UNIVERSITY OF CALIFORNIA, SAN DIEGO

A Peer–to–Peer I/O System in Support of I/O Intensive Workloads

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science

by

Kevin Roland Fall

Committee in charge:

Professor Joseph C. Pasquale, Chair
Professor Thomas E. Anderson
Professor Jeffrey Elman
Professor William Griswold
Professor Keith Marzullo

1994

Copyright

Kevin Roland Fall, 1994

All rights reserved.

The dissertation of Kevin Roland Fall is approved, and it is acceptable in quality and form for publication on microfilm:

Chair

University of California, San Diego

1994

To my parents and to Vicki, who have been a continuing source of encouragement and support.

TABLE OF CONTENTS

LIST OF FIGURES

I with to thank my advisor, Joseph Pasquale, for his valuable advice, inspiration, and patience. I would also like to thank other members of the Computer Systems Laboratory at UCSD, especially P. Keith Muller who provided me with many thought-provoking conversations and comments during the formation of the ideas contained herein.

VITA

November 1, 1966	Born, Hollywood, California.
1985–1986	Systems Programmer, TRW Electronics and Defense
1986–1988	Programmer Analyst, DASH Project, UC Berkeley
1988	B.A. Computer Science, University of California, Berkeley
1988-1989	Programmer Analyst, Comp. Systems Research Group, UC Berkeley
1989	Programmer Analyst, Project Athena, Massachusetts Institute of Technology
1990	M.S. Computer Science & Engineering, University of California, San Diego
1991–1992	Programmer Analyst, Center for Research in Language, University of California, San Diego
1991–1994	Research Assistant, University of California, San Diego
1992-1994	Networking Consultant, San Diego Supercomputer Center
1992-1994	Teacher, University of California San Diego Extension
1994	Doctor of Philosophy University of California, San Diego

PUBLICATIONS

"TCP/IP and HIPPI Performance in the CASA Gigabit Testbed", Proc. Usenix Symposium on High–Speed Networking., Aug., 1994.

"Improving Continuous-Media Playback Performance with In-Kernel Data Paths." Proc. 1st. Intl. Conf on Multimedia Computing and Systems., May, 1994.

"Improving I/O System Performance with In–Kernel Data Paths." Proceedings UniForum Winter Conference., Mar., 1993.

"Exploiting In-Kernel Data Paths to Improve I/O Throughput and CPU Availability." Proceedings USENIX Winter Conference., Jan., 1993.

"Sequoia 2000 Network (S2KNet) Handbook." Sequoia 2000 Technical Report 92/15., Nov., 1992.

"The Distributed Laboratory: An Interactive Visualization Environment for Electron Microscopy and 3D Imaging." Communications of the ACM., Jun., 1992.

"Internet Throughput and Delay Measurements Between Sequoia 2000 Sites." Sequoia 2000 Technical Report 91/7., Dec., 1991.

"Network and Operating System Support for Multimedia Applications." UCSD CSE Technical Report CS91-186., Mar., 1991.

"Current Research by the Computer Systems Research Group of Berkeley." Proceedings European UNIX User's Group., Apr., 1989.

FIELDS OF STUDY

Major Field: Computer Science and Engineering
Studies in Operating Systems.
Professor Joseph C. Pasquale

Studies in Computer Networks.
Professor Joseph C. Pasquale
Hans–Werner Braun, San Diego Supercomputer Center

ABSTRACT OF THE DISSERTATION

A Peer-to-Peer I/O System
in Support of I/O Intensive Workloads

by

Kevin Roland Fall
Doctor of Philosophy in Computer Science
University of California, San Diego, 1994
Professor Joseph C. Pasquale, Chair

The I/O subsystem interface present in most modern operating systems stems from the Multics system of the late 1960's. The Multics I/O design places a user program in the center of the flow of I/O data, requiring data movement between I/O devices and user processes. This dissertation argues that an I/O system supporting peer-to-peer communication between I/O objects improves programmability and increases overall system performance in terms of throughput and minimization of transaction delay over the traditional model. Peer-to-peer communication refers to the ability to transfer data between I/O objects (i.e. entities upon which typical read and write I/O operations may be performed) without direct user processes intervention. User processes define data stream endpoints, but need not actively transfer data between objects to facilitate I/O data flow. A system interface and architecture is described which implements a peer-to-peer I/O model. A set of example applications and measured implementations substantiate the claimed flexibility and efficacy of the mechanisms described.

Chapter 1

Introduction

Improvements in computer hardware have enabled the development of complex applications with enormous I/O demands. Providing adequate system performance for such applications poses a significant challenge to operating systems, especially with the growing popularity of multimedia applications and systems. Although both application demands and hardware performance have witnessed great gains in recent years, I/O system software performance has not improved commensurately. Furthermore, fundamental assumptions manifested in an I/O system's structure may limit achievable performance by introducing unnecessary overheads.

I/O Intensive applications are those applications with large transput (input or output) demands—on the order of hundreds or thousands of megabytes. Many applications, especially multimedia applications, require the movement of large volumes of data between devices or files in a timely fashion with minimal intermediate manipulation or processing. Concepts useful for improving I/O system performance for these applications include minimization of data movement within memory, and separating I/O control from I/O data transfer [Pas92].

The design of most conventional I/O system architectures dates back to the Multics system of the late 1960's and UNIX [ATT78] of the late 1970's. The UNIX I/O model provides a continuous untyped stream of bytes between processes and devices. The original UNIX I/O system provided no inter-process communication (IPC) mechanism for unrelated

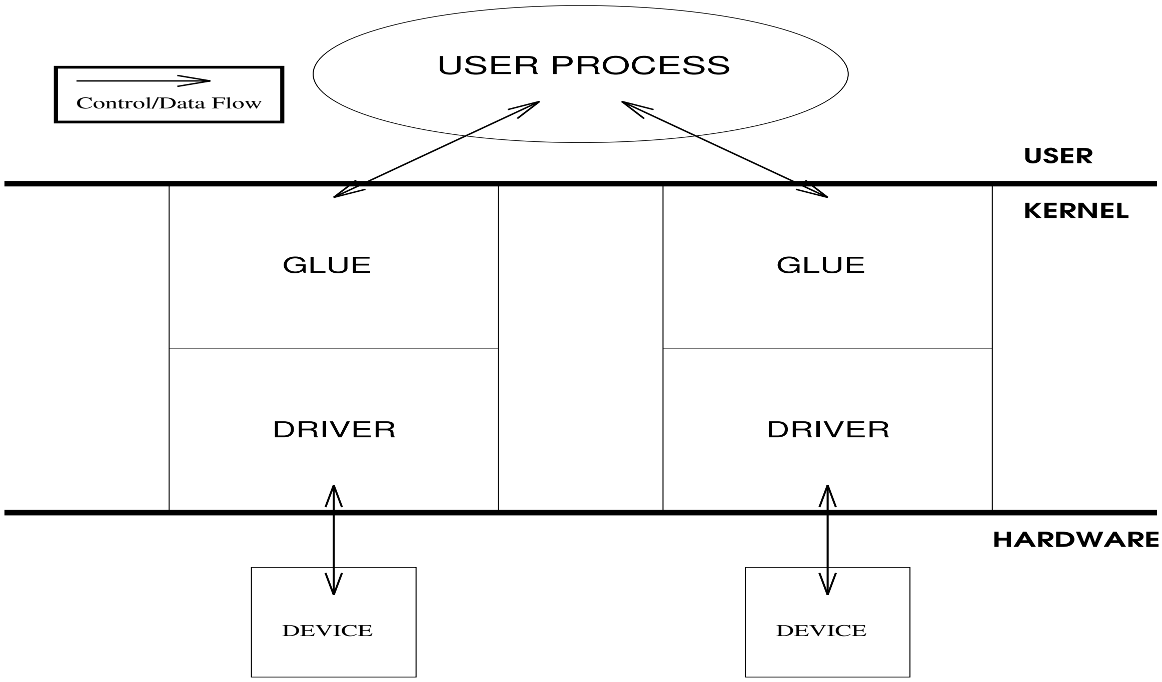

Figure 1.1: Conventional Memory-Oriented I/O Model

processes to communicate with each other.[1] IPC was successfully added to UNIX with the introduction of *named pipes*, and has evolved to include other mechanisms based on shared memory, network communication, and specialized procedure calls.

Although the various IPC mechanisms create a number of ways for processes to communicate with one another (and also with devices in some cases), little software support exists for the direct transfer of data between hardware devices. With the emergence of multimedia computing, the notion of the computer's operating system as an "I/O director" becomes natural, and the I/O architecture should directly support this conceptualization. User processes specify the sources and sinks of data streams, leaving the job of data transfer up to the operating system.

[1]The UNIX *pipe* facility allowed related processes (those sharing a common creation ancestry) to communicate in a simplex manner.

1.1 The Traditional Memory-Oriented I/O Model (MIO)

Figure 1.1 illustrates the *memory–oriented* I/O model (MIO) present in the I/O subsystems of most operating systems today. In MIO, a process wishing to transfer data between two devices acquires a handle to each device, executes a loop of copy operations to move I/O data into its own address space, then moves the same data out of its address space to a destination device. The MIO loop structure operates generally as follows:

```
handle1 = open("device1", READ)
handle2 = open("device2", WRITE)
allocate buffer1 (size is specified by programmer)

do until no more {
        fill buffer1 from handle1 (read)
        output buffer1 to handle2 (write)
}
```

In this illustration, both control and data requests flow between user processes and operating system. There are several benefits of the MIO structure. MIO provides great flexibility by allowing user processes to perform any necessary data transformation on I/O data as it is moved between I/O devices. Manipulation of I/O data is simple with MIO because I/O data is made directly available to user processes in user-specified buffers. Manipulation is done generally as follows:

```
handle1 = open("device1", READ)
handle2 = open("device2", WRITE)
allocate buffer1 (size is specified by programmer)

do until no more {
        fill buffer1 from handle1 (read)
        manipulate buffer1 or alter control flow
                based on contents of buffer1
        output buffer1 to handle2 (write)
}
```

Unfortunately, MIO suffers from several performance problems: data copying across the user/kernel boundary and overhead associated with scheduling and context switches. These overheads are discussed in more detail in the following section. In the application context of multimedia systems, for example, data copying limits ultimate achievable throughput (e.g., frame rate), and scheduling/context switching can adversely affect continuity of playback for multimedia applications (skips and loss of synchronization).

1.2 Programming Difficulty of Current Systems

Consider the complexity of coding a delay-sensitive application responsible for transferring digitized audio samples from a network connection to an audio DAC (digital to analog converter) using an MIO structure. The programmer must carefully select:

- the read and write transfer unit sizes
- flow control (in particular, how to handle too fast a sender)
- how often a program must execute to satisfy delay bounds

Subtle tradeoffs arise when attempting to make appropriate selections for the above metrics. Furthermore, each is sensitive to overall system and network loading. For example, small read/write transfer sizes provide decreased delay but may limit throughput (a potential problem for a video application). Large sizes can cause the reverse phenomenon: high throughput but large delay. Currently, these sizes are determined for delay-sensitive programs in an *ad–hoc* fashion. When system loading changes, new values must be determined "on-the-fly."

Flow control is similarly challenging. Most systems combine flow control and queuing to achieve reliable data delivery between communication endpoints (both for local IPC and network communication with reliable protocols like TCP [Pos81]). For the delivery of uncompressed audio, dropping certain information is tolerable, but specifying this fact to current systems is awkward at best. One technique involves using non-blocking I/O and discarding information when I/O would otherwise block. Unfortunately, this method can lead to losses at inappropriate moments, producing a "glitch" in the final audio output.

In conventional non-real-time operating systems, a programmer faces considerable difficulty specifying when processes run. Execution of user processes is required in conventional I/O systems because both data input request operations (i.e., reads) and data output request operations (i.e., writes) need to be performed by user processes. Even in real-time systems, where programmer's can specify when processes execute, the interfaces provided require the execution of user processes and tend to include overly-conservative scheduling policies to provide execution-time guarantees.

1.3 Performance Implications of Current Interfaces

As discussed in the previous section, MIO-based systems must invoke user processes to facilitate the flow of I/O data. Invoking user processes implies several system overheads:

- crossing protection boundaries (system calls)
- context switches
- TLB misses
- loss of locality (caching inefficiency)
- data copying

Crossing the user/kernel protection boundary occurs during the execution of any system call, page fault, or exception. However, Anderson et. al [ALBL91] point out that while microprocessor integer speeds have improved steadily, support for the operations required by operating systems has not scaled commensurately. For example, their measurements indicate that a R3000 RISC CPU receives only a factor of 3.9 speed improvement over its CVAX predecessor for the handling of the null system call,[2] while this factor is 6.7 for applications. Similarly, a SPARC-1 CPU can execute a null system call in about the same amount of time required by the CVAX, but achieves a factor of 4.3 speed improvement for applications. As the popularity of RISC processors grow, the performance divergence

[2] A common metric used to evaluate system performance is the time required to execute a "null" system call, including a simple entry and exit from the kernel without any in–kernel processing. Generally, the cost of a null system call is small (tens of microseconds or less).

between operating system and user codes is likely to increase because the techniques employed to improve applications performance on RISC architectures do not apply equally well to operating system code. In particular, incorporating large register sets and pipelines, the processing required to field interrupts, manage threads, examine, save, and restore state are improved little by deeper pipelines and large register sets.

As with system calls, the overhead involved in performing a context switch has not been reduced commensurately with the increase in microprocessor integer performance. Context switches are performed by an operating system to improve CPU utilization during periods of I/O. Performing a context switch requires selection of a process to run, plus a change of MMU page table entries to the context of the next process. Context switch overhead accounts for approximately 60% of the time required to perform a null IPC between protection domains (the rest of the time is spent in procedure calls and trap handling) [BALL90]. In measurements made by Ousterhout [Ous90b], most RISC machines showed context switch speeds of about 50% one would expect from the MIPS rating of the particular CPU tested.

Any replacement of MMU page table entries requires the invalidation of the TLB (if process tags are not used). TLBs are flushed during context switch operations to ensure new processes do not receive access to the address space of a previous process. In some modern processors, TLB reloading is relegated to software, improving flexibility at the expense of greater complexity in handling TLB misses. Bershad et. al [BALL90] estimates 25% of the time required to do an LRPC (a highly optimized local RPC) is attributable to TLB misses occurring during virtual address translations.

Change of process context can also have a negative effect on cache performance. For multiprocessing workloads, where various processes may accumulate cached data, Mogul and Borg [MB91] estimate poor cache performance induced by context switching can account for a full order of magnitude performance degradation. Unfortunately, modern RISC architectures are hurt badly by loss of cache locality. An uncached memory reference will cost about three times as much or more than a comparable cached memory reference. As CPU performance has improved at a rate of about 18% to 35% before 1985 and at a rate of between 50% and 100% per year thereafter, the row–access times of DRAM-based memory systems has improved much more slowly—at a rate of about 7% per year [HP90] (sec. 8.4). This trend suggests the cache miss penalty is likely to increase rather than decrease in the future.

Perhaps the largest source of performance degradation in conventional systems is copying of data. Data copying is performed by an operating system to move information from one buffer to another or between device and system memory. In the traditional UNIX I/O system, a program moving data between two devices incurs four data copy operations: I/O adapter to kernel buffer, kernel buffer to user-specified process buffer, process buffer to new kernel buffer, and finally kernel buffer to I/O device. The first and last operations are accomplished by DMA or by programmed I/O; the intermediate two are accomplished by direct CPU copying. Because DMA to physical memory results in a modification to physical memory without a CPU memory reference, common practice requires a cache flush (or partial cache flush) when DMA is performed.

Interestingly, the problem of data copying has had its greatest affect on network I/O performance. When only disks were attached to computer systems, the system bottleneck was the performance of the I/O device; the CPU could easily copy data to and from user space with plenty of time to spare. With faster devices (such as network adapters), where the device I/O rate may be comparable to the CPU execution rate, the bottleneck shifts from the I/O device rate to the performance of data copies, and thus the memory system. Several groups credit data copying as a primary problem with respect to system performance (see [Ous90b], [PCMI91], and [CT90] as examples).

1.4 Summary of MIO

The problems associated with current operating systems supporting a memory-oriented I/O structure for I/O intensive applications are twofold. First, programming is made difficult because of the need to select buffer sizes and handle flow control. Second, the current I/O system interface requires the execution of user processes to move I/O data.

Execution of user tasks degrades performance by requiring protection crossings, context switches, and unnecessary data copies. Moreover, as a consequence of the MIO programming interface, shared buffering and inter-peripheral hardware transfers are precluded because I/O data *must* generally be made available to a user's address space, and may be modified by user processes even after it is submitted to the operating system for transfer.

1.5 Peer-to-Peer I/O (PPIO) Philosophy and Goals

The previous section highlights the primary problems associated with the MIO structure. Specifically, the MIO structure includes a programming interface requiring programmer specification of I/O transfer unit sizes and the overall MIO structure requires user process execution for supporting I/O data transfer. This section introduces an alternative I/O structure known as *Peer-to-Peer* I/O (PPIO). In PPIO, a programmer need not choose I/O transfer sizes and user process execution is *not* required for the flow of I/O data.

1.5.1 Research Goals and Methodology

The following sections of this chapter outline the primary components needed to achieve the Peer-to-Peer I/O system. These components are described in greater detail in Chapters 3, 4, and 5.

1.5.2 Guiding Principles

The goals of the research presented in this dissertation are to address the problems exhibited by traditional MIO systems as described above, following several guiding principles:

Principal 1 improve performance for a rich set of application programs
Principal 2 assure interface enhancements are conceptually simple
Principal 3 take advantage of special-purpose hardware, if available
Principal 4 enhance sytem interface without duplicating existing functionality

1.5.3 Philosophy

PPIO takes a different view of data movement as compared with MIO. In PPIO, user processes specify the producer and consumer of a data flow. The operating system responds by creating an internal *association* used to keep track of which data producers are “connected” to which data consumers. The operating system can optimize the movement of data between producer and consumer by techniques such as buffer sharing. In PPIO, an application wishing to move data between a data producer and consumer would execute the following program segment:

```
handle1 = open("device1", READ)
handle2 = open("device2", WRITE)

ASSOCIATE handle1 with handle2 until EOF
{OS handles data flow from handle1 to handle2}
```

Supporting associations within the operating system has two potentially significant influences on performance. First, the user process establishing the association need not execute (and be scheduled) for data to be transferred between `handle1` and `handle2`. Secondly, no process-level buffering is used. Removing the intermediate buffer implies I/O data need not necessarily be made available to user processes. Thus, I/O data can be manipulated even if it is not present in main memory (e.g., if it resides only on I/O adapter memory).

The technique described above is often called *streaming* in the literature (see Section 2.4), and it is the data handling model supported by PPIO. Streaming is essentially a method which “short-circuits” the data path between producer and consumer. Critics are quick to point out the most obvious drawback: intermediate processing is difficult in such an I/O model. Fortunately, a number of techniques (some of which are currently under investigation by other researchers) provide the ability to introduce intermediate processing into a data stream without requiring a user process implementation. These techniques are detailed in Chapter 4. By extending the PPIO *association* model with intermediate processing capability, PPIO provides the benefits of streaming with the flexibility of module execution. A process would would create associations in the following general way:

```
handle1 = open("device1", READ)
handle2 = open("device2", WRITE)
code1 = open("intermediate_code.exec", READ and EXECUTE)

ASSOCIATE code1 with handle 1
ASSOCIATE handle1 with handle2 until EOF
{OS handles data flow from handle1 through code1 to handle2}
```

In this case, the operating system manages data originating from `device1`, invokes processing described by `code1` and deposits the result to `device2`. The code referenced by `code1` may be executed by any device capable of code execution. More specifically, the code referenced by `code1` need not be executed on the primary CPU, but may instead be executed on another CPU or on a special intelligent I/O adapter.

1.5.4 Evaluation

The PPIO philosophy adheres to the principles described in Section 1.5.2 as follows. Many applications require simple movement of I/O data from a source to sink, including copy programs, application-level gateways (for networking), and video/audio capture and playback applications. The basic PPIO philosophy provides support for these applications, and broadens support by incorporating processing modules. Thus, a rich set of applications are supported (principle 1).

The philosophy rests on creating associations between a data producer and consumer. Connecting I/O objects together in this fashion is not a conceptual leap for a programmer. For example, telephone operators (before the advent of electronic switching) are commonly pictured as connecting together a pair of communicating entities with a “patchcord”. Conceptually, the creation of associations is analogous to the notion of a patchcord. Additionally, the concept is similar to an inverted *pseudoterminal* as described by Ritchie [Rit84] a decade ago. The association concept is simple, thus meeting principle 2.

As previously described, the philosophy includes no buffer interface and does not require user-level process execution. For these reasons, processing of I/O data need not occur on the main CPU and I/O data need not necessarily reside in main memory. Systems equipped with outboard processors and memory are capable of offloading processing to peripheral devices. Systems lacking outboard processors but equipped with adapter memory can perform some data manipulations without moving adapter-resident data to main memory (thus avoiding a DMA or Programmed IO data copy). Thus, principle 3 is met.

The last two principles of Section 1.5.2 constrain the relationship between mechanisms introduced in supporting PPIO with respect to those already present for supporting MIO. Most existing systems have a `read` and `write` system function but lack any function capable of associating the source of a read with the sink of a write. The PPIO philosophy does not overlap functionality with either read or write, and is not intended to replace either of these functions (principal 4). Rather, it may be employed by applications wishing to improve data flow performance.

1.5.5 System Components

To be a useful system design philosophy, PPIO must be implementable in conventional operating systems. The PPIO approach requires additional operating system level mechanisms above and beyond those provided with conventional systems. Movement of data through an operating system is accomplished and initiated using several components, the following of which are of primary importance when realizing a PPIO system:

1. An I/O Interface for User Programs
2. A buffering system for holding in-transit data
3. An execution thread for moving data when necessary
4. A processing module architecture, for performing data transformations

1.6 Contributions

The PPIO design represents a new paradigm for the manipulation of I/O intensive data flows. This dissertation provides a critical review of the memory-oriented model of

I/O present in most operating systems and provides an alternative based on a peer-to-peer design.

The system software support required for implementing a peer-to-peer I/O system is described, including a novel system interface based on the manipulation of inter-object *associations* specified by user processes. A characterization of I/O objects based on *attributes* represents a new approach to flow control capable of capturing the behavior of time-sensitive devices. A specification of flow-control algorithms based on attributes completes the flow control model.

Performance characterizations for three prototype PPIO implementations indicate a 2–3 factor improvement in throughput and latency for software-based PPIO implementations with fast I/O devices. The prototypes span several relevant areas, including disk file manipulation, network-based video playback, and network packet forwarding.

1.7 Organization of the Dissertation

In Chapter 2, work relating to the content of this dissertation is presented. In Chapter 3, the system interface supporting the peer-to-peer model is introduced, along with several examples. Chapter 4 describes the execution of processing modules and code required to support I/O data streaming, including the use of kernel level threads and code execution invoked via interrupt processing. Chapter 5 investigates the handling and manipulation of data buffers within the peer-to-peer architecture. Chapter 6 discusses performance measurements taken from prototype implementations of the PPIO architecture. Some of the preliminary results have been published in [FP93] and [FP94]. Chapter 7 presents conclusions, a critique, and future work.

Chapter 2

Survey of Related Work

This chapter is devoted to past and ongoing work which relates to the formation of the peer-to-peer I/O design (PPIO). The design originated as a result of considering the effects of manipulating multimedia data objects using a general purpose operating system. Such objects are generally large and lack temporal locality and may not require arbitrary bit manipulation by user processes. These attributes make efficient handling of such objects difficult with conventional operating systems, which often improve performance between devices and user processes by use of caching.

The development of the PPIO design emerges by questioning the assumptions made in most systems that the operating system's job (with respect to I/O) is to connect devices to user processes. The design also borrows from recent work in both operating system design and high-speed network protocol implementation. Moreover, the PPIO design has stemmed from work in several areas, which are discussed in this chapter:

- development and standardization of common I/O interfaces
- buffering techniques
- protocol processing
- streaming systems
- module handling
- processing in other protection domains

2.1 Development of I/O Interfaces

One of the earliest I/O systems provided by a general multiprocessing operating system appeared in Multics [FO74]. The interface was of particular significance due to the use of opaque *handles* which provide abstract names for hardware resources and enable redirection of input and output. The interface and corresponding implementation provide access to hardware resources by user processes, and the interface has not changed drastically since its inception.

With the creation of UNIX in the 1970s [ATT78], much of the I/O interface from Multics was copied, with some function names changed. Over the years, the interface remained largely unchanged.[1] Additional functions were added with the integration of the ARPA protocols in the BSD4.2 distribution [LJF83].

The original interface underwent further specification and became incorporated in the POSIX 1003.1 standard [Ins88], which specifies an operating system interface borrowed heavily from the experience with UNIX. Other popular commercial systems use a similar structure. For example, in WindowsNT [Cus93], a *file object* includes *service procedures* such as `OpenFile`, `ReadFile`, and `WriteFile`. OS/2 [DK92] provides similar facilities.

The interfaces provided by these systems are very similar, differing primarily in function name rather than operation performed. They each provide for the reading and writing of I/O objects, along with low-level manipulations. They all provide movement of data between devices and user processes, and do not address the need to establish data paths which do not involve an intermediary user process.

[1] Minor changes include the adoption of the `select` call to permit multiplexing of I/O descriptors and the `mmap` call to permit the mapping of device memory to user processes. These capabilities are used extensively by display server processes.

2.2 Buffering

I/O systems rely on buffers to hold data in transit and to help speed–match objects which produce and consume data at different rates. The uses of buffering and types of buffering available in current systems are detailed in Chapter 5. Buffering systems have evolved with new device technology.

2.2.1 Buffering in UNIX

Early UNIX systems used buffers designed to accommodate character and block oriented I/O devices. Character oriented I/O devices were generally attached by asynchronous lines, and produced or consumed variable but small numbers of characters at a time. Such devices included card readers, terminals, etc. Block I/O devices primarily included tape and disk devices which were accessed with a fixed size block of some nominal number of bytes (approx. 0.5KBytes).

With the addition of network protocol support in the kernel, the buffering system had to be improved to accommodate variable-sized network *packets*. A more sophisticated form of buffering called *mbufs*, specifically designed for the manipulation of packet data, was created with 4.2BSD UNIX. Mbufs support a number of special manipulations required of network protocol processing including prepending of headers and appending of trailers and are found in most BSD-derived systems today. They have continued to evolve with the advent of new networking protocols such as OSI.

The mbuf system incorporates functions to allocate and release mbufs, prepend or append headers and trailers, reference-count data, and align buffers. The data storage portion of an mbuf is either an entire page or individual chunks provided by the general kernel allocator. Mbufs exist entirely in the kernel's address space and are not pageable. Data is copied between kernel-resident mbufs and free-form user process buffers as needed.

To provide a common data structure for all processing modules, Streams [Rit84] includes a *message block* data structure with *message buffers*. Message blocks are a very simple buffering scheme discussed in the original Streams design, and include a header describing a corresponding data buffer, a current read and write offset and maximum or limits. The scheme was extended and slightly modified in SystemV [USL92] and Plan 9 [Pre90].

The original Streams buffering structure is similar to the buffering structures present in the *standard I/O package* (STDIO) provided in conjunction with an ANSI C Programming environment, and specified in the *Language-Specific Services for the C Programming Language* chapter of the POSIX 1003.1 specification [Ins88]. In the Streams case, message buffers are entirely kernel-resident and not visible to user processes. In the STDIO case, I/O buffers are present entirely in user process space and not referenced directly by the kernel.

2.2.2 Buffering Optimizations

The buffering schemes mentioned so far are *address-space-local*. That is, they exist only within single address spaces and data transferred between address spaces is typically copied. A number of efforts have focused on reducing the overheads associated with copying. The copy problem is generally the high cost associated with moving data between protection boundaries. The user/kernel protection boundary is a particularly important special case.

The simplest way of avoiding copying is by utilizing *shared memory*. Memory may be shared between address spaces, including the kernel's. The unit of sharing is typically a *page*, and data must be aligned and wired-down. Rendezvous is implemented manually by the programmer. Synchronization primitives serialize access to shared data when necessary. Most existing systems provide such capability (UNIX examples include the SystemV `shmat, shmget, shmdt` interface and the BSD `mmap` interface).

A number of efforts attempt to optimize performance and provide a higher level interface than system pages. In the DASH message passing system [Tzo91], a virtual address range is reserved in all address spaces for the purposes of I/O data manipulation. The system has *move semantics*, meaning data is moved from one address space and into another; the initial address space relinquishes a reference to the data. Such movement requires unmapping in the sending address space and mapping into the destination address space by MMU table manipulation. Generally, such manipulations also require a TLB flush. The V Kernel [Che88] also implements such a scheme. The advantage of the scheme is the relative cost of updating page tables versus copying. For messages of reasonable size, page table updates are significantly cheaper than copying data. Both the DASH and V systems employ a buffering scheme which is visible to both kernel and user address spaces.

