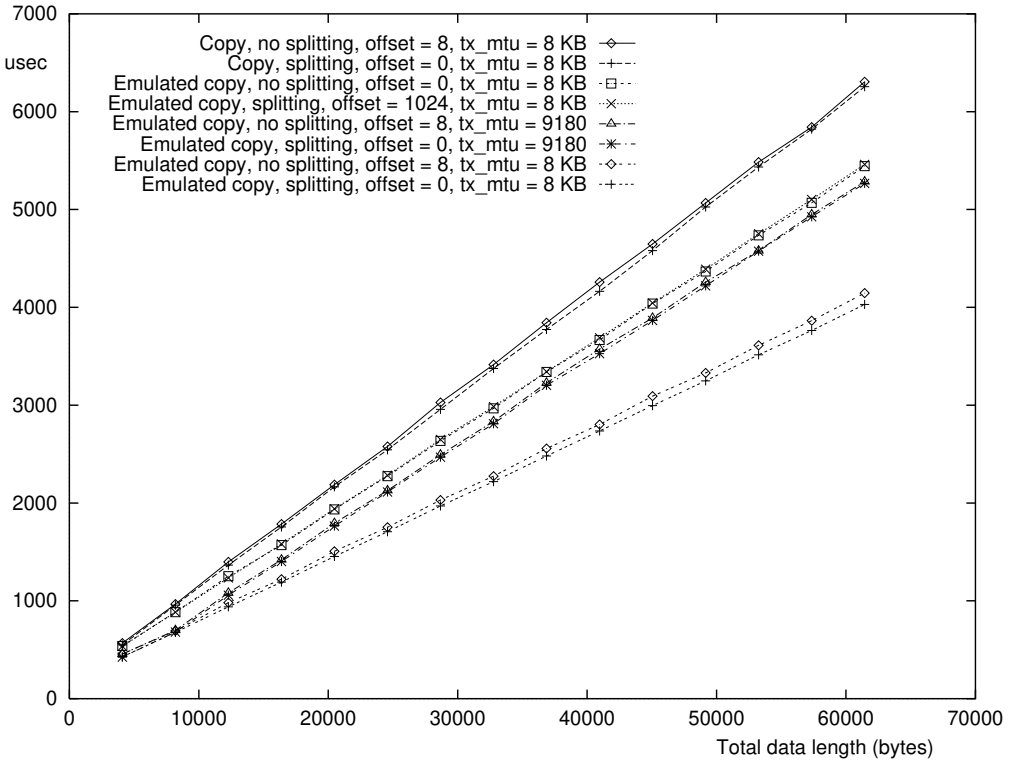


Figure 8.7: Multiple-packet end-to-end latency with pooled in-host buffering at 155 Mbps. The performance of client-aligned buffering is not substantially affected by the presence or absence of header/data splitting.

be aligned and swapped if the client used non-contiguous input buffers, with each segment conforming to the preferred offset and length.) The throughput for 60 KB decreased further to 90 Mbps when the client input buffer already started misaligned, so that all data had to be copied, although the MTU was a multiple of the page size (8 KB). Even in this last case, emulated copy still gave much better performance than did copy. This advantage is due to TCOW's copy avoidance on output. The top two curves indicate that the throughput for 60 KB using copy was 78 Mbps. This value is fairly insensitive to header/data splitting, client input buffer alignment, and MTU.

Figure 8.8 shows multiple-packet end-to-end latencies at 155 Mbps with early demultiplexing, eight-byte packet headers, and no transport-level flow control. Client input buffers were offset from a page boundary by 1 KB. The throughput for 60 KB with copying was 80 Mbps and fairly insensitive

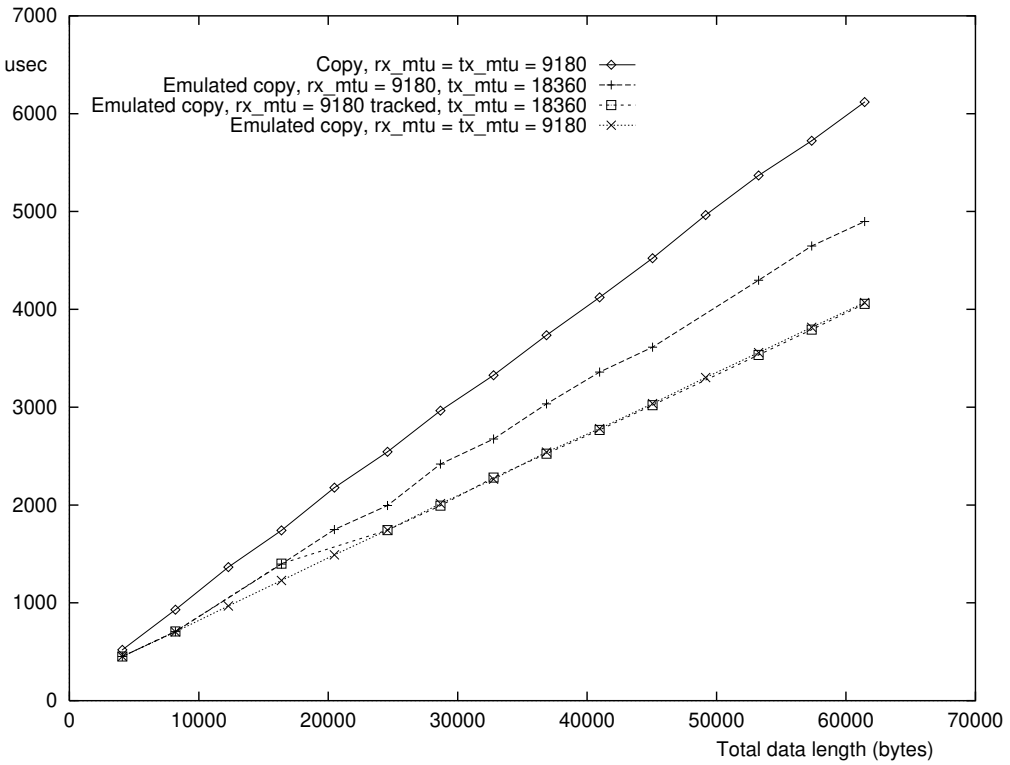


Figure 8.8: Multiple-packet end-to-end latency with early demultiplexing at 155 Mbps. Copying is avoided regardless of client input buffer alignment and MTU. The MTU can be quickly estimated by MTU tracking.

to the MTU. The throughput for 60 KB with emulated copy was 121 Mbps if the actual MTU (tx_mtu) was the same as that expected by the receiver (rx_mtu), 9180 bytes. The actual MTU could vary widely without affecting performance, as long as it was the same as that expected by the receiver. When the actual MTU was 18360, twice the size expected by the receiver, the throughput for 60 KB fell to 100 Mbps, because data beyond the first two pages ended up not aligned in system buffers and had to be copied to the client input buffer for delivery. The figure also shows latencies for the first run using MTU tracking, that is, estimating the MTU based on the longest packet received. After data longer than the actual MTU was received, MTU estimation converged to the actual value, and performance became the same as when the MTU was fixed and known.

Buffer snap-off extends the conditions under which server-aligned buffer-

ing occurs. However, buffer snap-off does not improve the performance in situations where server-aligned buffering already occurs. Consequently, performance with buffer snap-off can be estimated to be roughly equal to the bottom line in Figure 8.8.

8.6 Summary

The main observations from this chapter's experiments are:

- Emulated copy implements copy semantics while giving performance almost as good as or better than the other copy avoidance schemes.
- Emulated share can offer the same programming interface as that of copy and provides the best overall performance. However, emulated share offers lower integrity guarantees than does emulated copy.
- Emulated copy and emulated share with client-aligned buffering impose fewer restrictions and give about the same performance as do migrant-mode data passing schemes. This is the case even if the network adapter has only the minimum level of support considered here, pooled in-host buffering (i.e., single scatter-gather list).
- Current technological trends tend to increase performance differences between copy and other schemes and reduce performance differences among schemes other than copy.
- Absence of checksumming support in network adapters reduces but does not invalidate the performance advantage of emulated copy relative to copy.
- Data fragmentation into packets imposes optimization conditions for emulated copy that can be successively reduced by each higher level of network adapter support: pooled in-host buffering, early demultiplexing, buffer snap-off, outboard buffering.
- Header-data splitting provides very little performance improvement and no reduction in optimization conditions relative to pooled in-host buffering.

Operation	Latency
Base	0.0598 B + 130
Copyin	0.0180 B - 3
Reference	0.000363 B + 5
Wire	0.00141 B + 18
Read only	0.000367 B + 2
Invalidate	0.000373 B + 2
Region mark out	3
Copyout	0.0220 B + 15
Unreference	0.000100 B + 2
Unwire	0.000237 B + 10
Swap	0.00163 B + 15
Region create	24
Region fill	0.000398 B + 9
Region fill & overlay refill	0.000716 B + 11
Region map	0.000474 B + 6
Region check, unreference, reinstate, mark in	0.000507 B + 11
Region check, unreference, mark in	0.000194 B + 6
Region check	5
Region mark in	1
Overlay allocate	7
Overlay	7
Overlay deallocate	0.000344 B + 12

Table 8.2: Costs of primitive data passing operations on the Micron P166 computer at 155 Mbps, in μsec. B is the data length in bytes.

Operation	Latency
Base	0.0190 B + 110
Copyin	0.0169 B
Reference	0.000346 B + 5
Wire	0.00113 B + 18
Read only	0.000375 B + 2
Invalidate	0.000366 B + 2
Region mark out	3
Copyout	0.0221 B + 11
Unreference	0.000105 B + 2
Unwire	0.000218 B + 9
Swap	0.00166 B + 13
Region create	21
Region fill	0.000398 B + 6
Region fill & overlay refill	0.000723 B + 8
Region map	0.000518 B + 5
Region check, unreference, reinstate, mark in	0.000501 B + 13
Region check, unreference, mark in	0.000181 B + 7
Region check	5
Region mark in	1
Overlay allocate	5
Overlay	6
Overlay deallocate	0.000327 B + 12

Table 8.3: Costs of primitive data passing operations on the Micron P166 computer at 512 Mbps, in μsec. B is the data length in bytes.

Operation	Latency
Base	0.0718 B + 179
Copyin	0.0440 B - 63
Reference	0.000649 B + 10
Wire	0.00256 B + 33
Read only	0.000648 B + 3
Invalidate	0.000707 B + 3
Region mark out	5
Copyout	0.0535 B + 23
Unreference	0.000186 B + 3
Unwire	0.000440 B + 18
Swap	0.00285 B + 27
Region create	43
Region fill	0.000714 B + 15
Region fill & overlay refill	0.00132 B + 20
Region map	0.000835 B + 12
Region check, unreference, reinstate, mark in	0.000972 B + 20
Region check, unreference, mark in	0.000327 B + 11
Region check	9
Region mark in	2
Overlay allocate	17
Overlay	15
Overlay deallocate	0.000543 B + 22

Table 8.4: Costs of primitive data passing operations on the Gateway P5-90 computer at 155 Mbps, in μsec. B is the data length in bytes.

Operation	Latency
Base	0.0710 B + 235
Copyin	0.00974 B - 5
Reference	0.000347 B + 8
Wire	0.00175 B + 23
Read only	0.000275 B + 4
Invalidate	0.00141 B + 4
Region mark out	5
Copyout	0.0182 B + 1
Unreference	0.000244 B + 6
Unwire	0.000450 B + 25
Swap	0.00443 B + 12
Region create	43
Region fill	0.000428 B + 8
Region fill & overlay refill	0.000730 B + 10
Region map	0.00175 B + 3
Region check, unreference, reinstate, mark in	0.00167 B + 16
Region check, unreference, mark in	0.000227 B + 12
Region check	7
Region mark in	2
Overlay allocate	17
Overlay	11
Overlay deallocate	0.000336 B + 28

Table 8.5: Costs of primitive data passing operations on the AlphaStation 255/233 computer at 155 Mbps, in μsec. B is the data length in bytes.

Scheme		Early demultiplexing 155 Mbps	Early demultiplexing 512 Mbps
Copy	E	0.0997 B + 141	0.0581 B + 141
	A	0.0998 B + 125	0.0568 B + 116
Emulated copy	E	0.0621 B + 153	0.0205 B + 153
	A	0.0622 B + 150	0.0214 B + 133
Share	E	0.0619 B + 165	0.0202 B + 165
	A	0.0621 B + 162	0.0210 B + 143
Emulated share	E	0.0602 B + 137	0.0186 B + 137
	A	0.0600 B + 137	0.0193 B + 109
Move	E	0.0628 B + 197	0.0211 B + 197
	A	0.0626 B + 202	0.0217 B + 172
Emulated move	E	0.0610 B + 151	0.0194 B + 151
	A	0.0609 B + 150	0.0201 B + 133
Weak move	E	0.0620 B + 173	0.0201 B + 173
	A	0.0615 B + 170	0.0206 B + 143
Emulated weak move	E	0.0603 B + 144	0.0187 B + 144
	A	0.0602 B + 143	0.0194 B + 127

Table 8.6: Estimated (E) and actual (A) end-to-end latencies on Micron P166 computers, in μsec. Estimates for both 155 and 512 Mbps were based on measurements at 155 Mbps. B is the data length in bytes.

Scheme		Client-aligned pooled	Unaligned pooled
Copy	E	0.0584 B + 146	0.0584 B + 146
	A	0.0579 B + 136	0.0569 B + 143
Emulated copy	E	0.0218 B + 154	0.0422 B + 153
	A	0.0215 B + 154	0.0442 B + 174
Share	E	0.0227 B + 179	0.0432 B + 178
	A	0.0228 B + 174	0.0436 B + 190
Emulated share	E	0.0214 B + 152	0.0418 B + 151
	A	0.0211 B + 151	0.0417 B + 154
Move	E	0.0225 B + 196	0.0225 B + 196
	A	0.0222 B + 199	0.0222 B + 196
Emulated move	E	0.0217 B + 162	0.0217 B + 162
	A	0.0215 B + 162	0.0214 B + 163
Weak move	E	0.0227 B + 187	0.0227 B + 187
	A	0.0222 B + 182	0.0222 B + 186
Emulated weak move	E	0.0214 B + 160	0.0214 B + 160
	A	0.0213 B + 162	0.0214 B + 162

Table 8.7: Estimated (E) and actual (A) end-to-end latencies on Micron P166 computers at 512 Mbps, in μsec. Estimates were based on measurements at 512 Mbps. B is the data length in bytes.

	Micron P166		Gateway P5-90	
Type of Parameter	Estimated	Actual	Estimated	Actual
Network-dominated	> 0.0570	0.0598	> 0.0570	0.0718
Memory-dominated	0.0228	0.0220	0.0548	0.0535
Cache-dominated	[0.0165, 0.0228]	0.0180	[0.0328, 0.0548]	0.0440

	AlphaStation 255/233	
Type of Parameter	Estimated	Actual
Network-dominated	> 0.0570	0.0710
Memory-dominated	0.0229	0.0182
Cache-dominated	[0.00586, 0.0229]	0.00974

Table 8.8: Data passing costs estimated according to effective network transmission rate (about 140 Mbps for raw transmission rate of 155 Mbps) and memory and L2-cache copy bandwidths are consistent with the actual values.

	Gateway P5-90			
Type of Parameter	Estimated	GM	Min	Max
Memory-dominated	2.40	2.43	2.43	2.43
Cache-dominated	[1.44, 3.33]	2.46	2.46	2.46
CPU-dominated multiplicative factor	> 1.57	1.79	1.58	1.92
CPU-dominated fixed term	> 1.57	1.83	1.53	2.59
	AlphaStation 255/233			
Type of Parameter	Estimated	GM	Min	Max
Memory-dominated	1.00	0.83	0.83	0.83
Cache-dominated	[0.26, 1.39]	0.54	0.54	0.54
CPU-dominated multiplicative factor	> 1.30	1.64	0.75	3.77
CPU-dominated fixed term	> 1.30	1.54	0.47	3.74

Table 8.9: Scaling of data passing costs relative to the Micron P166. "GM", "Min", and "Max" are the geometric mean, minimum, and maximum values of the ratios of parameters of each given type.

Chapter 9

Evaluation of I/O-oriented IPC

Microkernel systems are often judged according to how well they match monolithic systems in: (1) compatibility with existing programs; and (2) performance. Hence, microkernel systems often include one or more servers emulating conventional APIs (e.g., that of Unix) and attempt to show that they can compile system code as fast as does an existing monolithic system [67].

The IPC facility determines, to a large extent, both the compatibility and performance of a microkernel system. The ultimate goals of an IPC facility, therefore, should be, *at the same time*: (1) to provide a client interface that is compatible with conventional APIs; and, (2) to give user-level servers performance that is, for practical purposes, as good as that of equivalent kernel-level servers.

