Improving Continuous–Media Playback Performance with In–Kernel Data Paths

Kevin Fall
kfall@cs.ucsd.edu

Joseph Pasquale
pasquale@cs.ucsd.edu

Computer Systems Laboratory
Department of Computer Science and Engineering
University of California, San Diego
San Diego, CA 92093-0114

Abstract

Continuous media playback suffers when a station's operating system offers insufficient I/O throughput. Conventional I/O system structures support a memory–oriented `read` *and* `write` *interface requiring the execution of user–level processes to facilitate playback, and can incur throughput degradation due to unnecessary data copies. Our* splice *mechanism supports a peer–to–peer model of I/O where a requesting application can associate a data source with its corresponding data sink, allowing for system optimizations in the data path implementation. In an experiment designed to simulate remote video playback, we present measurements indicating that use of our techniques resulted in a 55% gain in throughput as compared with conventional systems.*

1 Introduction

Computer and networking technology are beginning to offer a digital alternative to current analog cable and broadcast systems. Transmission and display of continuous media information including video is presently achievable with conventional workstations and local area networks (and some wide area networks) under lightly loaded conditions. Under moderate to heavy load, continuous playback of multimedia data is adversely affected.

Figure 1 illustrates an example multimedia transmission and playback environment. Camera–generated images are digitized, compressed, and sent on the delivery network at the transmitting station. Image data frames are transmitted in a compressed form at compression ratios of 200:1 or better, and are delivered to a destination station where they are decompressed and displayed in a window as part of a graphical user interface environment. Figure 1 assumes image decompression occurs in the user process for simplicity.

Although the transmitting station need only manipulate compressed data, the receiving station must manipulate decompressed data for display on a conventional computer frame buffer. Thus, the receiver of video data experiences a tighter demand on its operating system to provide high throughput.

This paper describes a general I/O model and mechanism which can be integrated into most operating systems to address the primary OS problems of multimedia playback: process switching overhead and insufficient throughput. The mechanism explicitly addresses well–known overheads associated with data copying, and also reduces the number of context–switch operations required to perform I/O operations. When combined with a scheme capable of improving network protocol processing performance (as is suggested in [KP93] for example), a greater than doubling above baseline performance may be achieved. The mechanism represents an augmentation of conventional operating system I/O interfaces, and is not a replacement for the `read` and `write` system functions. The functionality and implementation of `read` and `write` remain unaffected.

This paper is organized as follows. Section 2 describes our architectural I/O model. Section 3 describes the *splice* mechanism, which allows an input stream and an output stream to be directly and efficiently connected. Section 4 provides a detailed example how splice may be used in a video playback application using a common windowing environment (e.g. the X11 window system). In Section 5, we de-