An alternative to move semantics is *copy semantics*, in which a sending domain retains at least a logical copy of transferred data. A *virtual copy* is performed by duplicating page mappings for the same physical space in multiple virtual address spaces and setting page permissions to read-only rather than read/write. When a write is performed to the pages in question, a write fault occurs, and a copy is performed. The scheme is known as *copy-on-write* and is used extensively in Mach [ABB$^+$86] and in its predecessor, Accent [FR86].

In the *fbufs* approach [DP93], buffers are made *immutable*, meaning only the originator of a buffer may have write-access to it. Given immutable buffers, there is no performance advantage of maintaining move rather than copy semantics. Copy semantics are implemented by sharing buffers. Further optimizations improve *fbuf* performance by elimination of zero-filling pages for trusted protection domains and assuming a received buffer is *volatile*. A volatile buffer may be asynchronously modified (this assumption relaxes the need to update page table entries and software VM maps in the sending domain).

In *container shipping* [PAM94], inter-domain transfers with move semantics are supported by the notion of encapsulating data in a sequence of *pallets* (contiguous virtual memory addresses). Data is mapped into a receiving domain only when requested by the application.

2.2.3 Buffering Summary

The early buffering systems mentioned here were created initially for simple storage of data while in transit from one location to another. Further developments included the incorporation of scatter/gather capabilities to accommodate variable-sized data objects (e.g., network packets). In such cases, the user processes control their own buffers, resulting in a copy across the user/kernel boundary.

Recent work with respect to buffering addresses the performance problems associated with the data movements implied by separate buffering mechanisms. These systems all share the common goal of reducing overheads, and also share the disadvantage of having to re-implement user programs in order to take advantage of the improved buffering regime. In addition, most of these systems all include the implicit assumption that the buffering system's purpose is to move data to and from user address spaces, with the possible exception of the container shipping scheme, which allows the application to decide.

2.3 Protocol Processing

IPC performance problems due to excessive copying are felt nowhere more strongly than in the processing of network protocols. The reason is simple: the two primary devices which bear the I/O load on most present–day systems are the network and disk subsystems, and the performance of network interfaces has tracked the performance of CPUs more closely than disks. Furthermore, manipulation of data buffers is considerably more complex for processing network packets.

The data manipulation problem (primarily data copying and checksumming computations) has been discussed repeatedly, including [CJRS89, DWB+93]. Much of the recent work in these areas have been the creation of methods to avoid intermediate (kernel-level) buffering.

The Afterburner [DWB+93] card is a network-independent network interface card built by HP Bristol based on Van Jacobson's "witless" network interface design. The idea was to build a dumb and fast interface with substantial on-board buffering and perform protocol operations on packets buffered in the network interface. The initial prototype was an FDDI interface called Medusa [BP93]. Afterburner was built after the success of Medusa. This system uses a *single copy* protocol stack: data is copied only once from network adapter to application address space. Checksum computation is included with the copy at no additional cost. This system has the advantage of requiring modification of only the operating system network buffering code, and does not imply re-coding of existing user applications.

Another approach which eliminates intermediate buffering to improve application performance for network protocol processing is discussed in the context of fast host interface design [ST93]. In this system, pages are mapped to and from user address space and are made accessible to the network interface. Earlier systems used similar approaches to optimize network transfer [JZ93, SB90] and local RPC [BALL90].

The efforts in protocol processing generally attempt to optimize the data path between network interface and user application. Issues include the proper type of buffering, modifications to the user interface, and partitioning of functionality and layering. The benefit of outboard processing remains an unresolved issue. The problems and approaches to network protocol processing with respect to data manipulation closely mirror the issues involved in buffer management discussed in the previous section. Once again, the provision for interconnection of data producer and consumer objects below the application is not available.

2.4 Streaming Approaches

A class of techniques known as *streaming* most closely resembles the design of PPIO. Streaming attempts to "short-circuit" a data path by interconnecting data sources and sinks below the application layer. Several types of streaming are discussed in [DAPP92], where the authors conclude low-level streaming is precluded because applications should be given access to continuous media (CM) data.

2.4.1 Types of Streaming

The authors of [DAPP92] identify four types of streaming:

- Hardware
- DMA-DMA
- OS Kernel

- User Level

The following four paragraphs summarize their observations and criticisms.

Hardware streaming refers to the ability to interconnect data sources and sinks at the level of the system bus, avoiding any operations that bring I/O data into main memory. This type of streaming is described by the authors to lack integration with workstation computer systems. Although hardware streaming precludes the direct processing by the CPU on I/O data, sufficiently sophisticated adapters have been implemented to handle data manipulations (see below).

DMA-DMA streaming refers to the inclusion of system memory in the data transfer path, but exclusion of processing by the main CPU. In such a system, conventional I/O devices supporting scatter/gather DMA can be used, and the authors acknowledge that generic devices could be used for supporting multimedia devices. This type of streaming architecture passes I/O data through the main memory system, and is likely to encounter a bottleneck at that point. Both hardware streaming and DMA-DMA streaming offer the potential advantage of *not* filling the CPU's data cache with I/O lacking locality.

OS Kernel Streaming refers to the DMA-DMA streaming model mentioned previously, but with the addition of software processing within the kernel's protection domain. I/O data passes through the CPU's data cache, which is advantageous for I/O data with locality and disadvantageous otherwise. To minimize manipulations and improve memory performance, Integrated Layer Processing (ILP) is used along the data flow. Although this approach offers general programmability across the data flow, processing must be performed in the protection domain of the operating system's address space.

User-Level streaming offers the greatest flexibility of data manipulation in the data flow as compared with the other techniques. Data flows through the kernel domain, through one or more user domains, and back through the kernel on output. All performance issues present in OS Kernel streaming are present with User Level streaming, with the added overhead of crossing protection domains while attempting to provide good performance. The authors begin from this point to develop a buffering method which is understood by both user and kernel software. This work eventually culminated later in the *fbufs* [DP93] design mentioned previously.

While the streaming approaches do suffer some of the problems the authors have illustrated, several observations can be made. First, the streaming approaches are not mutually exclusive. A system can make use of low-level (hardware) streaming when available and little intermediate processing is required, but may use higher-level streaming when appropriate or when processing is required. In addition, processing need not be done only within the user protection domain on the primary CPU.

The PPIO design can make use of low-level streaming when devices supporting such functionality are available. Data transformations to be introduced in the stream are first coded as algorithms and compiled for execution either in the main CPU or in an external attached processor. Hardware-streaming-capable devices are downloaded with user-provided code. Otherwise, OS streaming is used. The interface used in PPIO for introducing processing modules into the kernel's address space (or to devices) is discussed in Chapter 3.

2.4.2 Examples of Streaming Systems

As discussed above, streaming can take place at a number of different software levels of a system in addition to directly between hardware devices if appropriately designed. Although streaming in general has not received much attention in the literature, there a number of hardware examples and a few software examples. The following two sections present examples of hardware and software systems, respectively, which employ streaming.

Hardware Examples

Several hardware systems have been implemented with support for some variety of hardware streaming. One such system is the SCSI [NCR90] bus used by most workstations and Macintosh[2] computers. Computers supporting SCSI (*hosts*) are equipped with a *host adapter* which provides the translation between the computer's native bus and the SCSI standard bus. *Initiators* are responsible for initiating a data transfer and act as a master with

[2]Macintosh is a trademark of Apple Computer Inc.

respect to *targets*. Host adapters typically operate as initiators, requesting I/O for a number of targets.[3] Most peripherals are targets, and respond to requests from initiators. Peripherals supporting the SCSI Copy operation may achieve inter-device hardware streaming between an initiator and target.

IBM's MicroChannel bus [Bow91] represents another popular bus supporting streaming capability. This bus defines peer-to-peer transfers as occurring between two *bus masters*. One master acts as a controlling bus master, the other a slave. The interface for controlling peer-to-peer operation is known as the *Subsystem Control Block Architecture*,[4] and is not technically part of the Micro Channel Architecture. The capability has been tested at the University of Pennsylvania [ST93] within the AURORA [CDF+93] project.

Moving from the bus level to the system level, the notion of streaming has also been implemented successfully in the Auspex NFS file server product [NFE92]. This system embraces the notion of *functional multiprocessing* (FM). In FM, various specialized processing boards are attached by a VME bus modified for high performance. Data flows between processing boards across the VME backplane, and is not staged intermediately in system memory. A conventional UNIX-based workstation attaches to the VME bus to provide control—I/O data does not ordinarily pass through its memory or CPU.

Another architectural approach to system construction is known as the *desk area network* (DAN) [HD91, D.93]. This work is being undertaken at the Unversity of Cambridge London within the Pegasus project [LDS92]. The DAN employs an ATM switch fabric to interconnect devices on a user's desktop as an alternative to the interconnection of intelligent devices on a traditional system bus. Attached devices offer a range of programmability and are managed by an ATM-attached processor. The approach is also being investigated by researchers at MIT [AHIT94].

[3] Host adapters may also act as targets when more than one host is connected together on a SCSI bus (a configuration supported by SCSI, but not often used).

[4] SCB Architecture is a trademark of IBM

Software Examples

The CTMS system [PCMI91] supports multimedia applications on a UNIX system base. Recognizing the problem with multiple data copies, the `ioctl` system call is modified to support the ability to pass file descriptors from one device driver to another. The resulting interface supports the ability to create inter-device-driver transfers. Essentially it supports either DMA-DMA streaming or OS Kernel streaming, depending on the types of devices used. Unfortunately, this implementation is specific to the Token Ring Device drivers the authors worked with, and the interconnection architecture is described only briefly in one paragraph.

Another system which uses a streaming concept similar to that of CTMS is the *Cross-Bar-Interface* (CBI) [Tho93], built by Los Alamos National Laboratory for constructing wide-area HIPPI networks. The device is a two-board set, each consisting of two HIPPI interfaces, control logic, and buffering, and is controlled by an EISA-bus PC running BSD/386.[5] Among other functions, the CBI can be used for offloading network protocol processing for hosts not well-equipped for protocol processing such as the Thinking Machine CM-2. Such hosts communicate with the CBI using a simple protocol called SHIP. The BSDI system running on the PC includes a construct called a *looping socket*. The looping socket is used to interconnect SHIP sockets with TCP/IP sockets, thus affecting a protocol translator which is controlled by a user process but whose implementation is in kernel space.

Discussion

Streaming has been implemented in a number of hardware systems and a few software systems. Although the Auspex product has been a reasonably successful commercial product and represents an exception, hardware streaming capability remains largely un-utilized, most likely due to the lack of software and the lack of hardware with enough capability to both perform inter-device transfers and perform enough processing to allow data to be mutually understood between more than one device.

[5] BSD/386 is a trademark of Berkeley Software Design Inc. (BSDI).

The software efforts at streaming have been of an *ad hoc* nature, being used in specific circumstances with specific devices. These systems have not taken a new approach to I/O as PPIO aims to do, but rather have optimized data paths of particular interest to the developers. PPIO breaks with the traditional goals of I/O systems by focusing on inter-device (object) streaming support as opposed to conveying data between application and peripheral devices.

2.5 Modules and Protection

Its initial goal of PPIO was to support the interconnection of I/O devices through the operating system using streaming techniques. Such a system lacks the ability to perform any intermediate processing and thus would be inappropriate for many applications. By incorporating kernel-managed software processing *modules* along the data path from data source to sink, processing can be performed on I/O data without the need to invoke the application. Furthermore, module processing may take place outboard, on specialized processors that are locally attached, or possibly on other devices which are network-accessible.

Processing modules are described in Chapter 4. If module code is permitted to be introduced into the operating system by user processes, some mechanism should address the issue of security. Specifically, execution of errant or malicious module code could cause failure of the entire system. Section 4.3.2 of Chapter 4 addresses these concerns and describes related work in this area.

Chapter 3

PPIO Interfaces

This chapter is concerned with the interfaces used to interact with PPIO. The *user interface* represents the programming interface encountered by persons wishing to use the PPIO facilities from a user program. The *system interface* refers to the interface used by developers of system-level code needing to control or manipulate PPIO objects. The interface may also be divided into the portion relating to the establishment of flows and the portion relating to the manipulation of processing modules. Each portion has a user and system component. The overall PPIO internal structure and interfaces are depicted graphically in Figure 3.1.

3.1 User Interface

The user interface to PPIO provides the capability of establishing and tearing down *associations*, adjusting the rate at which data flows between sources and sinks, and specifying intermediate processing modules. The user interface consists of two primary parts: functions and *descriptors*. Descriptors are opaque handles used by user processes to identify kernel-resident objects.

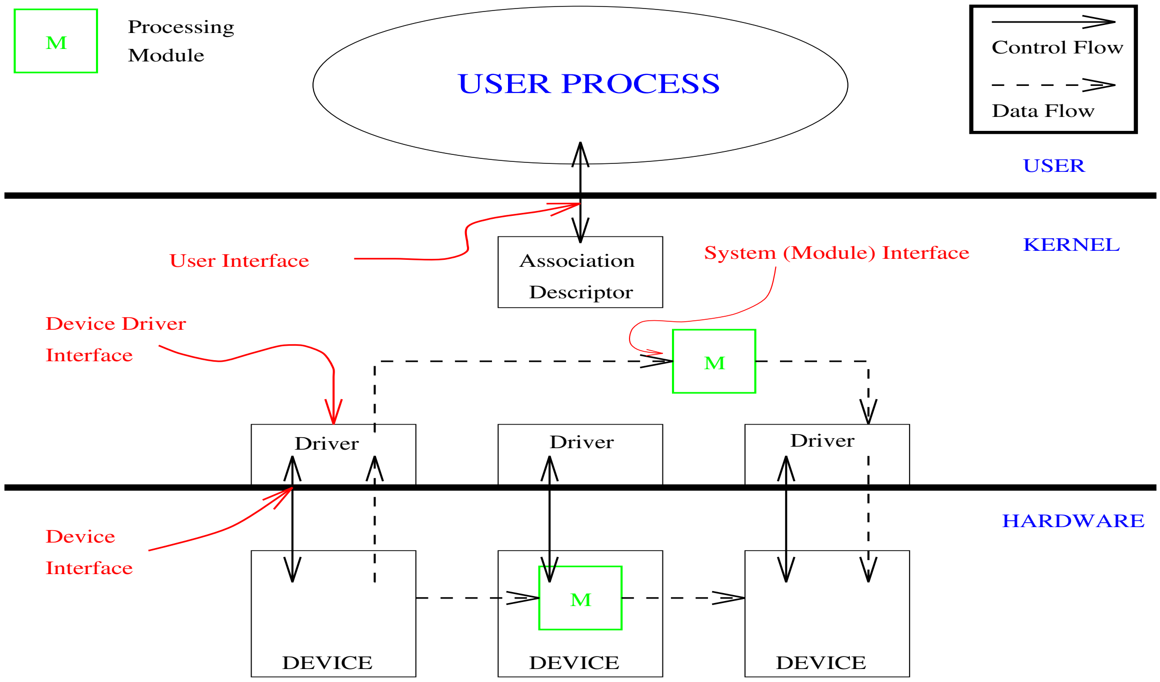

Figure 3.1: Location of interfaces in PPIO system.

3.1.1 Descriptors

Data sources and sinks must be identifiable to entities both at the user and system level. Most systems provide both a name (convenient for humans) and a *handle* or *descriptor* to refer to objects capable of performing I/O. Although names are convenient for user processes establishing initial associations between handles and accessible objects, they are generally not used beyond the initial association due to performance degradation caused by having to interpret character strings on each access. Most systems provide an abstract *descriptor* which refers to an object upon which I/O operations can be performed.

Descriptors are *opaque* to user processes, and refer to objects not directly accessible to user processes. Instead, they refer to underlying objects maintained by the operating system. Possession of a descriptor implies some set of access privileges to the underlying object. Common access privileges include *read* and *write*, and may also include *append*, *truncate* or *change owner*.

3.1.2 User Interface Functions

The user interface functions provide for user-level manipulation of associations and modules. This section describes the functions used to manipulate associations.

The `splice` Function

The PPIO architecture relies on the creation of kernel-maintained *associations* between data sources and sinks. User processes specify the source and sink endpoints of an association and the operating system manages data transfers between them. The function of greatest importance is called `splice`, and it is the primitive used by user processes to create associations. Using C-like notation, it is called as follows:

```
/*
 * "file_descriptor"s refer to I/O objects and
 * "splice_descriptor" refers to the association
 * between them.  In many systems these will
 * be equivalent types
 */

        file_descriptor   dsrc, dsink;
        splice_descriptor dnew;

        dnew = splice(dsrc, dsink);

        if (dnew == ERROR) {
           /* handle error */
        }

        /*
         * now dnew refers to the association
         * between dsrc and dsink
         */
```

In the above code fragment, `dsrc` and `dsink` represent descriptors referencing I/O objects. `Dsrc` and `dsink` refer to the underlying source and sink objects, respectively. After a successful completion of the `splice` call, `dnew` represents a handle of an association or "splice" and will be referred to as the *splice descriptor*. No data is flowing at this point; the `splice` call is used to establish the association and allocate any necessary kernel-resident data structures.

Once an association is created with `splice`, it may be manipulated in a number of ways. For generality, manipulations are performed by writing control messages to the splice descriptor. Thus, a sequence of `write` operations on the splice descriptor may be used to change the state of the splice, and `read` operations may be used to gather status. Performing a `close` operation on the splice descriptor removes the association from the system.

Controlling an Association

Control information relating to an association is exchanged between user and kernel environments by means of a *splice control message* data structure. This structure is represented in C-like notation as follows:

```
struct SPLICE_CTRL_MESSAGE {
        unsigned int       op;        /* operation */
        int                increment;
        int                cnt;
        void               (*handler)();
        unsigned int       stat;
 };

 #define SPLICE_OP_STARTFLOW         0x01
 #define SPLICE_OP_STOPFLOW          0x02

 #define SPLICE_INCREMENT_DEFAULT    -2
 #define SPLICE_INCREMENT_INFINITE   -1
 #define SPLICE_INCREMENT_NOOP       0

 #define SPLICE_HANDLER_NONE          ((void (*)()) -2)
 #define SPLICE_HANDLER_DEFAULT       ((void (*)()) -1)
 #define SPLICE_HANDLER_NOOP          ((void (*)()) 0)

 /* see section on Association Descriptors */
 #define SPLICE_STAT_ERROR            0x01
 #define SPLICE_STAT_ACTIVE           0x02
 #define SPLICE_STAT_WANTED           0x04
```

This structure provides a consistent way of manipulating the condition of a splice. Operations are specified in the `op` field by specifying one of the operations (defines beginning with `SPLICE_OP_`). The `SPLICE_OP_STARTFLOW` operation indicates the user process wishes data to begin flowing across the association. `SPLICE_OP_STOPFLOW` indicates the converse and is used to stop the flow of an association. New operations may be added as needed.

The `increment` field specifies the number of bytes to transfer across an association before returning control to the calling user application. When control is returned, data flow is stopped. A `SPLICE_OP_STARTFLOW` must be executed to restart data flow. The increment represents an important concept in PPIO, and refers to the amount of data the user process is willing to have transferred by the operating system on its behalf. In effect, it specifies the *level of delegation* the user process is willing to give to the system. Specifying `SPLICE_INCREMENT_DEFAULT` indicates the system should choose an appropriate increment. This will generally be a buffer size deemed convenient by the operating system. `SPLICE_INCREMENT_NOOP` indicates the increment should not be changed. Specifying `SPLICE_INCREMENT_INFINITE` indicates to the system the user process is willing to delegate the entire data flow to the operating system. Data will continue to flow between a source and associated sink until EOF is reached on the source, an error occurs, or the calling process is interrupted.

Associations operate synchronously be default, blocking the user process until `increment` bytes of data have been transferred. An association may also operate asynchronously by returning control to the calling process immediately, and optionally notifying the application by way of an upcall [Cla85]. The `handler` field is used to specify the entrypoint of the routine in user-space to be invoked during the upcall. Specifying `SPLICE_HANDLER_DEFAULT` indicates the system should execute a default handler, which should raise an error condition. `SPLICE_HANDLER_NONE` indicates no handler is to be used, thus effectively disabling the upcall. `SPLICE_HANDLER_NOOP` indicates the system should not alter the present handler entrypoint.

Given an existing association (created by a call to `splice`), the standard `write` call may be used to control the association. Here is a simple example:

```
/* dnew is a splice descriptor */

struct SPLICE_CTRL_MESSAGE      sm;

sm.op = SPLICE_OP_STARTFLOW;
sm.increment = SPLICE_INCREMENT_DEFAULT;
sm.handler = SPLICE_HANDLER_NONE;

if (write(dnew, &sm, sizeof(sm)) != sizeof(sm)) {
        /* handle error */
}
```

In this example, the control message `sm` is filled in with the instruction to commence flow on the splice with the default `increment`. `SPLICE_HANDLER_NONE` in the `handler` field indicates no routine should be called upon I/O completion. This is a synchronous splice; providing a handler would not be useful (and would not be used).

When a `SPLICE_OP_STARTFLOW` operation is given to a splice, the `write` call initiating the control message blocks until an amount of data equal to the increment is transferred, EOF is reached, or an error occurs. In the case of an asynchronous splice, the `write` call returns immediately, and the specified handler is invoked when the amount of data specified by the increment is reached, EOF occurs, or an error is encountered. An asynchronous splice is illustrated below:

```
/* dnew is a splice descriptor */

struct SPLICE_CTRL_MESSAGE      sm;
void                      splice_handler();

fcntl(dnew, F_SETFL, FASYNC); /* set async flag */
sm.op = SPLICE_OP_STARTFLOW;
sm.increment = SPLICE_INCREMENT_DEFAULT;
sm.handler = splice_handler;
```

```
        if (write(dnew, &sm, sizeof(sm)) != sizeof(sm)) {
                /* handle error */
        }
        ...

splice_handler()
{
        static struct SPLICE_CTRL_MESSAGE continue_msg = {
                SPLICE_OP_STARTFLOW,
                SPLICE_INCREMENT_DEFAULT,
                splice_handler
        };
        static int sz = sizeof(continue_msg);
        if (write(dnew, &continue_msg, sz) != sz) {
                /* handle error */
        }
}
```

In this example, the `fcntl` call changes the splice to perform asynchronously. The `write` call in the main section returns immediately, and when the default increment number of bytes have been transferred between the data source and sink, `splice_handler` is invoked. At this point, the data flow is interrupted. A subsequent call to `write`, this time in the handler, causes another default increment number of bytes to flow through the splice. The user may specify an infinite increment (`SPLICE_INCREMENT_INFINITE`), in which case the splice will continue moving data until an error or EOF is reached.

Status of an Association

To gather status from a splice, the `read` call may be used on the splice descriptor. It operates using the same `SPLICE_CTRL_MESSAGE` structure used by the `write` call. It is called as follows:

```
/* dnew is a splice */

struct SPLICE_CTRL_MESSAGE      sm;

if (read(dnew, &sm, sizeof(sm)) != sizeof(sm)) {
        /* handle error */
}
```

```
printf("splice status: 0x%x\n", sm.stat);
printf("bytes transferred: %d\n", sm.cnt);
```

In this example, the `sm.stat` field indicates the status of a splice as described as given above. The status field is an array of bits indicating various conditions, and may be extended as needed. Active splices have data flowing across the association, as indicated by the `SPLICE_STAT_ACTIVE` bit being set. Splices which have encountered an error have the `SPLICE_STAT_ERROR` bit set, and those which have other processes or threads awaiting their completion have the `SPLICE_STAT_WANTED` bit set. Status bits may not be modified by user programs; they are read-only.

The `sm.cnt` field indicates the number of bytes which have been transferred across the association thus far. This field may be used by applications to verify or inspect the progress of an active association.

3.1.3 Module Manipulation

Associations may be created using the interface provided in the previous section. Creation of an association does not imply any data transformation across the association, although it may imply conversion between buffering types. To provide the flexibility of incorporating processing between a connected source and sink, processing modules may be introduced between an associated source and sink. As mentioned previously, the PPIO interface for manipulating processing modules is based heavily on Streams (originally proposed by Ritchie in [Rit84]). Streams has become a *de-facto* standard since the release of ATT UNIX SVR3. Its most recent description may be found in the ATT UNIX SVR4 documentation [USL92].

Although the entire Streams interface will not be included here, the details relevant to PPIO shall now be described. In Streams, each processing module is a kernel-resident set of two *queues*, and a set of interface functions. The modules form a pipeline connecting drivers to user processes. Modules with fan-in or fan-out exceeding one are known as *multiplexors*. Modules are added or removed from a pipeline in a LIFO fashion, and are thus known as *stackable modules*. The "stack" is maintained such that the module closest to the user process is considered the topmost module.

To be effective in supporting data manipulation in PPIO, the user interface for manipulation of modules must provide the following functions:

- pushing a module on top of a stream
- removing a module from the top of a stream
- (de)associating multiplexors with neighboring modules

The method for adding a module to the top of the stream "stack" is accomplished by the `I_PUSH` *ioctl* call:

```
file_descriptor fd;

if ((fd = open("device_name", O_RDWR)) < 0) {
        .. error 'device open' failed ..
}

if (ioctl(fd, I_PUSH, "module_name") < 0) {
        .. error 'push' failed ..
}
```

The method for removing the topmost module from the stack is accomplished by the `I_POP` *ioctl* call:

```
/* fd is a stream descriptor */
if (ioctl(fd, I_POP, NULL) < 0) {
        .. error 'pop' failed ..
}
```

The methods for establishing relationships between streams and multiplexors is considerably more complicated. The `I_LINK` *ioctl* call is used to add streams to one side of a multiplexor, and the `I_UNLINK` *ioctl* call is used to uncouple streams from a multiplexor. The reader is referred to the Streams Programmer's Guide [Sun92] (Chp. 10), for more details on the use of multiplexors.

Although the specific Streams interface is somewhat UNIX-specific (due to the use of the `ioctl` call above), it is easily generalized by the addition of appropriate library or system calls to other operating system environments.

3.1.4 Dynamic Loading

Dynamic loading refers to the ability to introduce code modules into a protected address space. It has been used for selectively loading device drivers based on what hardware is available. With respect to PPIO, dynamic loading provides the ability to introduce code compiled and linked in user space into the running operating system's protected address space. Several current operating systems support dynamic loading, including Linux's *modules*, OSF's `ldr_xload` interface, SunOS's `modload` facility and AIX's `sysconfig` subroutine [IBM92]. Although many systems provide this capability, no standard presently exists for the user interface.

The `kload` Function

The functionality required by PPIO is similar to that provided by many of the systems mentioned above. To support push and pop operations on modules, PPIO requires the following capabilities:

- load a module into the kernel's address space
- name the module
- return a reference for the module
- unload a module

In PPIO, the following call, `kload`, provides the needed support:

```
mod_descriptor    md;

md = kload("module_name", "location of binary containing code");
if (md < 0) {
     ... error loading ...
}

close(md);   /* unloads module */
```

In the example above, `kload` is used to load user-level code into the operating system environment. Note that with appropriate hardware support, `kload` may actually load code to an attached peripheral device rather than the kernel's address space. The descriptor `md` is a *module descriptor* and is used to refer to the particular module just introduced into the operating system.

The second argument to `kload` specifies the implementation code for the module being loaded. If the second argument is `NULL`, `kload` searches the current state of the operating system for the named module. It will return a valid module descriptor provided the named module has already been loaded either by a previous call to `kload`, or initially at operating system build time (as would be the case for conventional Streams modules, for example). If the second argument is non-NULL, it specifies a file system object (e.g., file name) indicating which code file should be loaded into the operating system. This method parallels that used by the existing systems mentioned above.

The module descriptor works similarly to the splice descriptor previously described in that control messages may be exchanged between kernel-resident modules and a controlling user process by applying `read` and `write` calls to the module descriptor. Modules in PPIO do not generally have as long a lifetime as modules in conventional Streams, and are thus expected to be moved into and out of the operating system more frequently. Modules and associations introduced by a process are removed prior to process exit completion.

3.1.5 Combining Splice, Streams, and Modules

The previous sections have described the user interfaces to PPIO, the stacking functions of Streams, and the `kload` interface used to load processing modules into the operating system. These interfaces may be used in combination by applications to create data paths managed by the operating system which includes in-band processing but does not need to pass through the application's address space. The following example illustrates how a user process might use PPIO to create a data flow between `source_device` and `sink_device` which employs an intermediate encryption processing module (error checks have been removed for clarity) :

```
#define MOD_NAME "des_encryption"
#define MOD_FILE "/usr/kmodules/crypt_des.o"

descriptor            sourcedev, sinkdev;
mod_descriptor        mdesc;
splice_descriptor     sdesc;
SPLICE_CTRL_MESSAGE   sm;

static char mykey[] = "encryption_key";

/* obtain handles to source and sink devices */
sourcedev = open("source_device", O_RDONLY);
sinkdev = open("sink_device", O_WRONLY);

/* load the module of interest */
mdesc = kload(MOD_NAME, MOD_FILE);

/* push the module on the destination device's stack */
/* (could also have used source device's */
ioctl(sinkdev, I_PUSH, MOD_NAME);

/* interact with module-- in this case, provide crypto-key */
write(mdesc, mykey, sizeof(mykey));

/* create association between source dev and sink stack */
sdesc = splice(sourcedev, sinkdev);

/* fill in splice descriptor, used to start data flow */
sm.op = SPLICE_OP_STARTFLOW;
sm.increment = SPLICE_INCREMENT_INFINITE;
sm.handler = SPLICE_HANDLER_NONE;

write(sdesc, &sm, sizeof(sm));
```

In the example above, the source and sink devices are opened and are referenced by the `sourcedev` and `sinkdev` descriptors, respectively. The executable image of the encryption processing module is contained in the file `/usr/kmodules/crypt_des.o`. It is introduced into the system by the call to `kload` which names the module ("des_encryption"). In cases where the module is precompiled into the operating system, the first argument to `kload` is the name of the precompiled module, and the second argument is `NULL`. A successful call to `kload` is indicated by a positive return value.