For applications whose I/O is primarily storage-related, such as compiling, the combined goals of compatibility and performance have often been achieved. Even if the system's IPC facility is inefficient and copies data multiple times between clients and servers, the API emulation library linked with applications can map open files and transparently implement explicit file I/O by reading and writing the corresponding mapped region. Alternatively, applications may access files using the mapped file I/O interface. In either case, data and control passing costs for file accesses are almost as good as those in a monolithic system. For network-related and other I/O with ephemeral server buffers using conventional APIs, however, microkernels typically have had poor performance.

I/O-oriented IPC targets specifically the latter case, which most needs improvement: network-related and other I/O with ephemeral server buffers.

	Micron PP200
CPU	Pentium Pro 200 MHz
SPECint95	8.20 (Dell XPS Pro200n)
L1-cache	8 KBI + 8 KBD, 2490 Mbps
L2-cache	256 KB, 668 Mbps
Memory	32 MB, 4 KB page, 374 Mbps

Table 9.1: Characteristics of the computers used in the experiments. The integer rating used for the Micron PP200 is the listed SPECint95 of the Dell XPS Pro200n, which has the same CPU.

Chapters 4 and 5 make the case that I/O-oriented IPC rates highly with respect to the first goal of an IPC facility, compatibility. This chapter evaluates how close I/O-oriented IPC comes to reaching the second goal, performance.

Experiments on the Credit Net ATM network at 512 Mbps demonstrate that I/O-oriented IPC gives user-level protocol servers performance approaching that of kernel-level ones. In fact, for data longer than about a page, the performance of a user-level server using I/O-oriented IPC and emulated copy is *better* than that of a kernel-level server with conventional data passing by copying.

9.1 Experimental set-up

In this chapter's experiments, user-level clients communicated over the Credit Net ATM network at 512 Mbps by making input and output requests to a kernel- or user-level datagram protocol server. I/O-oriented IPC was used in the user-level server case. The kernel- and user-level protocol servers were identical, and both subcontracted a kernel-level network driver. Original clients used data passing schemes compatible with conventional explicit I/O interfaces: copy, emulated copy, and emulated share.

The experiments were performed on Micron PP200 PCs with the characteristics shown in Table 9.1. Scaling tests were also performed on Gateway P5-90 and Micron P166 PCs (Table 8.1). In all tests, pairs of identical machines were used. The operating system used was NetBSD 1.1 with the modifications described in Section 8.1. The measurement methodology was similar to that described in Section 8.1.

Both experiments with kernel-level and user-level servers were on NetBSD,

	Copy	Emulated copy	Emulated share
Kernel-level	178	323	346
User-level	175	307	320

Table 9.2: Equivalent throughputs for single 60 KB datagrams (Mbps).

a monolithic system. This set-up has the virtue of keeping the most factors constant, but, arguably, experiments with user-level servers should be on a microkernel system. I/O-oriented IPC extensively optimizes the data path between user-level clients and servers. The resulting data passing overheads are competitive to, if not better than, those of existing microkernel systems. However, the implementation of I/O-oriented IPC described here does not yet optimize the control path. IPC goes though the standard trap and context switching mechanisms of NetBSD. These mechanisms are much more heavy-weight than those of a true microkernel system, such as L4. The context switch time of L4 on the Pentium processor at 166 MHz is 0.74 μs [51], more than an order of magnitude less than that of NetBSD on the same processor[1]. Consequently, the experimental set-up used here is biased against user-level servers, particularly for short data, where context switch costs are most significant.

9.2 Measurements

Figure 9.1 shows the end-to-end latency for communication between user-level clients making requests to kernel- or user-level protocol servers (KLS and ULS, respectively). Data passing between user-level client and system buffers was implemented by the copy, emulated copy, or emulated share scheme. The curves indicate that for data longer than about a page, end-to-end performance was dominated by the network latency and presence or absence of data copying in the end systems. IPC avoidance (KLS vs. ULS) had relatively low impact, and the particular copy avoidance scheme used (emulated copy or emulated share) also had little importance. Table 9.2 shows the equivalent throughputs for single 60 KB datagrams.

Figure 9.2 shows the corresponding latencies for short data. The *ULS*

[1]I/O-oriented IPC has a much more aggressively optimized data path than that of L4, however. For arbitrarily located client data, L4 does not completely avoid copying; it simply reduces the number of copies in IPC to one.

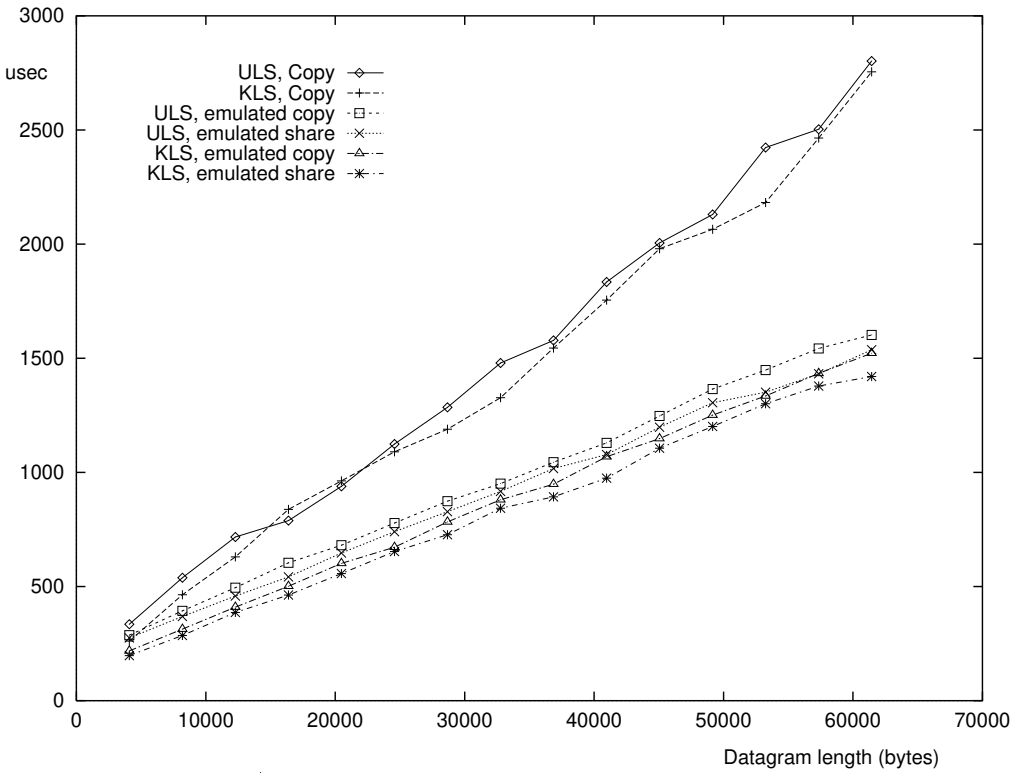


Figure 9.1: With I/O-oriented IPC, the performance of user-level servers (ULS) approaches that of kernel-level ones (KLS). For data longer than about a page, user-level servers had *better* performance, with emulated copy, than that of conventional kernel-level servers, with data passing by copying.

overhead, i.e. the difference in end-to-end latencies with user- or kernel-level servers and a given data passing scheme, was about 60 μs for 4-byte datagrams.

Figures 9.1 and 9.2 indicate that the ULS overhead is fairly insensitive to data length and data passing scheme, that is, ULS overhead is more of a control passing overhead than a data passing overhead. In I/O-oriented IPC, much of the ULS overhead is due to a full context switch between user-level client and user-level protocol server at each end host. This clearly would not be the case in conventional IPC facilities, which pass data by copying. Additionally, it should be observed that much of the ULS overhead measured here may be an artifact of the experimental set-up used; it is well possible that I/O-oriented IPC would achieve much lower ULS overheads on the same

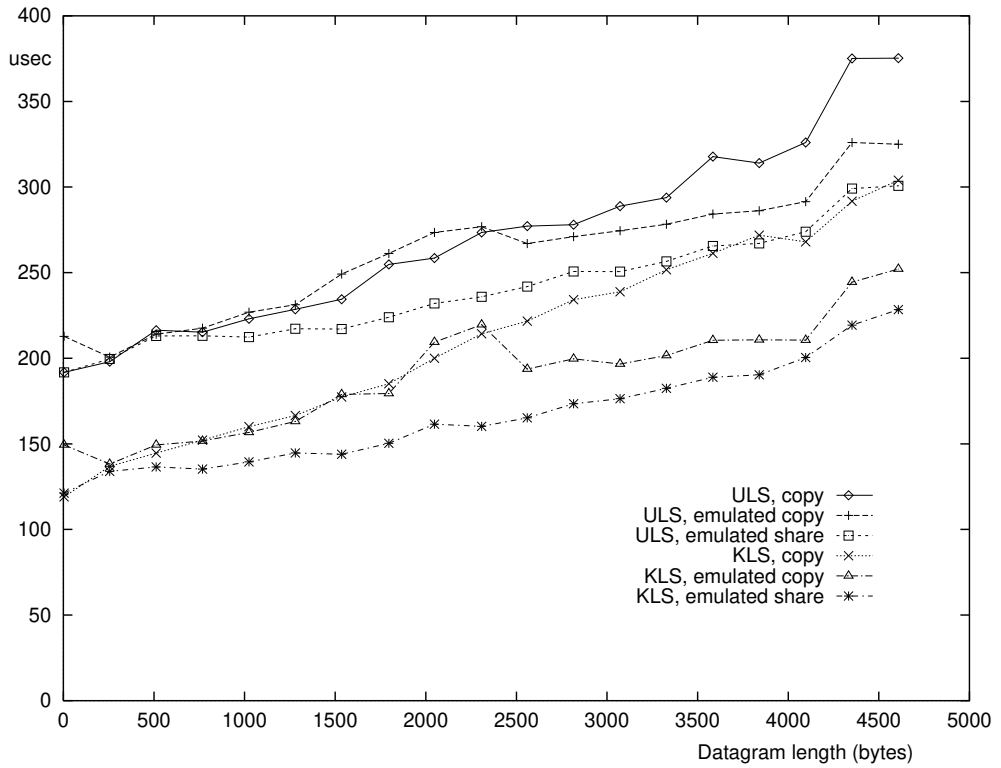


Figure 9.2: With I/O-oriented IPC, the higher latencies of user-level servers are due mostly to context switching, not to copying.

hardware if tests had been made on a true microkernel system.

Figure 9.3 shows end-to-end short-datagram latencies using emulated copy on the same network but with different end hosts: pairs of Pentium 90 MHz (P90), Pentium 166 MHz (P166), or Pentium Pro 200 MHz (PP200) PCs. The curves demonstrate that faster processors reduce not only end-to-end latency with kernel-level servers (with a lower bound at the device latency), but also the ULS overhead. The 4-byte latency with kernel-level servers on the P90 was close to that with user-level servers on the P166, a processor introduced about one year later than the P90. Similarly, the 4-byte latency with kernel-level servers on the P166 was close to that with user-level servers on the PP200, a processor introduced about one year later.

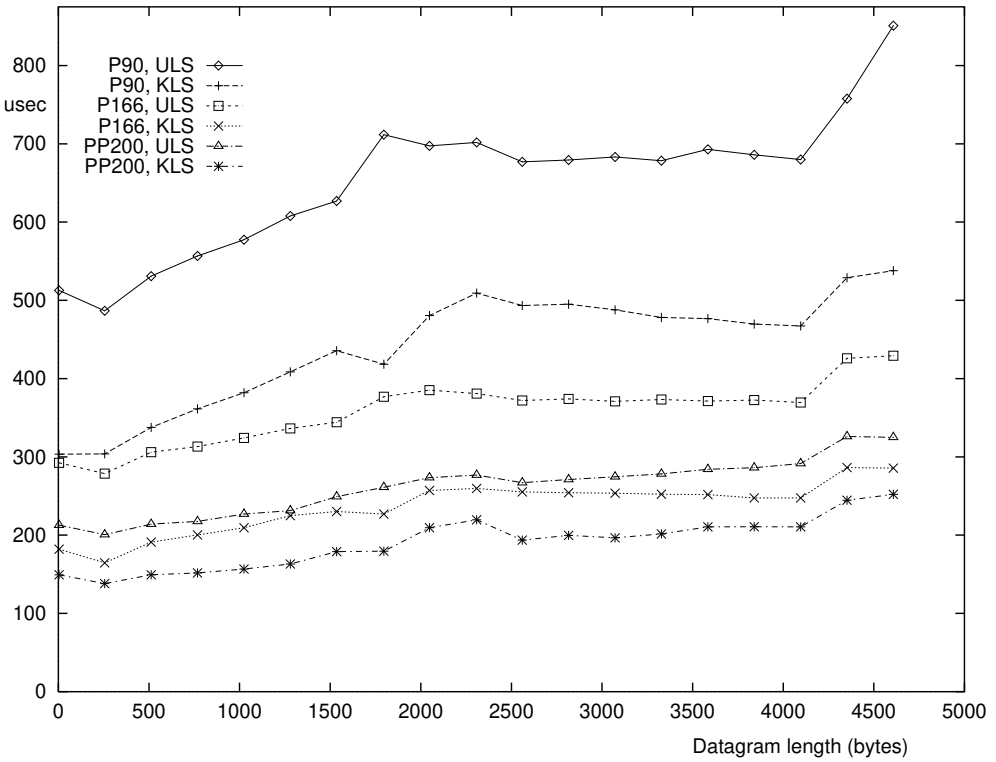


Figure 9.3: Processor improvements are quickly reducing the ULS overhead.

9.3 Analysis

In the measurements with user-level servers, the latter were the lenders of system buffers (as is the default case), and therefore neither zero completion nor copying between system and server buffers was necessary. In these conditions, I/O-oriented IPC involves no more data-touching operations than does a system call, and one may expect ULS overhead to scale according to processor performance, rather than memory performance. Table 9.3 validates this model by comparing scalings predicted from the processors' SPECint95 benchmarks with the actual scalings computed from Figure 9.3.

Extrapolating based on this scaling model, if processor performance continues to improve as fast as in the recent past (more than 50% per year [40]), the ULS overhead and the potential benefit of IPC avoidance relative to I/O-oriented IPC will drop quickly.

Processor	SPECint95	ULS overhead (μs)	Predicted scaling	Actual scaling
Pentium 90 MHz	< 2.88	209	> 2.85	3.32
Pentium 166 MHz	4.56	110	1.80	1.75
Pentium Pro 200 MHz	8.20	63	1.00	1.00

Table 9.3: Scaling of ULS overhead of each platform relative to that of the Pentium Pro 200 MHz PCs.

9.4 Summary

The main observations from this chapter's experiments are:

- Sufficiently long data allows the copy avoidance gains of I/O-oriented IPC to more than offset the higher control passing costs of IPC relative to a system call. For data longer than about a page, user-level servers using I/O-oriented IPC can have better performance than that of conventional kernel-level servers.

- Unlike conventional IPC facilities, I/O-oriented IPC has overheads that are due mostly to control passing, not data passing.

- The overheads of I/O-oriented IPC have been decreasing very fast, inversely to processor performance.

Chapter 10

Interoperability of I/O with Ephemeral and Cached Server Buffers

The new copy avoidance optimizations described and evaluated in Chapters 3 to 9 target specifically I/O with ephemeral server buffers. However, many applications perform I/O that involves both ephemeral and cached server buffers. For example, Unix's `ftp` may input data from a file and output that data over a network, or vice-versa. File I/O usually involves cached server buffers, whereas network I/O involves ephemeral server buffers. This chapter examines the interoperability of both types of I/O.

Copy avoidance in I/O with cached server buffers is provided by mapped file I/O. Section 10.1 describes how mapped file I/O can be integrated with the copy avoidance techniques described in the previous chapters for I/O with ephemeral server buffers. Such integration allows applications to pass data without copying between file systems and networks. Legacy applications, however, may still use explicit I/O, instead of mapped file I/O, for access to files. Section 10.2 discusses that case. Experiments in Section 10.3 show that both mapped file I/O and emulated copy can reduce response times and I/O processing times in a distributed application with remote file accesses over a network. Mapped file I/O and emulated copy are synergistic: Their combined effect is greater than the sum of their individual benefits.

10.1 File access by mapped file I/O

Data from a mapped file region can be output to a network using any of the schemes described in Chapters 3 to 7, passing data from file system to network without copying. If output is with move semantics, it implicitly unmaps the file from the corresponding region and deallocates the region. A potential problem is that a page with nonzero output reference count may migrate to a server's cache when the last region that contains the page is unmapped. To avoid TCOW faults, it is desirable to change the file server so that it refrains from writing on a page with nonzero output reference count. The server should instead write on a new page, complete the new page in a manner analogous to reverse copyout (Section 4.1), and then swap pages in the server's cache.