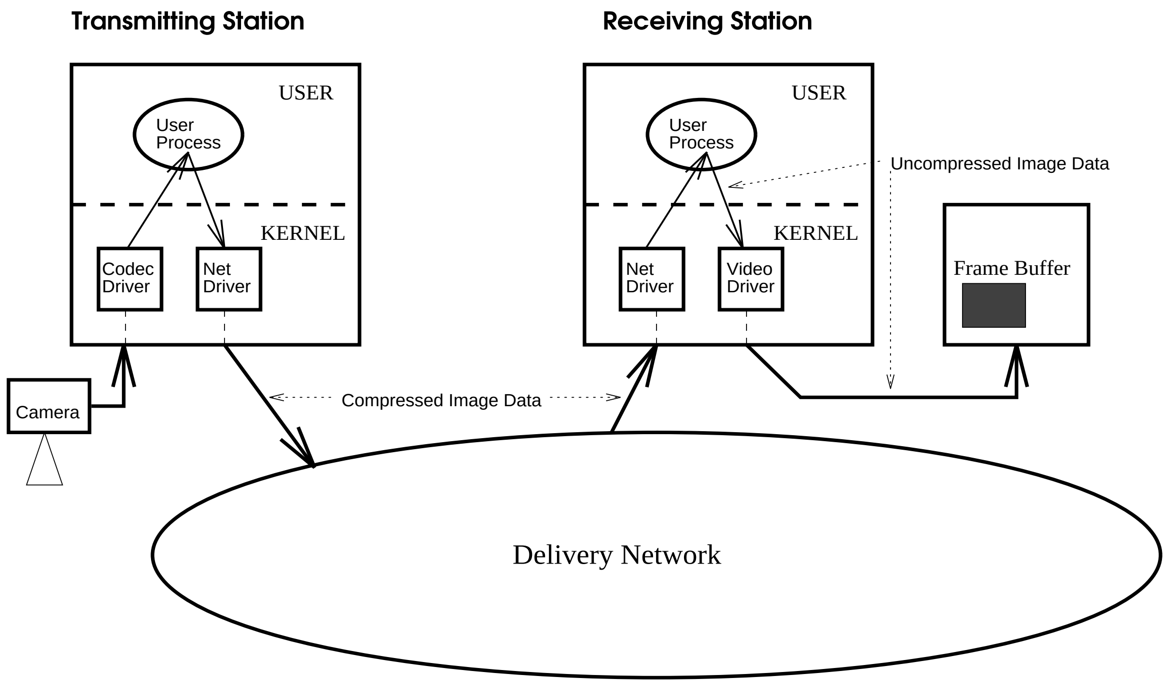

Figure 1: Multimedia Distribution Environment

scribe the implementation of splice in a conventional UNIX system equipped with Berkeley sockets. Section 6 contains a performance evaluation of splice for a simulated video display environment. In Section 8, we present conclusions.

2 Conventional Device Access

In Figure 1, user processes are responsible for the initiating the capture, compression, and network transfer of video data. On the receiving station, a peer process receives incoming compressed data, arranges for the data to be decompressed (by hardware or software, but performed by the receiving user process in this example) and delivers the uncompressed image to an ordinary framebuffer through the local operating system. I/O data is passed through the address space of processes on both the sender and receiver. In many situations, such processes perform little or no direct manipulation of I/O data and act only as forwarding agents. Many situations which do require processing require only common data transformations to be performed (e.g. format conversion, compression, etc.).

Figure 2 illustrates the *memory–oriented I/O* (MIO) model 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. Both control and data flow between devices through user processes. 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. Unfortunately, it also suffers from performance problems commonly associated with user–level process implementations: data copying across the user/kernel boundary, and overhead associated with scheduling and context switches. Data copying limits ultimate achievable throughput, and scheduling/context switching can adversely affect continuity of playback for multimedia applications (skips and loss of synchronization). A kernel–based *peer–to–peer* I/O (PPIO) model of data movement can address both performance issues without requiring an entirely new I/O programming interface, process control architecture, or inter–domain data transfer facility. Such a mechanism should reduce data copies and minimize process switching, yet allow applications to control data transformations and meter the rate of data transfers. The next section describes a mechanism that addresses these concerns.