At this point, the user process contains descriptors for the source and sink devices, plus the processing module. The "des_encryption" module becomes associated with the sink device by the `I_PUSH` *ioctl* call. In this case, the module could have been pushed on either the source or sink device's descriptor. The encryption module is an example of a module which requires control information to operate; it requires a secret key to perform encryption. Control information is passed to the module by way of the standard `write` call applied to the module descriptor. After successful completion of the `write` call, the application calls `splice` to connect the source and sink descriptors together. After the splice is created, a `write` operation on the splice descriptor using the `STARTFLOW` operation commences data flow. Data flows from the source device, through the splice, through the encryption module, and out to the sink device. The infinite increment in the example implies the write will block until data flow is complete.

3.2 System Interface

The PPIO *system interface* refers to the programming interface used by modules executing within the operating system's protected address space.[1] The purpose of the interface is to address two primary goals:

1. provide inter-module data flow
2. provide control and status to user processes

The system interface is designed based on an amalgam of elements taken from the original Streams design [Rit84] as extended by Plan 9 [Pre90], as well as the present UNIX device driver interface.

3.2.1 Streams Modules in Plan 9

In the case of Streams modules, the Plan 9 `Queue` structure contains an external interface consisting essentially of only the `put` routine. The entire structure is given

[1] Strictly speaking, these interfaces could also exist in other user processes for systems employing user level servers for OS services. In addition, such interfaces could be used for modules executing in peripheral devices (i.e. outboard processors).

in [Pre90] as follows:

```
typedef struct Queue Queue ;
struct Queue {
     Blist;                       /* linked list of blocks */
     int nb;                      /* # blocks in queue */
     Qinfo *info;                 /* line discipline defn */
     Queue *other;                /* opposite direction */
     Queue *next;                 /* next queue in stream */
     void (*put)(Queue*,Block*);  /* "put" procedure" */
     Rendez r;                    /* flow ctl rendezvous point */
     void *ptr;                   /* queue's private data */
};
```

The structure given here for Plan 9 includes enough fields to invoke the “put” procedure, discover the next downstream module, discover the peer “opposite-direction” module, maintain a block list, maintain a link to an object common to all modules in a Stream (the `info` field), deschedule and restart a module (the `r` field), and keep private data. The external interface of this structure is simply the `put` procedure, although the `next`, `nb`, and possibly `Blist` fields may be of interest to neighboring modules or the rest of the operating system.

3.2.2 UNIX Device Driver Interface

Typical current UNIX systems provide two primary interfaces to device drivers, depending on the type of device and abstraction being supported. *Character devices* represent the bulk of devices, including serial lines, (raw) disks, multimedia devices, etc. These devices are typically accessed directly by user processes. The other primary category of devices are *network devices*. These devices are typically not read from or written to *directly* by user processes, but instead communicate with kernel-resident protocol implementations.[2]

The UNIX interface to character device drivers is given by a structure known as `cdevsw`. This structure is given as follows:

[2]The other type of device drivers are for *block* devices. These device drivers employ a caching buffer system and are used primarily in support of file systems.

```
struct cdevsw {
        int     (*d_open)();      /* open driver */
        int     (*d_close)();     /* close driver */
        int     (*d_read)();      /* read data */
        int     (*d_write)();     /* write data */
        int     (*d_ioctl)();     /* control device */
        int     (*d_reset)();     /* reset device (if used) */
        int     (*d_select)();    /* await I/O */
        int     (*d_mmap)();      /* map to user space */
};
```

Although the structure listed above is visible in kernel address space only, the functions supported are very similar to the I/O calls available to user processes in most UNIX systems (`open, close, read, write, ioctl, select, mmap`).

As mentioned above, drivers for network devices are distinguished from character devices because of the way they are accessed by user processes. They use the following structure:

```
struct ifnet {
        char       *name;                /* intf name */
        ...
        /* procedure handles */
        int        (*if_init)();         /* init routine */
        int        (*if_output)();       /* output routine */
        int        (*if_ioctl)();        /* ioctl routine */
        int        (*if_reset)();        /* bus reset routine */
        int        (*if_watchdog)();     /* timeout routine */
        ...
};
```

The first obvious difference between these two structures is the omission of the `close, read, select,` and `mmap` routines in the network interface. The `close` routine is not used because user network interfaces are not initialized or freed as a result of user process action. Instead, interfaces are typically initialized at system startup time (at which time the `if_init` routine is called) and are not cleared (thus no need for a `d_close` routine). The `read` routine is not present because of the active output nature of network devices. Such drivers are invoked via an interrupt thread (upcall), and instead call the "input" routine of the appropriate higher-layer protocol. The `d_select` and `d_mmap` routines are not present because these are present only in support of the corresponding user process calls `select` and `mmap`.

3.2.3 The PPIO Module System Interface

By combining attributes of the interfaces described above, the PPIO module interface is constructed as follows:

```
struct module_extern_interface {
    int        (*m_put)();
    int        (*m_open)();
    int        (*m_close)();
    int        (*m_ctlread)();
    int        (*m_ctlwrite)();
    int        (*m_select)();
};

struct module_internal_state {
    struct module_extern_interface *next; /* next module */
    Module_ctrl_block mcb;                /* thread state */
    Blist bl;                             /* list of blocks */
    int nb;                               /* # bytes here */
    int nmsgs;                            /* # messages here */
};
```

This interface addresses the two primary goals indicated above. Inter-module simplex data flow is achieved by calling the `put` procedure from an adjacent module. Note that there is no "opposite direction" module reference due to the simplex character of modules in PPIO. Control of modules is provided to user processes in several ways.

The `m_open` routine is invoked as a result of a user process' call to `kload`. This routine is used to initialize any necessary data structures and notify the operating system of the presence of the module. The `m_close` routine is invoked when the user process executes a `close` call on the module descriptor. The result of a `close` call on a module descriptor is the removal of the specified module from any active data flow. Any module resources may be released if the system is loaded, or the module may remain in the system in hopes that it may be re-used in the future. This is known as *module caching*.

The other module interface routines deal with modules which have already been introduced into the operating system and initialized. The `m_ctlread` and `m_ctlwrite` calls support exchange of control information between kernel-resident modules and applications running in user space. As illustrated above, a module-specific protocol is established between user-space caller and the referenced module. A user's `write` of a control message results in the execution of a `m_ctlwrite` function in the referenced module. The module is responsible for interpreting the message provided by the user. A user's `read` call results in the execution of a module's `m_ctlread` function, and is used to provide status information originating at the module and destined for the user process. In support of a user process `select` call, the `m_select` call is used for synchronous multiplexing of module descriptors. Since module descriptors are only used for the exchange of control information, applying `select` in this fashion can be used to multiplex control information to/from multiple modules.

3.2.4 The PPIO Driver Interface

The driver interface is a system interface used to support the PPIO system and to program devices for I/O operations. Device drivers generally represent a substantial fraction of the code comprising modern operating systems, and are generally not easy to modify, write, or rewrite. PPIO is conceived to be integrated with an already-existing I/O subsystem, and therefore this section will deal with those requirements PPIO makes of device drivers without encouraging an entirely new driver architecture.

Drivers are generally divided into two sections, the *top half* and *bottom half*.[3] The top half is invoked synchronously, usually on behalf of a currently-executing user process or thread. The bottom half is invoked asynchronously, and is generally an *upcall* which is not necessarily related to the currently-executing user process.

[3] This terminology is common to both OS/2 an UNIX operating system, and possibly others.

Upcalls and Demultiplexing

Upcalls initiated due to a device interrupt must be demultiplexed to determine the destination of data which has been acquired during an I/O operation (e.g., data acquired as the result of a `read` operation). Requested I/O operations are typically encapsulated in a data structure which identifies the destination of the I/O data. In addition, the structure may contain a *handler* which is invoked to complete the upcall. For example, the UNIX `buf` structure is used to perform block-oriented I/O operations to disk:

```
struct buf {
    ... linked list fields ...
    long         b_flags;     /* too much goes here to describe */
    long         b_bcount;    /* transfer count */
    long         b_bufsize;   /* size of allocated buffer */
    ...
    daddr_t      b_lblkno;    /* logical block number */
    daddr_t      b_blkno;     /* block # on device */
    ...
    struct proc *b_proc;      /* proc doing physical or swap IO */
    int          (*b_iodone)(); /* function called by iodone */
    ...
};
```

In this example, the `buf` structure includes meta information about a disk block used when I/O is performed on a buffer. The link fields are used when buffers are linked together in a hash table using chaining, which forms the UNIX *buffer cache*. Write I/O is generally initiated after a user process request by a mechanism known as *delayed write*. Delayed write attempts to keep disk blocks cached under the expectation they will be modified again before a physical disk I/O is performed.

When physical I/O is performed on a block, the `b_iodone` function is invoked upon completion. In the case of the buffer cache, I/O completion may signal the need to awaken a blocked user process and perhaps free some data structure locks. There is no provision for completing the upcall in such a way as to schedule a corresponding downcall. In most systems, the `b_iodone` handler does no noteworthy work.

The example illustrates one mechanism which may be used to introduce I/O demultiplexing. In this case, demultiplexing is *dynamic* in that a different routine may be called on a per-buffer basis by modifying the `b_iodone` field. In the case of UNIX network protocol implementation the demultiplexing is *static* because incoming network packets are delivered to a fixed set of queues, based on a type field included in packet headers. For PPIO, a device driver's upcall must ultimately result in a lookup to determine whether an association is present for the data that triggered the upcall. By providing a reference in each I/O request, the required demultiplexing is easily accomplished.

Code Download

A unique characteristic of PPIO is its capability to take advantage of devices which support outboard processing to avoid data movement through main memory. When such devices are available, they must be programmed in such a way as to interact directly with their peers over a shared system bus. Such devices can be PPIO sources or sinks, or can be specialized processors which implement module functions. In either case, such devices execute code provided by the operating system or specified by user processes.

The user interface described in Section 3.1 remains fixed whether code modules are executed in the kernel's address space or in an external device. The `kload` call results in a call to the module's `open` routine, which is responsible for causing its own executable image to be loaded into the appropriate device. The interface used to accomplish this loading is specific to whichever operating system environment is used to implement PPIO. In the case of UNIX, the addition of a new `ioctl` for those devices supporting code download would require minimal modification to existing device drivers.

Chapter 4

PPIO Processing Model

This chapter describes the PPIO processing model. Section 4.1 provides the operating system concepts and background necessary for understanding the PPIO architecture. Section 4.2 describes the processing and flow control algorithms employed to construct inter-object data flows, and Section 4.3 describes the execution environment for processing modules.

4.1 Definitions and General Concepts

This section reviews the essential operating system abstractions and objects used in the PPIO design. First, data sources and sinks are described, followed by processing abstractions. A discussion of flow control and its effects concludes the section.

4.1.1 Sources and Sinks

Data moved within an operating system is described as being originated at a data *source* and consumed at a data *sink*. The terms source and sink are abstract descriptions, and may be applied equally to either hardware or software entities. Movement of data from a source to an associated sink is generally facilitated by some device with processing capability; common examples are CPUs (for copying data) and DMA units (used to stream data between system memory and hardware peripheral devices). DMA is usually preferred

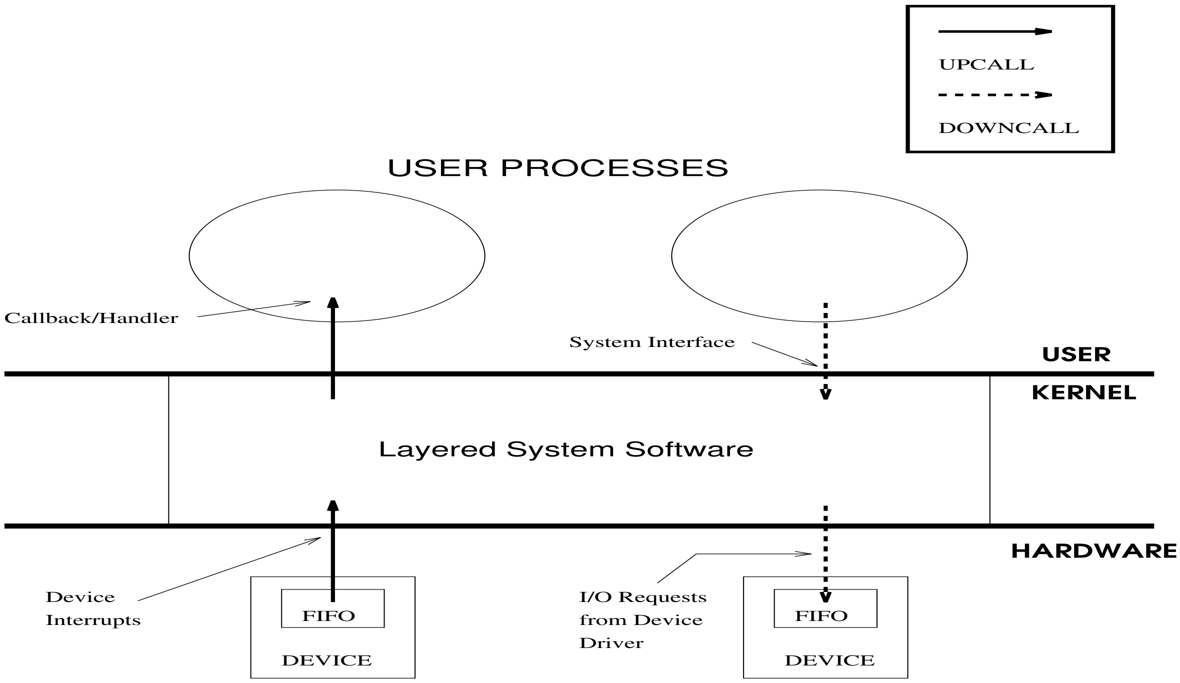

Figure 4.1: Upcalls and Downcalls in I/O System Structure

to programmed I/O because it allows for parallelism (in-cache CPU execution during simultaneous device I/O) and often offers superior bus transfer rates.

4.1.2 Upcalls and Downcalls

Call direction refers to the relative orientation with respect to software layering between a calling and called procedure. Calls originating near the "top" of the system software and proceeding downward are called *downcalls*, and those originating from hardware (usually interrupts, traps, and exceptions) are known as *upcalls*. Figure 4.1 illustrates the concepts of upcalls and downcalls. The illustration depicts upcalls completing ultimately at the user process, although many upcall schemes terminate upcalls within the kernel, and do not reach the user process layer. Systems such as Swift [Cla85] and X-Kernel [HP91] perform upcalls all the way to user space.

4.1.3 Processes and Threads

The term *process* has been used informally above. To be more precise, a process generally includes a single execution abstraction existing in its own unique address space. To support multiprocessing, an operating system performs *context switches* between processes (and hence address spaces) to simulate the concurrent execution of more than one activity. When a multiprocessor is available, multiple processes may actually run simultaneously, requiring locks for shared data structures and re-entrant shared code libraries.

When a process executes a downcall requesting I/O, the requested I/O data must generally be fetched from some I/O device. At this time the process is usually *blocked* (made to wait) until the requested data is obtained. When a process is blocked, it is not capable of doing further useful work until its I/O data is obtained, and the system responds by descheduling the blocked process and allowing another process to execute (by performing a context switch). The blocked process is *resumed* (allowed to continue execution) when its I/O data is available.

Creating processes in unique address spaces for concurrent tasks can be time consuming. Furthermore, highly cooperative concurrent tasks implemented as separate processes must employ IPC to communicate, often at significant performance cost. The operating systems community has responded to these issues by introducing a number of additional models of local (same machine) execution: coroutines, threads (user and kernel-supported), lightweight processes, continuations, LRPC, etc. Such abstractions exist within the following degrees of freedom:

- degree of parallelism
- amount of overhead
- protection (isolation)

Of the execution abstractions listed above, *threads* have become most popular. Threads generally allow the concurrent existence of more than one execution abstraction within the same address space. Sharing of data structures between threads is zero-cost, except for synchronization. In addition, when threads are kernel-supported, blocking of one thread need not block additional threads in the same address space. Such an arrangement provides the ability for a requesting program to continue execution when awaiting I/O. *Asynchronous I/O* generally refers to the ability to simultaneously handle other tasks while awaiting I/O. Threads provide a mechanism to implement asynchronous I/O.

Downcall execution of processes or threads generally operates in the *top half* of the system software. While executing in the top half, a process context is available, and blocking may occur by stopping the execution of the process and switching to another as mentioned above. Furthermore, resources such as a process' stack may be conveniently located, by identifying the presently-executing process. Execution in the *bottom half* refers to operations invoked on behalf of an asynchronous event, typically a hardware interrupt. Bottom-half execution is invoked at arbitrary points in time, and is generally unrelated to the presently executing process(es). Furthermore, bottom-half upcalls are not allowed to block. Synchronization between bottom and top halves is often achieved by disabling interrupts during critical code sections in the top half.

4.1.4 Flow Control

The execution abstraction is closely linked with the method employed to achieve flow control. *Flow control* refers to the task of speed-matching a fast data source with a slower data sink. Blocking is often used to achieve flow control for both intra-machine and inter-machine communication. A source is blocked until an associated sink is able to consume some amount of outstanding data. Reliable protocols like TCP [Pos81] include provisions for delivering flow–control information, used to control data sources, across a network. For most systems, a source/sink combination with a faster sink requires no flow control. Temporally sensitive systems, however, (e.g., multimedia delivery) may need to attenuate source data production rate even when a faster sink is available. Generally, when a source produces data at a rate in excess of the sink's consumption rate, and the source rate cannot be adjusted, flow control is not possible. Loss of flow control results in data loss, which is intolerable in traditional systems, but may be acceptable in some current application domains (e.g., *loss-tolerant* multimedia applications).

4.2 Processing and Flow Control in PPIO

This section describes how flow control is implemented in PPIO. The first subsection provides a set of attributes used to categorize devices. These attributes are used to dictate the ways in which differing types of I/O objects may be slowed to achieve flow control.

4.2.1 I/O Object Attributes

In this section, an "I/O object" refers to any hardware or software entity capable of performing I/O. I/O objects may be classified by defining a set of boolean *I/O attributes* which describe the way I/O may be performed on most objects:

- active or passive output
- quenchable
- synchronous writes
- synchronous reads

An object with *active output* is any object capable of sourcing I/O data asynchronously (i.e., without an associated request). Thus, the term *output* is used from an object's point of view. Typical active objects include network interfaces or transport layer network connections (or datagrams for non-connection-oriented transport layers), in addition to periodic devices like video or audio digitizers. These objects produce data without a `read` operation from the operating system. An object with *passive output* performs I/O only after a corresponding request for data. Active and passive devices are assumed to be capable of asynchronous I/O unless specifically identified as only supporting synchronous reads or writes (below). The characterizations of passive and active I/O are derived from by Black in [Bla83].

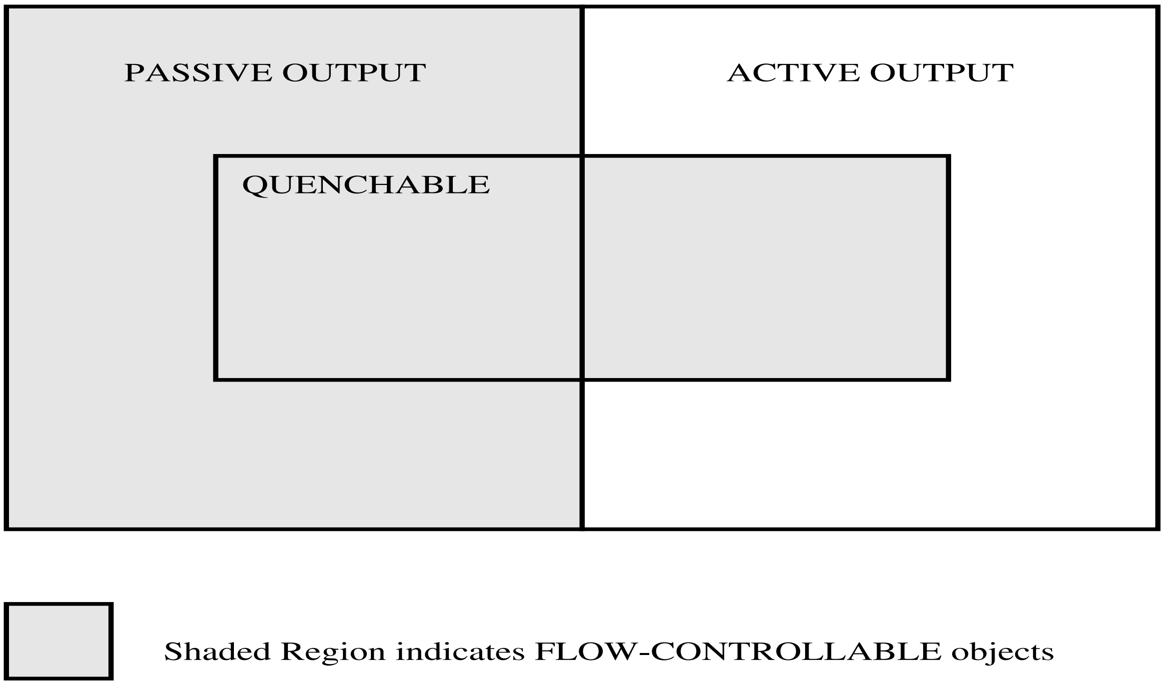

Figure 4.2: Flow-Controllable I/O Objects

Black describes how traditional operating systems support only *active operations* for user-process I/O. Active I/O operations are the familiar `read()` and `write()` operations. In his description of the Eden system, he observes that data may flow between a source and sink not only by employing the traditional active I/O operations, but also by either creating a system with no active output operations or no active input operations (these latter situations are duals). Furthermore, such a system requires roughly half as many data movement operations to be performed as compared with traditional systems. Although his description is accurate and would otherwise be a sufficient basis for describing objects in PPIO, the PPIO system must deal with not only abstract software objects but also hardware *devices*, which may not in general be assumed to fit into either category. Thus, the following attributes augment the active and passive I/O descriptions given in [Bla83].

Quenchable objects are those I/O objects whose data production rate may be adjusted when acting as a source by means other than the traditional method of blocking reads (usually by adjusting some "quality" parameter resulting in a smaller throughput source rate). Quenchability is discussed in more detail below in Section 4.2.3. An I/O object is said to be *flow-controllable* if it is either quenchable or passive. This relationship is illustrated in Figure 4.2. For passive objects, flow control is achieved by reducing the read request rate.

I/O objects requiring *synchronous writes* demand the CPU copy data to the object. Any device supporting only programmed I/O is considered to require synchronous writes. In addition, common objects such as memory or files are characterized by synchronous writes.

Those I/O objects requiring *synchronous reads* cannot perform data movement out of the object without CPU intervention. Programmed I/O devices require synchronous reads. Note that any passive object is also flow-controllable, as data production rate can be arbitrarily modulated by reducing request rate. Table 4.1 illustrates these attributes for a number of common I/O objects.

Table 4.1: Attributes for some common I/O Objects

Object	Active Output	Quenchable	Sync Writes	Sync Reads
file			√	√
UDP dgram	√		√	
TCP data	√	√		
UART	√	√	√	√
video dig	√	√		
disk ctlr		√		
net intf (PIO)	√		√	√
net intf (DMA)	√			
memory			√	√

4.2.2 Generic Source and Sink Procedures

In PPIO, I/O data can flow between any combination of sources and sinks described above. In general, asynchronous I/O is used when available. For those objects requiring synchronous I/O operations, kernel-level threads are employed. In general, all PPIO sources execute the source algorithm depicted in Figure 4.3 and all PPIO sinks execute the sink algorithm given in Figure 4.4. Flow control is achieved by executing the flow algorithms discussed in Section 4.2.3.

In the generic routines provided in Figures 4.3 and 4.4, source objects not requiring synchronous reads utilize upcalls to facilitate data flow by invoking the `read_handler`. When an upcall is executed, the corresponding sink object is looked up and a downcall is performed to the `sink` routine. On the sink side, the situation is reversed. When synchronous writes are required, the `sink` routine loops performing individual `write` operations until data transfer is complete. If asynchronous writes are possible on the sink object, the `write_handler` routine is invoked on write completion, and the process repeats itself.

4.2.3 Flow Control and Quenchability

In PPIO, objects identified as *flow-controllable* may be forced to reduce their source rate. Generally, when non-flow-controllable objects are data sources, flow control cannot be guaranteed, resulting in the possibility of data loss. As stated previously, data loss may be tolerable for certain classes of applications.

Intuitively, flow-controllable objects include all passive objects in addition to all objects directly supporting flow control (e.g., serial lines with hardware flow control, transport-layer protocol connections providing software-based flow control). In addition, flow-controllable objects include all quenchable objects. For example, a video digitizer might operate at a constant rate of 30 frames per second, yet be able to adjust its resolution, and thus the number of bytes per frame sourced. A digitizer of this variety would be considered quenchable, and thus flow-controllable.

Generally, flow-control may be accomplished by stopping a process or thread performing `read` requests on source objects, or by adjusting the throughput demands of a source object (i.e., for quenchable objects). The flow control algorithm given in Figure 4.5 completes the `source` and `sink` algorithms given above. A return code of `DONE` from the flow algorithm indicates a source should cease data production. The `CONTINUE` return code data production should be continued (i.e., the source does not require flow control).

For sources supporting asynchronous reads, upcalls may be used to facilitate data flow. Upcalls from `read_handler` are converted into downcalls to `sink` which request or perform I/O on the sink. For asynchronous sinks, an upcall from `write_handler` is converted into a down call to `source`. The overall structure creates a "ping-pong" effect in which control is exchanged cyclically between source and sink.

For data flow between objects requiring synchronous reads or writes, a thread of control is required to execute the `source` or `sink` routines. This thread may block as needed. Several systems provide kernel-level threads capable of such operation (see [ABB+86] for such an example). Systems lacking direct kernel-level threads typically support a *callout list* [KMQ89], where procedures can be queued for later execution [FP93].

4.3 Processing Modules

So far, this chapter has described a PPIO system with data sources and sinks. If sources are directly associated with a corresponding sink, a simple data routing function is performed. Such functionality may be useful for application layer gateways in networks and routing of uninterpreted multimedia data to display devices. When data flowing between an I/O source and sink object must be interpreted or transformed, intermediate *processing modules* (PMs) may be required. This section describes how PMs relate to PPIO.

4.3.1 Definitions

Many systems have incorporated some notion of a set of routines or processes (called PMs above) "connected together" in a linear *pipeline*. The pipeline transforms data originating at a source on its way to a sink according to the the transformations implemented in the PMs. Borrowing terminology from Pilot [Xer88], Figure 4.6 illustrates the fundamental concepts. Data originates at a source, where a *transducer* provides an interface to an I/O object on one side and a module interface on the other. *Filters* provide a module interface on both sides. They receive incoming data from an *upstream* (or previous) module, process it, and provide (possibly transformed) data to the next module in the pipeline.

Modules typically provide two distinct types of processing: *immediate* and *delayed*. Immediate processing occurs when received data takes the fastest path possible through the pipeline, often without buffering. Delayed processing typically implies processing occurs at a later time invoked by a clock-driven scheduler.

Several systems have been implemented with this basic design. In Streams [Rit84], immediate processing is invoked by *put* procedures and delayed processing is invoked by *service* procedures which run as coroutines. In [Lin94], immediate processing is called *in-band processing* and is invoked by *send* and *receive* procedures. Delayed processing or *out-of-band processing* is implemented in user space with the *callbacks* in the *tcl* [Ous90a] language.

4.3.2 Module Types: flexibility, security, and performance

The PPIO design is predicated on a desire to improve performance for operating systems providing protection. Much of its benefits derive from its ability to handle data transfers in a manner decoupled from user process execution. In PPIO, modules are implemented within the protected operating system and are therefore isolated from errant or malicious user processes. To provide PPIO with flexibility comparable to user-process based approaches, kernel-executed processing modules should be selected or provided by user processes. Privileged execution of user-provided processing modules introduces the issues of security and fault tolerance. Specifically, the operating system should not be made insecure or faulty by executing errant or malicious processing modules. There exists an important tradeoff between module flexibility and security in such an environment. Generally, flexibility and security are inversely proportional.