Similarly, data can be input directly from a network to a mapped file region, passing data from network to file system without copying. A potential difficulty is that data may arrive from the network preceded by an application-level header. The header may, for example, specify the file, offset from the beginning of the file, and length of the data. Because the application does not know the final destination of the data before examining the header, copy avoidance will usually require page swapping. However, page swapping will only be possible when data is received from the network in a buffer that starts at a page offset equal to the file offset modulo the page size. For simplicity, let us assume that the file offset o is always a multiple of the page size, and what may vary is only the file and, possibly, the length of the header (h) and of the data (l). Page swapping then requires that data be received from the network in a page-aligned buffer, stripped of any headers. The technique used to achieve page alignment will depend both on the data passing semantics used for input from the network and on the type of input buffering provided by the network adapter. The case of emulated copy and server-aligned buffering achieves copy avoidance using only standard techniques. Other non-copy data passing schemes and input buffering types, however, require application-level optimization to achieve copy avoidance.

If network input is by emulated copy with server-aligned buffering, the application informs the network server that input is expected, consisting of an arbitrarily aligned segment of h bytes, followed by a page-aligned segment of l bytes. The server correspondingly scatters the input. The application does not immediately receive the input. Instead, the application first *peeks* at the header (e.g., using Unix's standard `recv` call with `MSG_PEEK` flag). After

decoding the header, the application inputs the first h bytes to a region that is not a mapped file region, followed by l bytes directly to the corresponding mapped file region (the application can use, e.g., Unix's standard `readv` call with a scatter list). No copying is necessary, but the network adapter must support server-aligned buffering, as discussed in Chapter 6. If h and l are fixed, early demultiplexing is sufficient. However, if h or l is variable, buffer snap-off is necessary.

If network input is by emulated copy with client-aligned buffering and the network adapter supports only pooled in-host buffering, copy avoidance becomes more difficult because data may arrive with arbitrary alignment from the network. Page alignment can be restored, however, by using the software technique discussed in Section 6.3, *header patching*. This technique requires that both sender and receiver agree on its use. Let h' be the preferred alignment for input from the network. Using header patching, the sender transmits the application-level header followed by the data starting at file offset $o+h'+h$ and of length $l-h'-h$, followed by data starting at file offset o and of length $h'+h$. The receiver peeks at the first $h'+h$ bytes of the input. After decoding the header, the receiver inputs the first l bytes directly to the address a corresponding to file offset o in the mapped file region, and the following $h'+h$ bytes also to an address a (the I/O interface itself patches the initial data of length $h'+h$ on top of the headers at address a).

If network input is by any non-copy scheme other than emulated copy, copy avoidance requires an addition to existing mapped file I/O interfaces. In-place input directly to the mapped file region is not possible because the application must first decode the header to determine the region. Likewise, migrant-mode input brings data into a newly allocated region, not the destination mapped file region. In these cases, the application should input the header and data to a page-aligned region other than its final destination, decode the header, and patch the header. The application then uses a new technique, *user-directed page swapping*, to pass the data to its destination without copying. User-directed page swapping is implemented by a new system call:

```
mswap(source, destination, length, deallocate)
```

where `source` is the start address of the source region, `destination` is the start address of the destination (mapped file) region, `length` is the length of both regions, and `deallocate` indicates whether pages from the destination region should be passed to the source region or simply deallocated.

`mswap` arguably is a fundamental call that is missing in mapped file I/O interfaces such as that of Unix. It is necessary, for example, for passing data between mapped files without copying (`mswap` would be used with a true `deallocate` flag for such purpose). To process a `mswap` call, the system verifies write access permissions to both regions, pages in any missing pages in the source region, marks all pages of the source region dirty, and swaps pages between the regions. Before swapping, if `deallocate` is true, the system removes and deallocates any pages in the destination region (without flushing to the respective pager), and therefore the source region ends up without any physical pages. After `mswap`, the contents of the destination region is the same as if the data had been copied.

In the in-place case, the source region would in general be needed for subsequent input, and therefore `mswap` could be used with a false `deallocate` flag. In the migrant-mode case, on the contrary, if the source region is to be deallocated, `mswap` could be used with a true `deallocate` flag.

10.2 File access by explicit I/O

Data input from a network can be output, using an explicit I/O interface, to a file. However, using only the new optimizations described in this dissertation, copy avoidance will be limited to network input; file output will copy data from client buffer to server cache. For copy avoidance also on file output, mapped file I/O can be used, as described in the previous section.

Conversely, data input from a file, using an explicit I/O interface, can also be output correctly, without modifications, to a network. However, to avoid TCOW faults and unnecessary page copying, the following refinement should be made in data passing from server cache to client buffer: For each page of the client buffer, if the client page has zero output reference count, data can be simply copied from server cache to client page. If, however, the client page has nonzero output reference count, data should be copied from server cache to a new page; if the new page is not completely filled, it should be completed by reverse copyout; and the new page should be swapped with the client page. This modification is transparent to clients and preserves copy avoidance gains of network output when the buffer is reused for file input. For copy avoidance also on file input, mapped file I/O can be used, as described in the previous section.

10.3 Demonstration

This section experimentally demonstrates that emulated copy interoperates efficiently with mapped files. The experiments measure the performance of a synthetic benchmark application when files and/or the network are accessed with or without copy avoidance. The goal of the experiments is to measure the effects of copy avoidance in best-case and worst-case conditions; the experiments are intended as demonstrative only and not as an evaluation of benefits for a typical load.

10.3.1 Experimental design

In the benchmark application, a *client* fetches or stores file data by making requests to a remote *server*. Requests contain a header that includes the code of the service requested, a request identifier that the server should include in the corresponding reply, and, in case of fetch or store requests, the file identifier, file offset, and data length. In the case of store requests, data follows the header in the same packet. Replies contain a header that includes the identifier of the corresponding request, the resulting status (e.g., *success* or *bad file*), and the length of the data actually fetched or stored. In the case of successful fetch replies, data follows the header in the same packet.

Both client and server run at user level. Each party communicates with the other by making requests directly to a kernel-level network driver. Both parties use the same data passing scheme, which may be either copy or emulated copy.

The server accesses files using either the conventional explicit I/O interface (Unix's `read` and `write`, with data passing by copying) or the mapped file interface (Unix's `mmap` using shared mode). The server maintains a list of open files per client.

The server uses several optimizations in the case of mapped files. To each open mapped file corresponds one or more mapping regions. Mapping regions have always the same size, so as to prevent fragmentation of the server's address space. To allow for expansion of files open for writing, the server maintains its own indication of the true length of the file. Before mapping such a file to a region whose end would lie beyond the end of file, the server repositions the end of file to correspond to the end of the region. The server truncates such a file to its true length before closing and unmapping it.

The client requests to fetch or store data always of the same file and of

	Micron XRU266
CPU	Pentium II 266 MHz
SPECint95	10.8 (Intel Portland)
L1-cache	16 KBI + 16 KBD, 3234 Mbps
L2-cache	512 KB, 626 Mbps
Memory	32 MB, 4 KB page, 390 Mbps
Disk controller	Adaptec AHA 2940 Ultra Wide SCSI-2
Disk	Seagate ST34371W "Barracuda", 4.35 GB, 512 KB cache, 7200 RPM, average/max seek 8.8/19 ms

Table 10.1: Characteristics of the computers used in the experiments. The integer rating used for the Micron XRU266 is the listed SPECint95 of the Intel Portland motherboard, which has the same CPU.

length and offset (from the beginning of the file) equal to a multiple of the page size. The experiments measure the response time observed by the client and the I/O processing time incurred at the server for each combination of data passing schemes for network and file access. Two types of measurement are made for each data length. The *miss* measurement is taken right after unmapping mapped regions (if any) of the file under study and performing explicit I/O on a very large unrelated file. The latter step removes the data of the file under study from all caches (processor caches, file system buffer cache, and cache of the disk). The *hit* measurement is taken right after the miss measurement, so that data is in the cache and, in the case of mapped file I/O, in a region mapped to the server. No "warm-up" measurement is taken. The reported values for each measurement are the averages of five tries. Because some measurements had large variance, this chapter's graphs include error bars. Each such bar corresponds to the standard deviation of the corresponding measurement.

The measurements do not include the case of double caching, where the server accesses files using explicit I/O but maintains its own cache of the data. On hits, double caching is equivalent to file mapping (except on total memory consumption, which with file mapping may be only half of that of double caching). On misses, double caching is equivalent to the case where the file is accessed with explicit I/O.

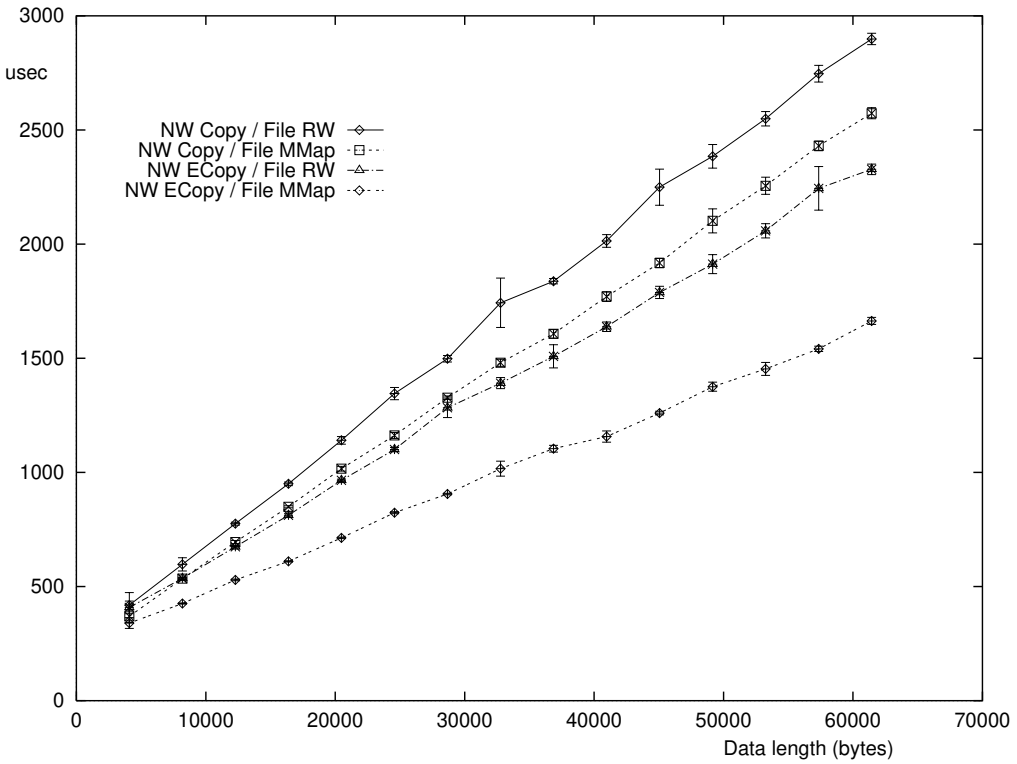


Figure 10.1: Fetch response time in the case of a hit.

10.3.2 Experimental set-up

Both parties ran on computers with the characteristics shown in Table 10.1. The operating system used was NetBSD 1.1 with the modifications described in Section 8.1. Parties were connected by the Credit Net ATM network at 512 Mbps. Unless otherwise stated, early demultiplexing was used.

Unfortunately, NetBSD 1.1 does not efficiently implement mapped file I/O. Contrary to many contemporary systems, NetBSD maintains separate pools of free pages for VM and the file system's buffer cache. Consequently, when mapping and unmapping a file, NetBSD has to copy data between pages of each pool. This inefficiency affects measurements in the miss case.

10.3.3 Response time

Figure 10.1 shows the *hit* response time for fetching data of each length using different data passing schemes. Whether `mmap` or emulated copy gives

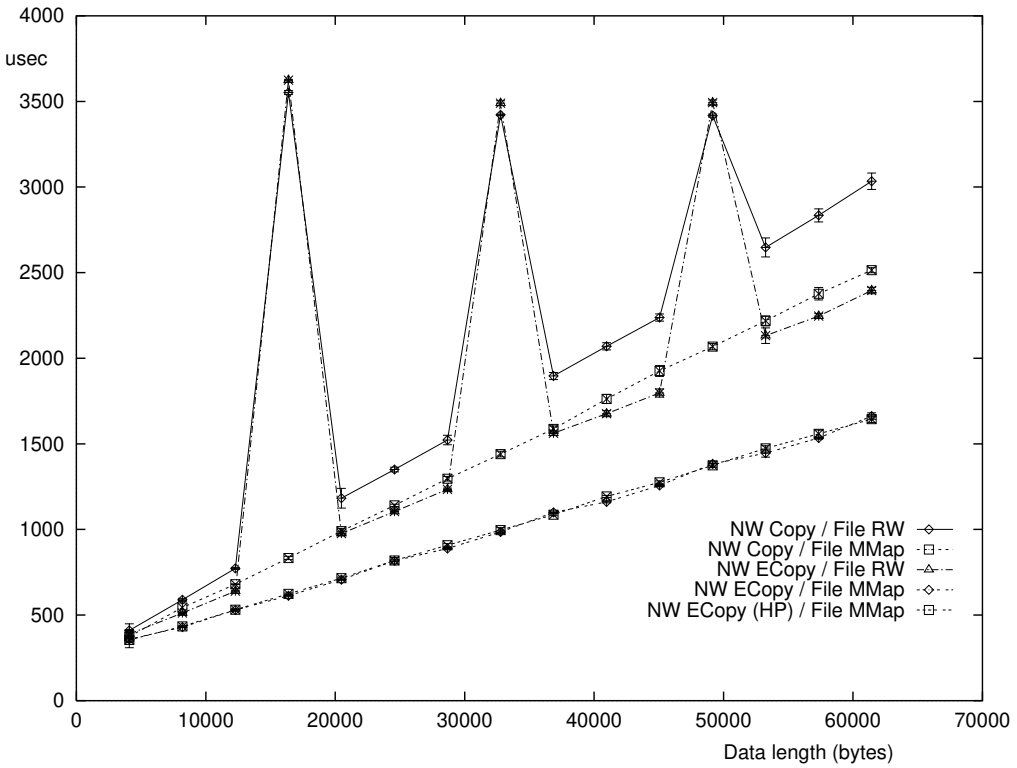


Figure 10.2: Store response time in the case of a hit.

by itself greater improvement depends on the data length. For data shorter than 8 KB, `mmap`'s fewer instructions executed in the server resulted in greater benefits than those of emulated copy. For data longer than 8 KB, however, emulated copy's avoidance of two copies (one at each party) had greater impact than `mmap`'s avoidance of only one copy (at the server). The combined effect of `mmap` and emulated copy is greater than the individual effect of each optimization. This synergy is due to cache effects. The benefits of copy avoidance are greatest when copying is avoided in the entire end-to-end data path. In this experiment, relative to data passing by copying, combined `mmap` and emulated copy reduced response time by 29% for 8 KB data and by 43% for 60 KB data.

Figure 10.2 shows the corresponding *hit* response times for storing data. The spikes that occur when the file is accessed by explicit I/O are due to NetBSD's file write policy. By default, NetBSD updates the disk synchro-

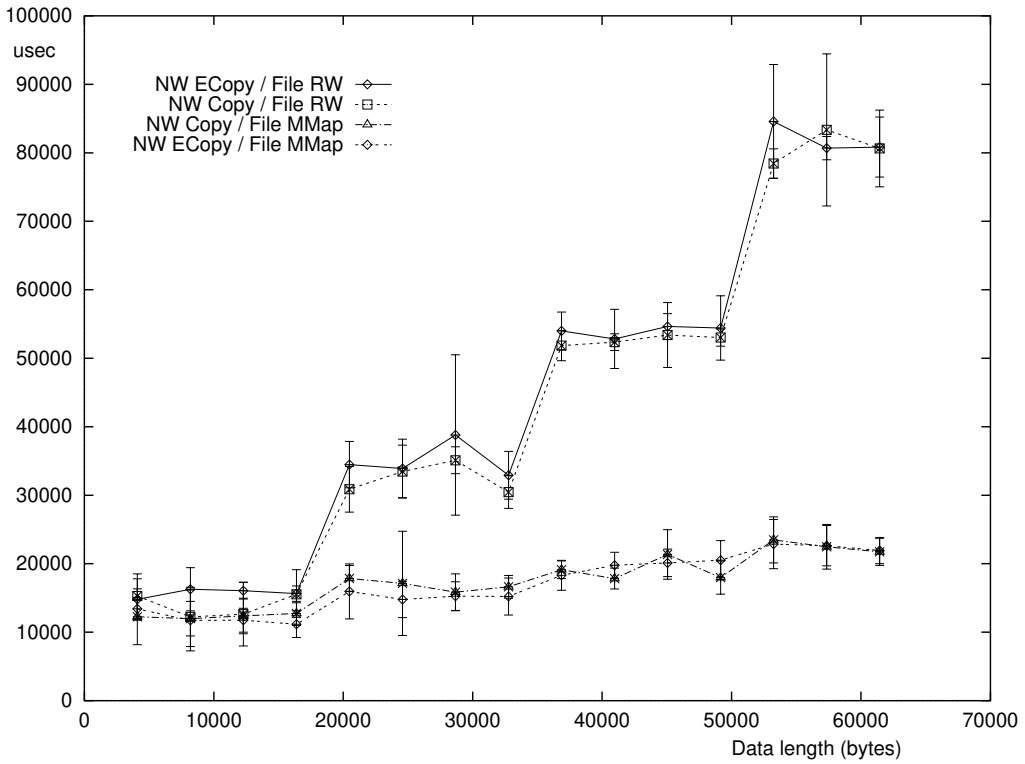


Figure 10.3: Store response time in the case of a miss.

nously[1] when data length is multiple of 16 KB. Otherwise, the effects of copy avoidance are similar to those of Figure 10.1. Relative to data passing by copying, combined `mmap` and emulated copy reduced response times by 27% for 8 KB data and by 45% for 60 KB data.