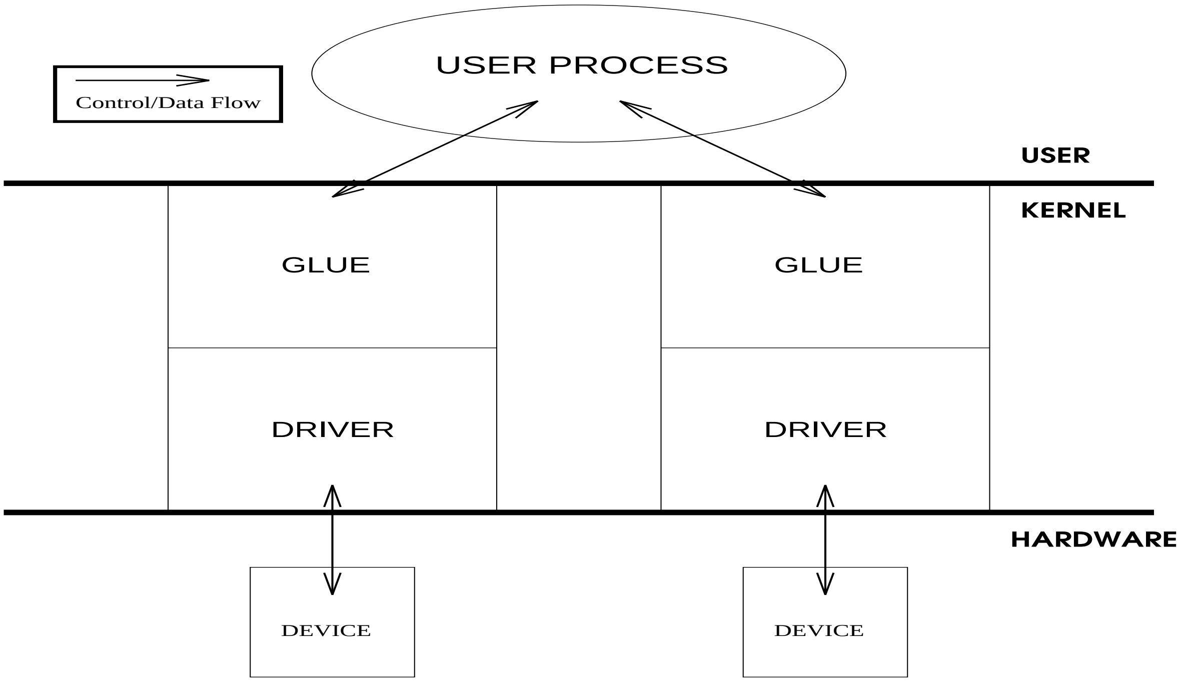

Figure 2: Conventional Memory–Oriented I/O Model (MIO)

3 The Splice Mechanism

The *splice* mechanism [FP93] is a system function used to establish a kernel–managed data path directly between I/O devices. In an MIO system exporting the traditional `read` and `write` interfaces, I/O data flowing between devices must pass through the address space of a user process. Figure 2 shows how data and control information flow in an MIO system. Data originating at one device is brought into operating system memory through the device driver, is manipulated in various layers of system "glue" code, until it is delivered to a buffer specified by the user–process. An analagous set of events occurs to deliver data from the user–process to the destination device.

With *splice*, an application may express an association between a data source and sink directly to the operating system using descriptors which do not refer to memory addresses (i.e. they are not buffers). The advantages of splice are improved throughput due to copy elimination and reduced context switching load. In a multimedia playback application, these improvements manifest themselves as an increased frame rate and improved continuity of animations (i.e. smoother video).

The programming interface to splice is a system call with the following calling convention:

```
splice(fd1, fd2, size)
```

The call establishes an *in–kernel data path* between a data source and sink specified by the references `fd1` and `fd2`, respectively. The splice mechanism eliminates copy operations to user space by removing the buffer interfaces present in the I/O functions `read` and `write`. By eliminating the user–level buffering, kernel buffer sharing is possible. More specifically, when block alignment is not required by an I/O device, a kernel–level buffer used for data input may be used subsequently for data output. [1].

In addition to removing the buffering interfaces, splice also combines the read/write functionality together in one call. The splice call indicates to the operating system the source and sink of a data flow, providing sufficient information for the system to manage the data transfer by itself without requiring user–process execution. Thus, context switch operations for data transfer are eliminated. Context switches consume CPU resources, degrade cache performance by reducing locality of reference [MB91], and effect the performance of virtual memory by requiring TLB invalidations [BALL90]. Additional motivations for addressing the issue of context switching performance are mentioned in [PCMI91], [CJRS89], and [CHKM88].

[1] A similar technique is used in the 4.3BSD Reno NFS Implementation [Mac91].

For applications making no direct manipulation of I/O data (or for those allowing the operating system to make such manipulations), splice relegates the issues of managing the data flow (e.g. buffering and flow control) to the operating system. For example, data movement may be accomplished within the operating system by a thread of control created due to a completion event (e.g. device interrupt), or in a synchronous fashion achieved by the use of kernel–level threads. Flow control may be achieved by selective scheduling of kernel threads or by only posting reads to data-producing devices when data-consuming peers complete I/O operations. A kernel–level implementation provides great flexibility in choosing which control abstraction is most appropriate.