Processing modules may be grouped into two broad categories: *compiled* and *interpreted*. Furthermore, they may be subdivided into *user*, *administrator*, and *system* modules. User modules may be written or submitted by any (non-privileged) user program and are incorporated into an optimized pipeline by some system software layer. Administrator modules must be submitted to the OS for inclusion into a pipeline by some privileged process. System modules are generally implemented by the OS developer and included at OS build time.

Streams [Rit84] offers a simple compiled/system module execution model. All modules are predefined and are compiled in to the operating system at system generation time. User processes select which modules comprise pipelines by dynamically *stacking* one module atop another. In Streams, the pipeline exists between a hardware device (or simulated hardware loopback agent called a pseudoterminal) and a user process.

In the modern commercial systems OSF/1 [Ope90] and Solaris [Sun90], a compiled/administrator module model is used. A compiled set of routines and data may be loaded in or unloaded from an executing operating system when needed. A privileged user process specifies which code files should be incorporated into the executing operating system, and invokes system calls capable of installing the specified code. Code loaded into the OS remains there until it is unloaded.

Several current UNIX-based systems now provide a facility known generally as a *packet filter*, initially described by Mogul in [MRA87]. Packet filters fall into the interpreted/administrator or interpreted/user model, depending on the level of privacy of network data desired. Systems of this kind require a user to express data manipulations in a special language which is interpreted and executed within the operating system.

A technique which has presently been applied only to user space entities but offers promise for operating systems as well is called *sandboxing* [WLAG93]. A related methodology is being undertaken on a larger scale in a new operating system effort called *SPIN* [BCE+94]. These systems offer a compiled/user execution model, and rely on code analysis implemented within a compiler to enforce restrictions on an arbitrary code segments' data access. The idea is to take user-provided code and execute it in another (possibly trusted) protection domain.

Systems supporting user modules must insulate module execution from the non-module portions of the system to guarantee system integrity. If a popular programming language is used, user convenience is maximized, whether the language is interpreted or compiled. Interpreted code provides enhanced security, as the system can determine harmful effects of code execution before certain operations are actually executed, but generally executes slower due to the interpretation overhead. In compiled systems, techniques such as *sandboxing* may be employed, but add execution overhead. Research in this area is ongoing.

Administrator modules generally offer better performance than user modules, but limit flexibility and depend on privileged process execution for security. On such systems, the operating system takes no measures to ensure the safety of introduced code. Errors, deadlocks, etc. may occur,[1] resulting in potential OS crashes or data corruption. These systems offer less flexibility than user modules by not allowing regular users to introduce their own data transformations in pipelines they create.

System modules offer the highest degree of security and performance at the price of minimum flexibility. Security and performance are maximized because modules can take full advantage of the data structures available in the protected operating system address space and module code may be written by the same programmers as the rest of the operating system. Flexibility is minimal, as users and administrators may only select among those modules provided by the operating system. Although additional modules can be developed, the majority of programmers prefer to avoid operating system code development and generation.

The PPIO model provides no restriction as to which module types can or cannot be used. Administrator, system, and interpreted user modules are directly supportable with present technology, and an example interface for such pipelines is provided in Chapter 3. Compiled user modules are not presently supportable with commonly existing compiler technology, but PPIO is well-positioned to take advantage of advances in this area which seem imminent. Moreover, PPIO supports any or all of the modules types discussed, and may be incorporated into most operating systems supporting a pipeline mechanism.

[1]The assumption here is that such errors are likely to occur more frequently in such a system as compared with system modules because administrator modules would generally not be included as part of the standard OS code base, and would thus have received comparatively less testing.

4.3.3 Execution Locus

In PPIO, unlike most conventional I/O systems, the PMs described above need not necessarily be executed on a main CPU. The *execution locus* (EL) of a PM refers to the hardware agent responsible for executing PM instructions. In a conventional uniprocessor system, the EL of a PM resides in the (single) CPU. In a conventional (shared or distributed memory; symmetric or asymmetric OS) multiprocessor, several PMs may execute concurrently, and the EL of each is likely to be on separate main processors. With PPIO, generalized *functional multiprocessing* (FM) may be exploited. Figure 4.7 illustrates the differences between FM and conventional uniprocessing. With FM, processors are generally dedicated to a small set of tasks and do not run general user or OS code. FM is described in more detail in [NFE92], and is a primary design principal for the hardware of the Auspex NFS file server product.

The flexibility of changing the EL for PMs in PPIO is achieved due to the decoupling of user process execution from the movement of data. A requesting user process need only request what processing needs to be performed, and the operating system handles the transfer. In such a system as *Streams*, stream modules could execute with a EL other than the main CPU, but data is ultimately routed to a consuming (producing) user process, which must execute on a main CPU for data to flow. This restriction is lifted for PPIO.

Figure 4.3: Source Algorithm: algorithm executed on source objects

```
source algorithm, executed by data sources
procedure source()
{

        allocate system-dependent resources ;

        /* if src_flow_downcall returns DONE, a thread
         * will have been created, taking care of the read
         */
        if (src_flow_downcall() == DONE) {
                return ;
        }

        /*
         * set up handler for async reads,
         * initiate read for passive objects
         * (active ones don't need initiation
         */
        if (!sync_reads || active_output) {
                if (first_time) {
                        install read_complete handler ;
                        first_time = FALSE ;
                }
                if (active_output) {
                        return ;
                } else {
                        initiate read ;
                }
        }
        /*
         * perform synchronous reads for objects that require
         * it, call sink when complete
         */
        if (sync_reads) {
                loop {
                        perform read ;
                }
                call sink ;
        }
        return ;
}
procedure read_complete handler()
{
        /*
         * can still have a need for sync reads, if so
         * do that here
         */
        if (sync_reads) {
                loop {
                        perform read ;
                }
        }
        /*
         * this call will drop the data if we're active
         */
        if (src_flow_upcall() == DONE) {
                return ;
        }

        call sink ;        /* give to peer */
}
```

Figure 4.4: Sink Algorithm: algorithm executed on sink objects

```
sink algorithm, executed by data sinks

procedure sink()
{

        /* install handlers for async writes */

        if (firsttime && !syncwrites) {
                install write_complete_handler ;
                firsttime = FALSE ;
        }

        initiate write ;

        if (syncwrites) {
                loop until done {
                        perform write ;
                }
                free system dependent resources ;

                /* if we're running under a thread (top half)
                 * the thread will perform the next read for us,
                 * so be careful to not call the source here, or
                 * we'd keep eating up stack with procedure calls
                 */
                if (threadcreated == TRUE) {
                        return ;
                } else  {
                        call source ;
                }
        }
        return;
}
procedure write_complete_handler()
{
        free system dependent resources ;

        /* same comment as above */
        if (threadcreated == TRUE) {
                return ;
        } else {
                call source ;
        }
        return ;
}
```

Figure 4.5: Flow Algorithms: procedures executed on source and sink objects to achieve flow control

```
procedure srcflowdowncall()
{

        boolean flowcontrollable = (passive || quenchable) ;
        /*
         * if we're fine or can't do anything anyhow, just return
         */
        if (!resourceslow || !flowcontrollable) {
                return CONTINUE ;
        }
        /*
         * the only way we can slow this source down is to
         * block a thread.  If we're already a thread
         * block until things are better.  If we're not,
         * create a thread (which will call source again),
         * and wind up blocking if necessary
         */
        if (passive && !quenchable) {
                if (threadcreated) {
                        block until !resourceslow ;
                        return CONTINUE ;
                } else {
                        create thread reader ;
                        threadcreated = TRUE;
                        return DONE ;
                }
        }

        /* at this point, must be quenchable */
        quench source ; /* adjust quality or perform f-ctrl */
        return CONTINUE ;
}
procedure srcflowupcall()
{
        if (active && resourceslow) {
                discard data ;
                free resources ;
                return DONE ;
        }

        return CONTINUE ;
}
thread reader()
{
        /*
         * only need to call the 'source' side because
         * all instances of 'source' will call 'sink' when
         * appropriate
         */

        loop until done {
                call source ;
        }
}
```

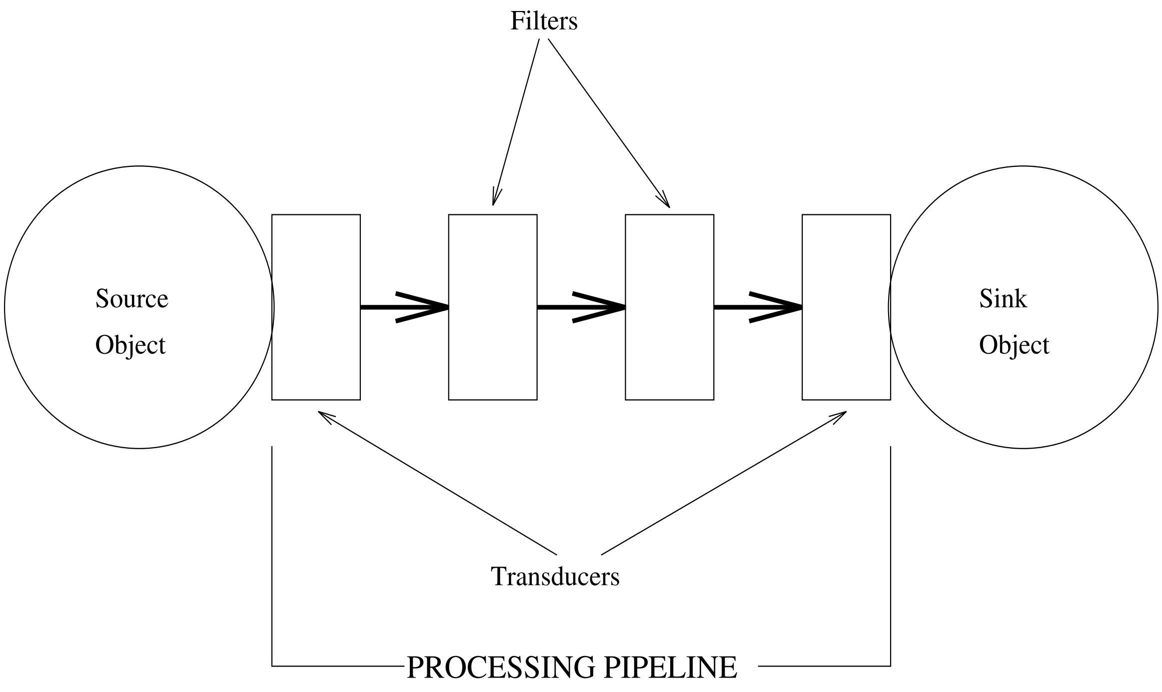

Figure 4.6: Processing Modules (using Pilot terminology)

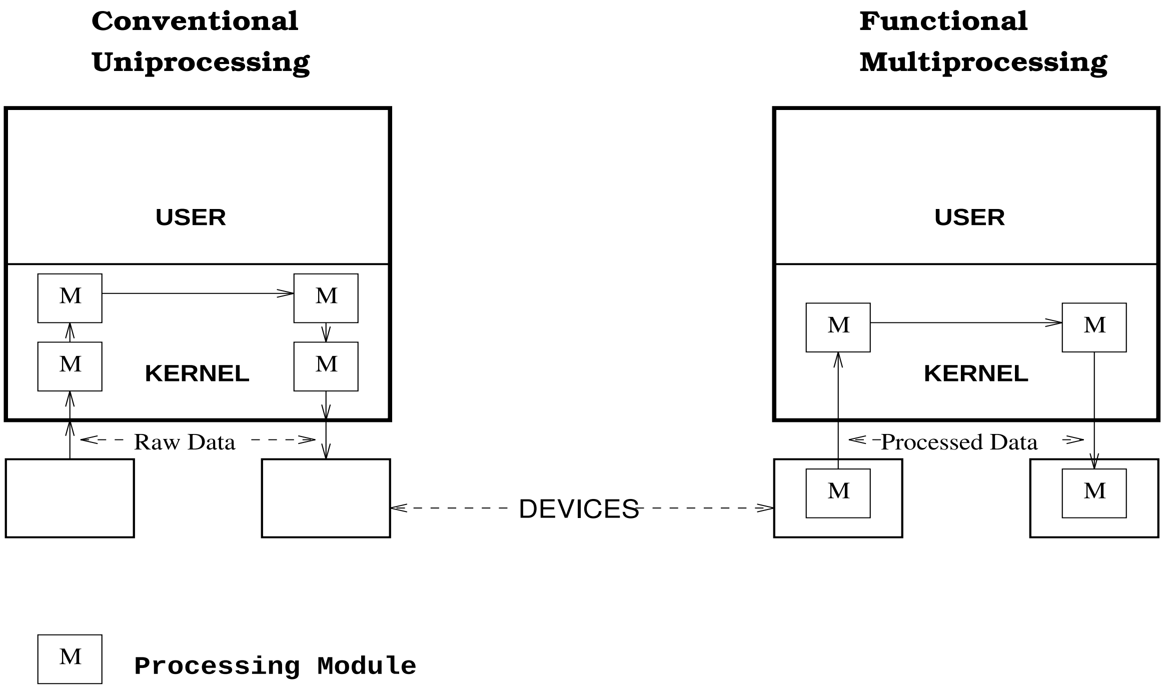

Figure 4.7: Comparative Illustration of PM Execution in Uniprocessor and in Functional Multiprocessing.

Chapter 5

Data Flow

This chapter discusses the way data is stored and manipulated in I/O systems. The first sections discuss the purpose and types of buffering used in conventional systems for I/O, followed by the methods employed by PPIO to manage data transfers. The final section discusses the kernel data structures required for implementing the PPIO system. Data handling in PPIO is based on a streaming model which reduces overhead by "short-circuiting" the data path between source and sink objects.

5.1 Buffering and Data Manipulation

This section begins with a review of the fundamental purposes and performance implications of buffering. Buffering is generally used for caching (to improve performance), or for temporary storage of data (e.g., during I/O transactions). Excess manipulation of buffers reduces performance by increasing demand on the main memory system.

5.1.1 Purposes of Buffering

Operating systems often manage I/O data in transit from some data source to some data sink. Data is generally stored in *buffers* (memory blocks), which may reside in a generally accessible location (*main memory*) or may reside on peripherals equipped with memory (*outboard memory*). Buffering generally improves throughput by reducing per–byte overhead at the expense of increased delay.

Buffers achieve four purposes in operating systems:

1. Temporary storage and manipulation of in–transit data
2. Speed–matching and burstiness smoothing (flow/rate control)
3. Caching
4. Performance enhancement (delay/throughput alteration)

Temporary storage of data is needed within an operating system when data must be stored before it is consumed. Such situations occur often with hardware device management. Many hardware devices employing DMA must have buffers supplied before I/O can be performed. This situation is especially representative for network interface devices, which have *active output* (see Chapter 4). In many cases (e.g., network protocol processing), data must be manipulated before it is useful to higher layer software.

Speed–matching and burstiness smoothing may be realized with a buffer by decoupling the buffer filling process from the buffer draining process. The function is similar in notion to an aqueous reservoir, which may be employed to capture rain water (a bursty source) and drain it later periodically (a constant-rate water tap). Speed–matching is achieved similarly, by adjusting the rates of the filling and draining processes separately.

Buffers are often used within operating systems to perform a caching function by providing a layer higher in the memory hierarchy than disk (or other devices). Cached file data becomes accessible on the order of nanoseconds from DRAM memory as compared with milliseconds when accessed with conventional disk drives or over conventional networks.

Buffers may be employed to alter throughput and delay performance by performing *data aggregation*, which may be illustrated by employing the following equations:

$$\lambda = \frac{L}{T_i + T_b * L} \tag{5.1}$$

$$\delta = T_i + T_b * L \tag{5.2}$$

In these equations, λ represents throughput, δ represents delay, L indicates the length of a block of information, T_i represents the fixed cost (per-block cost) of performing an I/O operation, and T_b represents the per-byte cost of performing an I/O operation. As the amount of buffering introduced in such a system is increased ($L \rightarrow \infty$) or decreased ($L \rightarrow 1$), we arrive at the following limits for λ:

$$\lim_{L \to \infty} \lambda = \frac{1}{T_b} \tag{5.3}$$

$$\lim_{L \to 1} \lambda = \frac{1}{T_i + T_b} \tag{5.4}$$

The corresponding limits for δ are as follows:

$$\lim_{L \to \infty} \delta = \infty \tag{5.5}$$

$$\lim_{L \to 1} \delta = T_i + T_b \tag{5.6}$$

Equations 5.1 and 5.3 indicate that as buffering is increased ($T_b * L \gg T_i$), achievable throughput approaches the *maximum channel capacity* of $1/T_b$. The disadvantages of large quantities of buffering include increased delay, as illustrated by Equations 5.2 and 5.5. Furthermore, physical memory requirements scale with L.

Assuming $\lambda > 0$, minimal buffering occurs when $L = 1$. In such environments, the following relationship typically holds: $T_i \geq T_b * L$. Throughput is minimized as per-byte overhead is maximized. Delay is correspondingly minimized, as the minimum possible delay of $T_i + T_b$ is achieved.

5.1.2 Uniform and Specialty Buffering Systems

Operating systems typically allocate buffers from a shared pool of physical memory. *Uniform buffering* refers to a common buffering scheme employed throughout an operating system. *Specialty buffering* refers to multiple distinct types of buffering used within the same system to interact with different system objects, typically different hardware devices.

Both types of buffering have been incorporated into real systems. For example, uniform buffering is incorporated in the *fbufs* scheme described by [DP93] and the buffering scheme used previously in the X-Kernel [HP91], along with *Interrupt Request Packets* (IRPs) of WindowsNT [Cus93] and *Request Packets* of OS/2 [DK92]. Specialty buffering is used by systems such as UNIX in its original form [ATT78] and when enhanced to support networking [LJF83].

Systems employing uniform buffering usually include a buffer manager used by all system software to allocate and free buffers. Buffers are allocated as needed when data must be stored, or may have to be preallocated in cases where data arrival cannot be predicted (i.e., active output I/O objects). They are released when data has been delivered to its destination. Systems with specialty buffering typically include several buffer managers which keep separate pools of memory for servicing allocation requests.

Buffering may be employed at all layers of system software. It is also commonly used by application software performing I/O operations. In protected operating systems, application-layer buffering is often distinct from operating-system buffering for reasons of protection and synchronization. When data must be moved between the operating system's address space and an application's, it is usually copied.

5.1.3 Buffer Manipulations

Operating systems spend significant amounts of time manipulating memory buffers. The largest cost is associated with those operations that require access to each byte of I/O data (in contrast to those which require only one operation per buffer). The most common operations requiring access to each data byte in a buffer are as follows:

1. moving data between devices and main memory
2. zero-filling main memory regions
3. copying data from one main memory buffer to another

In most conventional operating systems, data is moved between I/O devices and main memory with DMA or Programmed I/O. In the former case, an on-board DMA engine performs *cycle stealing* by competing with the CPU for access to the bus interconnecting devices and main memory. A small on-board memory aggregates I/O data which is moved into main memory by the DMA engine. Programmed I/O requires the CPU to perform the bus accesses directly. DMA operation can have the following negative performance impacts: reduction of CPU cache fill or flush rate due to memory bus contention and cache invalidation needed following a successful DMA input transaction.

Zero-filling refers to setting the contents of memory buffers to the NULL value. Zero-filled buffers are generally needed in a buffer manager's free memory pool for servicing allocation requests across mutually distrusting requesters. In cases where requesters are mutually trusting (i.e., data privacy is not required), zero-filling may be avoided.

Data Copies

Unnecessary copying of data buffers presents a substantial concern for high-performance operating systems. Copying is performed for the following primary reasons:

- Alignment
- Gather/scatter manipulations ((de)serialization)
- Data movement between protection domains (isolation)

Data must be aligned within an operating system for a number of reasons. Consider *block-oriented* devices such as disk controllers which perform low-level I/O operations only on individual data sectors. Buffered I/O must generally be aligned on a block or word boundary. Copying a portion of data from one file (i.e., sequence of blocks) to another which does not begin at a properly-aligned point requires a data copy for alignment as illustrated in Figure 5.1. In this illustration, the destination requires block alignment.[1] The data to be transferred begins half way through block 1 of the source block list and ends half way through block 4. A copy is required to align the beginning of I/O data on a block boundary, and indicated by the dashed arrows.

Data is sometimes required by hardware to be in contiguous memory addresses. Conversely, some hardware is more easily programmed or can offer superior performance by using non-contiguous buffers. *Scattering* or *deserializing* data refers to splitting a block

[1] This situation occurs in the UNIX buffer cache system.

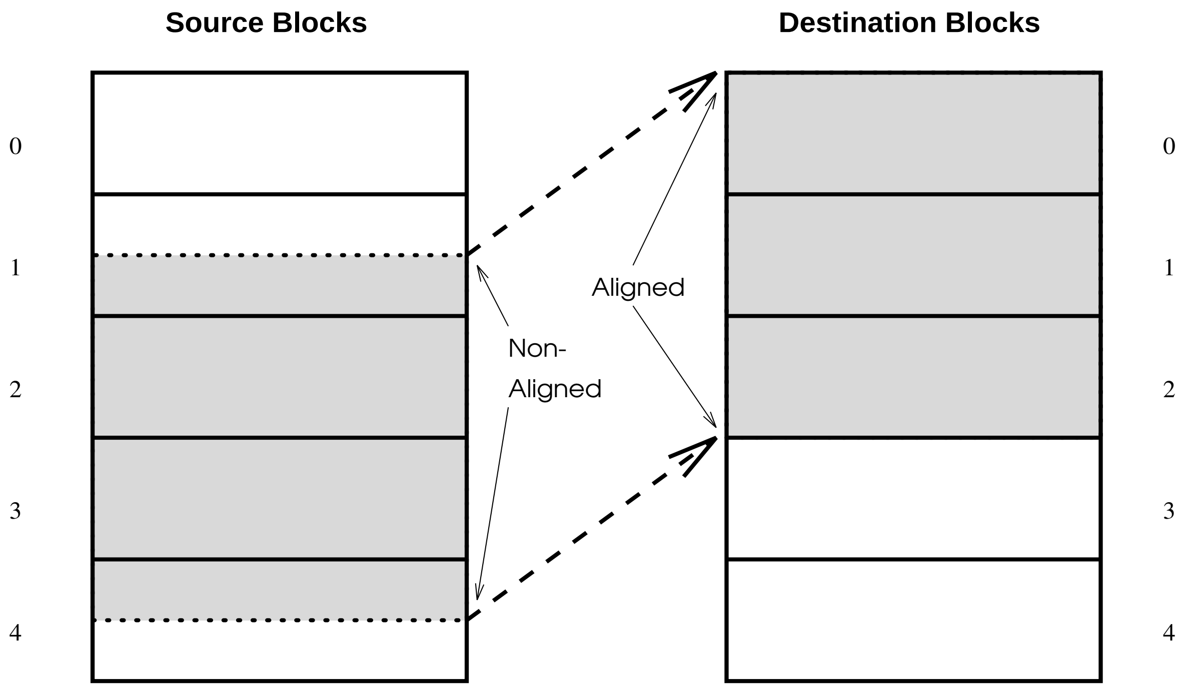

Figure 5.1: Copying a file from a non-aligned source offset. A data copy is required when the destination requires block (or word) alignment not satisfied by the source.

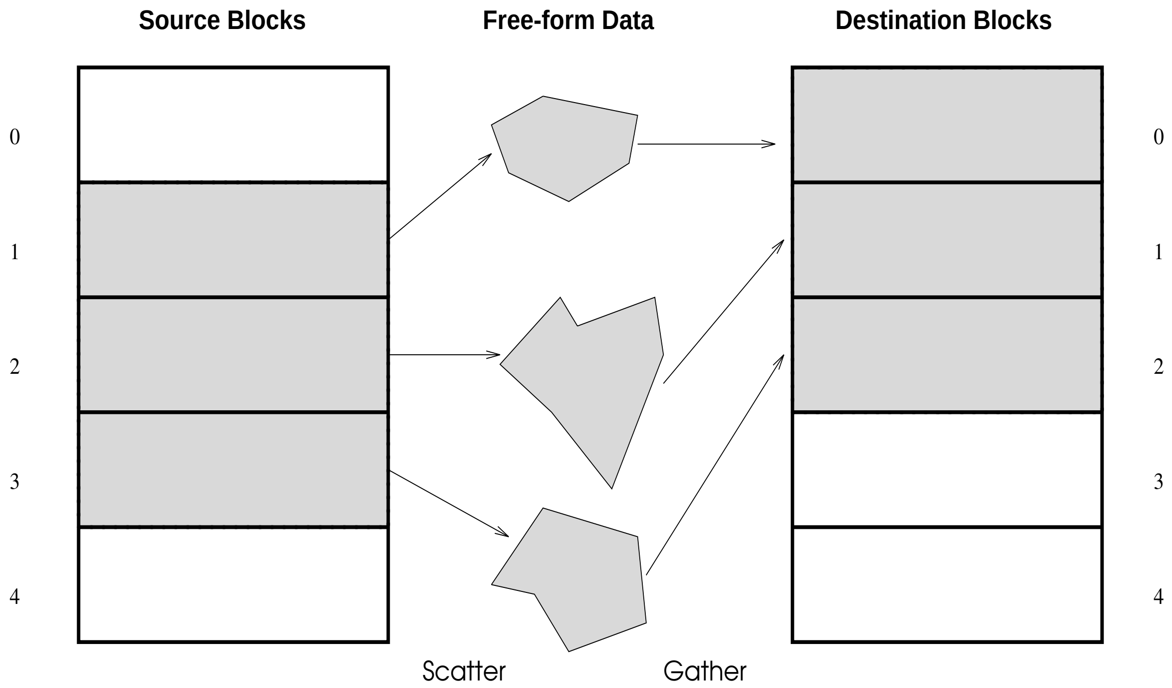

Figure 5.2: Scatter and gather data operations. A data copy is usually required to serialize or deserialize arbitrarily-located data.

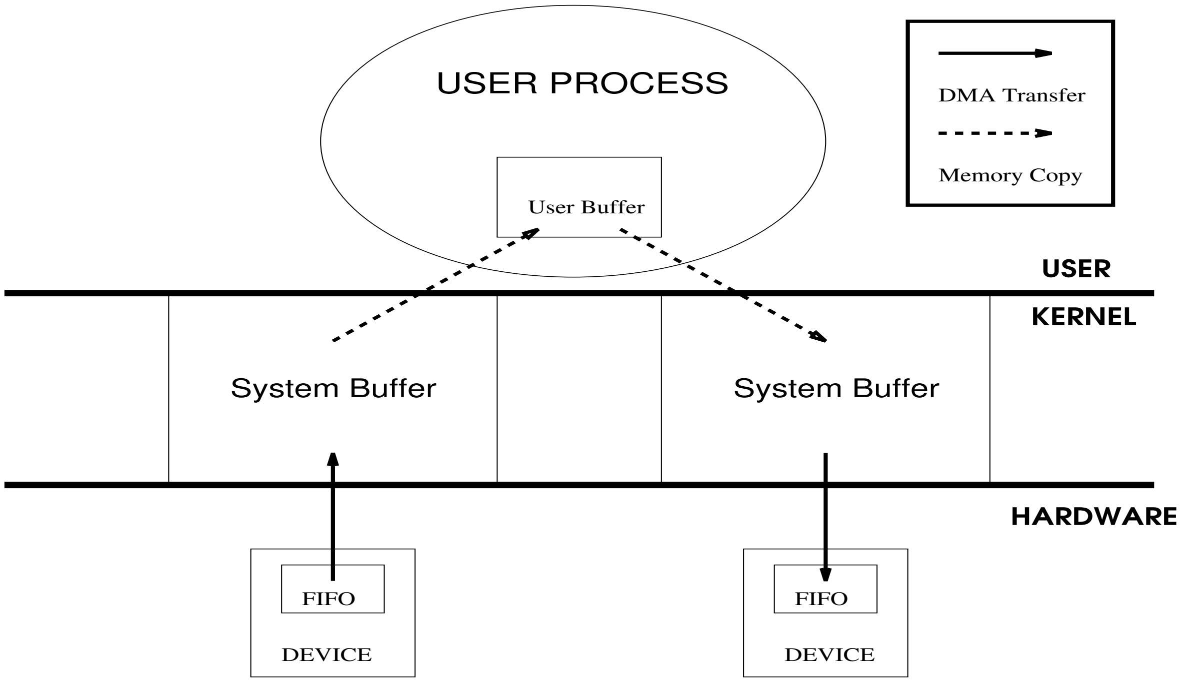

Figure 5.3: Data copies performed during I/O data routing. I/O data is copied between user and kernel address space on both incoming and outgoing operations.

of contiguous data into non-contiguous sections. The reverse operation is *gathering* or *serializing*. These operations are illustrated in Figure 5.2.

Isolation and protection is easily achieved by copying data. By producing a second copy of data, any corruption or manipulation of one copy is isolated from the other. Such copies are typical of interprocess communication mechanisms, either between user processes or between the operating system and a user process. For example, a user process moving data between I/O devices is illustrated in Figure 5.3.

5.1.4 Effects of Data Manipulations

The buffer manipulations described in the previous section have an important impact on system performance. Each of the manipulations can be optimized in various ways.