Figure 10.2 also shows the response time when header patching (HP) and pooled in-host buffering are used. The response time is essentially the same as that with early demultiplexing (the default in all figures).

The *miss* response time for fetching data is very variable because of disk latencies (especially rotational latency). The average response time measured for 4 KB data was 14 ms, with standard deviation of 4.5 ms. For 60 KB data, the average response time was 25 ms, with standard deviation of 3 ms. Response time differences among data passing schemes for network I/O and

[1]In a comparative study of the file systems of several Unix-derived systems, [40] (in its figures 6.40 and 6.41) shows that a surprising number of them have a synchronous write policy. In NetBSD, the system administrator can configure asynchronous writing when mounting the file system, but system manuals strongly discourage that option.

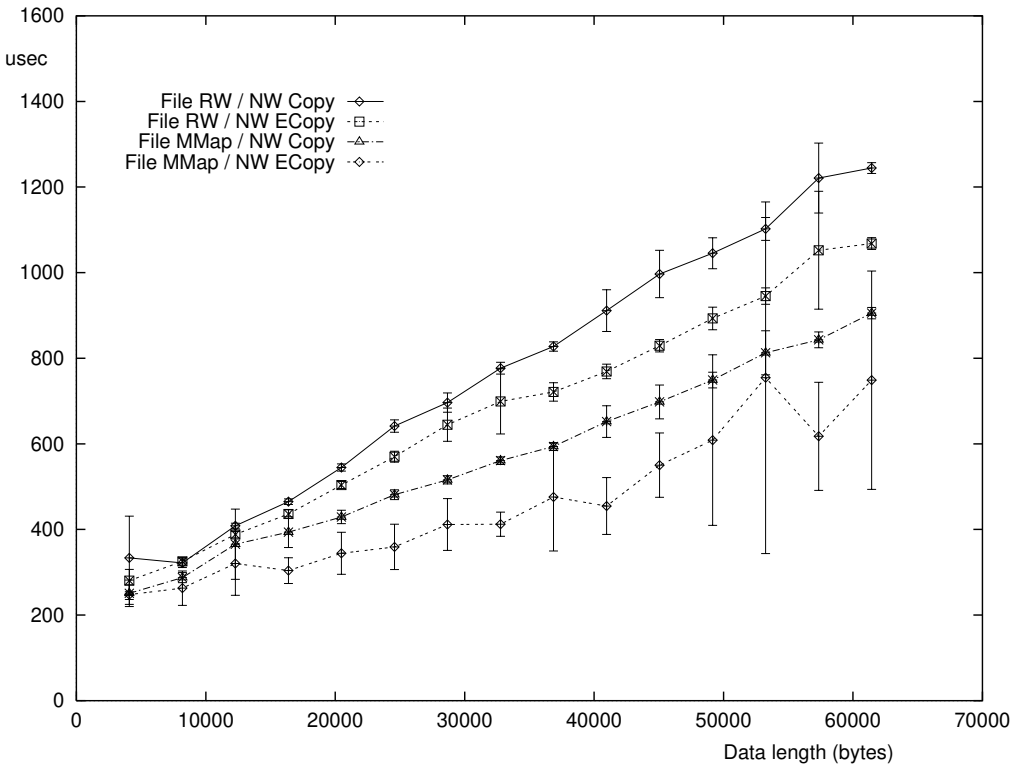


Figure 10.4: Fetch I/O processing time in case of a hit.

for file I/O were not statistically significant.

For storing data, however, the *miss* response time has a marked dependence on whether explicit I/O or mapping is used for the file, as shown in Figure 10.3. The steps that occur when the file is accessed by explicit I/O and the data has length that is a multiple of 16 KB are due to NetBSD's synchronous write policy. Differences between corresponding cases with copy or emulated copy were not statistically significant because of the large variability in disk latency.

10.3.4 I/O processing time

Figure 10.4 shows the *hit* I/O processing times measured in the server's system for fetching data of each length using different data passing schemes. `mmap` and emulated copy each avoid one data copy at the server. `mmap` has by itself a greater impact than does emulated copy because, on a hit, `mmap` also avoids executing many file system instructions. In this experiment, relative

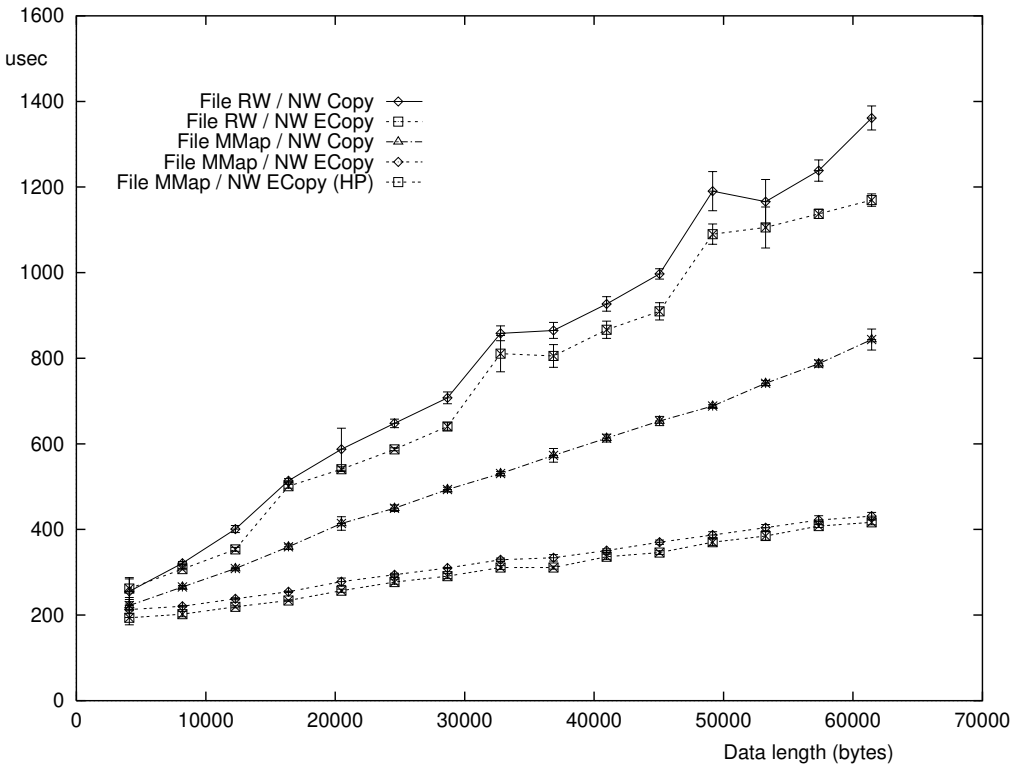


Figure 10.5: Store I/O processing time in case of a hit.

to data passing by copying, combined `mmap` and emulated copy reduced I/O processing times by 18% for 8 KB data and by 40% for 60 KB data.

Figure 10.5 shows the corresponding *hit* I/O processing times for storing data. The effects of copy avoidance are greatest when both network and file accesses are copy-free. Relative to data passing by copying, combined `mmap` and emulated copy reduced I/O processing by 31% for 8 KB data and by 68% for 60 KB data.

Figure 10.5 also show the I/O processing time when header patching (HP) and pooled in-host buffering are used, instead of early demultiplexing. Header patching reduces I/O processing time noticeably. In the early demultiplexed case, the host processor has to insert and remove special buffer segments for headers and trailers (the Credit Net 512 Mbps cards do not have automatic depadding). This overhead is avoided in the header patching/pooled case. However, header patching is not transparent to clients, while early demultiplexing can be. A higher level of adapter support (buffer

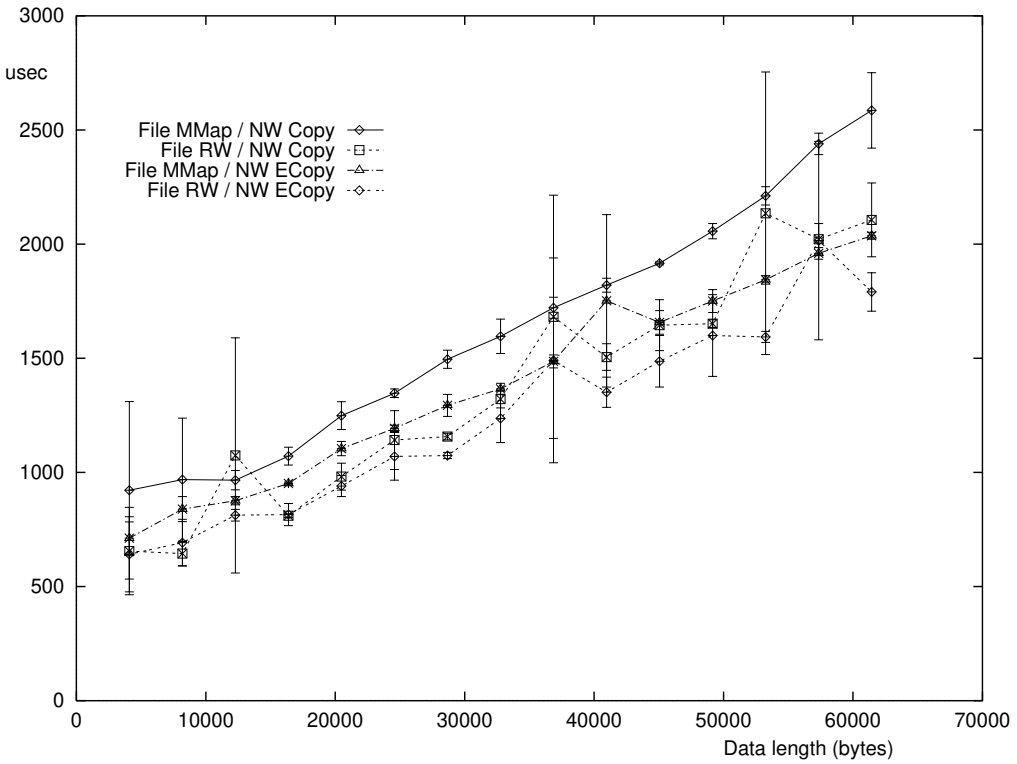


Figure 10.6: Fetch I/O processing time in case of a miss.

snap-off) should give performance similar to that of header patching and transparency greater than that of early demultiplexing.

Figure 10.6 show the I/O processing times for fetching data on a *miss*. Unfortunately, unlike many contemporary systems, NetBSD's implementation of `mmap` does not avoid copying, because NetBSD maintains separate pools of free pages for VM and the file system's buffer cache. Therefore, the full benefits of copy avoidance could not be demonstrated for this case. Relative to data passing by copying, emulated copy alone reduced I/O processing time by 15% for 60 KB data (for 8 KB data, measured differences were not statistically significant).

Figure 10.7 shows the related I/O processing times for storing data on a *miss*. Again, NetBSD's `mmap` implementation did not allow the full benefits of copy avoidance to be demonstrated. Relative to data passing by copying, emulated copy alone reduced I/O processing times for 60 KB data by 16% (for 8 KB data, differences were not statistically significant).

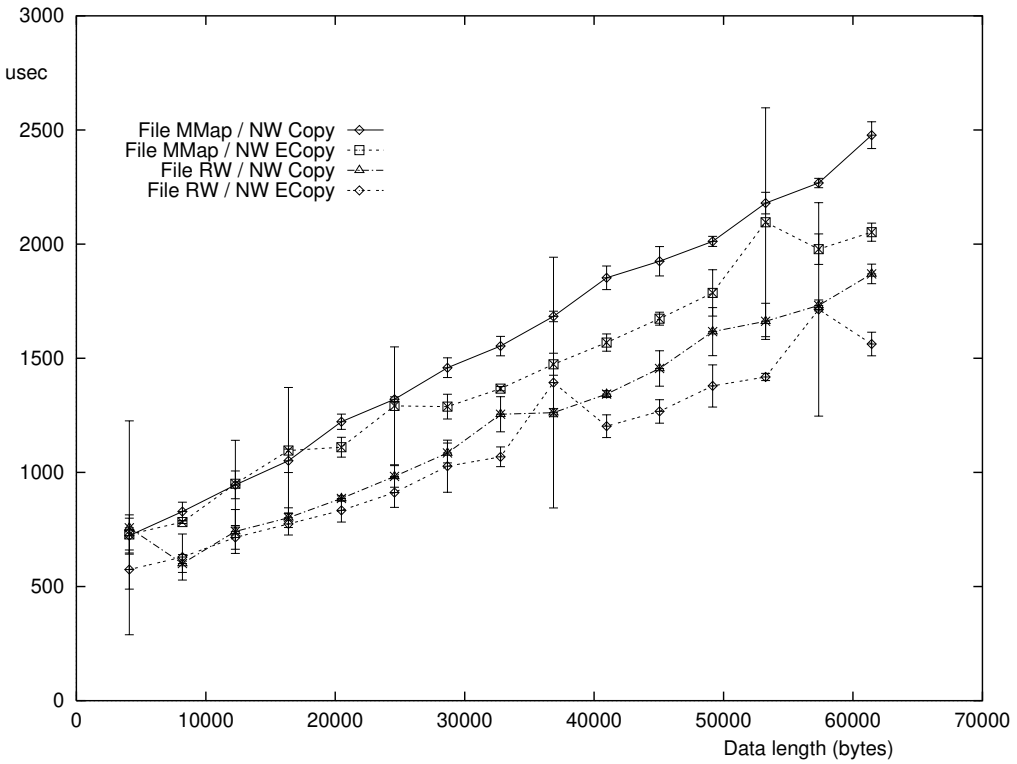


Figure 10.7: Store I/O processing time in case of a miss.

Figures 10.6 and 10.7 suggest that, in this version of NetBSD, it is more efficient to use double caching than mapped file I/O.

10.4 Related work

This dissertation's copy avoidance approach tightly integrates the I/O and VM subsystems. Applications and the network and file I/O subsystems share or exchange pages allocated from the VM pool, which can result in transparency relative to applications. IO-Lite [32] takes a different approach, where applications and the network and file I/O subsystems share read-only buffers allocated from the IO-Lite (as opposed to the VM) pool of buffers. Consequently, IO-Lite can avoid copying only for applications that use the IO-Lite API. Unlike the solution presented here, IO-Lite has to copy data between application buffers and IO-Lite buffers for applications that use a conventional explicit I/O interface with copy semantics or a mapped file I/O

interface. IO-Lite requires early demultiplexing, whereas this work's solution can provide copy avoidance even with only pooled in-host buffering.

10.5 Summary

Copy-free output of file data to a network may require a few modifications in the file system to avoid the overwriting of pages with outstanding output references. Copy-free input of file data from a network can be achieved with standard techniques in the case of network input by emulated copy with server-aligned buffering. However, server-aligned buffering requires special hardware support (early demultiplexing or buffer snap-off). If network input is by emulated copy with client-aligned buffering and pooled in-host buffering, the page alignment necessary for copy avoidance can be achieved by a software technique, *header patching*. For network input by other non-copy schemes, header patching can be used for alignment and a new technique, *user-directed page swapping*, can be used to pass data to its final destination without copying.

However, the measurements presented in this chapter show that fetch response time is much more sensitive to cache misses than to data copying. For optimization, the first priority is to improve the prefetch and cache policies, using, for example, application-controlled file caching [19], informed prefetching and caching [65], or compiler-inserted I/O prefetching [58]. After a high cache hit ratio is achieved, copy avoidance can considerably further improve the fetch response time.

Likewise, store response time is more sensitive to a synchronous write policy than to data copying. Although an asynchronous write policy is more prone to inconsistency in case of a system crash, the costs of the synchronous policy appear excessive. A reasonable compromise, used in many systems [49, 40], is to process writes asynchronously but update the disks each 30 seconds. When writes are asynchronous, copy avoidance can significantly further improve response time.

The server's I/O processing time can always benefit from copy avoidance, whether or not a miss occurs. The measurements presented in this chapter show that copy avoidance can provide improvements ranging from negligible (for misses on short data) to very significant (for hits on long data). If the server is CPU-bound, copy avoidance may significantly improve the server's throughput.