Supporting unrestricted data transformations in a kernel–implemented data flow presents challenges. There are four techniques presently available for introducing code dynamically into the kernel's address space:

1. Loadable Kernel Modules/Drivers
2. Streams Modules
3. Software–Isolated Fault Domains
4. Interpreters

Loadable kernel modules (LKMs) are now supported by many operating systems. LKMs are relocatable object files loaded into kernel address space by priviledged processes. Code from LKMs are then executed by the kernel. Faulty modules may cause failure of the entire system. *Streams modules*, as described first in [Rit84], offer a set of predefined operations implemented within the operating system that a user process may choose to be executed on its behalf. Standard data transformations (e.g. compression or other commonly–used coding algorithms) would be appropriate candidates for a streams–style implementation. *Software–Based Fault Isolation* [WLAG93] prevents untrusted code modules from accessing other regions of memory by restricting the set of machine–level instructions a module may execute. The scheme provides enhanced fault tolerance in exchange for some execution overhead required to examine a module's address references. In such a system, a user would compile a module with a compiler specially modified to produce "safe" code. The system can verify the "safeness" of code and incorporate the module into its address space. *Interpreted languages* may also be used, as reported in [MRA87] and [MJ93]. These systems place a small interpreter in the operating system. Code is written in the specialized language.

Applications requiring completely generalized direct modification of intermediate data should use the traditional `read` and `write` interfaces, manipulating data as needed. Thus, splice augments the traditional I/O interface by providing enhanced performance for those applications able to utilize direct data streaming or one of the code loading techniques listed above. Note that an application may alternate between the use of splice and read/write to accomplish a single task, depending on the nature of the data flow. When intermediate processing is required, read/write are used; when no intermediate processing is required, splice is used. Applications which use only the `read` and `write` I/O interface remain completely unaffected.

Eliminating a memory-based interface provides not only the advantages listed above, but also the possibility of connecting data source and sink devices together directly at the system's I/O bus and managing the transfer by the kernel in a general way. Some current architectures (e.g. IBM's Microchannel [Bow91]) support this notion as "peer-to-peer DMA" or "stream-mode transfers." These architectures provide I/O data transfer between interface adapters directly on the system bus. During such a transfer, I/O data is not brought into the system's main memory unit. For a MIO system, taking advantage of stream–mode transfers would be difficult due to the need to make I/O data visible to user processes. With the peer–to–peer model offered by splice, use of such a direct transfer mechanism is natural.

4 Extended Example

In this section we describe how splice may be used to optimize receiver performance in a video playback application. Figure 3 illustrates the environment on a multimedia receiving station equipped with hardware decompression.

The receiving application can optimize its data flow by connecting together the network protocol stack with a hardware decompression unit. To set up the receiver, the multimedia receiver process establishes handles to a network connection as well as to a hardware decompression device using a conventional device open function. It then establishes an *in–kernel data path* between the I/O devices by invoking the splice mechanism. The process may terminate a splice at will, or may also indicate that a splice is to operate for only a specified number of bytes. Data received at the network interface is processed by the system's protocol implementation in the conventional