For access to I/O devices supporting DMA, devices compete for bus access and are allowed to transfer some number of bytes per transaction. The bus may be shared with the main memory unit, in which case contention between the CPU and I/O devices can occur. Furthermore, I/O data written to the memory unit requires a cache invalidation to prevent the CPU from reading stale data from cache.[2] DMA engines are typically programmable, and throughput is maximized by employing the highest number of bits per word available in addition to maximizing the DMA burst transfer size. With respect to buffering, DMA-based peripherals may require contiguous and/or block-aligned I/O buffers. For devices with active output, DMA buffers must generally be pre-allocated.

I/O devices supporting programmed I/O require CPU action to complete I/O requests. Requiring CPU action (obviously) prevents the CPU from performing other tasks simultaneously while I/O is occurring. PIO devices are generally less expensive than their DMA-based counterparts. In addition, the cache invalidation required by DMA-based operation is avoided. Finally, for PIO devices, I/O buffers can generally be allocated at the point I/O data is moved from peripheral memory to main memory.

Zero-filling can be a time-consuming procedure on systems, especially when much IPC is performed between mutually distrusting processes. In such cases, old data from other processes must not be allowed to be visible in new buffer allocations. When zero-filling is performed, the processor cache is affected by filling it with many zero values, reducing its effectiveness.

Copying of data is considered to be one of the most important sources of overhead in most I/O systems. It is, for example, the greatest source of overhead for processing of network protocols at high throughput like TCP in UNIX [CJRS89]. A system's copy performance is given by the following equation:

$$\frac{1}{T_r} + \frac{1}{T_w} = \frac{1}{T_c} \tag{5.7}$$

Rearranging terms, we have:

$$T_c = \frac{T_w * T_r}{T_w + T_r} \tag{5.8}$$

[2]Note that some modern architectures, including the DEC Alpha [Sit92], update cache entries when DMA takes place, thus obviating the need for cache invalidation.

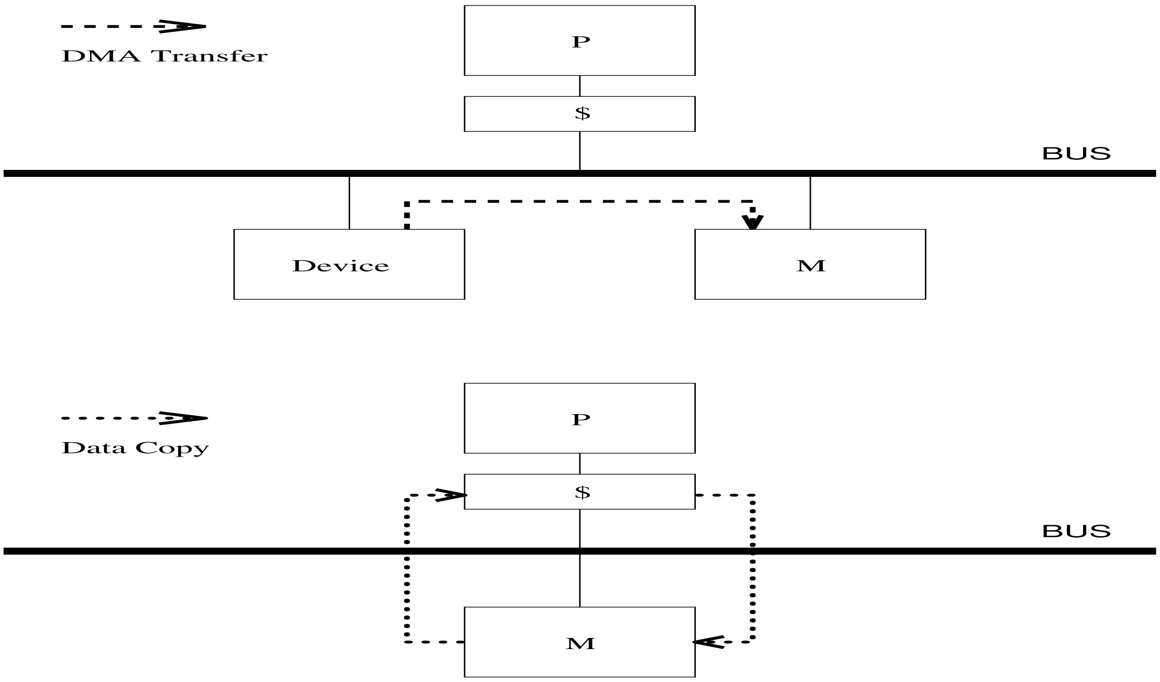

Figure 5.4: Architectural effect of DMA and copy operations. DMA operations require cache invalidation while copy and PIO operations fill cache entries.

In these equations, T_r, T_w, and T_c refer to memory system uncached read, write, and copy throughput performance, respectively. These values are generally given in MBytes/sec or MBits/sec.

DMA transfers and data copying also affect the performance of processor caches. Figure 5.4 illustrates data flow during DMA and copy operations. In the case of DMA operations, data is transferred from device to main memory, bypassing the processor cache. The processor may continue execution during this period, but cached entries for the region of memory involved in the DMA operation must be invalidated to force a cache line fill upon access by the CPU. In the second portion of the figure, a data copy is illustrated. Copies are performed by the CPU a word at a time between distinct portions of the cache. Copying data which exhibits little locality (i.e., is not accessed again in the future) is detrimental to good cache performance. In particular, filling precious cache entries copied data which lacks locality forces out other data which may exhibit superior locality.

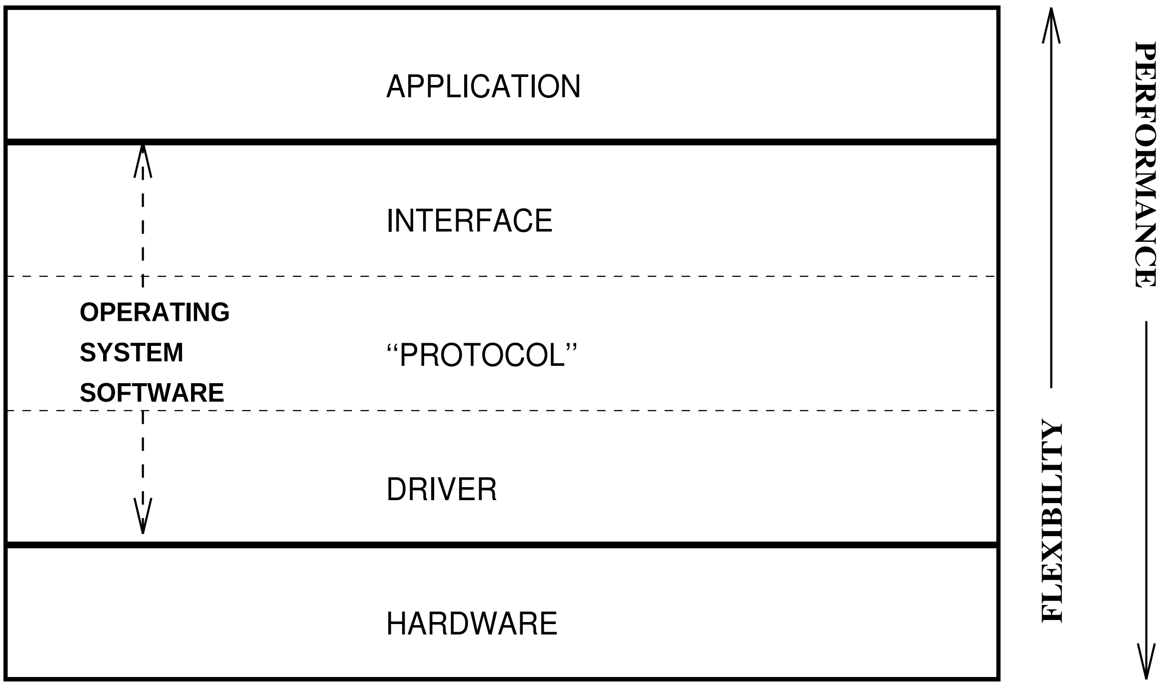

Figure 5.5: Possible layers for implementing streaming

5.2 Data Flow in PPIO

This section turns to the approach taken in PPIO to address several of the data movement problems described in the previous section. The PPIO design is focused on the concept and mechanisms needed to support a more direct data flow between I/O objects.

5.2.1 Data Streaming

Data streaming was introduced in Section 2.4. It may be thought of as coupling a data source with a corresponding sink at a software layer generally below the application. In the following discussion of streaming the *Interface* layer represents an extension of the layers presented in [DAPP92]. Several distinct layers of software may be considered for the implementation of OS-supported streaming:

- Interface - topmost layer of the OS, just below application
- OS Kernel (protocol) - layer between interface and device driver

- DMA-DMA (driver) - bottommost software layer
- Hardware - between I/O device

The layers of possible implementation are illustrated in Figure 5.5. With streaming approaches, greater performance is generally achieved at the expense of reduced flexibility.

Interface streaming requires comparatively little modification to the operating system. In such a system, operating system code substitutes for application code responsible for copying data between kernel and user. The OS code executes just above the interface layer. For protected operating systems, this avoids protection domain switches between kernel and user environments, and also avoids accompanying context switches. It also eliminates the data copy between user and kernel space. However, buffer re-use may be difficult or impossible at this layer (especially in specialty buffering systems) because potentially incompatible buffering schemes may still require a data copy (in this case between different portions of kernel virtual space).

Protocol streaming is similar to interface streaming, but is realized at one software layer closer to the hardware. The term "protocol" is used here loosely. It refers not only to network protocols, but also to other software abstractions implemented above device drivers. Protocol here would include abstractions such as files for disks, streams for connection-oriented network protocols, and virtual address spaces created for processes. Streaming at the protocol layer is the preferred location to implement streaming on systems lacking hardware streaming support. There are several reasons for choosing this layer for streaming. First, at this layer, I/O data in transit is typically stored in the original buffer it was created in (i.e., it has not yet been copied), and buffers may be passed and shared between different "protocol" modules (especially in unified buffering systems). Second, placing streaming above the device driver keeps most higher-level functionality isolated from the details of device handling (a good software engineering technique). Third, within the protocol layer semantic meaning becomes associated with I/O data, and meta-information is interpreted and removed from the data stream.

Providing streaming support at the device driver level is possible but unattractive. At the point I/O data emerges from a device driver it is generally an uninterpreted collection of data bytes with in-line meta-information. Meta-information such as protocol headers are generally not understood by protocols other than those to which they were designed, and should not be passed between I/O objects. Doing so would require interpretation of meta-information which is already implemented at the protocol layer and would thus constitute a redundant implementation (and possibly a layer violation, although this is a comparatively minor concern).

5.2.2 Hardware Streaming

Providing streaming at the hardware layer is possible and is supported by a small number of systems as discussed in Section 2.4. Supporting such systems requires appropriate hardware combined with supporting system software. Appropriate hardware support generally implies inter-device data transfer capability, plus sufficient processing capability to stage data in a format acceptable to the receiving peer device.

Advocating processing on external devices has encountered resistance. The primary counterargument is to assert that equipping external devices with processors requires those processors to progress in performance at a rate equal to or greater than the main CPU. The cost of replacing more than one processor should be used to implement a symmetric multiprocessor. There is a response to this argument.

External processors need not necessarily progress in performance at the rate of regular microprocessors. Onboard processors have the task of staging data at a rate commensurate with external I/O being performed, and when an I/O adapter is engineered for a particular external interface, a processor of the appropriate speed is selected. The speed of the processor need not be enhanced unless the speed of external I/O is increased. External I/O interfaces are generally specified by a standards process and do not generally improve in speed spontaneously (or even quickly as compared with improvements in microprocessor technology).

5.3 PPIO Data Structures

The PPIO architecture relies on the establishment of associations between data sources and sinks. When an association is established, any relevant descriptors are recorded in an *association descriptor* (AD).

5.3.1 Association Descriptors

The association descriptor (AD) is a dynamically allocated aggregate data type, created one-to-one for each active association. The AD contains at least the following information; additional information may be required on a per-implementation basis:

- descriptor to source I/O object
- descriptor to sink I/O object
- descriptor of process initiating association
- association state
- current offset in source (if appropriate)
- current offset in sink (if appropriate)

The source and sink descriptors are references to aggregate data structures describing a data source or sink, including functions to initiate I/O, change state, or perform object-specific operations (e.g., device resets or programming, etc). For example, although they may not refer to conventional files, they are known as *file descriptors* in UNIX and POSIX.

The process descriptor is used to identify the process responsible for initiating the association. It is used for accounting (keeping track of amount of I/O performed, etc) and for delivery of asynchronous events and completion state. Should an association terminate normally, the process descriptor is used to identify the particular process which should be signaled. Note that while I/O data flows through an association, a process need not be resident. In addition, exceptional conditions such as aborts and errors cause delivery of asynchronous events (signals) to user processes. User processes therefore retain full control over the handling of exceptions.

The association state encodes the present condition of an association. Associations can include the following state information: active, wanted, error. Active associations represent data flows in operation without error. The wanted state indicates a process is awaiting completion of an association. Associations may operate asynchronously, and a process may opt not to await the completion of an association. In such cases, the wanted condition will remain clear. The error condition is recorded in the AD when an I/O error occurs on the source or sink object or some other internal inconsistency is encountered, and ultimately results in the termination of the association.

Current offsets are maintained for I/O objects requiring offsets. Randomly-addressable objects such as disks generally require offsets. Subsequent `read` and `write` operations use the offset field to advance sequentially through a source or sink I/O object. The offset can be manipulated randomly with a *seek* directive such as UNIX's `lseek`.

5.3.2 Demultiplexing Reference

The PPIO architecture makes use of *upcalls* to execute the algorithms described in Section 4.2.2. When executed as a result of an interrupt condition, scheduling delays and intermediate buffering can be minimized. Upcalls are implemented by installing a *demultiplexing reference* (DR) prior to initiation of an I/O transaction. Completion of an I/O transaction includes execution of a procedure referenced by the DR. In cases where demultiplexing is explicitly encoded in a subsystem (such as higher layer protocol or port numbers in network protocols), an existing OS data structure is typically available (e.g., protocol control block) in which the DR may be added. Alternatively, in cases where state is maintained on a per-I/O-request basis (such as for disk blocks), the DR may be encoded in the meta-data portion of an I/O buffer.

5.3.3 Modules

The data structures required for module support include a buffering mechanism and support for process and timer execution. The PPIO architecture adopts the Stream

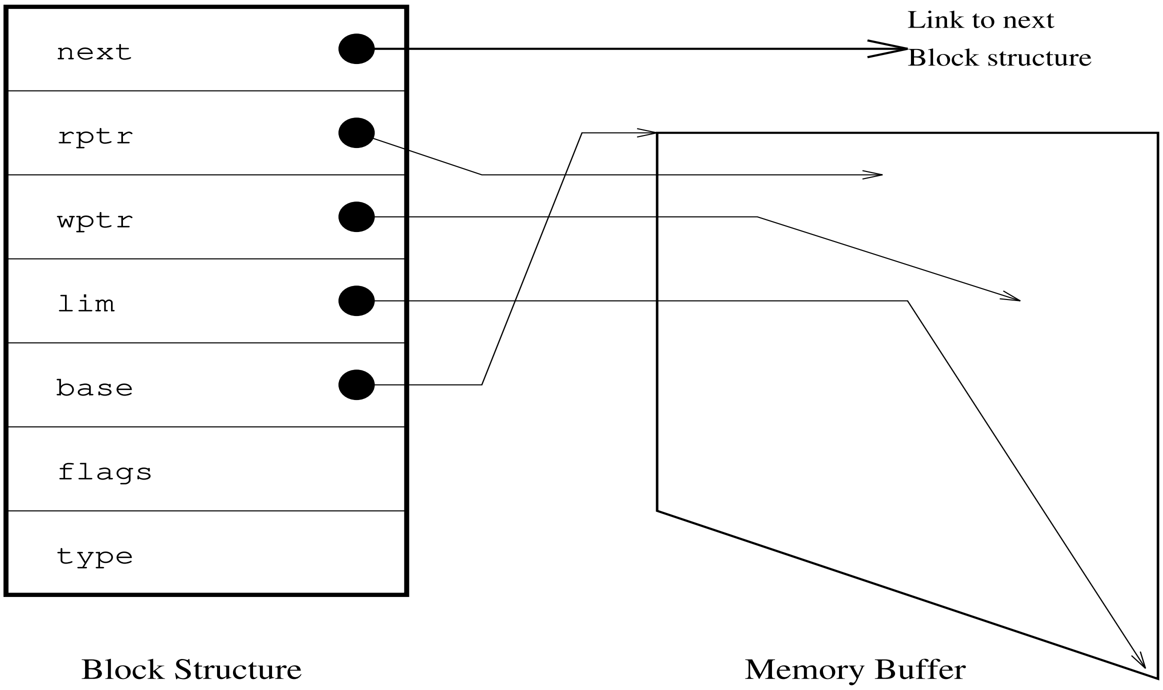

Figure 5.6: The Block Structure from Plan 9

architecture suggested originally by Ritchie [Rit84] in the 8th Edition UNIX system and enhanced for use in Plan 9 [Pre90]. Other versions have evolved commercially from SystemV Release 3.

Buffering mechanisms have been discussed in Section 5.1.2. In Ritchie's Streams, data and control information is passed in *message blocks* containing a simple header. The header includes *read*, *write*, and *limit* pointers which refer to a variable sized data buffer. In Plan 9, the header also includes a *base* pointer, and the message structure is now called a *Block*. It is illustrated in Figure 5.6. The `read` and `write` pointers indicate where data is read from or written to, respectively. The `base` and `limit` pointers indicate absolute boundaries for data and provide bounds for the read and write pointers.

All systems presently available incorporating Streams or similar architectures utilize a common buffering structure known to all modules. Examples of such systems include Plan 9, System V UNIX, Windows NT, and the X-Kernel. Minimum complexity is achieved by supporting a single buffer abstraction as modules need only accommodate the single mechanism. The module architecture could be used in a specialty buffering system with some additional effort. In such cases, modules may be coded to handle all possible buffering abstractions utilized in the existing operating system, or translation modules may be constructed to convert from the various specialty buffers into a canonical form. The former strategy trades additional complexity for improved performance. The latter trades performance (by possible need for data copies) for flexibility and opportunity for code re-use.

In addition to the data structure required to pass data between modules, data structures are required to support the execution model of processing modules. In the original Streams design, module processing is done with coroutines, requiring minimal state between execution of module service routines. The design was later changed to allow modules to run as processes. Module processes are lightweight, as they all coexist within the kernel address space. Systems which support kernel-level threads (e.g., Mach, X-Kernel, OSF, WinNT) may use their respective thread facilities for scheduling and activating modules. For systems lacking such support, a Streams scheduler is typically grafted to the already existing operating system.

Chapter 6

Implementation and Performance

This chapter focuses on experience gained by implementing prototypes of the PPIO design described in the previous chapters of this dissertation. The implementations have spanned several months at which time the interface itself has been evolving. Therefore, the prototypes implement a simplified version of the interfaces presented in the Chapter 3. The performance results, however, are insensitive to the interface specifics.

Three performance studies are discussed in this chapter. The first study shows the effect upon CPU utilization and throughput when using PPIO to accomplish a file copy operation. The second study investigates how PPIO can be used for improving the movement of continuous media data delivered over a network in a video playback scenario. The final study examines the use of PPIO in an application-level gateway and is focused on measuring end-to-end latency and stability under high load.

6.1 Experiment 1: File System

The first implementation and performance study of the PPIO ideas and the `splice` system call was prepared in 1992 and is described in [FP93]. Splice was implemented as a system call under Ultrix 4.2A, and has been tested on a DECStation 5000/200 and DECStation 5000/240. Ultrix is a derivative of Berkeley UNIX supplied with Digital Equipment Workstations based on the MIPS R3000 and MIPS R4000 microprocessors.

The PPIO code comprises about 3000 lines of C source code (including comments), and increases the kernel's object size by about 10%.

6.1.1 Background

The file system prototype implementation of PPIO supports file-to-file copies between files residing on local disk storage devices using `splice`. The following discussion outlines only the portions of the implementation relevant to the 4.2BSD-based file system. The `splice` implementation uses a buffer cache kernel interface, and makes use of the following buffer cache routines: `bmap()`, `bread()`, `getblk()`, `bawrite()`, `brelse()`, as well as the dynamic kernel memory allocator and `callout` list. This section assumes basic familiarity with these functions. They are discussed in more detail in [KMQ89].

6.1.2 Implementation and Operational Details

Assuming an entire file is to be copied, `splice` operates generally as follows. First, the size of the source file is determined from information present in the `gnode`.[1] A special *association descriptor* (AD) (see Section 5.3.1) is dynamically allocated to keep state information about the data transfer. Placing all necessary information in this descriptor allows I/O to proceed without requiring the availability of the calling process' context.

A file copy proceeds by first acquiring the entire list of all physical block numbers comprising the source file. The physical blocks are determined by repeatedly calling the `bmap()` function with increasing offsets. The destination file is mapped similarly to the source file, except a special version of `bmap()` is used for improved performance which avoids delayed-writes of freshly allocated, zero-filled blocks. The list of physical blocks is stored in a dynamically allocated table in the AD. At this point, all information necessary to proceed with an asynchronous data transfer has been stored in the AD, and user-mode execution of the calling process may be resumed.

[1] An Ultrix data structure describing a generic file system node.

6.1.3 Read-Side Operation

Data transfer between the source and destination files must be allowed to proceed without blocking; no guarantee can be made as to the availability of the calling process' context. New versions of the kernel routines `bread()` and `getblk()`, with the calls to `biowait()` removed (see below), provide most of the needed functionality. The physical block number is retrieved by indexing into the table in the AD by the logical block number on the source file. A call to the new `bread()` will schedule a read request and return immediately, instead of blocking awaiting buffer completion in `biowait()`. A handler function is installed in the buffer preceding the call to the driver's strategy routine by setting the `B_CALL` bit and `b_iodone` fields in the buffer header. When a read completes the read handler is invoked, which in turn schedules a write by placing a reference to the write handler at the head of the system `callout` list.

6.1.4 Write-Side Operation

The write side is invoked via the `callout` list with a locked buffer containing valid data just acquired from the source file. The `callout` list is used to decouple the I/O access periods at the source and destination I/O devices. Lock-step behavior is avoided by introducing the asynchrony provided by the callout list; this improves performance by allowing I/O operations at the source and destination points to proceed simultaneously. New fields in the buffer header structure indicate the AD and logical block number which are associated with a buffer's data. Thus, several buffers may be in transit simultaneously and need not be maintained in sequential order.

The logical block number, retrieved from the read-side buffer header, is used to index into the AD to determine the destination physical block number for the current buffer's data. The physical block number is used to request a buffer header using a modified version of `getblk()` which avoids allocating any real memory to the buffer,[2] but instead only sets the `b_bcount` field in the new buffer header to the requested size. The data

[2]Ordinarily, `getblk()` allocates both block headers and associated physical and virtual memory pages.

pointer in the new buffer header is saved and altered to point to the same address the data pointer in the read-side buffer does, so both buffers share a common data area. Thus, copying between cache buffers is avoided.

The `size` and `flags` fields in the buffer header are also saved and updated to match the corresponding fields in the read-side buffer header. At this point a write handler is installed in the header (by assigning the `b_iodone` in the buffer header), and an asynchronous write is performed by calling `bawrite()`. The write handler begins execution after the asynchronous write has completed. It retrieves a pointer to the source-side buffer for the current logical block number from the buffer just written and frees it by calling `brelse()`. It then frees the buffer just written similarly. Finally, a read request restarts the entire cycle.

6.1.5 Flow Control

Flow control cannot be achieved by causing the calling program to block. The calling process is not directly responsible for initiating intermediate `read` or `write` requests (these are done by the operating system), so causing it to block would provide little benefit. Instead, rate-based flow control based on the completion rate of write requests is employed.

Each AD maintains a count of the number of pending read and write requests. If the number of pending reads and the number of pending writes drop below pre-specified watermarks (currently 3 and 5, respectively), the write handler will issue up to a pre-specified number of additional reads (currently 5). These values must be set such that the source is not underutilized and the destination is not overwhelmed.

6.1.6 Performance Experiments

Several experiments were performed to measure the effectiveness of `splice` for performing file copies. The goal of these experiments is to demonstrate improvement in CPU availability and I/O system throughput. These improvements are achieved by reducing copying and switching overheads when using splice rather than using user-level read/write system calls to transfer data between files.

Configuration

All file system experiments were performed on a DECStation 5000/200 equipped with 32 MB memory using a 3.2 MB buffer cache. The DECStation 5000/200 MIPS R3000 processor is clocked at 25 Mhz and includes a 64 KByte instruction and 64KByte write-through data cache. Cached memory read throughput is 21 MB/s, uncached CPU read rate is 10 MB/s, and partial-page write throughput is 20 MB/s [DEC90].

Digital's RZ56 and RZ58 SCSI disks were used for performance measurements. The RZ56 provides an average rotational latency of 8.3 ms, average seek time of 16 ms, and a to/from media peak data transfer rate of 1.66 MB/s. The RZ58 provides an average rotational latency of 5.6 ms, average seek time of of under 12.5 ms, and to/from media peak data transfer rate of 3.1-3.9 MB/s. The RZ56 provides 64 KB of read-ahead cache, and the RZ58 provides 256 KB of read-ahead cache segmented into 4 read-ahead requests [DEC92].

The performance improvement of splice is most pronounced when applied to devices producing or consuming data at high rates relative to the CPU execution rate. To determine how splice would perform when using fast devices, RAM disk was implemented. The RAM disk is a device driver with a character-special and block-special device interface upon which a UNIX file system may be created. Consequently, the effect of using fast versus relatively slow devices on splice's performance could be evaluated. Either device required execution of the same file system code. The RAM disk driver uses 16MB of statically allocated memory from the kernel's BSS region, leaving a free memory pool of 5MB.

CPU Availability Test

The goals of measuring CPU availability and throughput is accomplished by executing a CPU-bound test program in three different environments:

IDLE execution of the test program with no other programs running

CP execution of the test program concurrent with a process executing the UNIX program `cp` copying a large regular file from a file system located on one physical disk to a file system on a different physical disk

SCP identical to CP, except a splice-based copy program `scp` is used rather than `cp`

Baseline performance indices are obtained by executing the test program in the IDLE environment and noting how long a fixed set of operations takes to complete. To measure changes in CPU availability, the amount of time required for the test program to complete the same number of operations is compared in the CP and SCP environments.

Table 6.1: Improvement in CPU availability using SCP vs. CP (copying 8MB file).

Disk Type	Contending with `CP`	Contending with `SCP`	Improvement of `SCP` over `CP`
RAM	49.3%	81.4%	65.1%
RZ58	63.6%	84.2%	32.4%
RZ56	63.8%	79.8%	25.1%

Table 6.1 and Figure 6.1 shows the relative performance degradation of a CPU-bound process when executing concurrently with a process copying an 8MB file using either `cp` or `scp` (i.e., the CP or SCP environments), with disks of various performance characteristics. Other tests were performed with significantly larger file sizes, but were not statistically distinguishable from the 8MB representative case listed above.

Referring to Table 6.1, column one lists the type of disks being used, including the two SCSI disks described above and the RAM disk driver (recall that the RZ56 is slowest disk, and the RAMDISK is the fastest). Columns two and three show the percentage reduction in execution rate experienced by the test program in the CP and SCP environments, respectively, as compared to the IDLE environment. Thus, a percentage reduction of $\chi\%$ means the process's execution rate was a factor of $\frac{\chi}{100}$ of the "IDLE rate," which is the execution rate in the IDLE environment. For example, in the CP environment using RAMDISKs, the test program executed at 49.3% of the rate it would execute in the IDLE environment, thus running approximately twice as long in the CP environment.

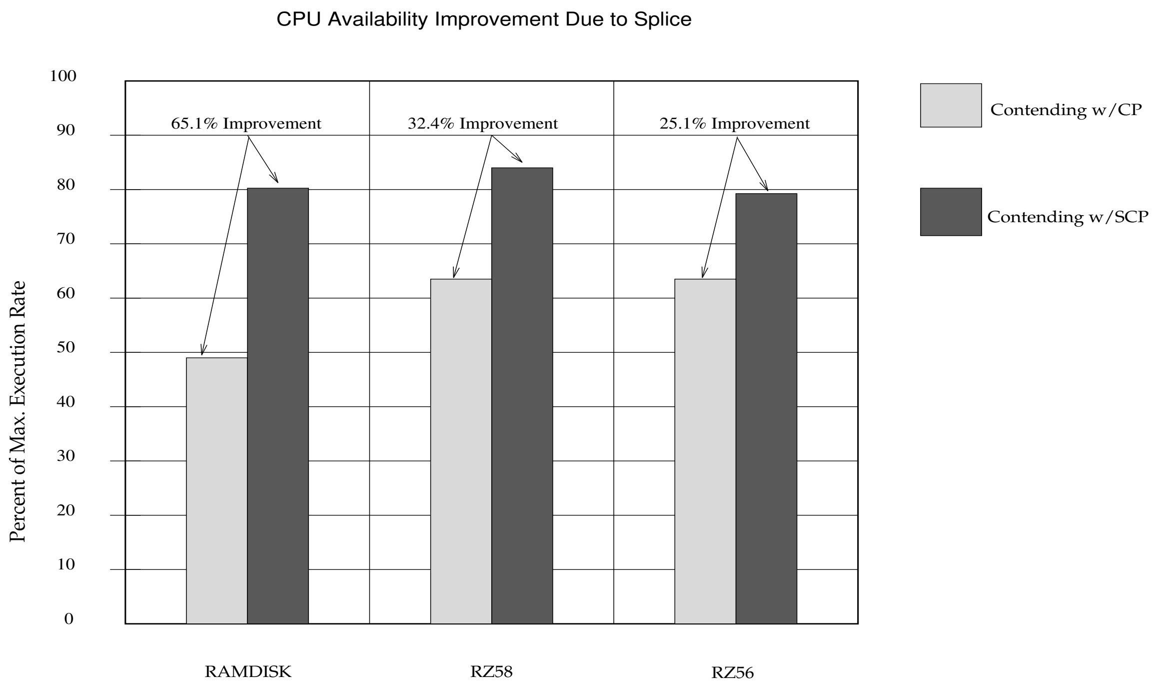

Figure 6.1: CPU Availability comparison of CP and SCP environments (8MB file copy).