Finally, the measurements presented in this chapter also demonstrate that for greatest copy avoidance benefits, all I/O data copying should be avoided. In applications involving both files and networks, this can be achieved, without changes in the semantics of existing interfaces, by using mapped files and emulated copy.

Chapter 11

Data Passing Avoidance and Scheduling Avoidance Optimizations

Chapters 3 to 10 describe and evaluate new optimizations for copy avoidance according to either the explicit or the mapped file I/O model. Those models assume that all I/O data must be passed to and from client buffers. As explained in Section 1.2.4, many clients do not actually have such requirement. For example, some clients may only pass data from one device to another, without processing that data, or pass the same data to multiple servers. It may be more efficient for such clients to use an alternative I/O model, with *data passing avoidance*, where data passing is reduced or eliminated, rather than merely made more efficient.

In an extensible system, a client may implement such model by installing an extension and having the data be passed to and from the extension, rather than the client itself. In the case of a kernel extension, the implementation may further depart from conventional I/O models by also including *scheduling avoidance*, that is, by executing the extension at interrupt or callout level. Scheduling avoidance reduces context switching overheads, but may also disrupt system scheduling.

This chapter presents Genie's new interface, *iolets*, which offers an I/O model with data passing avoidance in systems with monolithic or microkernel structure. Iolets allow *limited hijacking*, a safe form of scheduling avoidance.

Experiments in Chapter 12 show that, for certain applications, iolets and limited hijacking can greatly improve I/O performance relative to that of a

conventional explicit I/O interface, which passes data by copying. However, the improvement is dramatically reduced if the conventional explicit I/O interface is optimized with the copy avoidance techniques of Chapters 3 to 7.

11.1 Iolets

An *iolet* is an I/O program, usually rather small, that executes at kernel level and contains multiple I/O requests. Clients specify iolets in a restricted, parametric, non-Turing-complete language with few features other than those strictly necessary for data passing avoidance and scheduling avoidance. An iolet interpreter executes an iolet by making the iolet's I/O requests as specified by the client. When the iolet completes, the iolet interpreter returns a status code to the client.

Clients can specify that an iolet be repeated a certain number of *iterations*, and that processing of the I/O requests of each iteration be:

1. *Sequential:* Next request processed after completion of the current request (useful for device-to-device data transfers, e.g., in network file servers);
2. *Periodic:* Similar to sequential, with possible delay between iterations to satisfy a specified period (useful for digital audio and video);
3. *Parallel:* Requests made in client-specified order, but completed asynchronously in arbitrary order (useful for multicast); or
4. *Selective:* Similar to parallel, but iteration completes when any of its requests completes (as in Unix `select`).

An iolet might, for example, specify that, for a certain number of iterations, periodically each $\frac{1}{30}$ second, data be input from a video digitizer card into a buffer and then output from that buffer to a video monitor and a network interface, as illustrated in Figure 11.1. The buffer used in the iolet could be multiserver or outboard (Section 1.2.4).

The interface for submitting an iolet is as follows:

```
iolet_submit(scheduling, iterations,
             actual_iterationsp,
             period,
             requests_number, requests,
             bufc, bufv, handlep)
```

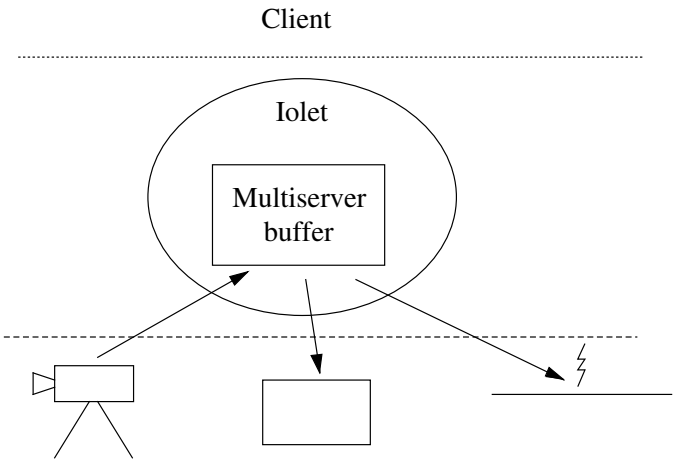


Figure 11.1: Iolet for video capture, display, and transmission.

where `scheduling` can be *sequential, periodic, parallel,* or *selective*, as explained above; `iterations` specifies how many times the iolet's requests should be repeated; `actual_iterationsp` points to number of times the requests were actually repeated (returned by the interface); `period` is the time interval between successive periodic iterations; `requests_number` is the number of requests in each iteration; `requests` specifies the requests of each iteration, in the desired order; `bufc` is the number of buffers used by the iolet; `bufv` is a vector of pointers to the descriptors of the buffers; and `handlep` points to an identifier returned by the interface. The client can use that identifier to synchronize with completion of the iolet or to abort the iolet (using `io_sync` or `io_abort`, as described in Section 7.1). A null `handlep` makes the iolet synchronous.

Requests of each iteration of an iolet are specified by a vector of structures of the following type:

```
struct io_req {server, service,
               bufc, ind_bufv,
               timeout, resultp}
```

where `server` specifies the server of the request, `service` defines the requested service, `bufc` is the number of buffers used by the request, `ind_bufv` is a vector of indices into the `bufv` vector of the `iolet_submit` call, specifying the buffers submitted for the given request, `timeout` is the time limit for

completion of the request, and `resultp` points to the result of the request in the last iteration executed (returned by the interface).

Buffer descriptors are the same as those for conventional I/O requests (Section 7.3). The client can specify, in a buffer scatter-gather element, the usual native-mode and migrant-mode data passing schemes, such as emulated copy and move. In addition to those, the *multiserver* and *outboard* data passing schemes are defined. In such cases the `location` of the scatter-gather element determines: (1) the lender of the buffer and (2) up to three integers to be interpreted by the lender, and which may specify, for example, a network connection, display window, or file. When the iolet is submitted, the interface automatically makes a buffer allocation request to the lender, and processing continues when the lender replies. Likewise, when processing of the iolet completes, the interface makes a buffer deallocation request to the lender, and processing continues when the lender replies. If the client does not specify the lender, the interface allocates and deallocates the buffer from and to the VM pool of free pages.

If any request in an iolet is to a user-level server, the selective transient mapping of the corresponding buffers occurs only in the first iteration of the iolet, and unmapping occurs when the iolet completes.

11.2 Limited hijacking

Genie's iolet interpreter executes at kernel level in the context of the client or, when processing of an I/O request completes asynchronously, at interrupt level. In the latter case, if the next request of the iolet is to a server that is integrated in the kernel, the interpreter can make that request immediately, without scheduling. Although such scheduling avoidance eliminates context switching latencies, it also implicitly assigns to the processing of the next request priority higher than that of any application processing[1]. The iolet in effect *hijacks* the CPU, which may disrupt system scheduling.

Limited hijacking prevents scheduling disruption by letting the iolet interpreter avoid scheduling only for requests that execute very fast[2].

Each service provided by an I/O server has associated *fixed* and *variable* costs. The iolet interpreter estimates the processing time of a request by

[1] Also true of processing as callout or software interrupt [36].

[2] This is consistent with scheduling policies, such as that of Unix, that assign higher priorities to more interactive processes.

adding the respective service's fixed cost and the product of the respective service's variable cost and the total length of the buffers passed in the request and respective reply. Kernel-level servers can be integrated with the kernel and have fixed and variable costs defined at system build time. After boot time, they can also be installed and have fixed and variable costs configured by the system administrator. User-level servers can't hijack the CPU, and therefore their fixed and variable costs are considered to be zero.

The iolet interpreter monitors the *accumulated processing time* of each iolet. This time is the sum of measured request processing times, reset to zero when the client process is rescheduled, there is a delay between periodic iterations, or a user-level server returns a reply to a request of the iolet. The iolet interpreter avoids scheduling if the sum of the accumulated processing time and next request's processing time estimate is less than the *hijack limit*, a configurable constant much lower than the scheduling quantum. Otherwise, if the iolet request is synchronous, the iolet interpreter wakes the client, and processing continues when the client is rescheduled; otherwise, processing continues when the client next checks the status of the iolet.

If processing of a request completes synchronously at the non-preemptive, non-interrupt-driven top half of a kernel, such as that of Unix, processing of the next request may also occur without rescheduling. In such cases, the iolet interpreter avoids scheduling if the sum of the accumulated processing time and next request's processing time estimate is less than the *I/O scheduling quota*, a configurable constant. Otherwise, the iolet interpreter yields the CPU for rescheduling.

11.3 Related work

Contrary to the ephemeral server buffers of the optimizations of Chapters 3 to 7, multiserver buffers can be either ephemeral or cached. This has been demonstrated by experiments with the *splice* interface on a network server and a display server, using ephemeral multiserver buffers [36], and on a file system, using cached multiserver buffers [35]. The splice interface can be considered to implement a special case of iolet where the number of requests is always two, the first request is always to input data, the second request is always to output data, and multiserver buffering is used.

The combination of services of multiple I/O servers using iolets is quite similar to that achieved in Scout *paths* [57]. Iolets allow, however, run-time

requests to any I/O server, whereas Scout paths are constrained to build-time *router graphs*.

IBM channel programs [42], like iolets, allow unprivileged applications to submit to the system sequences of I/O requests. However, channel programs do not support data passing avoidance. A given channel program can use only one device and always passes data between that device and client buffers. Channel programs also do not use scheduling avoidance, because they run on a dedicated I/O processor (called a *channel*).

Iolet's use of a restricted, parametric, non-Turing-complete language for special-purpose extensibility in monolithic and microkernel systems has many precedents, including packet filters [54], application-controlled file caching [19], and application-controlled VM caching [48]. A recent paper [32] calls this form of extensibility *inherently safe customization* because it rules out the potential safety problems that come with general extensibility using a Turing-complete language. Such problems include potential breach of system protection or integrity and hoarding of resources, such as CPU and memory.

SPIN [9] and VINO [69] are examples of systems that, contrary to iolets, use Turing-complete languages (Modula-3 and C++, respectively) for extensions. Compared to those systems, iolet's model of extensibility is much more restricted. Iolets allow clients to combine the services of multiple I/O servers in client-specific ways; however, kernel-level I/O servers are trusted and can only be installed by a privileged user. For more general extensions, unprivileged users can install user-level servers. The iolet extensibility model is similar to that of Unix shells, which allow combining the functionality of multiple programs. SPIN and VINO, on the contrary, allow unprivileged users to install general application-specific programs as untrusted kernel extensions. Such generality does come at a price, however: To make execution of untrusted extensions safe, SPIN requires extensions to be written in a type-safe language, and VINO has to encapsulate extensions for software fault isolation. These requirements have been reported to make code from 10% to 150% slower [70][3].

Because iolets are very compact and, given their special purpose, can be efficiently interpreted, iolet submission is much simpler than the installation of a SPIN or VINO extension. The latter have to be compiled, checked, or sandboxed, dynamically linked, and downloaded into the kernel for installa-

[3]Interpreted Java was found to make the same code from 13 to 113 times slower [70]. Compiled Java should have much less overhead, however.

tion, and unlinked and deallocated for tearing down.

I/O interfaces that allow clients to allocate multiserver or outboard buffers must prevent clients from hoarding such buffers. Iolets and *splice* [36] solve this problem by not allowing clients to hold such buffers outside the scope of the call that uses those buffers. The iolet interface allocates multiserver or outboard buffers when the client submits an iolet. When that iolet completes, the iolet interface automatically deallocates the respective buffers. Container shipping [64] and SPIN [9], on the contrary, allow untrusted user- or kernel-level clients to hold multiserver buffers outside the scope of a call. The referenced papers do not clarify whether or how those systems would prevent multiserver buffer depletion. VINO uses a transactional facility that allows it to arbitrarily abort extensions and free any resources that the extension may have acquired while processing. Such facility could be used to deal with extensions that appear to be hoarding a multiserver buffer. However, the overhead of the transactional facility is high [69].

Scheduling disruptions due to interrupt-level processing have been reported in several recent papers [56, 31, 57]. LRP [31] and Scout [57] demonstrate substantial throughput improvements under high receiver load by executing in the context of scheduled processes code that in the BSD Unix protocol stack is interrupt-driven.

11.4 Summary

The iolet interface implements an alternative I/O model with data passing avoidance and scheduling avoidance. An iolet is a program that consists of I/O requests. Clients can drop iolets into the kernel using the iolet interface. Iolets can reduce data passing overheads for applications that do not require access to I/O data or that output the same data to multiple servers. Iolets can also reduce control passing overheads, because control does not return to the client at completion of each request. Iolets are specified in a parametric, non-Turing-complete language that is inherently safe and can be used for this restricted form of extensibility in monolithic and microkernel systems.

Limited hijacking makes scheduling avoidance safe by limiting the amount of time that an iolet may execute before rescheduling. Before making a request, the iolet interpreter estimates, based on data length, the processing time of the request. If the sum of the actual execution time since rescheduling plus the estimate is too high, the iolet interpreter yields the CPU.

Chapter 12

Evaluation of Iolets and Limited Hijacking

This chapter experimentally evaluates data passing avoidance and scheduling avoidance in: (1) device-to-device data transfers (Figures 1.7); and (2) multicasting of the same client data to multiple devices (Figure 1.8).

Data passing avoidance and scheduling avoidance have been experimentally evaluated before. Previous works have reported large reduction in data passing overheads [35, 36, 64, 9, 77]. Evaluations involving scheduling avoidance have also suggested [35] or reported [77] reduction in control passing overheads. However, those previous comparisons were biased against conventional I/O models because the latter were not optimized. The experiments in Chapters 8 and 9 and Section 10.3 demonstrate that the performance of conventional I/O models can be greatly optimized by copy avoidance. However, before this work, comparisons between data passing and scheduling avoidance, on the one hand, and copy avoidance, on the other, had been missing in the literature. This chapter provides such comparisons.

12.1 Experimental set-up

Experiments were performed on a Micron P166 personal computer. The operating system used was NetBSD 1.1. A new system call was added to support Genie's iolet interface, which is described in the previous chapter. Measurements were performed according to the methodology described in Section 8.1. Similar variances were observed.

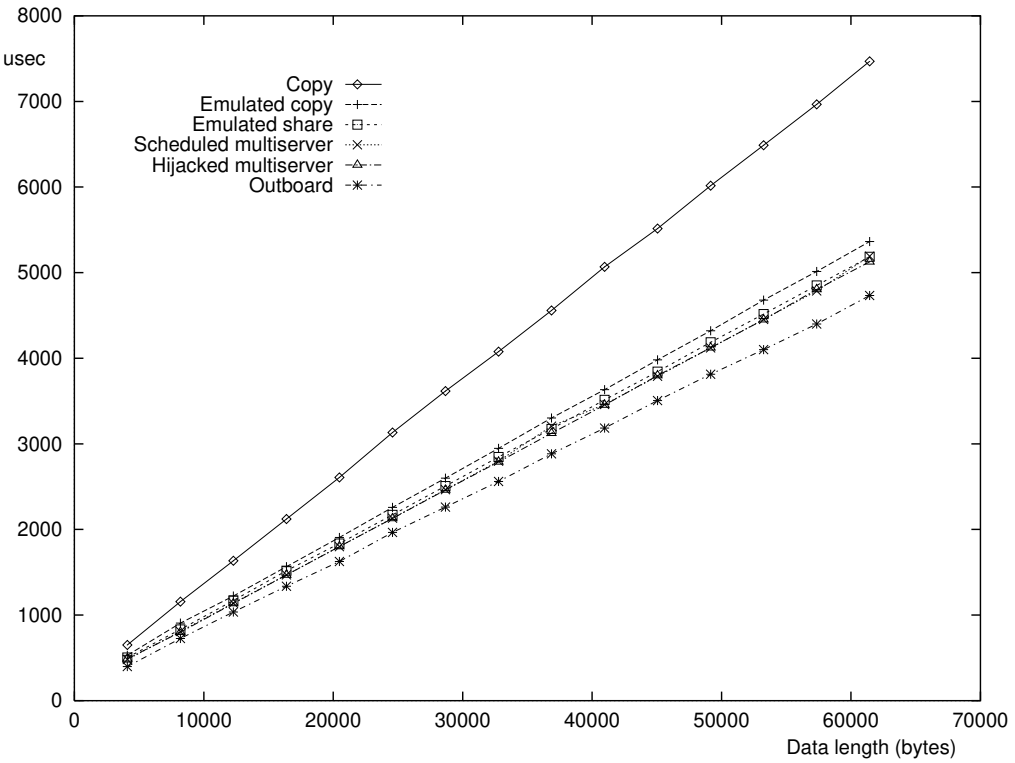


Figure 12.1: Total latency of data transfer from Cyclone board to ATM network.