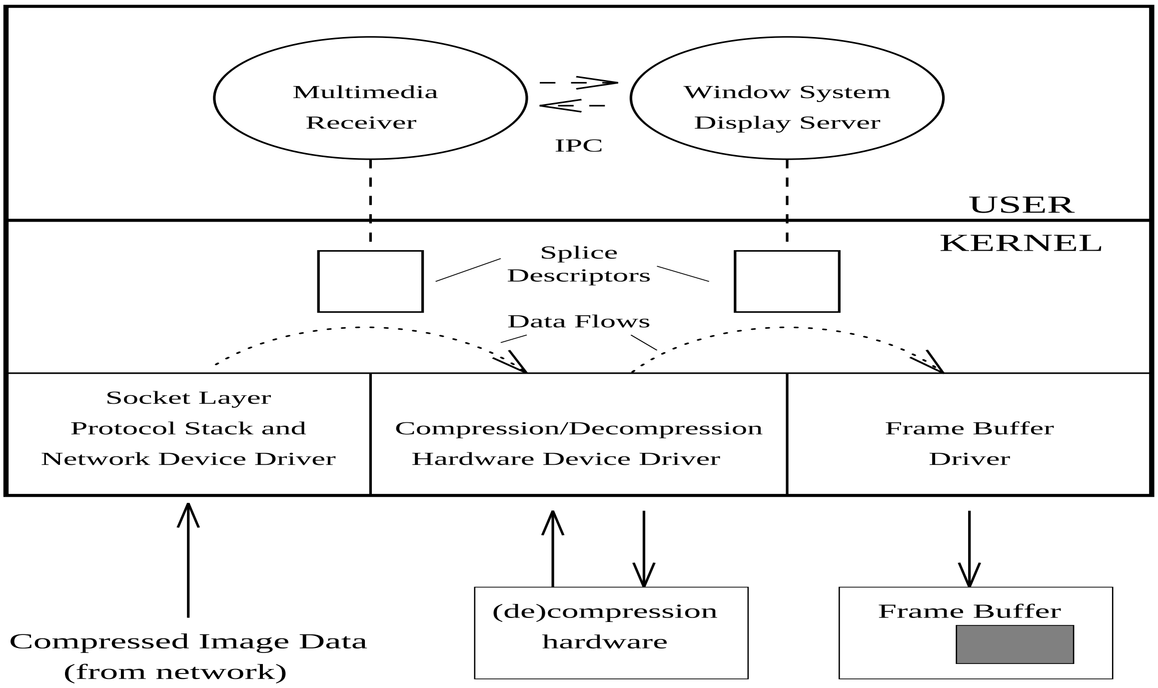

Figure 3: Example Use of Splice in Video Display Application

way, but instead of being delivered to the user process' address space, the compressed images are instead delivered to the hardware decompressor. In effect, an upcall is turned into a downcall by splice. Assuming control information is handled separately, the multimedia receiver need not make direct manipulation of the image data. In addition, the multimedia receiving process can acquire a handle to the output of the decompressor, and pass this handle to the window display server using interprocess communications mechanisms (IPC). Flow control, if implemented in this environment, is accomplished by providing flow control updates (increased window size advertisements for window–based flow control protocols like TCP) to the network source at a rate proportional to the processing rate of the decompression hardware.

The window display server responds to requests for image display by evaluating which windows obscure others and directing data to appropriate screen areas and clipping when necessary. When displaying an unobscured video window, the display server can optimize its data flow by establishing an in–kernel data path between the output of the decompressor and the frame buffer device driver. In such cases, the display server need not execute for image data to be displayed on the framebuffer screen. Because display servers tend to be large processes, avoidance of this particular context switch is especially appealing.

Should the server receive an event indicating an unobscured video window is to become obscured, it may abort the splice and perform individual `read` and `write` calls. If the window becomes unobscored once again, another splice can be created. Although display servers often improve performance by writing directly to the frame buffer using shared–memory, they must still be scheduled and must be allowed to execute a sufficient amount of time to draw a complete video screen on the frame buffer. Under heavy load, the server may fail to have sufficient time to perform this operation (i.e. its throughput is insufficient), ultimately resulting in discontinuous video playback and frame loss. Splice need not suffer from this form of degradation, as splice processing may be implemented without the possibility of pre-emption, as a result of a control thread created during interrupt processing.

5 Prototype Splice Implementation in UNIX

The splice programming interface was introduced in Section 3. We now turn to the prototype UNIX implementation. A data source and sink are specified by UNIX file descriptors. In the present prototype, these file descriptors may refer to regular files, sockets or device–special files. This section describes a network–