Finally, column four indicates the percentage improvement in execution rate when executing in a SCP environment compared to a CP environment; an improvement in execution rate of 100% is a doubling in execution speed. This improvement factor is a measure of how much more of the CPU becomes available to a process as a result of the efficiencies due to `splice`-based I/O over `read/write`-based I/O. For example, when using RAMDISKs, a program's execution rate will improve by 65.1%, effectively shortening the execution time by a factor of approximately $\frac{3}{5}$ due to splice-based I/O (rather than using `read/write`).

When contending with `cp`, the test program executes between $\frac{1}{2}$ and $\frac{2}{3}$ of its speed without contention. However, when contending with `scp`, which uses the CPU and memory more efficiently when doing I/O, the test program executes at $\frac{4}{5}$ or more of its speed without contention. Thus, processes will experience a 25 to 65 percent improvement in execution speed when contending with `splice`-based I/O versus `read/write`-based I/O, depending on the device speeds. With faster devices, `splice`'s effect on performance improvement becomes more dramatic.

Throughput Tests

To measure device-to-device file I/O throughput, a read cache cold start condition was ensured by performing large file I/Os through the buffer cache before taking measurements. Write-through behavior for the cache in the case of writes was ensured by using only asynchronous writes for SCP and calling `fsync()` on the destination file for CP. Many of CP's delayed-write blocks are forced to disk in any case because the file sizes tested are larger than the buffer cache size.

Table 6.2: Throughput Improvement with Splice based on environment.

Disk Type	SCP Throughput (MB/s)	CP Throughput (MB/s)	%-Improvement of `SCP` over `CP`
RAM	3.34	1.88	77.7%
RZ58	0.59	0.55	7.3%
RZ56	0.37	0.37	0.0%

Table 6.2 and Figure 6.2 shows the achievable throughput using `scp` vs. `cp` when copying files. For the throughput tests, the test program used to produce Table 6.1 is disabled, so the figures in Table 6.2 represent maximum attainable throughput measures assuming an otherwise idle CPU. Column one indicates the disk type, columns two and three represent the throughputs measured for copying an 8MB file using `scp` and `cp`, respectively. The fourth column indicates the percentage improvement in throughput of `scp` compared to `cp`. Thus, splice-based copying can operate at just under 1.8 times the maximum throughput of read/write-based copying using fast devices (in this case, RAMDISKs). However, when using relatively slow devices such as today's SCSI disks, the disk transfer time dominates the overall throughput measurement and the benefit of splice is minor.

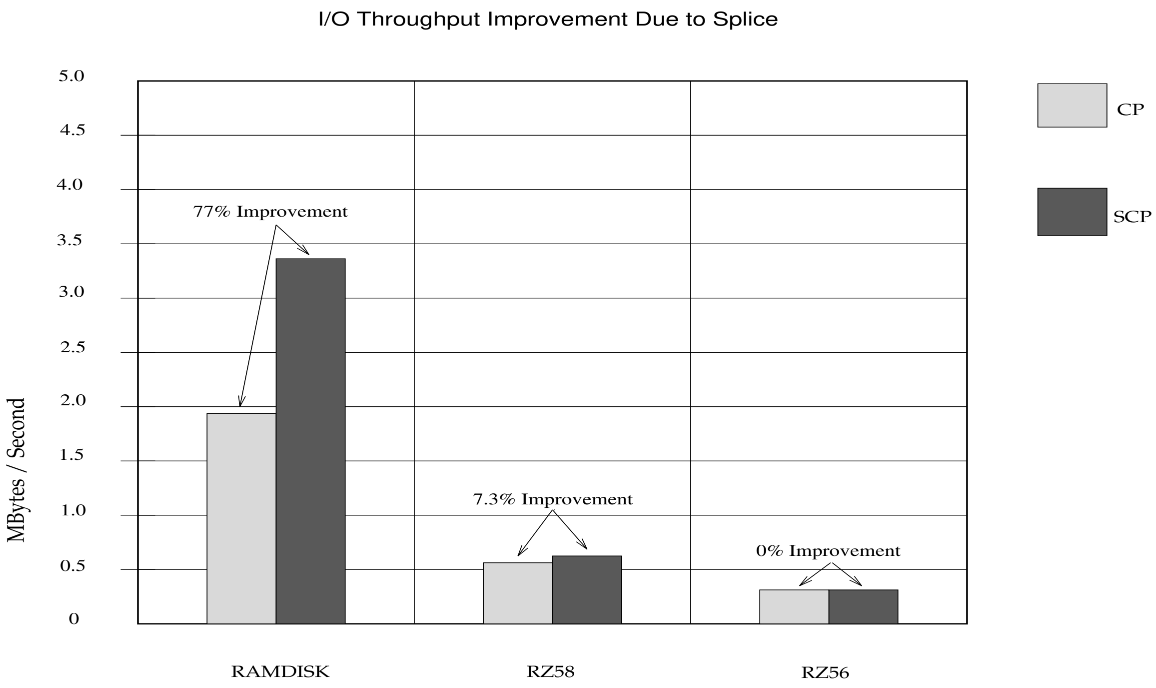

Figure 6.2: Throughput Performance of Uncontested `CP` and `SCP` I/O Programs

Discussion

The performance improvements achieved by splice result from two modifications to the I/O subsystem:

- shortening the path data must travel between devices by eliminating the need to move data to and from user space
- bypassing context switch overhead between the reading of the input device and writing to the output device, leaving flow control and timing of block transfers (within a single splice operation) to the kernel

Except for modifications for non-blocking behavior, no fundamental modifications have been made to the buffering, scheduling, or block allocation strategies present in most UNIX systems.

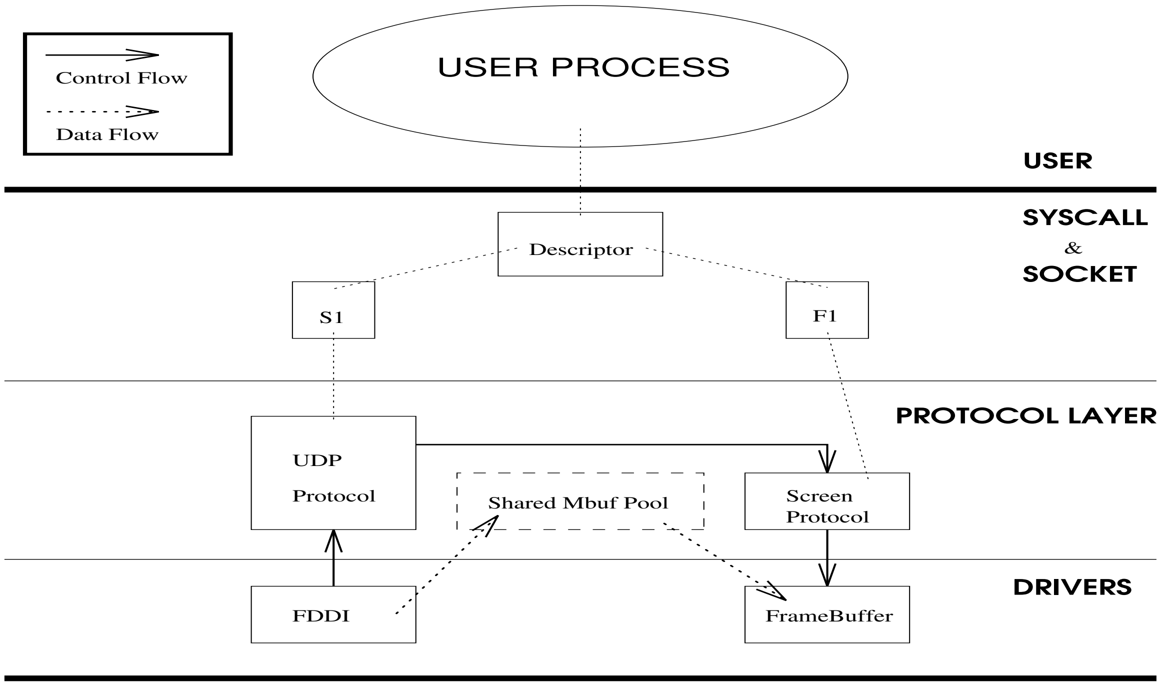

Figure 6.3: Network-to-Framebuffer Splice Implementation at the Receiver

6.2 Experiment 2: Network to Display

This section describes a network–to–framebuffer (NtoF) prototype implementation of PPIO and the `splice` system call. The NtoF splice is representative of the task required at a multimedia receiving station and is described in [FP94]. Figure 6.3 illustrates the NtoF splice implementation described below and used for performance evaluation. A Berkeley–based network structure [LJFK86] was used to construct the prototype.

The `splice` call provides sufficient information for the operating system to manage the flow of data between the I/O objects specified by the calling process. For any active splice, a dynamically allocated kernel–resident *association descriptor* (AD) maintains references to socket or file structures plus any additional information required by the implementation.[3] For Internet domain sockets, a new field in the `inpcb` structure indicates the presence of an active splice for the associated socket by pointing to a valid association descriptor.

[3] The descriptor provides enough information for splice to operate asynchronously as previously described, although this feature has not been exercised.

Implementing a prototype NtoF splice within the Berkeley network code framework is straightforward. The Berkeley network framework essentially uses an upcall[4] scheme until data is delivered to a *socket buffer*. When a socket buffer is ready, a user process is awakened which continues its kernel–mode execution, copying data in the socket buffer (in the form of `mbuf` data structures) to its own contiguous user address space.[5] By intercepting the upcall before an append operation is performed on the socket buffer, flow of control may be diverted for splice processing.

At the time flow of control is diverted to splice processing, data received from the network is in the form of *mbufs*. Mbufs represent the network buffer abstraction present in most Berkeley–derived network implementations. Received packets are placed in mbufs or a linked list of mbufs called an mbuf *chain*, and may be coalesced as required by protocol processing. Using the IP protocol [ISI81], packet sizes are limited to 64KBytes which is adequate to hold most compressed frames (for example, the largest JPEG–compressed image we have encountered is about 20KBytes, with typical values of 7–8KBytes).

6.2.1 Performance Experiments

We constructed an experiment designed to simulate the video display environment of Figure 6.4. The experiment illustrates the relationship between overhead introduced by protocol processing and overhead associated with data copies and checksum operations experienced at the receiving workstation.

Experimental Setup

Figure 6.5 illustrates the experimental setup. The throughput performance of a continuous media receiving program using the *splice* mechanism to move data between the network and display device was compared against an equivalent user–level process implementation. The effect of protocol checksum computation on overall throughput was also included in the study.

[4]Clark [Cla85] describes the notion of upcalls in detail, but does not discuss the Berkeley networking implementation, which does not perform an upcall all the way to user space.

[5]Scatter/gather operation is supported with the `readv()` and `writev()` system calls.

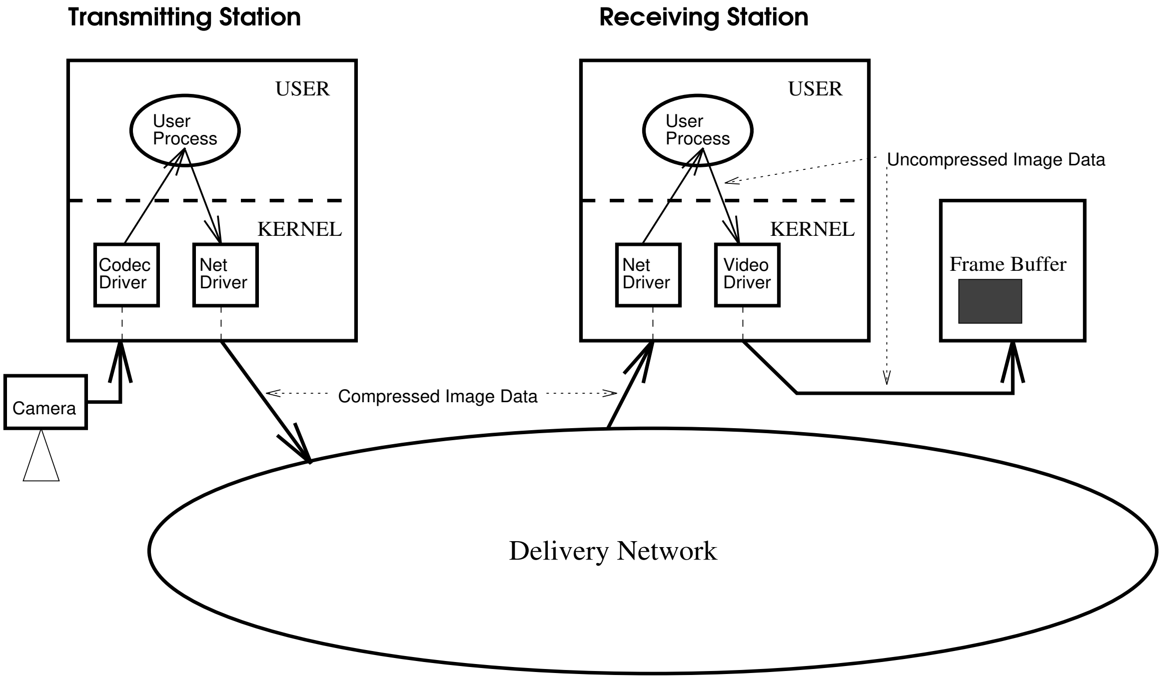

Figure 6.4: Multimedia Distribution Environment

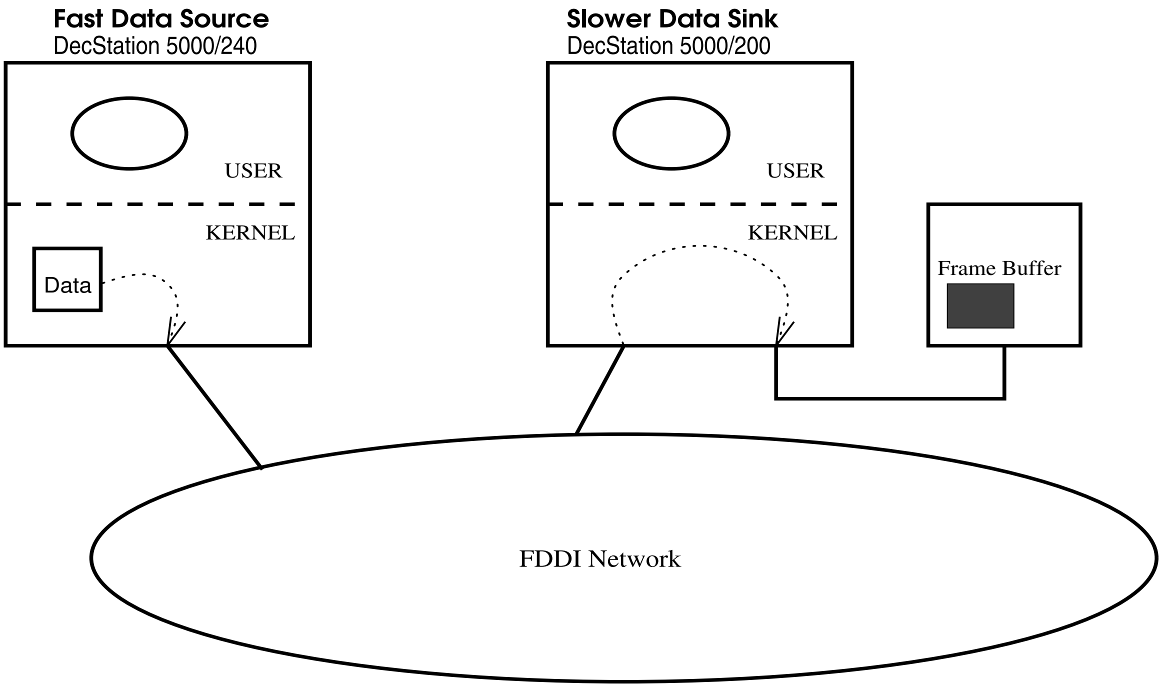

Figure 6.5: NtoF Splice Test Setup

The experiment consists of a DECStation 5000/240 data sender and DECStation 5000/200 data receiver connected by an FDDI network. They have 25MHz and 40MHz R3000A CPUs, respectively. Their SPECint ratings are 19.5 and 27.9. Each system runs the Ultrix 4.2A operating system modified to support the *splice* mechanism. The faster sender simulates a more powerful computational resource at the data source, as might be typical of a video file server application. Theoretical FDDI bandwidth is 100 Mbit/s, although measured performance at the network interface (a DEC DEFZA FDDI adapter, which provides no send–side DMA) is 56-64 Mbit/s [KP93].

For the sender employing `splice`, 12,000 bytes of 8-bit image data are sent per frame, divided into three FDDI packets using the UDP transport protocol [Pos80] to send a sequence of 50 frames (150 FDDI packets). Image data is stored in a statically allocated kernel memory region in the sender, and is passed directly to the network protocol subsystem to be sent. For the user–level sender, the same image data is statically allocated in a user process, and sent employing the UNIX socket layer [LJFK86]. A data copy is performed between the user and kernel address spaces.

For the user–level receiver, incoming data is passed up through the UDP protocol layer, delivered to the socket layer, and the receiving application is scheduled to run and eventually copies data from the network buffers to its own user-level buffers. With `splice`, data is not delivered to the socket layer, but is instead moved directly to the destination (e.g., the frame buffer) in the context of a network software interrupt handler. The data appears in the frame buffer as a 120x100 video window.

Data Loss at Receiver for User–Level Process

The UDP transport protocol provides no flow control or error recovery. Flow control is generally necessary when a sender is faster than the associated receiver, to avoid buffer overrun at the receiver. In this experiment, the splice–based mechanism at the receiver allowed data transfer to occur between data source and sink with no data loss. However, the user–process implementation experienced considerable data loss.

For the user–level experiment, if the sender is permitted to send as fast as possible

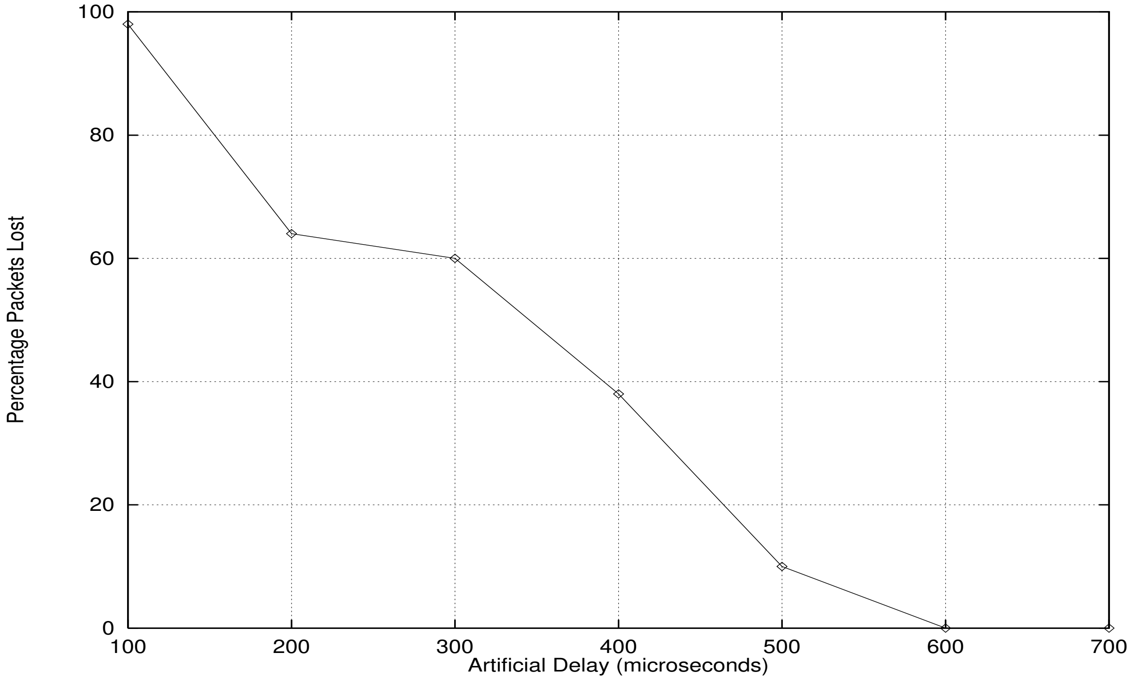

Figure 6.6: UDP Packet Loss vs. Artificial Delay (checksum enabled)

(not flow controlled in any way), approximately $\frac{2}{3}$ of the data is lost at the receiver. To avoid buffer overrun at the receiver during the user–level test, a rate–based flow control mechanism for UDP was constructed by introducing artificial delay between network packets at the sender. Figure 6.6 illustrates the percentage of packets lost as a function of the amount of artificial delay introduced at the sender. Because the amount of delay required to avoid overrun is a function of processing load at the receiver, the correct setting of time delay depends on whether or not checksums are computed. The optimal delay value is the smallest amount of time which still permits all data to be delivered successfully. We empirically determined 600 microseconds to be the optimal value when checksums are computed; 400 microseconds is used when checksums are disabled.

In ordinary operation, UDP performs a checksum to detect corrupt data and discards datagrams whose checksums fail. The checksum operation is the dominant processing overhead associated with UDP/IP [CHKM88]. For uncompressed video, corrupt data could be delivered to the frame buffer without catastrophic result. In addition to removing checksumming for loss–tolerant data such as uncompressed audio or video, checksums may be

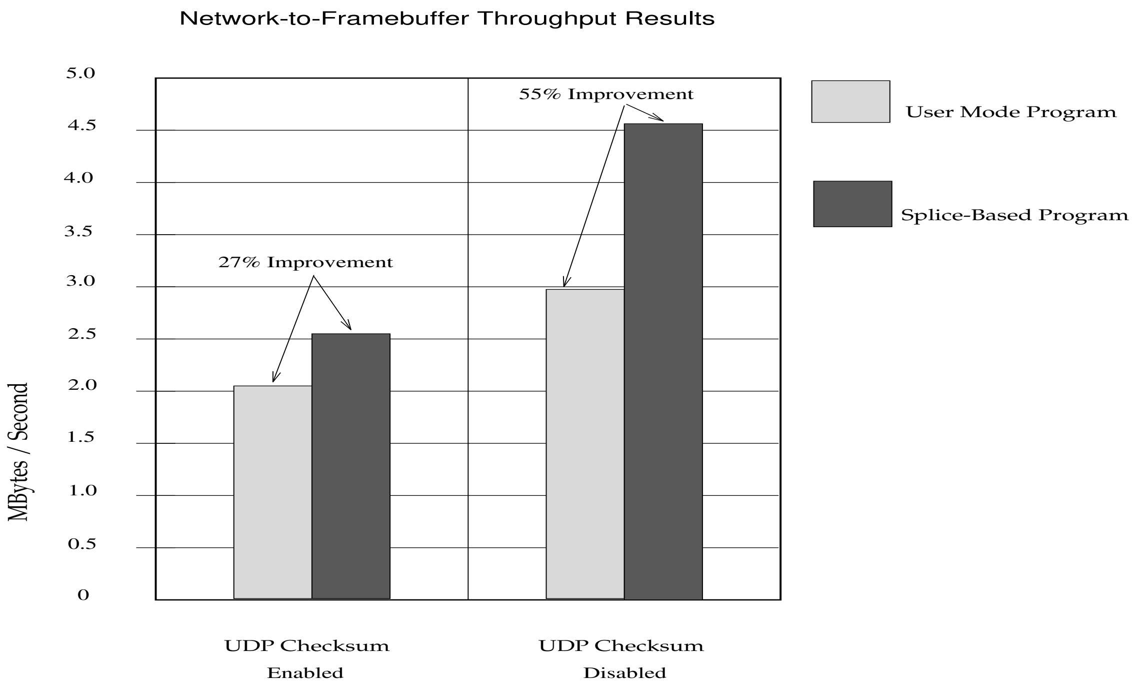

Figure 6.7: Net-to-Display Throughput (Splice vs. User Process)

disabled in certain restricted operating environments (e.g., when hardware checksumming is performed [KP93]). To evaluate the relationship between splice and checksumming, the experiment was performed with checksums enabled and again with checksums disabled.

Measured Results

Figure 6.7 depicts the differences in measured throughput in MBytes per second between the conventional user-mode program and the splice-based implementation. Note that the user mode program uses *memory mapped I/O* for frame buffer access thus avoiding a data copy when writing to the framebuffer. The graph shows differences between user mode-based data transfer (not using splice) versus splice-based data transfer, both with the UDP checksum computation enabled and disabled.

Discussion

The results show that throughput performance is improved by a factor of 1.25–1.5 when using splice, depending on whether the checksum computation is enabled or disabled.

The checksum computation is the dominant factor in UDP processing; with it removed, data copying becomes the dominant factor. Because splice directly addresses the problem of data copying, it combines synergistically with elimination of the checksum. The combined result offers a greater than doubling in throughput.

6.3 Experiment 3: Network to Network

The previous two experiments reported throughput gains achieved with an I/O structure based on the guidelines of PPIO. These tests measured a substantial throughput increase by performing *macro* experiments, and did not precisely account for where the end-to-end performance gain comes from. In this experiment, *micro* measurements are used to account for the specific tasks improved by the PPIO approach, and to what extent performance is improved. In addition, the stability of such a configuration under overload is investigated.

6.3.1 Purpose of Experiment

The purpose of the Network-to-Network (N2N) experiment is to use the previously described `splice` mechanism to interconnect two network streams at the application layer and compare the *latency* and *stability* of this configuration against that of a conventional interconnection (i.e., using regular user processes). Latency refers to the time required to receive an incoming datagram, process it (including protocol and checksum computations), and forward it using another network connection. Stability refers to the system's response to increasing offered load as the network packet arrival rate is increased.

6.3.2 Setup

Figure 6.8 illustrates the experimental setup. Three workstations are assembled as depicted in the figure. The source machine is a DEC 3000/400 (Alpha CPU, 133 Mhz), and the sink machine is a DEC 3000/800 (Alpha CPU, 200Mhz). The test machine is a

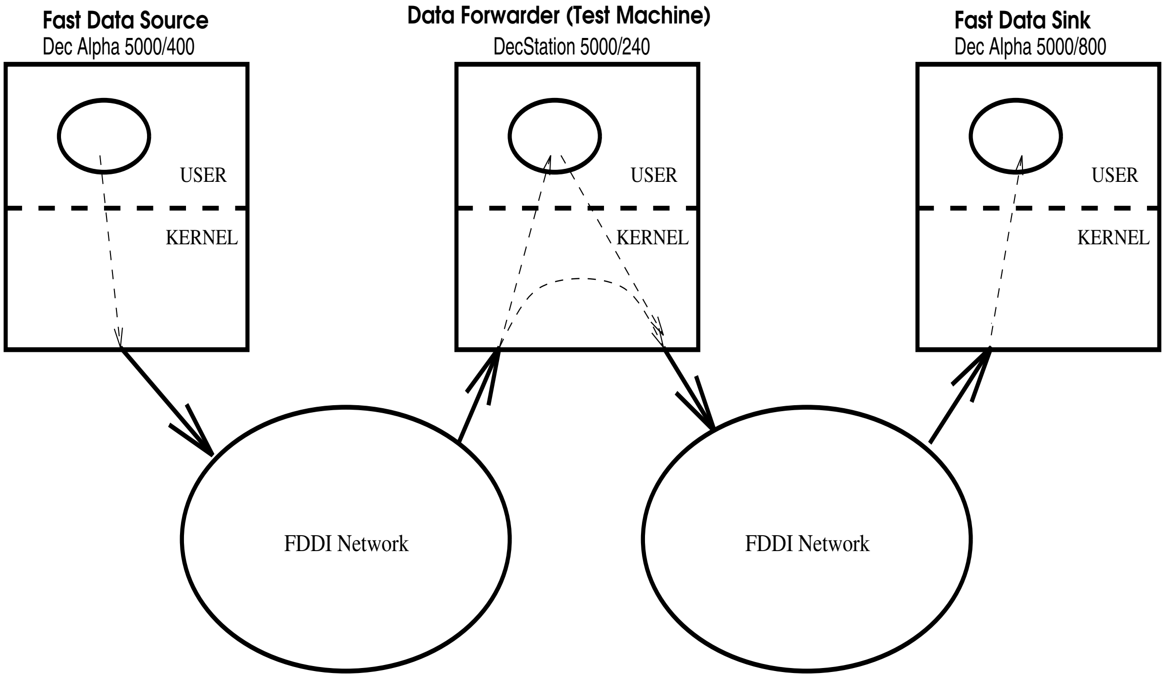

Figure 6.8: Experimental Setup for Net-to-Net test

DECStation 5000/240 (MIPS CPU, 40Mhz). Two separate FDDI networks, one connecting the source and test machines, the other connecting the test and sink machines, are used to transport data using the UDP protocol [Pos80]. All machines are equipped with Digital DEFTA FDDI interfaces. The source and sink Alpha machines are running DEC-OSF versions 1.3 and 2.0, respectively. The test machine runs a version of Ultrix4.2A modified as described below.