The device-to-device experiments input data from a Cyclone board and output the same data to a Credit Net ATM adapter at 155 Mbps. The multicast experiments output the same client data to the Cyclone board and to an ATM adapter, in that order. The Cyclone board contains an Intel 960 processor, 2 MB of memory, and a bridge with DMA engine between the board's bus and the host's PCI bus. The Credit Net ATM adapter is described in Section 8.1.

12.2 Device-to-device latency

12.2.1 Measurements

Figure 12.1 shows the total latency for transfers of data with length equal to increasing multiples of the page size from the Cyclone board to the ATM net-

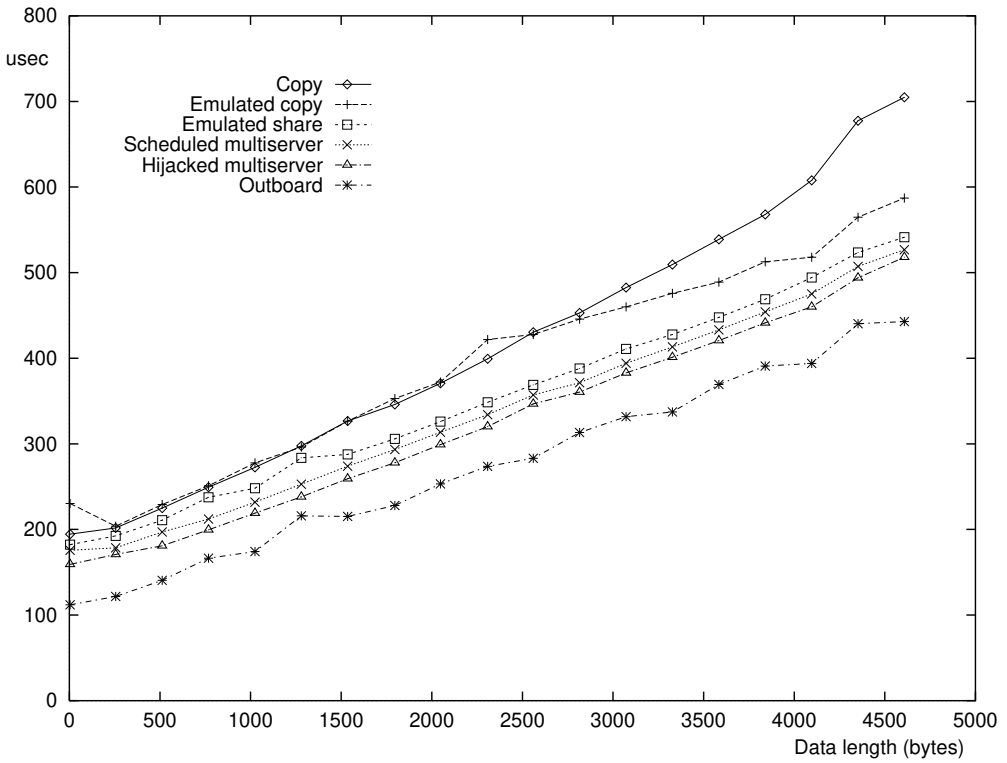


Figure 12.2: Total latency of short data transfer from Cyclone board to ATM network on otherwise idle system.

work (as shown in Figure 1.7) as AAL5 packets, without checksumming. For the copy, emulated copy, and emulated share schemes, separate synchronous input and output requests were used. For the multiserver scheme, single synchronous sequential iolets were used, each with an input request followed by an output request. The hijack limit was manipulated to achieve scheduled or hijacked processing. For the outboard scheme, synchronous output requests were used, specifying the driver of the Cyclone board as the outboard buffer lender. For 60 KB data, relative to copy, emulated copy reduced latency by 28%, and both emulated share and multiserver schemes reduced latency by 31%. The outboard scheme provided modest additional latency reduction.

Figure 12.2 shows the total latency of short data transfers between the same devices as in Figure 12.1. The fairly constant and small difference between latencies of the multiserver scheme with scheduled or hijacked processing corresponds to the cost of rescheduling (16 μs). NetBSD avoids con-

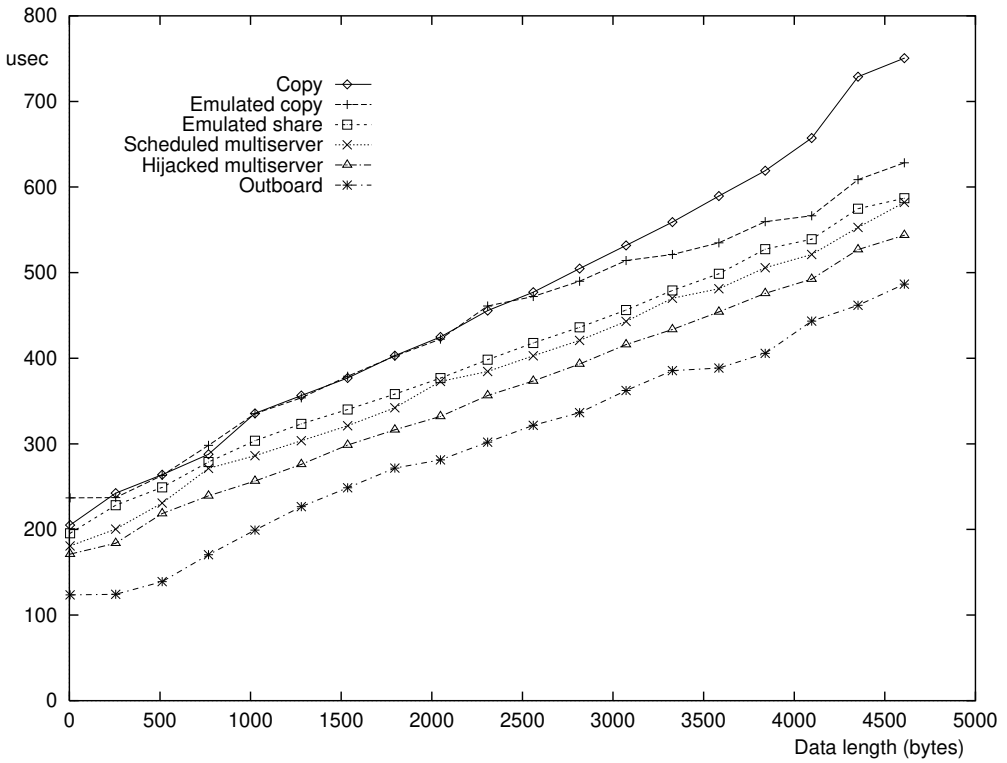


Figure 12.3: Total latency of short data transfer from Cyclone board to ATM network with ten concurrent compute-bound processes.

text switching when the same process is running before and after rescheduling, even after an idle period (guaranteed in Figure 12.2 because there were no other runnable processes). Latencies were slightly higher with emulated share than with scheduled multiserver because of an extra system call and page referencing/unreferencing costs in the emulated share case. Latencies were considerably lower with emulated copy than with copy for data longer than about half a page because of Genie's *reverse copyout* optimization (Section 4.1).

The experiments of Figure 12.2 were repeated with the host running ten concurrent compute-bound processes. When returning to user mode after an interrupt, exception, or system call, NetBSD reschedules the CPU if a higher-priority process is runnable. Compute-bound processes in NetBSD and other Unix-derived systems attain low scheduling priority as they accumulate running time, whereas I/O-bound processes, such as the device-

to-device benchmark used here, receive high priority. The device-to-device latency measurements, therefore, should be affected by only modest context switching overheads. This was not true when the experiments were first repeated because NetBSD did not guarantee strict priority scheduling. Rescheduling leaves interrupts enabled, after which events may make another higher-priority process runnable *before* the CPU returns to user mode. In such cases, the current lower priority process may be allowed to run until the next timer interrupt (spurious latencies of up to 10 ms were observed). This priority inversion was eliminated by, after rescheduling and immediately before returning to user mode, disabling interrupts and again checking if rescheduling is needed; if not, interrupts are atomically reenabled as part of the return from interrupt instruction. This modification guarantees that whenever the processor is in user mode, it is running the highest-priority process. Figure 12.3 shows the latencies after the modification. The differences between Figures 12.2 and 12.3 are quite small. Each rescheduling plus context switching cost on average 28 μs.

12.2.2 Analysis

The code that is executed in the experiments reported in Figure 12.1 was instrumented to measure the latencies spent in each section of the code. Table 12.1 shows the least-squares linear fit of latencies that correlate well with data length and the averages of nearly-constant latencies.

Table 12.2 compares, for each data passing scheme, the total latency estimated by adding the corresponding breakdown latencies, shown in Table 12.1, with the least-squares linear fit of the actual total latencies reported in Figure 12.1. The good agreement between predicted and actual total latencies suggests that the latency breakdown model is quite accurate for the data lengths considered, which are multiples of the page length.

The enter and leave latencies for iolets (c, d) were higher than those for simple requests and replies (a, b) because of the larger number of parameters of iolets. The delay between sequential requests in iolets (e, which includes limited hijacking calculations) was, however, quite low.

Table 12.1 reveals why total latencies for long data were so similar for emulated share, emulated copy, and hijacked or scheduled multiserver schemes: All breakdown latencies other than copyin (p) and copyout (q) were strongly dominated by device latencies (j and l).

In principle, ATM output from the Cyclone board (m) could be as efficient

as ATM output from host memory (l), and latency with outboard buffering should be much lower than with any other data passing scheme (by at least the cost of input from Cyclone board to host memory (j)). In practice, however, m will depend on the performance of the bridge between outboard memory and I/O bus. In the case of the experiments, the performance of the bridge was clearly less than optimal.

The low overhead of rescheduling (g) or rescheduling plus context switching (h), relative to total latencies, explains why, even for short data, hijacking provided little latency improvement.

The unspectacular benefit of hijacking found here contradicts the dramatic latency reductions reported for ASHs [34, 77]. The source of this discrepancy is that ASHs run on Aegis, which has a round-robin scheduler, whereas the measurements presented here were on NetBSD, with a debugged implementation of the multiple-priority scheduling policy common to most Unix-derived systems.

12.3 Device-to-device throughput

The scheduler idle loop was instrumented to determine how long the processor was idle during the measurements of Figure 12.1. Subtracting idle times from total latencies, the I/O processing times ($t_{I/O}$) shown in Figure 12.4 were obtained.

I/O processing times (Figure 12.4) were considerably more sensitive to data passing scheme than were total latencies (Figures 12.1) because device latencies contribute to the latter but not to the former.

Referring to the analysis in Section 8.3, Figure 12.4 suggests that, for data passing schemes other than copy, even client processing no more complex than copying the data may give $t_C \gg t_{I/O}$ and therefore give the same θ for all schemes.

Assuming, on the contrary, $t_C \ll t_{I/O}$ (e.g., network file server), Table 12.3 indicates the minimum value of θ_{PHYS} ($L/t_{I/O}$) necessary to saturate the CPU.

The bandwidth of the OC-3 ATM card limits θ_{PHYS} to about 135 Mbps. According to the values in Table 12.3, such θ_{PHYS} will not saturate the CPU for any data passing scheme other than copy. Therefore, θ will be 135 Mbps for all data passing schemes other than copy, but only 104 Mbps for copy and $L = 4$ KB.

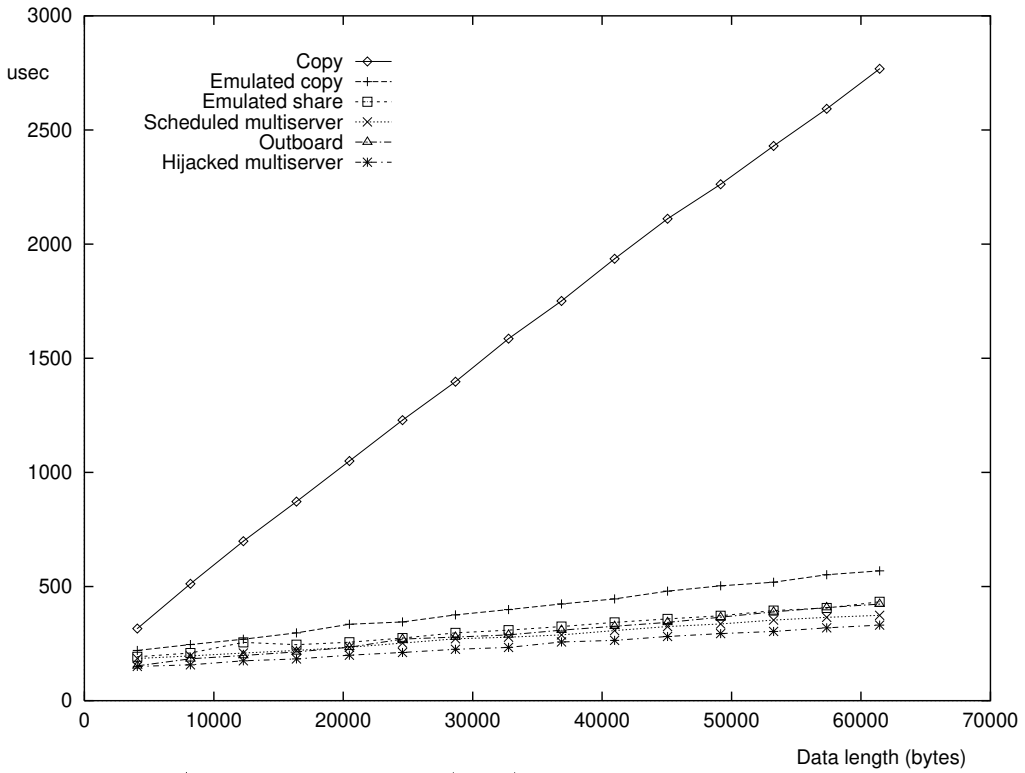


Figure 12.4: I/O processing time ($t_{I/O}$) of data transfer from Cyclone board to ATM network.

Table 12.3 shows that higher-bandwidth (perhaps parallel) devices and higher L can considerably increase θ, especially with data passing schemes other than copy, up to the PCI bus limit, 800 Mbps. For $L = 4$ KB, replacement of copy with emulated share, which is possible in conventional programming interfaces, can, using faster devices, improve θ by up to 64%; the hijacked multiserver scheme, which requires introduction of a new programming interface, can further improve θ by up to another 27%. For $L =$ 60 KB, however, θ is insensitive to data passing scheme, as long as the latter is not copy, because of the upper bound set by the I/O bus.

12.4 Multicast latency and throughput

Figure 12.5 shows the total latency for output of the same data (as shown in Figure 1.8) to the Cyclone board and to the ATM network, in that order.

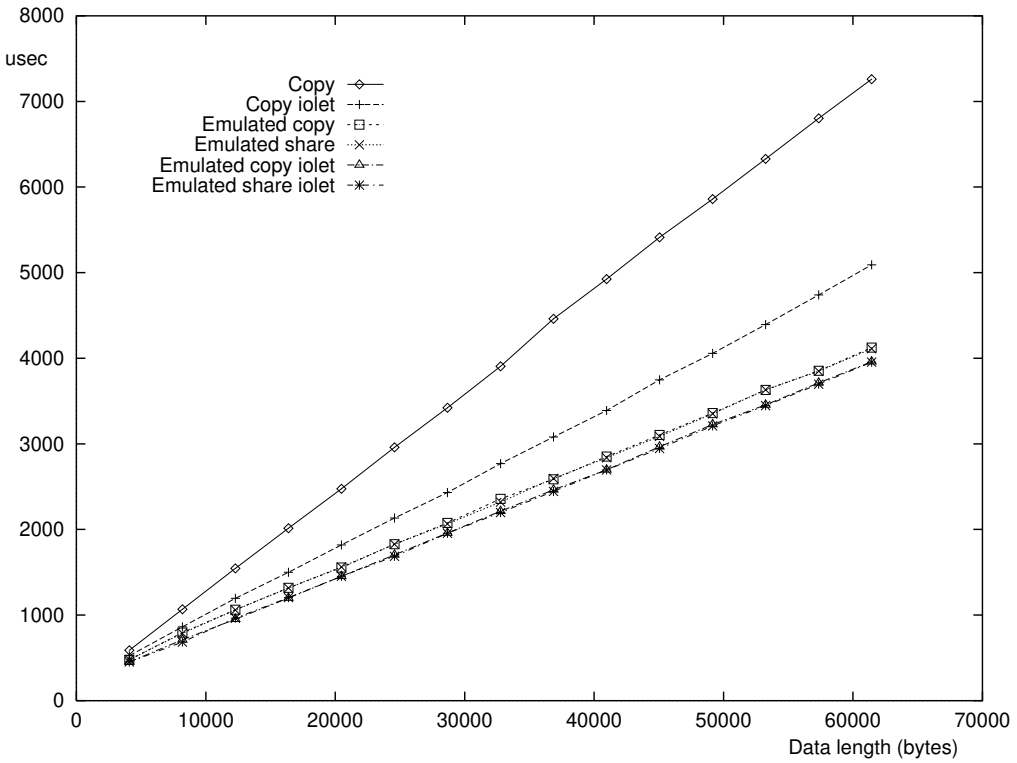


Figure 12.5: Total latency to output the same client data to the Cyclone board and to the ATM network.