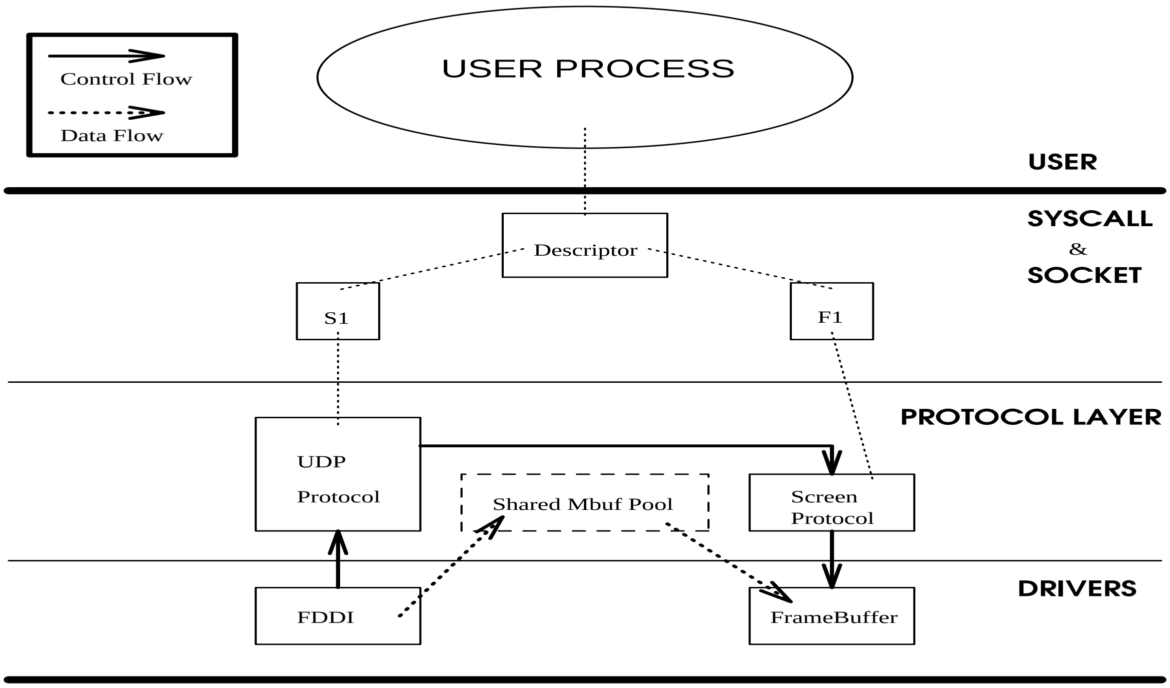

Figure 4: Network-to-Framebuffer Splice Implementation at the Receiver

to–framebuffer (NtoF) prototype implementation of splice; the implementation for regular files has been described in [FP93]. The NtoF splice is representative of the task required at a multimedia receiving station.

Figure 4 illustrates the NtoF splice implementation described below and used in the subsequent performance evaluation section. A Berkeley–based network structure [LJFK86] was used to construct the prototype.

The splice call provides suffient 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 *splice descriptor* maintains references to socket or file structures (depending on the purpose of the splice; see [FP93]), plus any additional information required by the implementation. [2] 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 splice descriptor.

Implementing a prototype NtoF splice within the Berkeley network code framework is surprisingly straightforward. The Berkeley network framework essentially uses an *upcall* [3] 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. [4] 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[Pos81], packet sizes are limited to 64KBytes which is adequate to hold most compressed frames (the largest JPEG–compressed images we have encountered is about 20KBytes, with typical values of 7–8KBytes).

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

[3] 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.

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

6 Performance Results

We now turn to the performance of the UNIX implementation described in Section 5. We constructed an experiment designed to simulate the video display environment of Figure 1. 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.

6.1 Experimental Setup

We compared the throughput performance of a receiving continuous media program using the splice mechanism to move data between the network and display device as compared against an equivalent user–level process implementation. We also compared the effect of protocol checksum computation on overall throughput.

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. When splice is being used, instead of delivering data to the socket layer, the data is instead moved directly to the destination (e.g. the frame buffer), in the context of a network

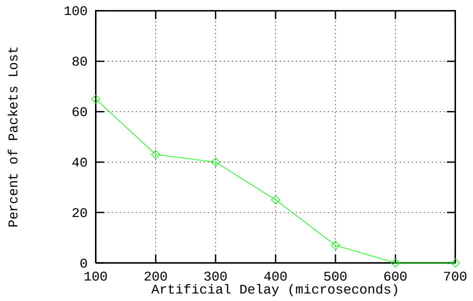

Figure 5: UDP Packet Loss vs. Artificial Delay

software interrupt handler. The data appears in the frame buffer as a 120x100 video window.