Using fast machines for generating and sinking data (as compared with the test machine) provides the ability to measure the degradation experienced by the test machine under overload condition without generating such an overload at either the source or sink.

6.3.3 Operation

The experiment operates by forwarding UDP datagrams between one network and the other using either `splice` or conventional forwarding. In either case, the forwarding operation is known as an *application level gateway*. Data is delivered up to and including the transport layer before it is re-sent to the other network. To be more precise, define the notation $(A, B), (C, D)$ to describe a UDP packet with source IP address A and port B destined for IP address C, port D. A forwarding agent on the test machine F creates receiving UDP endpoints at (F_a, F_r) and a sending endpoint at (F_b, F_s), where F_a is the IP address of F's FDDI interface on network a and F_b is the IP address of F's FDDI interface on network b. The source machine emits packets of the form $(S, S_s), (F_a, F_r)$. These packets are rewritten to the form $(F_b, F_s), (D, D_r)$ where D is the sink machine's IP address and D_r is its receiving port.

The application level gateway forwarding is of practical importance for conditions in which specialized policy may be introduced on a per-port or per-service basis.[6] This type of forwarding may be achieved either by a system providing `splice`, in which case forwarding is performed at the operating system layer (or below), or by a conventional user-level program. In the user-level program, the required operations are as follows (in the `splice` case, the loop is replaced with the `splice` call):

1. create two UDP endpoints (send and recv)
2. *bind* receive side to F_r
3. Loop:
   (a) read data from network
   (b) write data to network

6.3.4 Implementation

The N2N UDP implementation is on a DECStation 5000/240 running a version of Ultrix 4.2A modified to support the splice mechanism and the DEC DEFTA FDDI.[7] The 4.2A networking subsystem is a derivative of the commonly available Berkeley BSD networking implementation. In this environment, creating the N2N UDP splice was straightforward.

An association descriptor is maintained in the Berkeley networking implementation in the `inpcb` structure. Incoming packets are inspected, and the destination port

[6] For example, in the use of *firewalls*.

[7] The DEFTA interface was not available when Ultrix4.2A was released, but a suitable device driver appears in later versions of Ultrix and DEC OSF/1.

number is used to associate an incoming datagram with its intended receiving process. Modifying the `inpcb` structure in a way similar to that previously described in the NtoF experiment allows incoming packets to be directed up the protocol stack and immediately down again without having to execute the calling user process. The kernel-resident *splice descriptor* holds all information required by the system to forward a received datagram.

End-to-End Forwarding

The end-to-end forwarding task measured in this experiment consists of several constituent operations. The salient performance differences between forwarding at the user level and forwarding using `splice` arise due to differential work performed in two fundamental categories:

- Data Manipulation
- Process Manipulation

In the case of UDP forwarding with `spice`, all process dispatching and scheduling overheads are eliminated. In addition, data manipulation (both copying and protocol processing) is greatly reduced.

To better understand the difference in work required between the user and splice-based forwarding schemes, the specific operations required by the user-level forwarder may be compared with those required by the splice-based forwarder. The user-level forwarder (and its supporting kernel) performs the following major operations:

1. handle hardware interrupt (receive FDDI packet)
2. queue received packet(s) and schedule IP protocol
3. run IP protocol via software interrupt (an upcall)
4. run UDP protocol and queue datagram for user process
5. switch to user process (performs copy in kernel mode)
6. run user process (completes read, performs write)
7. copy data to network queue structure (kernel mode)

8. run UDP protocol
9. run IP protocol
10. run FDDI driver
11. run user process (to restart cycle)

The user-level forwarder performs the eleven operations listed above. The splice-based forwarder can reduce this execution path and instead performs only the following eight operations:

1. handle hardware interrupt (receive FDDI packet)
2. queue received packet(s) and schedule IP protocol
3. run IP protocol via software interrupt (an upcall)
4. run UDP protocol
5. pass through splice (will modify UDP headers)
6. run UDP protocol
7. run IP protocol
8. run FDDI driver

As can be seen from this comparison, the splice-based forwarder has about $\frac{3}{4}$ of the number of operations to perform as compared to the user-level forwarder. In addition, the PPIO approach can make an efficient optimization not available to the user-level forwarder with respect to protocol processing.

UDP Protocol Processing and PPIO

The UDP protocol is a "thin" protocol, providing little functionality beyond the features of IP [ISI81]. For this reason, it is a useful protocol for evaluation of I/O and network architecture performance because UDP does not include any dynamic control which might perturb experimental results. Although UDP provides little functionality, it does provide demultiplexing to the process level and a data checksum not present in IP. The most costly operations (for large datagrams) in the processing of UDP is well-known to be the checksum computation and copying between user and kernel address spaces (for examples, see [CJRS89] or [KP93]).

The user-level forwarding breakdown listed above includes the checksum and copy operations in operations numbered 4,5,7 and 8. Operations 5 and 7 are data copies; 4 and 8 are checksum computations. Although `splice` obviously eliminates operations 5 and 7 because no user level process is directly involved in the data transfer, it can also improve the checksum by eliminating its computation in the outgoing case.

Computation of the UDP checksum for the forwarding operation is performed by the user-level forwarder at steps 4 and 8. It is performed twice, once on the incoming side (step 4) for correctness, and once on the outgoing side (step 8) to establish a new checksum for subsequent transmission. Although the data portion of datagrams remain invariant across the forwarding operation, the UDP header *is always* changed and arbitrary datagram data *might be* changed by the user process. The operating system's protocol implementation has no knowledge as to whether the user process has actually modified any data and is therefore forced to compute a checksum covering both the new UDP header *and* the entire data portion.

In contrast to the user-level forwarder, the `splice` implementation has full knowledge over what end-to-end code path is executed during the forwarding operation, and can ensure no UDP user data has been modified since the moment it was received. Thus, the new (outgoing) UDP datagrams contain the same user data as when they were received, but with different header information only. This fact can be used to create a checksum for the outgoing datagram by computing only an *incremental checksum*.

Due to its dramatic impact on protocol performance, considerable effort has been invested in the development of efficient methods for computing the UDP (Internet) checksum [BP88]. Recently, *incremental update* methods for computing the checksum have been refined [Rij94]. With incremental update, only the changes in a message need be summed. The N2N UDP splice makes use of incremental checksum computation, and computes the outgoing checksum as follows:

$$new\_sum = orig\_sum - sum(orig\_header) + sum(new\_header)$$

Fortunately, the UDP header is of small and fixed size (8 bytes), and the computation runs in $O(1)$ time rather than $O(n)$ time for n bytes of data, as is characteristic of the traditional checksum performed by the user-level forwarder.

6.3.5 Methodology

To obtain accurate performance measurements of latency and stability, the Ultrix 4.2A kernel clock handling routines (e.g., `microtime()`) were replaced, boosting the clock resolution from 4ms to about 1 microsecond [Mil94]. In addition, the executed code path was instrumented with trace points. At each tracepoint, timestamps are written to a dynamically allocated kernel buffer. Timestamps are retrieved from the kernel by a user process at the conclusion of each experiment. The timestamps are used to compute a number of statistics described below.

Generally, all measurements are made both with UDP checksums enabled and disabled to compare the difference. When splice is used with checksums enabled, the incremental checksum technique described above is used.

Trace Points

For the user-level forwarder, the following ten trace points (points at which timestamps are collected) are used to measure latency of individual operations:

ENQ hardware interrupt handler queues incoming datagram
IPI IP protocol is invoked by software interrupt
WSO UDP protocol queues datagram in *socket buffer*
BSO user-level forwarder completes copy from network to user buffer
SST user-level forwarder re-enters kernel by system call (writing)
UDP UDP protocol (output) started
OUT IP protocol completed
DRV send device driver code completed
SSB user-level forwarder completes kernel-mode execution
TSO user-level forwarder re-enters kernel and blocks for reading

The time differences between adjacent trace points account for the times to complete required processing operations:

IPI-ENQ dispatch software interrupt and invoke handler
WSO-IPI UDP and IP input processing, including checksum
BSO-WSO wake-up user process and perform network/user data copy
SST-BSO run user process and enter kernel
UDP-SST perform user/network data copy
OUT-UDP UDP and IP output processing, including checksum
DRV-OUT driver transmit overhead
SSB-DRV return to user process (kernel mode)
TSO-SSB run user process and re-enter kernel

For the splice-based forwarder, the following six tracepoints are used:

ENQ hardware interrupt handler queues incoming datagram
IPI IP protocol is invoked by software interrupt
WSP invoke `splice` code
UDP UDP protocol (output) started
OUT IP protocol completed
DRV send device driver code completed

The metrics derived from the differences between adjacent tracepoint are analogous to those described for the user-level forwarder.

For measuring stability, a *macro* type experiment is used. The forwarding system is subjected to an increasing packet-per-second load with a fixed packet size, and the number of successfully forwarded packets is recorded.

Statistics

The timestamp difference metrics described above are used to measure the time overhead associated with each step in the forwarding process. In addition, **TSO-ENQ** and **DRV-ENQ** are used to evaluate the overall end-to-end forwarding time in the user-level and splice cases, respectively.

For the latency tests, a sample size of 200 packets of each size along the interval 1. . . 4000 bytes with increment 100 bytes are measured with a constant packet inter-arrival time of approximately 20ms. This period ensures the forwarder will not become overloaded by too high an arrival rate, and has been determined empirically. The stability experiments, described below, examine the effect of increasing the arrival rate.

The arithmetic mean of each time metric is computed in addition to the mean of the end-to-end forwarding times, and each mean calculation is accompanied by a confidence interval at the 95% level of confidence.

For the stability tests, two distinct packet sizes are chosen for experimentation: 1500 and 4000 bytes. The packet arrival rate is varied between 500 packets per second (pps) and 4500pps with an increment of about 200pps. A sample size of 10000 packets are sent for each packet rate. The 1500 byte packet size is a reasonable choice considering the wide availability of ethernet networks with 1.5KByte MTU sizes and the increased deployment of MTU Path Discovery [MD90], which will take advantage of this size over the default size of 576 bytes. The 4000 byte size is used to measure stability at maximum throughput (up to the FDDI rate of 100Mb/s).

For each packet rate, the number of successfully forwarded packets is recorded and divided by the number of transmitted packets to produce an estimate of the probability of successful packet delivery $\hat{p}$. Assuming the probability of successful forwarding follows a Bernoulli distribution with parameter p, $\hat{p}$ is a maximum likelihood estimator of p [All90](p. 435).

6.3.6 Results

Latency Experiment

Figure 6.9 illustrates the comparison of means of the overall end-to-end forwarding time for the user-level forwarder and the splice-based forwarder with and without UDP checksums. The small vertical bars represent confidence intervals at a 95% level of confidence. For 1 byte packets (the left side of the graph), the checksum and data copy overheads are dwarfed by other protocol processing and process manipulation. The user-level forwarder takes about 780 microseconds, and the splice-based forwarder takes about 300 microseconds, representing an improvement in latency of a factor of 2.6. In other words, the splice-based forwarder requires only about 40% of the time the user-level forwarder requires.

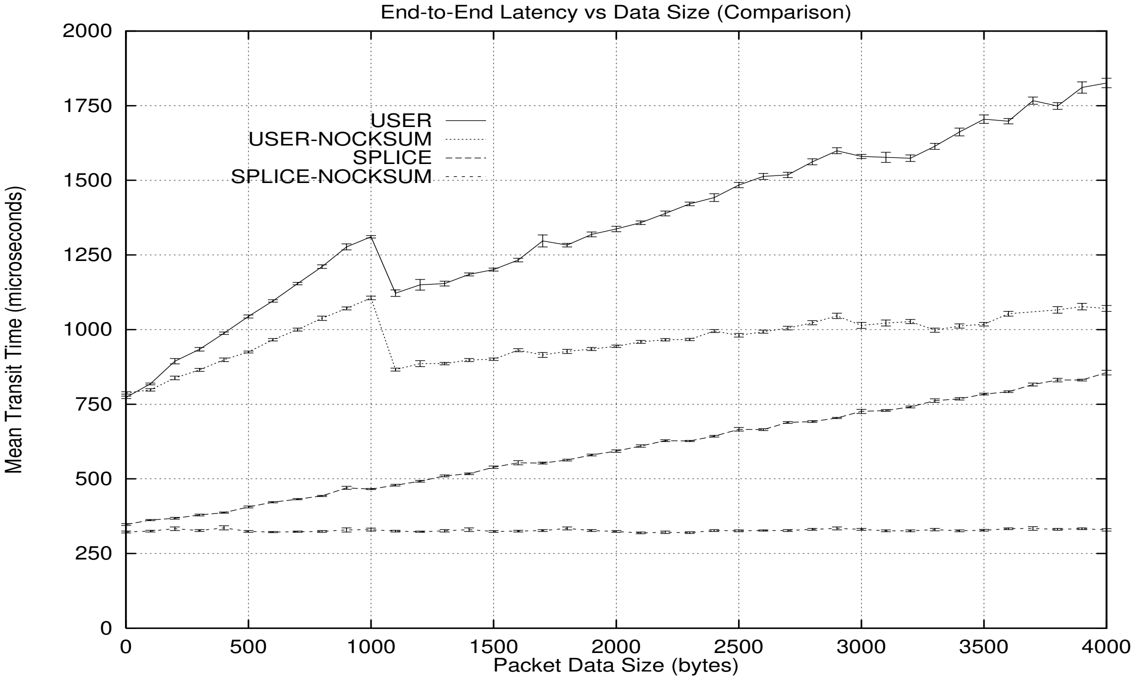

Figure 6.9: Comparison of forwarding latency, user vs splice.

For larger (4000 byte) packets, checksum and data copies dominate the time required for the user-level forwarder to operate. With UDP checksums enabled, the user-level forwarder requires 1826 microseconds, while the splice-based forwarder requires only 876 microseconds. Thus, the latency reduction due to `splice` is about a factor of 2. With UDP checksums disabled, the user-level forwarder is able to complete a packet forwarding in 1071 microseconds, while splice is able to do so in 329 microseconds. This represents a latency improvement of a factor of 3.3, or more than a tripling in latency improvement.

In Figure 6.9, the user-level forwarders experience a nonlinearity between 1000 and 1100 bytes. This discontinuity is a result of the kernel routine `sosend()`, which is invoked to copy user data from a application's buffer to a linked list of kernel-resident network buffers (*mbufs*) on output. If `sosend()` is given a user buffer less than `NCLBYTES` in size, it creates a chain of mbufs, each of which contain `MSIZE` bytes of data. If the user buffer size exceeds `NCLBYTES`, it will instead opt for using mbufs of size `M_CLUSTERSZ`. In Ultrix4.2A, the values of `NCLBYTES`, `MSIZE`, and `M_CLUSTERSZ` are 1024, 128, and

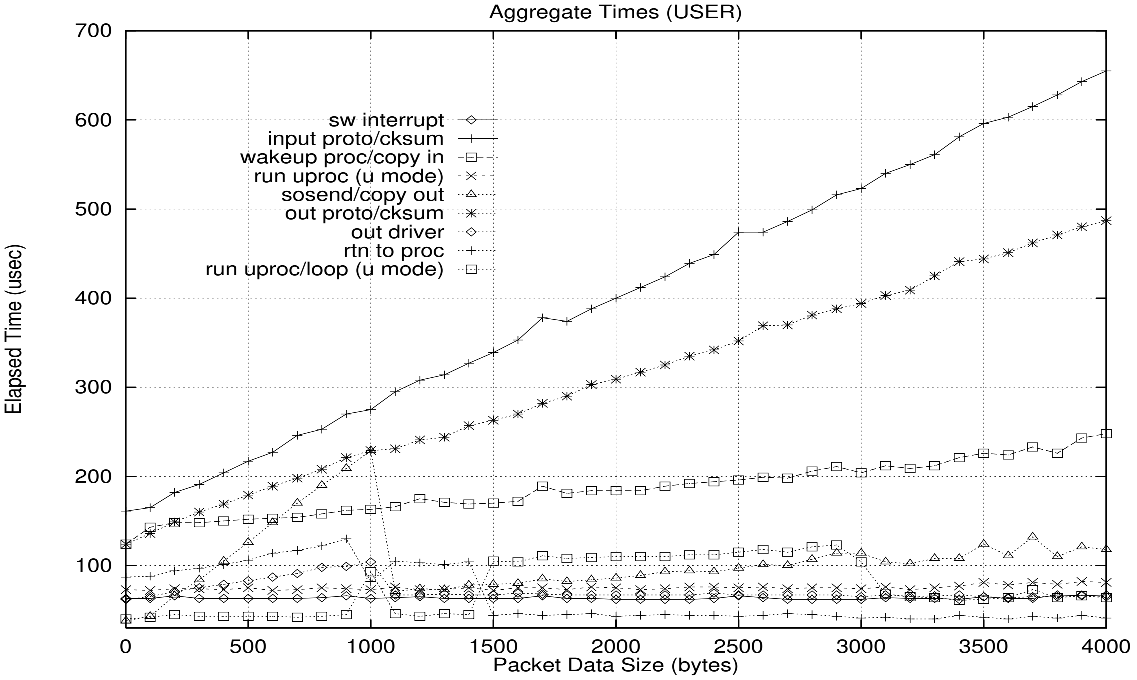

Figure 6.10: Times for aggregate operations (USER)

4096 bytes respectively.[8] Thus, once the 1024 byte threshold is crossed, a single mbuf is used until the next increment of 4096 bytes is reached. Splice does not exhibit the same discontinuity because `sosend()` is never invoked in the splice forwarding case.

Figures 6.10, 6.11, 6.12 and 6.13 indicate the time spent in each required processing operation listed in Section 6.3.5 above. In Figure 6.10, the most costly operations for the user forwarder are the protocol processing operations (including checksums) and incoming data copy. For single-byte packets, these measurements represent the fixed per-packet costs, because the checksum and copies need only manipulate a single byte. Thus, the lower bound on the time consumed by the checksums together is $161 + 124 = 285$ microseconds of the 780 total time, or about 37%. The next most costly operation is the incoming data copy (again, of a single byte). The lower bound for the copy is 124 microseconds or 16%. The remaining 47% of the overall time is spent in process manipulations and running the DEFTA device driver.

[8]These values may be modified by changing the file `mbuf.h` and recompiling the operating system from sources.

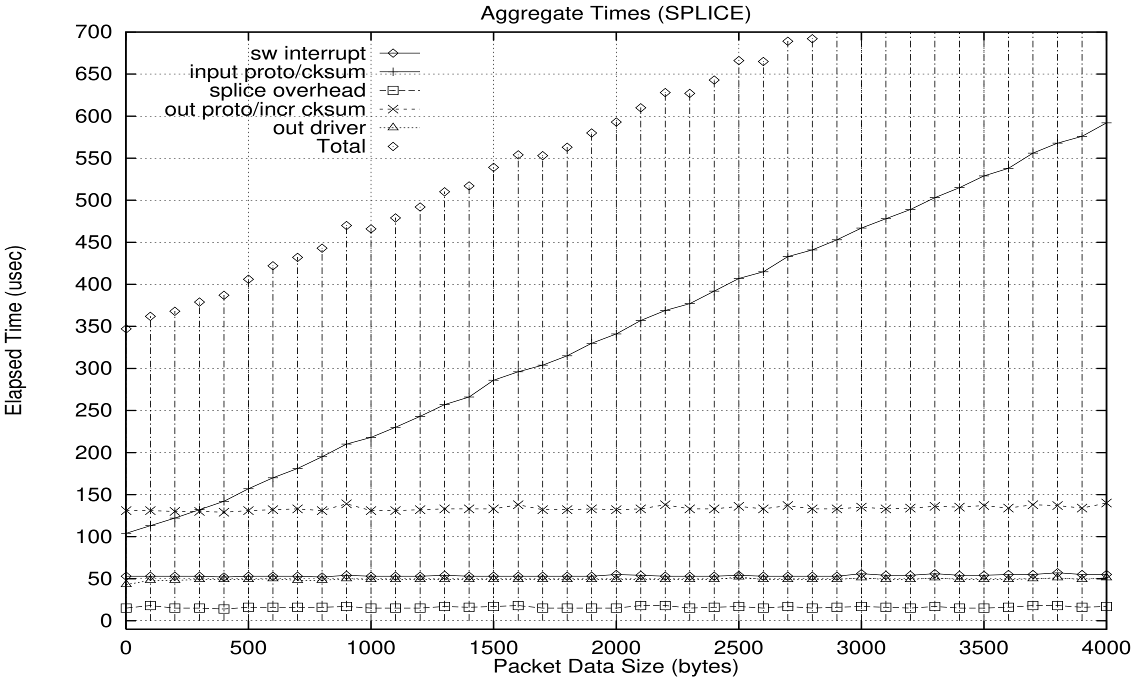

Figure 6.11: Times for aggregate operations (SPLICE)

For larger (4000 byte) packets, Figure 6.10 indicates the impact of the protocol processing and checksums to be $655+487 = 1142$ microseconds, or 63% of the overall 1826 microseconds. The data copies account for an additional $245 + 118 = 363$ microseconds, or 20%. The remaining 17% is spent manipulating processes and running the device driver. The total time in the driver comprises approximately 7% of the total time at 128 microseconds.

Turning to the performance graph of the splice-based forwarder in Figure 6.11, the topmost line with ordinate value at 350 microseconds and going off-scale at abscissa 2700 bytes represents the sum of the aggregate times and is equivalent to the line labeled SPLICE in Figure 6.9. (The corresponding line for the user-level forwarder was omitted because it is above 700 microseconds for all packet sizes and would have been entirely off-scale.) In Figure 6.11 only the line representing the incoming checksum computation has a non-zero slope, starting with 104 microseconds for 1-byte packets and growing to 576 microseconds for 4000 bytes packets. All other operations are approximately constant, with the outgoing (incremental) checksum requiring 132 microseconds, the combined input

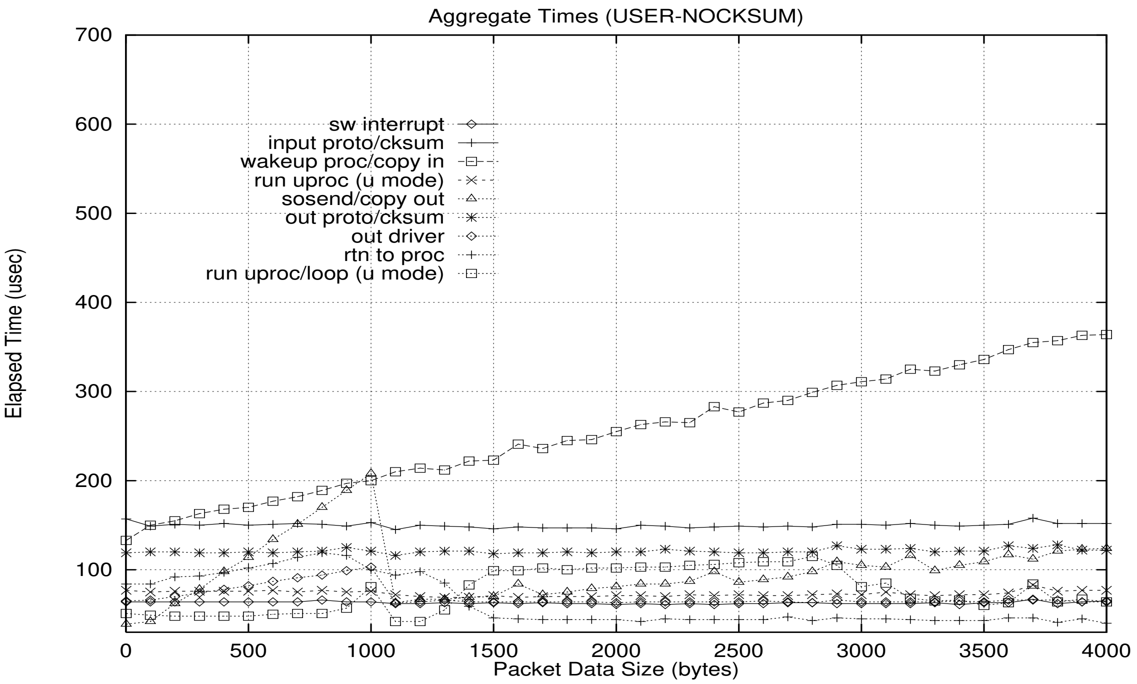

Figure 6.12: Times for aggregate operations (USER-NOCKSUM)

and output driver processing about 100 microseconds, and the splice overhead at about 15 microseconds.

Figures 6.10 and 6.11 reveal several key differences between the performance of user-level forwarding as compared with the splice-based approach. The dominant cost with increasing packet size is the checksum computation. The user-level forwarder must compute the checksum twice, giving rise to 1142 microseconds of overhead, as compared with the splice-based forwarder which, by using the incremental checksum, can accomplish the same task in 710 microseconds for 4000 byte packets. The user-level forwarder loses a total of 366 microseconds to copying data into and out of the user process at this size, while the splice-based forwarder never requires any such copy. In addition, the splice-based forwarder does not exhibit the nonlinearity at the 1Kbyte packet size. Finally, the user-level forwarder spends about 180 microseconds in process manipulation while the splice-based forwarder performs no process manipulation.

In Figure 6.12, the UDP checksums have been disabled for the user forwarder. The time taken by protocol processing remains constant across packet size at $150 + 120 = 270$

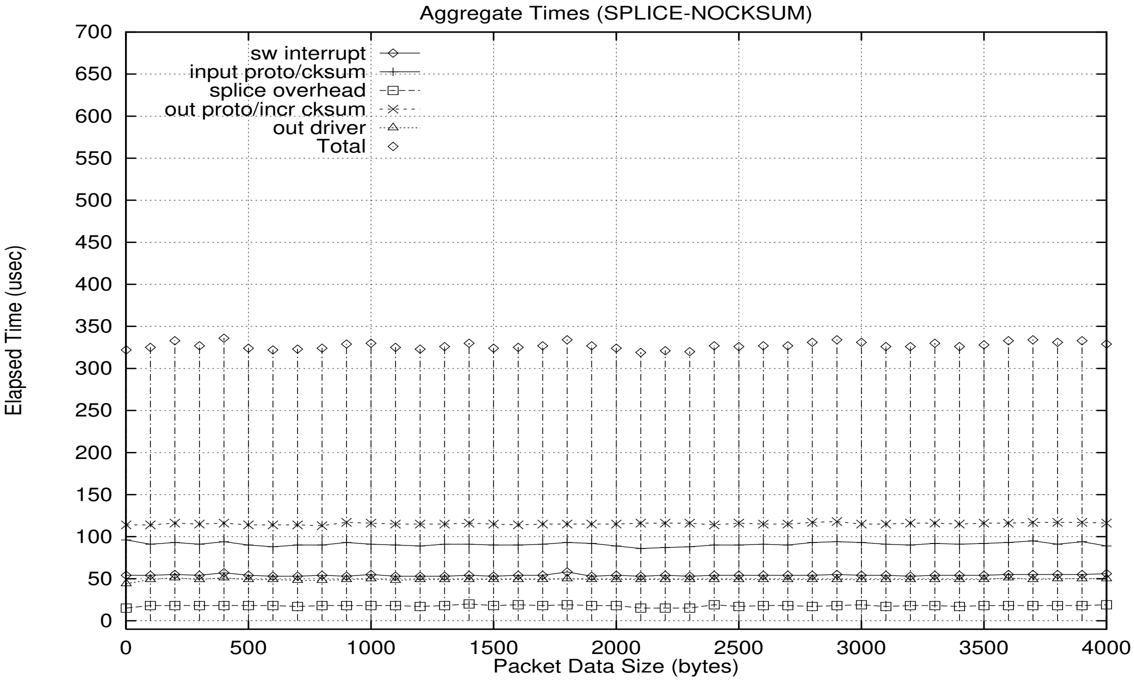

Figure 6.13: Times for aggregate operations (SPLICE-NOCKSUM)

microseconds. Without checksums, the dominant processing operation is the incoming data copy which grows linearly from 133 to 364 microseconds for 1 byte and 4000 byte packets, respectively. The slope of this line is greater than the corresponding line in Figure 6.10, indicating the incoming data copy is more costly when checksums are disabled. This result can be explained by noting that the checksum computation in Figure 6.10 has caused a filling of data cache with packet data, resulting in a lower time to perform the data copy. The MIPS R3000 architecture does not automatically insert I/O data into its data cache when a DMA operation is performed, and any such data must therefore be read from memory rather than cache.