Copy avoidance schemes used separate asynchronous output requests to each driver. Data passing avoidance schemes used single synchronous parallel iolets with one output request to each driver. Latencies differed little between emulated copy and emulated share because of an optimization: Genie avoids making output pages that are already read-only again read-only (Table 7.1). This optimization is normally enabled but had been disabled in the experiments of the previous chapters so as to evaluate emulated copy output under worst case conditions. In the experiment of Figure 12.5, measurements were made for output of increasing lengths of the same buffer. Given that lengths were multiple of the page size, each successive measurement required making only one additional page read-only. In this experiment, relative to copy, copy avoidance alone (emulated copy) reduced latency by 43%, more than did data passing avoidance alone (copy iolet), 30%. Combined copy avoidance and data passing avoidance (emulated copy iolet) reduced latency by 46%.

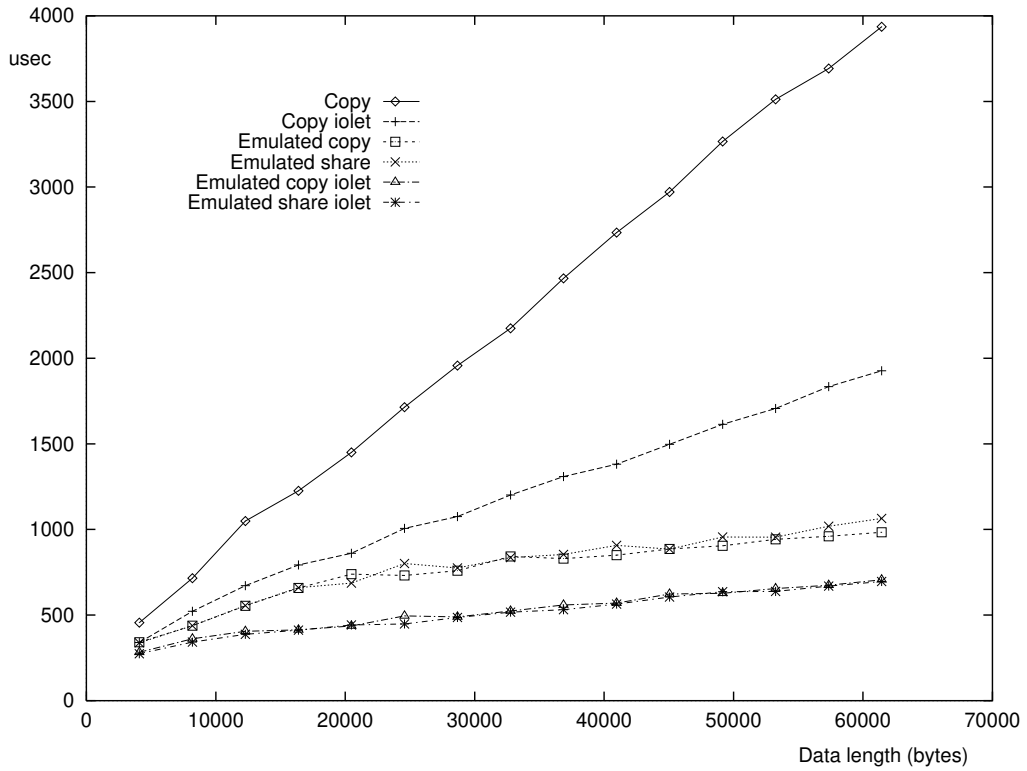


Figure 12.6: I/O processing time ($t_{I/O}$) to output the same client data to Cyclone board and ATM network.

Subtracting idle times from the latencies of Figure 12.5, the I/O processing times ($t_{I/O}$) shown in Figure 12.6 were obtained. For long data, copy avoidance alone (emulated copy) reduced $t_{I/O}$ (improving throughput) much more than did data passing avoidance alone (copy iolet), and copy avoidance and data passing avoidance combined synergistically in emulated copy iolet.

12.5 Summary

The main observations from this section's experiments are:

- In device-to-device data transfers:
  - Emulated copy and emulated share can greatly reduce data passing overheads without changing the programming interface.

  - Multiserver and outboard schemes can give smaller further improvements and require change in programming interface.
  - Performance differences among copy avoidance and data passing avoidance schemes tend to be obscured by client processing or saturation of physical I/O subsystem.

- In multicast:
  - Conventional requests with emulated copy or emulated share perform better than iolets with data passing by copying.
  - Copy avoidance and data passing avoidance combine synergistically.

- Limited hijacking makes scheduling avoidance safe, but benefits can be modest if the scheduler correctly enforces priorities.

a	Request enter	11
b	Reply leave	11
c	Iolet enter	20
d	Iolet leave	13
e	Iolet sequence	3
f	Multiserver buffer turn-around	1
g	Reschedule	16
h	Reschedule and context switch	28
i	Cyclone driver - input	$0.000156\ B + 9$
j	Cyclone board $\rightarrow$ host memory	$0.0211\ B + 13$
k	ATM driver - output	$0.000520\ B + 30$
l	Host memory $\rightarrow$ ATM network	$0.0590\ B + 17$
m	Cyclone board $\rightarrow$ ATM network	$0.0746\ B + 23$
n	System buffer allocate	$0.000162\ B + 5$
o	System buffer deallocate	$0.000125\ B + 5$
p	Allocate and copyin	$0.0168\ B$ - 5
q	Copyout and deallocate	$0.0205\ B + 15$
r	Aligned buffer allocate	$0.000226\ B + 4$
s	Swap and deallocate	$0.00222\ B + 19$
t	Reference	$0.000424\ B + 10$
u	Unreference	$0.000111\ B + 4$
v	Reference and read-only	$0.000808\ B + 13$
w	Outboard allocate	$0.0000868\ B + 8$
x	Outboard deallocate	$0.0000481\ B + 8$
y	Prepare descriptors	4

Table 12.1: Latency breakdown for device-to-device data transfers, in μs. B is the data length in bytes.

Scheme	Model	Estimated	Actual
Copy	$2*(a+g+b)+i+j+$ $+k+l+y+n+q+p+o$	$0.118B + 169$	$0.119\ B + 187$
Emulated copy	$2*(a+g+b)+i+j+$ $+k+l+y+r+s+v+u$	$0.0841\ B + 189$	$0.0841\ B + 193$
Emulated share	$2*(a+g+b)+i+j+$ $+k+l+2*(t+u)$	$0.0818\ B+ 173$	$0.0817\ B + 167$
Scheduled multiserver	$c+2*g+d+2*e+f+$ $+i+j+k+l+y+n+o$	$0.0811\ B + 155$	$0.0814\ B + 138$
Hijacked multiserver	$c+g+d+2*e+f+i+$ $+j+k+l+y+n+o$	$0.0811B + 139$	$0.0811\ B + 140$
Outboard	$a+g+b+k+m+w+x$	$0.0753\ B + 107$	$0.0753\ B+ 103$

Table 12.2: Estimated and actual total latencies for device-to-device data transfers, in μs, on an otherwise idle host. If other processes are active, h should replace g. B is the data length in bytes.

	4 KB		60 KB	
Scheme	$t_{I/O}$	$L/t_{I/O}$	$t_{I/O}$	$L/t_{I/O}$
Copy	316	104	2770	178
Emulated copy	220	149	569	864
Emulated share	192	171	433	1140
Scheduled multiserver	185	177	374	1310
Hijacked multiserver	150	218	331	1490
Outboard	153	214	423	1160

Table 12.3: I/O processing time ($t_{I/O}$, in μs) and throughput for CPU saturation ($L/t_{I/O}$, in Mbps) in device-to-device data transfers.

Chapter 13

Conclusions

Operating systems often copy I/O data between local clients and servers. Data copying is convenient because it decouples the parties and allows devices to have simplified buffering. However, the lagging improvements in the performance of memory relative to the performance of processors and many devices make copying highly and increasingly undesirable.

There have been numerous previous proposals for copy avoidance or even data passing avoidance. However, most such proposals have been incompatible with existing applications and systems. Optimizations that preserve compatibility, COW and page swapping, have long been known, but they have often been thought more restrictive than they need be [27, 29, 23, 60].

Page swapping, in particular, has typically been underutilized or wholly neglected. This dissertation reveals several more general or new ways to exploit it: input of data of arbitrary alignment and length, IPC, interoperation of explicit and mapped file I/O. Page swapping is arguably as fundamental a VM technique as is COW. In fact, they are dual: COW passes output request data, while page swapping passes input reply data. Both COW and page swapping pass data to or from client buffers with copy semantics (when page swapping is coupled with input alignment) and to or from server buffers with migrant semantics.

Page swapping has not been used more widely perhaps because the tools necessary for understanding it were missing. A major contribution of this dissertation is establishing systematic models of how I/O can be organized, how data and control passing overheads can be optimized, and how I/O data can be passed. These models conceptualize previous and new techniques in an innovative way, emphasizing structure and aspects relevant to compat-

ibility, such as programming interface, integrity, restrictiveness, criticality, symmetry, and protection, instead of implementation details.

These concepts provide the substrate on which the dissertation's main theme is developed: the integration of the I/O and VM subsystems for copy avoidance. Integration has long been available for storage I/O, in the form of mapped files. This work broadens integration to the case of network I/O using conventional explicit I/O interfaces with copy semantics. The resulting solution allows copy-free I/O both for storage and network I/O while preserving compatibility with existing interfaces and systems.

This dissertation describes and evaluates new optimizations for each data passing semantics and for IPC, network adapter support, data passing avoidance, and scheduling avoidance. In particular, two new data passing schemes for network I/O are introduced: *emulated copy*, for data passing with copy avoidance but preserving copy semantics between user-level client and system buffers, and *I/O-oriented IPC*, for efficient data passing between system and user-level server buffers.

Several previous works proposed integrating application and system buffers of the network I/O subsystem [29, 12] and, more recently, also those of the storage I/O subsystem [44, 32]. The advantages of the VM-based approach presented here are: (1) data passing semantics compatible with that of existing interfaces, and (2) optimization conditions that are advisory only and whose restrictiveness decreases according to the hardware support available. At one extreme of hardware support, pooled in-host buffering, some applications not aware of optimizations may already satisfy conditions and benefit from copy avoidance; at the other extreme, outboard buffering, all applications benefit from copy avoidance. In contrast, solutions that incompatibly change data passing semantics cannot without copying support previously written applications, regardless of hardware support available.

The experiments reported here cover a broad range of data and control passing optimizations for network I/O, perhaps the broadest in a single study. The pattern that emerges with remarkable regularity from the experimental results is that end-to-end performance is dominated by network bandwidth and latency and presence or absence of data copying at the end systems. Data passing semantics and whether protocol servers are implemented at kernel or user level have only secondary importance, when emulated copy and I/O-oriented IPC are used.

Considering the many and sometimes radically different optimizations previously proposed for alleviating data and control passing overheads in

network I/O, the similarity of their performance improvements can cause surprise. However, by and large such optimizations had been previously compared only to conventional data passing by copying and were never pitted against each other.

In hindsight, the dominating effects of physical network constraints and of copying should actually be expected. Some data passing semantics and operating system structures may indeed allow lower data and control passing overheads. However, the performance of both networks and memory have long been improving more slowly than that of processors. If copying or data passing is avoided, network performance can easily dominate I/O processing, including data and control passing.

The main conclusion is that optimizations that preserve data passing semantics and system structure can give end-to-end improvements almost as good as or, in some cases, better than those of data and control passing optimizations enabled by changes in data passing semantics or system structure. Additionally, scaling measurements suggest that differences in end-to-end performance using various data and control passing optimizations tend to decline, given current technological trends.

These results clearly confirm the thesis. Emulated copy can provide large network I/O performance improvements while preserving copy semantics and its programming interface. I/O-oriented IPC gives user-level protocol servers performance approaching that of kernel-level ones without changing operating system structure.

Emulated copy and I/O-oriented IPC establish a new baseline for judging future network I/O optimizations. Proposals for changing data passing semantics or operating system structure are not fairly supported by comparisons only with unoptimized implementations of conventional I/O interfaces: Emulated copy and I/O-oriented IPC can greatly improve the latter without changing semantics or structure.

The experimental results and analysis presented here suggest that the new baseline set by emulated copy and I/O-oriented IPC will be more difficult to significantly outperform than is the conventional one, set by interfaces that pass data by copying. Rather than simply reduce data and control passing overheads, new optimizations will possibly have to improve the physical I/O subsystem or find new ways to use the latter for specific applications or classes of applications (e.g., specialized protocol implementations or disk layouts, better prefetch and cache policies, application adaptation to available bandwidth, or trading bandwidth for latency).

The following sections make recommendations based on this dissertation's findings and point to relevant future work.

13.1 Recommendations

The results of Chapter 8 recommend the adoption of emulated copy in systems that have an explicit I/O interface with copy semantics, such as Unix and Windows NT. Emulated copy can significantly improve network end-to-end performance and is compatible with existing interfaces, such as sockets. The results suggest that little further improvement would be possible by radically changing the data passing semantics of existing interfaces.

For full emulated copy support, four additional calls would be useful:

1. Configure input and output thresholds;
2. Inform preferred alignment and length for client input buffers;
3. Specify variable lengths of application-level headers and trailers; and
4. Synchronously flush previous output requests.

The second call supports client-aligned buffering. The third call supports server-aligned buffering when application-level headers or trailers have variable length. In such cases, the application receives headers and trailers out-of-band. This call may not be necessary if headers and trailers have fixed length, since then the application can simply intersperse segments for headers and trailers in the input request's scatter list (e.g., using Unix's `readv`). It may also not be necessary if headers are patched (Section 6.3). The fourth call supports output buffer reuse by the client (the client can also avoid TCOW faults by using circular buffers with appropriately set socket window sizes, as discussed in Section 4.3). In Unix, these calls can be implemented as new `ioctl` requests.

Emulated copy requires a few simple VM modifications, as explained in Chapters 3, 4, and 10 and Section 7.4. In the socket framework, emulated copy would also require the following new interfaces in socket and protocol layers [49], in addition to those necessary for the above new calls:

1. Request data input into given buffers; and
2. Abort a previous request for data input into given buffers.

These interfaces support server-aligned buffering.

Chapter 8 also shows that in some cases, e.g. input of data of length around half a page using customized system buffers, emulated share can be considerably more efficient than emulated copy. It is therefore advisable to support both emulated copy (by default, for compatibility) and emulated share (for best performance). Emulated share can use the same programming interface as that of emulated copy. Output that uses any of the techniques explained in Section 4.3 to avoid TCOW faults will also work correctly, without modifications, under emulated share. To avoid corruption of important data on input with emulated share, applications may have to rearrange their buffers. Alternatively, applications may checkpoint their data periodically and roll back in the unlikely case of input failure. Emulated share requires, in addition to the modifications for emulated copy, two new calls:

1. Switch data passing to emulated share; and

2. `mswap` (Section 10.1).

The first call can be implemented as an `ioctl` request. The second call supports efficient interoperation with mapped files.

For application programmers, the recommendations are to use client-aligned buffering and emulated share whenever possible in new applications. Client-aligned buffering allows good performance even without special network adapter support. I/O with emulated share, if supported, gives the best performance, and, if not supported, can be transparently converted into emulated copy. Programmers should also strive to issue input requests (perhaps asynchronously) ahead of the corresponding physical input. This allows server-aligned or in-place input.

Similar recommendations apply for maintainers of existing applications. Although some applications may already post input requests early enough for server-aligned input or unwittingly use client-aligned buffers, other applications may benefit from changes to achieve copy avoidance. The changes need not be complex, and can include, for example, anticipating input requests, aligning client buffers, or flushing previous output requests before reusing output buffers. The changes are strictly for performance, however; applications will also work correctly without any changes.

The results of Chapter 9 suggest that protocols can be implemented at user level without major performance degradation, using I/O-oriented IPC.

User-level servers would make protocol and driver development and maintenance much easier than they currently are in systems such as Unix and Windows NT.

For copy avoidance, it is desirable that new protocols be designed with fixed-size headers and trailers. If variable-size headers are unavoidable, the results of Section 10.3 suggest the use of header patching. Header patching at one protocol layer is transparent to layers above and below that layer. Because header patching allows the preferred alignment to be fixed, it can simplify the network adapter support necessary for copy avoidance.

For designers of network adapters, Chapter 6 recommends support for buffer snap-off, which allows copy avoidance under very general conditions. In the case of fixed-size headers, the implementation of buffer snap-off is particularly simple. The benefits of buffer snap-off should be particularly visible in Gigabit Ethernet adapters. The high data rates of Gigabit Ethernet make data copying undesirable; however, because Ethernet packets are smaller than a page, the concatenation of multiple packets is essential to allow copy avoidance by page swapping. When buffer snap-off or early demultiplexing is supported, other important recommendations are: (1) allow specific input buffers to be reclaimed, and (2) use pooled in-host buffers when the data buffer list of a reception port is empty (rather than drop the packet). In adapters for Gigabit Ethernet, ATM , and other high-speed networks, support for checksumming and encryption may significantly improve performance.