6.2 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 our 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 (not flow controlled in any way), approximately 2/3 of the data is lost at the receiver. To avoid buffer overrun at the receiver during the user–level test, we constructed a rate–based flow control mechanism for UDP by introducing artificial delay between network packets at the sender. Figure 5 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 catas-

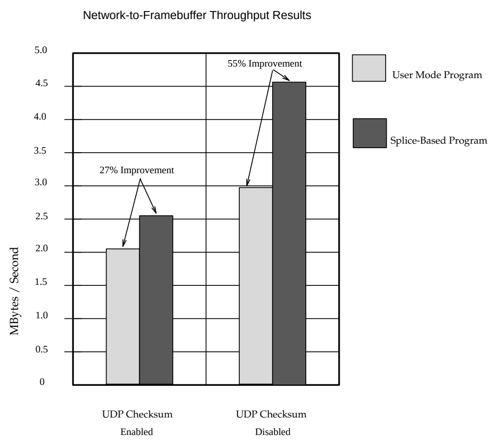

Figure 6: Throughput Performance of Splice and Non–Splice Network–to–Framebuffer Access

trophic result. In addition to removing checksumming for loss–tolerant data such as uncompressed audio or video, checksums may be disabled in certain restricted operating environments (e.g. when hardware checksumming is performed [KP93]). To evaluate the relationship between splice and checksumming, we performed the experiment with checksums enabled and again with checksums disabled.

6.3 Measured Results

We now describe measured results based on the use of splice and the effect of checksum computation. Figure 6 contains the differences in measured throughput in MBytes per second. The graph shows differences between user mode-based data transfer (not using splice) versus splice-based data transfer, both with the checksum computation enabled and disabled.

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.

Because of the hardware used in the experiment, we were limited to data transfers using system memory as an intermediate buffering area between the network and display adapter. We expect a greater performance improvement for device–to–device transfers capable of taking advantage of hardware offering direct peer–to–peer data transfer (e.g. "peer–to–peer DMA", "hardware data streaming"). The DECstation Turbochannel system used for the experiment does not support such transfers. The splice interface removes details of any intermediate buffering from the caller. The presence or absence of intermediate buffering is detectable by the caller only in terms of performance.

7 Related Work

A new I/O system for Mach 3.0 described by Forin et. al. [FGB91] suggests the problem of data copying is best addressed by the use of user–level device drivers capable of accessing device registers directly by mapping when possible. Remote device access is achieved by an IPC–based device interface relying on RPC.

In the CTMS system described by Pasieka et. al. [PCMI91], the problem of excess data copies is addressed by a modification to the UNIX I/O model of inter–device transfers. They introduce new *ioctl* calls capable of passing data between specific devices without passing directly through a user–level process.

A study of I/O design by Druschel et. al. [DAPP92] suggests the use of low–level data streaming inhibits innovation in application development by disallowing applications direct access to I/O data. While their criticism of low–level streaming is valid, the technique should not be precluded in systems because of the need to bring I/O data into the address space of a user–level process for some applications. Many applications not requiring direct manipulation of I/O data can benefit from streaming. Furthermore, for applications requiring well–known data manipulations, kernel–resident processing modules (e.g. like Ritchie's Streams [Rit84]) or outboard dedicated processors are more easily exploited within the kernel operating environment than in user process.

There are at least three systems which address the problem of minimizing physical data movement across domains: DASH IPC[TA91], Fbufs[DP93], and Container Shipping[PAM94]. Each of these systems uses virtual memory remapping techniques to transfer data between protection domains without physical data copies. The DASH IPC system allows processes to exclusively own regions of a restricted area of a shared address space. Consequently, an interprocess data transfer occurs by simply changing the ownership of a region from one process to another. The Fbufs design uses a similar technique, enhanced by caching the previous owners of a buffer, allowing reuse among trusted processes and eliminating MMU updates associated with changing buffer ownership. In the Container Shipping design, transfer and access are decoupled: an inter-domain transfer does not imply accessibility in the receiving domain. A process explicitly requests that portions of transferred data be made accessible, thus only incurring remapping costs on an as-needed basis. Although these systems all achieve high throughput and low latency, they must still schedule an intermediate process to forward data from a source (device or process) to a sink (device or process). This causes an inherent limitation in performance, which is characteristically absent from the in–kernel data path approach.