Figure 6.13 depicts the splice-based forwarder with checksums disabled. The top line once again represents the aggregate sum, but now remains isoheightual for all packets sizes. Remarkably, no constituent operation grows with packet size for splice-based forwarding with checksums disabled, suggesting that the adapter/memory DMA transactions executed by the interfaces is not adversely impacting forwarding performance. This is not especially surprising, considering the DEC TurboChannel operates at 100Mbytes/s,

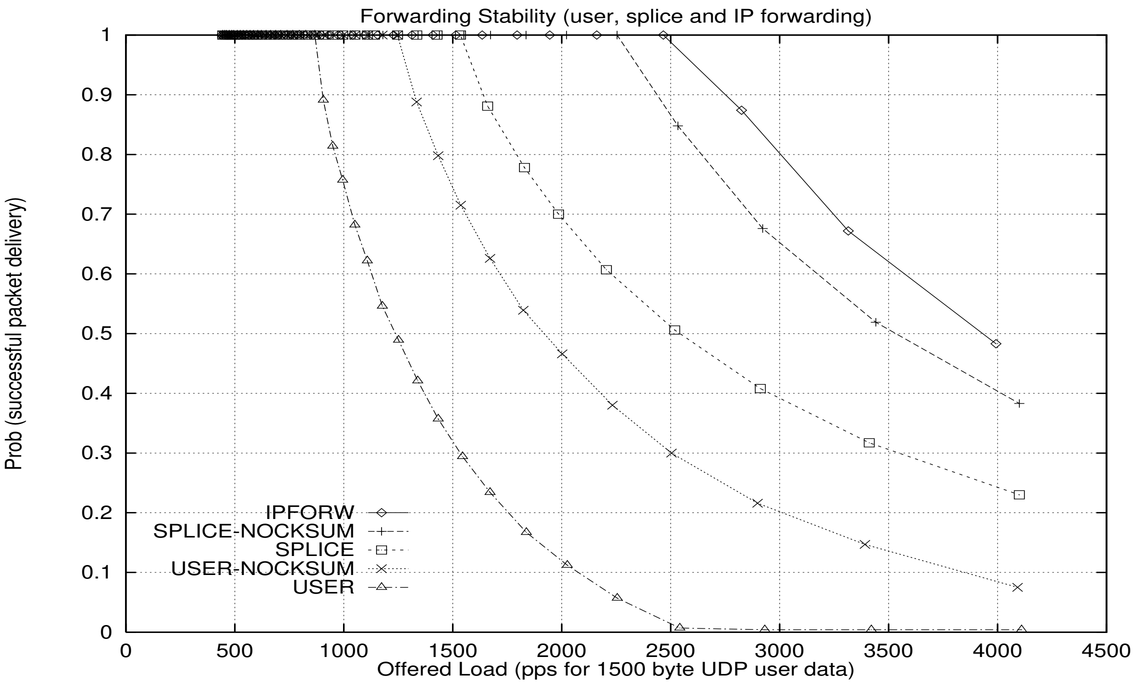

Figure 6.14: Stability Comparison (1500 bytes)

considerably faster than the 100Mb/s FDDI maximum rate.

Stability Experiment

The results of the stability experiments are depicted in Figures 6.14 and 6.15 for 1500 byte and 4000 byte packets, respectively. The probability of successful delivery will drop below 1.0 when the system is forced to drop packets due to a packet arrival rate exceeding its own forwarding rate. In this test environment, discarding is performed at the **ENQ** tracepoint described above when the interface's hardware interrupt handler attempts to add a new packet to the IP protocol queue (`ipintrq`). The maximum length of this queue is defined by the constant `IFQ_MAXLEN` and is typically set to 50 packets.[9]

For 1500 byte packets with checksums enabled, the user-level forwarder becomes unstable (i.e. the forwarding probability drop below 1.0) at about 900 pps as compared with 1600 pps for the splice based forwarder. With checksums disabled, the user-level

[9]This constant is contained in the kernel file `if.h` and may be altered by recompiling the operating system from sources.

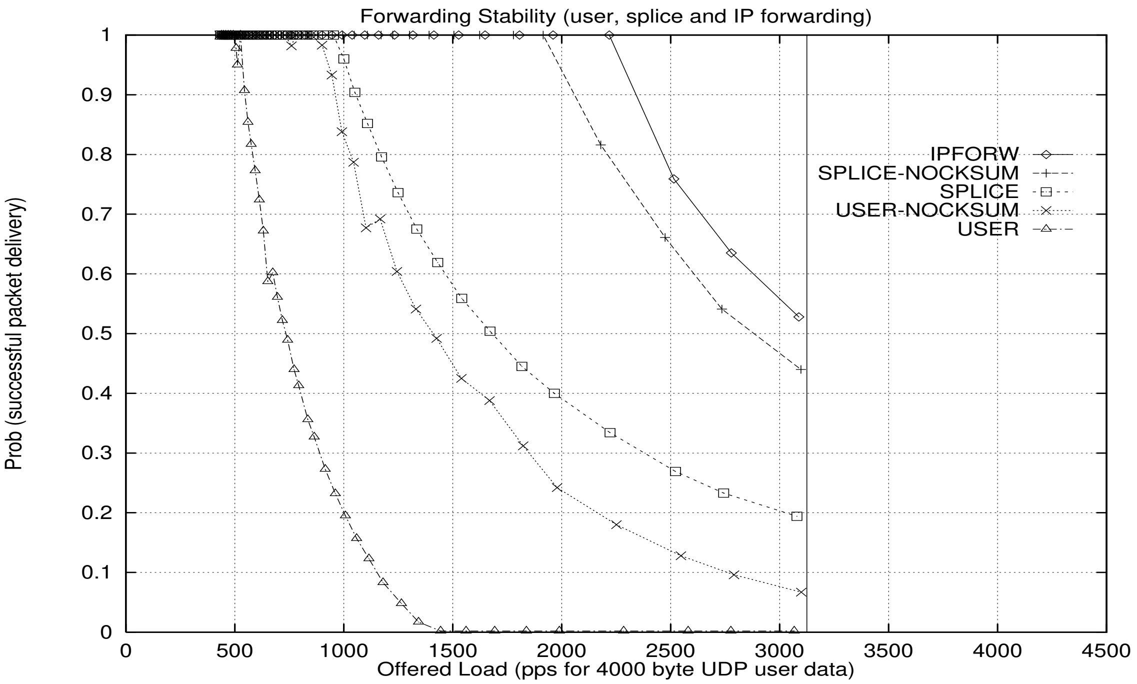

Figure 6.15: Stability Comparison (4000 bytes)

forwarder becomes unstable at about 1300 pps as compared to about 2300 for the splice-based forwarder. In either case, the splice-based forwarder can tolerate a factor of 1.8 greater offered load. For 4000 byte packets, Figure 6.15 indicates the 1.8 improvement factor is maintained when checksums are enabled. When checksums are disabled, the improvement of splice grows to a factor of about 2.4.

6.3.7 Latency and Stability Conclusions

The performance results of this experiment reveals the PPIO approach can be used to significantly reduce end-to-end forwarding latency. For small messages, where the cost of checksum and data copies is small, the PPIO approach requires only 40% of the time required by the conventional user-level forwarder. For larger messages, the cost of checksum and copies becomes dominant. When checksums are enabled, PPIO requires 50% of the time required by the user-level forwarder. With checksums disabled, PPIO requires only 30% of the time.

The result for small messages is significant in light of the many efforts to reduce copying and data checksum computations. For small messages, these operations are of negligible cost, revealing the 60% performance improvement due to PPIO is primarily a result of reducing process manipulation overhead rather than data manipulation overhead. Thus, other systems which attempt to address the data manipulation overheads by techniques such as memory-mapped I/O will still suffer the overheads of process manipulation.

For large messages with checksums in PPIO, only the incoming checksum computation scales linearly with message size; all other overheads are approximately constant and the PPIO forwarder does not exhibit any latency nonlinearities. For large messages with checksums, PPIO gains a significant advantage because of the incremental checksum. This optimization is the direct result of having knowledge about the code path traversed during the forwarding operation and observing that no user data is modified. Similar optimizations are possible in many circumstances with PPIO and are a general characteristic of the approach.

For large messages without checksums, the user-level forwarding time is dominated by data copying. If all data copying were removed from the user-level forwarder, the forwarding time would drop to approximately $1071 - 364 - 120 = 588$ microseconds. Even for a situation so favorable to the user-level forwarder, process manipulation accounts for about 18% of this overhead and the PPIO approach performs 40% better (329 microseconds).

With respect to stability under overload, the splice-based forwarder is able to sustain about twice the packet rate without dropping packets, except in the case for large packets when checksums are disabled, where it is able to sustain about 2.4 times the rate of the user-level forwarder. The point at which packet drops begin is when the packet arrival rate exceeds the forwarding rate, and is thus directly proportional to the reciprocal of the forwarding latency. In terms of maximum throughput (achieved at the 4000 byte packet size with checksums disabled) the best case for the splice-based forwarder is 61.3 Mb/s while the user-level best case is 21.8Mb/s. With checksums turned on, these numbers drop to 30.5Mb/s and 16.8Mb/s, respectively.

Generally, the PPIO approach for this environment leads to a drop in forwarding latency by a factor of between two and three over a comparable user-level forwarder. The result has the direct positive effect of delivering data across the forwarding agent in a more timely manner (reducing end-to-end latency), but also improves maximum throughput by a similar factor as a consequence. As explained, the implementation is straightforward and is able to take advantage of explicit knowledge about the code path taken to forward data.

Chapter 7

Conclusions and Future Work

The I/O subsystems present in most modern-day and historical operating systems were designed with a memory-oriented model of I/O. Data is moved between peripheral devices to main memory where it is operated upon by user processes and is eventually moved out of main memory to some other peripheral device. This approach suffers from a number of performance problems, especially when no intermediate manipulation is required as data is passed between peripherals.

The PPIO system described in this dissertation was originally conceived to address the issues posed by the desire to manipulate continuous media (digital audio or video sample data). Continuous media data lacks locality; it is not often accessed multiple times in a short span of time. These characteristics limit the effectiveness of conventional memory hierarchy designs which depend on locality for efficient operation.

7.1 Conclusions

PPIO improves the performance of applications by *streaming* data between source and sink I/O objects at the "protocol" layer of software or hardware. The "protocol" layer includes not only conventional network protocols, but also any subsystem which adds semantic meaning to raw data (e.g., file systems would also be considered a protocol layer). Streaming improves latency by reducing the number of entities traversed in the pipeline of a data flow. It also improves throughput by limiting costly data copy operations. Carefully implemented, PPIO can offer improved cache performance by not filling cache memory with data lacking locality. Restated, streamed data need not necessarily pass through main memory and may therefore avoid being cached, depending on characteristics of the I/O architecture.

The dissertation has argued the following points, that PPIO:

- is a novel approach to I/O subsystem design
- is intuitive to the programmer
- can be implemented in conventional operating systems
- performs better than existing systems for a large class of important applications

The PPIO design is an I/O system based on data streaming with processing accomplished by the introduction of kernel-maintained processing modules. It builds on previous work in the areas of operating systems and networking. Most previous work aims to improve the performance of I/O data flows using the existing memory-oriented I/O structure; PPIO shortens the I/O data path. Unlike PPIO, the few existing examples of hardware or software streaming have been implemented for specific applications and have not been suggested as a general structuring principal for I/O system software.

The PPIO system interface allows user applications to specify the endpoints and intermediate processing modules of I/O data flows. The interface lacks buffer addresses and sizes, allowing the operating system to optimize these values. With conventional I/O systems, an application programmer wishing to perform a data transfer between I/O objects must generally select a buffer size and address, but has little guidance in selecting optimal values. In addition, the conventional interface requires user processes to execute in order for I/O data to flow between objects. The *rate* at which I/O is performed is generally awkward for the user process to predict (or control). With PPIO, both of these issues as solved by relegating control to the operating system. The user interface supports the establishment of *associations* which interconnect I/O objects, and is analogous to the old-style telephone central office “patch panel”. This abstraction has a long history and is easily understood by most programmers.

The implementation of PPIO requires modest modification to existing operating system software. In PPIO, a regime of *object attributes* are used to characterize most I/O objects (and pseudodevices). Attributes are used to implement flow control between I/O objects specified by user processes. Unlike conventional flow control approaches, PPIO includes generic data source and sink procedures which are sensitive to the attributes of associated objects. Objects may be "slowed down" either by reducing I/O request rates or by adjusting quality parameters when possible. The source and sink procedures use an interrupt-driven structure (for such objects capable of invoking upcalls) to keep data in flight, and rely on a separate control thread only when necessary, thereby avoiding unnecessary context switches and minimizing latency.

Implementation of PPIO may be divided into two parts: handling of processing (execution thread), and handling of data (data movement with respect to main memory). Processing is generally initiated by way of *upcalls* or by way of kernel-supported threads in the case of nontrivial modules (or when queuing is required). Data manipulations in PPIO are accomplished in the native buffering scheme of the host operating system, and thus require no re-implementation of existing buffer managers. Generally, the implementation environment of PPIO requires only asynchronous event notification and thread execution which are common in most operating systems today.

An implementation of PPIO can improve performance for a broad range of applications. It has been shown to improve the CPU utilization, latency, and throughput of several applications, including file copying, displaying multimedia data, and routing network traffic at the transport layer. The architecture is highly adapted for the routing of raw data between objects or devices, and is ideally suited to other common operations such as backup and file system service (fileserver operation). With appropriate processing modules, arbitrary data transformations are possible, although transformations of sufficient complexity are likely to more easily maintained using the conventional I/O model. Filtering and coding of continuous media data (digital filtering, spectral transformations, encryption) appear to be ideally matched to PPIO with appropriate module implementations.

With respect to latency and throughput performance, the PPIO approach generally performs fewer operations than comparable conventional systems. It avoids all user/kernel copying operations, and can often make computational optimizations (such as the incremental checksum described in Section 6.3). Sharing of kernel buffers between associated I/O objects is possible in PPIO when data alignment restrictions are met. PPIO does not invoke user processes to route data, and can therefore eliminate all process manipulation overheads.

Three software prototype implementations of PPIO indicate CPU availability improves by 30% or more and throughput and latency improve by a factor of 2 to 3, depending on the speed of I/O devices. Generally, the latency and throughput performance improvements offered by PPIO improve with faster I/O devices, indicating PPIO scales well with new I/O device technology.

7.2 Critique

The new paradigm suggested by PPIO relies on the concept of *streaming* introduced in Section 2.4. Streaming is amenable to high-performance implementations due to its elimination of software layer overheads and its capability to be driven by asynchronous events. The most serious difficulty of streaming (and thus with PPIO as well) is the introduction of processing in a data path based on streaming.

In PPIO, kernel modules are used to perform arbitrary functions on I/O data and may be executed either in kernel or on attached processors. Ensuring the security of modules requires either an interpreted or compiled approach, as discussed in Section 4.3.2. Although the PPIO design discussed in this dissertation enumerates the techniques available for module security, no "best" solution is currently offered. The selection of module security mechanism(s) requires further investigation into the performance, flexibility, and security offered by such mechanisms as *sandboxing* and other compiled techniques.

In addition to the security issues of module processing, the level of functionality most appropriate for module implementation is not clear. Although a system supporting

PPIO would likely be delivered with a variety of pre-written modules, the task of writing one's own module may be formidable.

Modules must be made to execute in the environment provided by either the kernel or an attached device, and may require system functions such as timers, queues, and schedulable threads. Such abstractions may prove difficult for non-experts to utilize effectively. A programmer requiring substantial processing of I/O data is not provided with clear guidance on whether to utilize processing modules and PPIO versus a traditional user process implementation, although it seems clear a truly I/O intensive application requiring little user interaction would stand to benefit substantially from the PPIO approach.

7.3 Future Work

The development of the PPIO design has raised a number of issues which deserve future exploration. The module architecture was introduced after the initial development of the PPIO approach. Although introduction of code into operating system address space is available in many systems today (as described in Section 3.1.4), the secure introduction of arbitrary user code into a privileged environment remains a research item, and work in this area would complement PPIO.

Most of the experience with PPIO to date is with the interconnection of devices lacking flow or rate control (with the notable exception of the file system experiment, which does provide flow control). The issue of efficient management of flow and rate control remains a fertile area of work, especially in non-realtime-scheduled operating systems. Such research might include an investigation of what quality of service levels could be provided by a PPIO association.

The PPIO experiments described in this dissertation were all based on software-based streaming. Although the performance results were worthwhile (factors of 2 to 3), taking advantage of adapters capable of hardware-streaming could offer far greater improvements. PPIO was conceived with such adapters in mind and is an appropriate system architecture for exploiting them. Unfortunately, few general-purpose systems make use of the hardware streaming technique today (the IBM Microchannel bus supports it, but it is not widely used). Incorporating such adapters into a PPIO system would provide valuable insight to bounds on achievable performance.

Finally, the stability graphs (Figures 6.14 and 6.15) indicate an exponential decay in probability of successful packet delivery using forwarders based on splice, user processes, and conventional gateway forwarding using IP. Although the non-overloaded queuing dynamics of similar systems have been understood for some time, the dynamics of loss under overload (and congestion in general) warrants further investigation. In particular, the effect of loss on common transport layer protocols running over new network technologies (e.g., wireless and cell-based networks which exhibit loss characteristics different from today's Internet) should be investigated.

Bibliography

[ABB+86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young, "Mach: A New Kernel Foundation for UNIX Development," *Proceedings USENIX*, 1986.

[AHIT94] J. Adam, H. Houh, M. Ismert, and D. Tennenhouse, "A Network Architecture for Distributed Multimedia Systems," *Proc. 1st. Intl. Conf on Multimedia Computing and Systems*, pp. 76–85, May 1994.

[ALBL91] Thomas Anderson, Henry Levy, Brian Bershad, and Edward Lazowska, "The Interaction of Architecture and Operating System Design," *Proc. ASPLOS-IV*, pp. 108–120, Apr 1991.

[All90] Arnold Allen. *Probability, Statistics, and Queueing Theory with Computer Science Applications, 2nd ed.* Academic Press, 1250 Sixth Avenue, 1990.

[ATT78] ATT. *AT&T Bell System Technical Journal*, volume 57. Murray Hill, New Jersey, July-Aug 1978.

[BALL90] Brian Bershad, Thomas Anderson, Edward Lazowska, and Henry Levy, "Lightweight Remote Procedure Call," *ACM Trans. on Computer Systems*, 8(1):37–55, Feb 1990.

[BCE+94] Brian Bershad, Craig Chambers, Susan Eggers, Chris Maeda, Dylan McNamee, Pazdyak Pyzemyslaw, Stefan Savage, and Sizer Emin Gon, "SPIN - An Extensible Microkernel for Application-Specific Operating System Services," *Dept. of Computer Science and Engineering FR-35*, Technical Report 94-03-03, University of Washington, Feb 1994.

[Bla83] Andrew Black, "An Asymmetric Stream Communication System," *Proc. 9th SOSP*, pp. 4–10, Oct 1983.

[Bow91] Pat Bowlds. *Micro Channel Architecture*. Van Nostrand Reinhold, New York, NY, 1991.

[BP88] Dave Borman and Craig Partridge, "Computing the Internet Checksum," *RFC 1071, Network Information Center*, Sep 1988.

[BP93] D. Banks and M. Prudence, "A High Performance Network Architecture for a PA-RISC Workstation," *IEEE Journal on Selected Areas in Communications*, 11(2), Feb 1993.

[CDF+93] David Clark, Bruce Davie, David Farber, I. Gopal, et al., "The AURORA Gigabit Testbed," *Computer Networks and ISDN Systems*, 25(6):599–621, Jan 1993.

[Che88] David Cheriton, "The V Distributed System," *Communications of the ACM*, 33(3):314–333, Mar 1988.

[CHKM88] Luis-Felipe Cabrera, Edward Hunter, Michael Karels, and David Mosher, "User-Process Communication Performance in Networks of Computers," *IEEE Trans. on Software Engineering*, 14(1):38–53, Jan 1988.

[CJRS89] David Clark, Van Jacobson, John Romkey, and Howard Salwen, "An Analysis of TCP Processing Overhead," *IEEE Communications*, pp. 23–29, June 1989.

[Cla85] David Clark, "The Structuring of Systems Using Upcalls," *Proc. 10th SOSP*, pp. 171–180, Dec 1985.

[CT90] David Clark and David Tennenhouse, "Architectural Considerations for a New Generation of Protocols," *Proc. SIGCOMM '90*, pp. 200–208, 1990.

[Cus93] Helen Custer. *Inside Windows NT*. Microsoft Press, Redmond, WA, 1993.

[D.93] McAuley D., "Operating System Support for the Desk Area Network," *Fourth Intl. Workshop on Network and Operating System Support for Digital Audio and Video*, Nov 1993.

[DAPP92] Peter Druschel, Mark Abbott, Michael Pagels, and Larry Peterson, "Analysis of I/O Subsystem Design for Multimedia Workstations," *Proc. 3rd. Intl. Workshop on Network and Operating System Support for Digital Audio and Video*, Nov 1992.

[DEC90] Digital Equipment Corporation, Palo Alto, CA. *DECStation 5000/200 KN02 System Module Functional Specification*, rev 1.3 edition, Aug 1990.

[DEC92] Digital Equipment Corporation, Palo Alto, CA. *Information and Configuration Guide for Digital's Desktop Storage: Focus on the RZ Series of SCSI Disk Drives*, Feb 1992.

[DK92] H. M. Deitel and M. S. Kogan. *The Design of OS/2*. Addison-Wesley Publising Co., Reading, Massachusetts, 1992.

[DP93] Peter Druschel and Larry Peterson, "*Fbufs:* A High-Bandwidth Cross-Domain Transfer Facility," *Proc. 14th SOSP*, Dec 1993.

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

[FO74] R. Feiertag and E. Organick, "The MULTICS Input/Output System," *Communications of the ACM*, 17(7):35–41, July 1974.

[FP93] Kevin Fall and Joseph Pasquale, "Exploiting In-Kernel Data Paths to Improve I/O Throughput and CPU Availability," *Proc. USENIX Winter Conference*, pp. 327–333, Jan 1993.

[FP94] Kevin Fall and Joseph Pasquale, "Improving Continuous-Media Playback Performance with In-Kernel Data Paths," *Proc. 1st. Intl. Conf on Multimedia Computing and Systems*, pp. 100–109, May 1994.

[FR86] R. Fitzgerald and Rick Rashid, "The Integration of Virtual Memory Management and Interprocess Communication in Accent," *ACM Transactions on Computer Systems*, 4(2):147–177, May 1986.

[HD91] Mark Hayter and McAuley Derek, "The Desk Area Network," *Operating Systems Review*, 25(4), Oct 1991.

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

[HP91] N. Hutchinson and L. Peterson, "The X-Kernel: An Architecture for Implementing Network Protocols," *IEEE Transactions on Software Engineering*, pp. 64–76, Jan 1991.

[IBM92] IBM, "AIX Version 3.2 Kernel Extensions and Device Support Programming Concepts," 1992.

[Ins88] The Institute of Electrical and Electronics Engineers, Inc., New York, NY. *IEEE Standard Portable Operating System Interface for Computer Environments, Std 1003.1-1988*, 1988.

[ISI81] ISI, "Internet Protocol," *RFC 791, Network Information Center*, Sep 1981.

[JZ93] David Johnson and Willy Zwaenepoel, "The Peregrine High-Performance RPC System," *Software Practice and Experience*, 23(2):201–221, Feb 1993.

[KMQ89] Michael Karels, Kirk McKusick, and John Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley Publishing Company, 1989.

[KP93] Jonathan Kay and Joseph Pasquale, "Measurement, Analysis, and Improvement of UDP/IP Throughput for the DECStation 5000," *Proc. USENIX Winter Conference*, pp. 249–258, Jan 1993.

[LDS92] I. Leslie, McAuley D., and Mullender S., "Pegasus: Operating System Support for Distributed Multimedia Systems," *ACM Operating System Review*, 27(1), Oct 1992.

[Lin94] Christopher Lindblad, "A Programming System for the Dyanamic Manipulation of Temporally Sensitive Data," *PhD Dissertation, Massachusetts Institute of Technology*, 1994.

[LJF83] Sam Leffler, William Joy, and Robert Fabry, "4.2 BSD Networking Implementation Notes, Revised July 1983," *4.2BSD System Manual*, July 1983.

[LJFK86] Sam Leffler, William Joy, Robert Fabry, and Michael Karels, "Networking Implementation Notes, 4.3BSD Edition," *4.3BSD System Manager's Manual*, June 1986.

[MB91] Jeffrey Mogul and Anita Borg, "The Effect of Context Switches on Cache Performance," *Proc. ASPLOS-IV*, pp. 75–84, Apr 1991.

[MD90] Jeff Mogul and Steve Deering, "Path MTU Discovery," *RFC 1191, Network Information Center*, Nov 1990.

[Mil94] David Mills, "A Kernel Model for Precision Timekeeping," *RFC 1589, Network Information Center*, Mar 1994. An updated version of this document from Apr, 1994 is available by anonymous ftp from `louie.udel.edu`.

[MRA87] Jeff Mogul, Rick Rashid, and Michael Accetta, "The Packet Filter: An Efficient Mechanism for User-Level Network Code," *Proc. 11th SOSP*, pp. 39–51, Nov 1987.

[NCR90] NCR Corporation. *SCSI: Understanding the Small Computer System Interface*, 1990.

[NFE92] Bruce Nelson, Raphael Frommer, and Auspex Engineering, "The Myth of MIPS for I/O: An Overview of Functional Multiprocessing for NFS Network Servers," *Auspex Technical Report 1, 6th ed.*, Aug 1992.

[Ope90] Open Software Foundation. *The Design of the OSF/1 Operating System*, 1990.

[Ous90a] John Ousterhout, "Tcl: An Embedded Command Language," *UC Berkeley Computer Science Division (EECS)*, Jan 1990.

[Ous90b] John Ousterhout, "Why Aren't Operating Systems Getting Faster as Fast as Hardware?," *Proc. USENIX Summer Conference*, pp. 247–256, June 1990.

[PAM94] Joseph Pasquale, Eric Anderson, and P. Keith Muller, "Container-Shipping: Operating System Support for I/O Intensive Applications," *IEEE Computer*, 27(3):84–93, Mar 1994.

[Pas92] Joseph Pasquale, "I/O System Design for Intensive Multimedia I/O," *Proc. IEEE Workshop on Workstation Operating Systems*, Apr 1992.

[PCMI91] Michael Pasieka, Paul Crumley, Ann Marks, and Ann Infortuna, "Distributed Multimedia: How Can the Necessary Data Rates be Supported?," *Proc. USENIX Summer Conference*, pp. 169–182, June 1991.

[Pos80] Jon Postel, "User Datagram Protocol," *RFC 768, Network Information Center*, May 1980.

[Pos81] Jon Postel, "Transmission Control Protocol," *RFC 793, Network Information Center*, Sep 1981.

[Pre90] David Presotto, "Multiprocessor Streams for Plan 9," *Proc. Summer UK UNIX User's Group*, pp. 11–19, 1990.

[Rij94] A Rijsinghani, "Computation of the Internet Checksum via Incremental Update," *RFC 1624, Network Information Center*, May 1994.

[Rit84] Dennis Ritchie, "A Stream Input-Output System," *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, Oct 1984.

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

[Sit92] Richard Sites, editor. *Alpha Archictecture Reference Manual*. Digital Press, One Burlington Woods Drive, 1992.

[ST93] Jonathan Smith and Brendan Traw, "Giving Applications Access to Gb/s Networking," *IEEE Network*, 7(4), July 1993.

[Sun90] Sun Microsystems, Mountain View, CA. *SunOS 4.1 Reference Manual, Rev A*, Mar 27 1990.

[Sun92] Sun Microsystems. *SunOS 5.1 Streams Programmer's Guide*, 1992.

[Tho93] Richard Thomsen, "Los Alamos Multiple Crossbar Network: Crossbar Interfaces," This document is available by anonymous ftp: `ftp.lanl.gov:pub/hippi/CBI/cbi.ps`, May 1993.

[Tzo91] D.P. Tzou, S.-Y. Anderson, "The Performance of Message-Passing using Restricted Virtual Memory Remapping," *Software–Practice and Experience*, (21):251–267, Mar 1991.

[USL92] USL, "UNIX System V Release 4, Programmer's Guide: STREAMS," 1992.

[WLAG93] Robert Wahbe, Steve Lucco, Thomas Anderson, and Susan Graham, "Efficient Software–Based Fault Isolation," *Proc. 14th SOSP*, pp. 203–216, Dec 1993.

[Xer88] Xerox Corporation, Sunnyvale, CA. *Pilot Programmer's Manual*, Sep 1988.