For processor designers, this work recommends support for unmapped access (Section 3.1.1), tagged TLB entries, and, in case of virtually addressed caches, tagged cache entries. The latter two optimizations avoid abrupt loss of locality on context switches. It is also important to insure that the costs of TLB updates in new processors and in multiprocessors scale well.

13.2 Future work

It would be interesting to determine experimentally, on various typical workloads, what percentage of existing applications unwittingly satisfy the conditions for copy avoidance with server- or client-aligned buffering and TCOW.

A comparison between emulated copy and emulated share in architectures radically different from those of the computers used in this dissertation could unveil interesting new results. In multiprocessors, for example, the cost of

TLB updates can be much higher than in a single processor. In such architectures, the advantage of emulated share may be higher, because it does not require TLB updates. A study of the Solaris zero-copy TCP scheme on multiprocessors [23] shows that COW and page swapping are considerably more efficient than copying even on multiprocessors without hardware support for TLB consistency. That study suggests that emulated copy would also be beneficial in such architectures, but does not clarify what further benefit would be obtained from emulated share.

Other architectural variations of interest include systems with much greater I/O bandwidth than that of personal computers, and devices with much lower latency than those used in this dissertation's experiments. Both of these variations may make the effects of data passing semantics and operating system structure more significant than found here.

Optimized control passing may significantly reduce the latency of I/O-oriented IPC for short data. It would be interesting to verify if techniques such as those of Peregrine [43], L4 [51], or Exokernel [34] could be applied to improve context switching in I/O-oriented IPC.

Bibliography

[1] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu and W. Zwaenepoel. "TreadMarks: Shared Memory Computing on Networks of Workstations", in *IEEE Computer*, 29(2):18-28, Feb. 1996.

[2] T. Anderson, H. Levy, B. Bershad, and E. Lazowska. "The Interaction of Architecture and Operating System Design", in *Proc. 4th ASPLOS*, ACM, Apr. 1991, pp. 108-120.

[3] R. Atkinson. "Default IP MTU for use over ATM AAL5". RFC 1626, Naval Research Laboratory, May 1994.

[4] M. Bailey, B. Gopal, M. Pagels and L. Peterson. "PATHFINDER: A Pattern-Based Packet Classifier", in *Proc. OSDI'94*, USENIX, Nov. 1994, pp. 115-124.

[5] D. Banks and M. Prudence. "A High-Performance Architecture for a PA-RISC Workstation", in *J. Selected Areas in Communications*, IEEE, vol. 11, no. 2, Feb. 1993, pp. 191-202.

[6] J. Barrera III. "A fast Mach network IPC implementation", in *Proc. USENIX Mach Symp.*, USENIX, Nov. 1991, pp. 1-11.

[7] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. "Lightweight Remote Procedure Call", in *ACM Trans. Comp. Systems*, 8(1):37-55, Feb. 1990.

[8] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. "User-Level Interprocess Communication for Shared-Memory Multiprocessors", in *ACM Trans. Comp. Systems*, 9(2):175-198, May 1991.

[9] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers and S. Eggers, "Extensibility, Safety and Performance in the SPIN Operating System", *Proc. 15th SOSP*, ACM, Dec. 1995, pp. 267-284.

[10] D. Black. "Scheduling Support for Concurrency and Parallelism in the Mach Operating System". in *Computer*, IEEE, May 1990.

[11] M. Blumrich, K. Li, R. Alpert, C. Dubnicki. E. Felten and J. Sandberg. "Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer", in *Proc. 21st Annual Int. Symp. Comp. Arch.*, IEEE/ACM, Apr. 1994, pp. 142-153.

[12] J. Brustoloni. "Exposed Buffering and Sub-Datagram Flow Control for ATM LANs", in *Proc. 19th Conf. Local Computer Networks*, IEEE, Oct. 1994, pp. 324-334.

[13] J. Brustoloni and P. Steenkiste. "Effects of Buffering Semantics on I/O performance", in *Proc. OSDI'96*, USENIX, Oct. 1996, pp. 277-291.

[14] J. Brustoloni and P. Steenkiste. "Scaling of End-to-End Latency with Network Transmission Rate", in *Proc. Gigabit Networking Workshop (GBN'97)*, IEEE, Kobe, Japan, Apr. 1997.

[15] J. Brustoloni and P. Steenkiste. "Copy Emulation in Checksummed, Multiple-Packet Communication", in *Proc. INFOCOM'97*, IEEE, Kobe, Japan, April 1997, pp. 1124-1132.

[16] J. Brustoloni and P. Steenkiste. "Evaluation of Data Passing and Scheduling Avoidance", in *Proc. NOSSDAV'97*, IEEE, St. Louis, MO, May 1997, pp. 101-111.

[17] G. Buzzard, D. Jacobson, M. Mackey, S. Marovitch and J. Wilkes. "An implementation of the Hamlyn sender-managed interface architecture", in *Proc. OSDI'96*, USENIX, Oct. 1996, pp. 245-259.

[18] *BYTE*, "Getting to the Kernel: Is NT Still Safe?", Oct. 1996, p. 130.

[19] P. Cao, E. Felten and K. Li. "Implementation and Performance of Application-Controlled File Caching", in *Proc. OSDI'94*, USENIX, Nov. 1994, pp. 165-177.

[20] J. Carter and W. Zwaenepoel. "Optimistic Implementation of Bulk Data Transfer Protocols", in *Proc. SIGMETRICS'89*, ACM, May 1989, pp. 61-69.

[21] J. B. Chen and B. Bershad. "The Impact of Operating System Structure on Memory System Performance", in *Proc. 14th SOSP*, ACM, Dec. 1993, pp. 120-133.

[22] D. Cheriton and W. Zwaenepoel. "The Distributed V Kernel and its Performance for Diskless Workstations", in *Proc. 9th SOSP*, ACM, Oct. 1983, pp. 129-140.

[23] H. J. Chu. "Zero-Copy TCP in Solaris", in *Proc. Winter Tech. Conf.*, USENIX, Jan. 1996.

[24] D. Clark and D. Tennenhouse. "Architectural considerations for a new generation of protocols", in *Proc. SIGCOMM'90*, ACM, Sept. 1990, pp. 200-208.

[25] E. Cooper, P. Steenkiste, R. Sansom and B. Zill."Protocol Implementation on the Nectar Communication Processor", in *Proc. SIGCOMM'90*, ACM, Sept. 1990, pp. 135-143.

[26] H. Custer. "Inside Windows NT". Microsoft Press, 1993.

[27] C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards and J. Lumley. "Afterburner", in *IEEE Network*, July 1993, pp. 36-43.

[28] R. Draves, B. Bershad, R. Rashid, and R. Dean. "Using Continuations to Implement Thread Management and Communication in Operating Systems", in *Proc. 13th SOSP*, ACM, Oct. 1991, pp. 122-136.

[29] P. Druschel and L. Peterson. "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility", in *Proc. 14th SOSP*, ACM, Dec. 1993, pp. 189-202.

[30] P. Druschel, L. Peterson and B. Davie. "Experience with a High-Speed Network Adaptor: A Software Perspective", in *Proc. SIGCOMM'94*, ACM, Aug. 1994, pp. 2-13.

[31] P. Druschel and G. Banga, "Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems", *Proc. OSDI'96*, USENIX, Oct. 1996, pp. 261-275.

[32] P. Druschel, V. Pai and W. Zwaenepoel. "Extensible Kernels are Leading OS Research Astray", in *Proc. HotOS-VI*, IEEE, May 1997, pp. 38-42.

[33] T. von Eicken, A. Basu, V. Buch and W. Vogels. "U-Net: A User-Level Network Interface for Parallel and Distributed Computing", in *Proc. 15th SOSP*, ACM, Dec. 1995, pp. 40-53.

[34] D. Engler, M. F. Kaashoek and J. O'Toole Jr., "Exokernel: An Operating System Architecture for Application-Level Resource Management", *Proc. 15th SOSP*, ACM, Dec. 1995, pp. 251-266.

[35] K. Fall and J. Pasquale. "Exploiting In-Kernel Data Paths to Improve I/O Throughput and CPU Availability", in *Proc. Winter Conf.*, USENIX, Jan. 1993, pp. 327-333.

[36] K. Fall and J. Pasquale, "Improving Continuous-Media Playback Performance with In-Kernel Data Paths", in *Proc. 1st Intl. Conf. on Multimedia Computing and Systems*, IEEE, May 1994, pp. 100-109.

[37] D. Golub, R. Dean, A. Forin, R. Rashid. "Unix as an Application Program", in *Proc. Summer Conf.*, USENIX, June 1990.

[38] R. Gopalakrishnan and G. Parulkar, "A Real-time Upcall Facility for Protocol Processing with QoS Guarantees", *Proc. 15th SOSP*, ACM, Dec. 1995, p. 231.

[39] P. Goyal, X. Guo and H. Vin, "A Hierarchical CPU Scheduler for Multimedia Operating Systems", *Proc. OSDI'96*, USENIX, Oct. 1996, pp. 107-121.

[40] J. Hennessy and D. Patterson. "Computer Architecture: A Quantitative Approach", 2nd. ed., Morgan Kaufmann Pub., San Francisco, CA, 1996.

[41] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham and M. West. "Scale and Performance in a Distributed File System", in *ACM Trans. Comp. Systems*, 6(1):51-81, Feb. 1988.

[42] IBM. "IBM System/370 Extended Architecture Principles of Operation". IBM publ. nr. SA22-7085-1, 2nd. ed., 1987.

[43] D. B. Johnson and W. Zwaenepoel. "The Peregrine High-Performance RPC System", in *Software — Practice and Experience*, 23(2):201-221, Feb. 1993.

[44] M. F. Kaashoek, D. Engler, G. Ganger and D. Wallach. "Server Operating Systems", in *Proc. SIGOPS European Workshop*, ACM, Sept. 1996.

[45] M. F. Kaashoek, D. Engler, G. Ganger, H. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti and K. MacKenzie. "Application Performance and Flexibility on Exokernel Systems", to appear in *Proc. 16th SOSP*, ACM, Oct. 1997.

[46] K. Kleinpaste, P. Steenkiste and B. Zill. "Software Support for Outboard Buffering and Checksumming", in *Proc. SIGCOMM'95*, ACM, Aug. 1995, pp. 87-98.

[47] C. Kosak, D. Eckhardt, T. Mummert, P. Steenkiste and A. Fischer. "Buffer Management and Flow Control in the Credit Net ATM Host Interface", in *Proc. 20th Conf. Local Computer Networks*, IEEE, Oct. 1995, pp. 370-378.

[48] C. Lee, M. Chen and R. Chang. "HiPEC: High Performance External Virtual Memory Caching", in *Proc. OSDI'94*, USENIX, Nov. 1994, pp. 153-164.

[49] S. Leffler, M. McKusick, M. Karels and J. Quaterman. "The Design and Implementation of the 4.3BSD UNIX Operating System", Addison-Wesley Pub. Co., Reading, MA, 1989.

[50] J. Liedtke. "Improving IPC by Kernel Design", in *Proc. 14th SOSP*, ACM, Dec. 1993, pp. 175-188.

[51] J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, G. Heiser, N. Islam and T. Jaeger. "Achieved IPC Performance (Still the Foundation for Extensibility)", in *Proc. HotOS-VI*, IEEE, May 1997, pp. 28-31.

[52] C. Maeda and B. Bershad. "Protocol Service Decomposition for High-Performance Networking", in *Proc. 14th SOSP*, ACM, Dec. 1993, pp. 244-255.

[53] C. Maeda. "Service Decomposition: A Structuring Principle for Flexible, High-Performance Operating Systems", Ph. D. Thesis, CMU-CS-97-128, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Apr. 1997.

[54] J. Mogul, R. Rashid and M. Acetta. "The Packet Filter: An Efficient Mechanism for User-Level Network Code", in *Proc. 11th SOSP*, ACM, Nov. 1987, pp. 39-51.

[55] J. Mogul and S. Deering. "Path MTU discovery". Network Working Group, RFC 1191, Nov. 1990. Available from `http://info.internet.isi.edu/in-notes/rfc`.

[56] J. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel", *Proc. 1996 Conf.*, USENIX, Jan. 1996, pp. 99-111.

[57] D. Mosberger and L. Peterson, "Making Paths Explicit in the Scout Operating System", *Proc. OSDI'96*, USENIX, Oct. 1996, pp. 153-167.

[58] T. Mowry, A. Demke and O. Krieger. "Automatic Compiler-Inserted I/O Prefetching for Out-Of-Core Applications", in *Proc. OSDI'96*, USENIX, Oct. 1996, pp. 3-17.

[59] T. Mummert, C. Kosak, P. Steenkiste and A. Fischer. "Fine Grain Parallel Communication on General Purpose LANs", in *Proc. 10th Intl. Conf. Supercomputing*, ACM, 1996, pp. 341-349.

[60] B. Murphy, S. Zeadally and C. Adams. "An Analysis of Process and Memory Models to Support High-Speed Networking in a UNIX Environment", in *Proc. Winter Tech. Conf.*, USENIX, Jan. 1996.

[61] G. Necula and P. Lee. "Safe Kernel Extensions Without Run-Time Checking", in *Proc. OSDI'96*, USENIX, Oct. 1996, pp. 229-243.

[62] S. O'Malley, M. Abbot, N. Hutchinson and L. Peterson. "A Transparent Blast Facility", in *Internetworking: Research and Experience*, vol. 1, 1990, pp. 57-75.

[63] J. Ousterhout. "Why aren't operating systems getting faster as fast as hardware?", in *Proc. Summer 1990 Conf.*, USENIX, June 1990, pp. 247-256.

[64] J. Pasquale, E. Anderson and P. Muller, "Container Shipping: Operating System Support for I/O-Intensive Applications", *Computer*, 27(3):84-93, IEEE, Mar. 1994.

[65] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky and J. Zelenka. "Informed Prefetching and Caching", in *Proc. 15th SOSP*, ACM, Dec. 1995, pp. 79-95.

[66] J. Postel. "The TCP Maximum Segment Size and Related Topics". Network Working Group, RFC 879, Nov. 1983. Available from `http:info.internet.isi.edu/in-notes/rfc`.

[67] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black,W. Bolosky and J. Chew. "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures", in *Proc. 2nd ASPLOS*, ACM, Oct. 1987, pp. 31-39.

[68] M. Schroeder and M. Burrows. "Performance of Firefly RPC", in *ACM Trans. Comp. Systems*, 8(1):1-17, Feb. 1990.

[69] M. Seltzer, Y. Endo, C. Small, K. Smith, "Dealing With Disaster: Surviving Misbehaved Kernel Extensions", *Proc. OSDI'96*, USENIX, Oct. 1996, pp. 213-227.

[70] C. Small and M. Seltzer, "A Comparison of OS Extension Technologies", *Proc. 1996 Conf.*, USENIX, Jan. 1996, pp. 41-54.

[71] P. Steenkiste, B. Zill, H. Kung, S. Schlick, J. Hughes, R. Kowalski and J. Mullaney. "A Host Interface Architecture for High-Speed Networks", in *Proc. 4th IFIP Conf. High Performance Networks*, IFIP, Dec. 1992, pp. A3 1-16.

[72] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and G. Plaxton, "A Proportional Share Resource Allocation for Real-Time, Time-Shared Systems", *Proc. 17th Real-Time Systems Symp.*, IEEE, Dec. 1996, pp. 288-299.

[73] T. Stricker, J. Stichnoth, D. O'Hallaron and S. Hinrichs. "Decoupling Synchronization and Data Transfer in Message Passing Systems of Parallel Computers", in *Proc. Intl. Conf. Supercomputing*, ACM, July 1995, pp. 1-10.

[74] C. Thekkath, T. Nguyen, E. Moy and E. Lazowska. "Implementing Network Protocols at User Level", in *Proc. SIGCOMM'93*, ACM, Sept. 1993.

[75] C. Thekkath, H. Levy and E. Lazowska. "Separating Data and Control Transfer in Distributed Operating Systems", in *Proc. 6th ASPLOS*, ACM, Oct. 1994, pp. 2-11.

[76] S.-Y. Tzou and D. Anderson. "The Performance of Message-passing using Restricted Virtual Memory Remapping", in *Software – Practice and Experience*, vol. 21(3), March 1991, pp. 251-267.

[77] D. Wallach, D. Engler and M. F. Kaashoek, "ASHs: Application-Specific Handlers for High-Performance Messaging", *Proc. SIGCOMM'96*, ACM, Aug. 1996, pp. 40-52.