8 Conclusions

Although current workstation and networking technology are sufficiently advanced to support processing and dissemination of multimedia data, the memory–oriented model of I/O present in most operating systems can present a bottleneck which adversely affects overall performance. Especially significant is the reduction in throughput experienced at stations receiving and displaying multimedia data. In conventional I/O systems employing the traditional memory–oriented `read` and `write` interfaces, user–level processes must actively move data between network and other local devices to facilitate remote device access.

Operating system overhead can be reduced by adopting a *peer–to–peer* model of device I/O as offered by the *splice* mechanism. By decoupling user–process execution from inter–device data flow, significant reductions in data copying and process execution overhead are achieved.

These performance improvements are achievable for the common situation of data transfers terminating on a station providing video and audio playback with a conventional bus architecture lacking direct device–to–device data transfer capability. Although the conventional memory–based I/O model would have difficulty taking full advantage of a sytem offering this capability (because of the need to provide data access to a user–processes' address space) we would expect a more significant throughput improvement for such machines.

References

[BALL90] Brian Bershad, Thomas Anderson, Edward Lazowska, and Henry Levy. Lightweight remote procedure call. *ACM Trans. on Computer Systems*, 8(1):37–55, February 1990.

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

[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, January 1988.

[CJRS89] D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of TCP processing overhead. *IEEE Communications*, pages 23–29, June 1989.

[Cla85] David Clark. The structuring of systems using upcalls. *Proc. 10th SOSP*, pages 171–180, December 1985.

[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*, November 1992.

[DP93] Peter Druschel and Larry Peterson. Fbufs: A high–bandwidth cross–domain transfer facility. *Proc. 14th SOSP*, pages 189–202, December 1993.

[FGB91] Alessandro Forin, David Golub, and Brian Bershad. An I/O system for mach 3.0. *Proc. USENIX Mach Symposium*, pages 163–176, November 1991.

[FP93] Kevin Fall and Joseph Pasquale. Exploiting in-kernel data paths to improve I/O throughput and CPU availability. *Proc. USENIX Winter Conference*, pages 327–333, January 1993.

[KP93] Jonathan Kay and Joseph Pasquale. Measurement, analysis, and improvement of UDP/IP throughput for the DECStation 5000. *Proc. USENIX Winter Conference*, pages 249–258, January 1993.

[LJFK86] Sam Leffler, William Joy, Robert Fabry, and Michael Karels. Networking implementation notes, 4.3bsd edition. *4.3BSD System Manager's Manual*, June 1986.

[Mac91] Rick Macklem. Lessons learned tuning the 4.3bsd reno implementation of the nfs protocol. *Proc. USENIX Winter Conference*, pages 53–64, 1991.

[MB91] Jeffrey Mogul and Anita Borg. The effect of context switches on cache performance. *Proc. ASPLOS-IV*, pages 75–84, April 1991.

[MJ93] Steve Mccanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. *Proc. USENIX Winter Conference*, Jan 1993.

[MRA87] J. Mogul, R. Rashid, and M. Accetta. The packet filter: An efficient mechanism for user-level network code. *Proc. 11th SOSP*, pages 39–51, Nov 1987.

[PAM94] Joseph Pasquale, Eric Anderson, and P.Keith Muller. Container shipping: operating system support for i/o intensive applications. *IEEE Computer Magazine*, 27(3):84–93, March 1994.

[PCMI91] Michael Pasieka, Paul Crumley, Ann Marks, and Ann Infortuna. Distributed multimedia: How can the necessary data rates be supported? *Proc. USENIX Summer Conference*, pages 169–182, June 1991.

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

[Pos81] Jon Postel. Internet Protocol. *RFC 791, Network Information Center*, September 1981.

[Rit84] Dennis Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984.

[TA91] Shin-Yuan Tzou and David Anderson. The performance of message-passing using restricted virtual memory remapping. *Software - Practice and Experience*, 21(3):251–267, March 1991.

[WLAG93] Robert Wahbe, Steve Lucco, Thomas Anderson, and Susan Graham. Efficient software–based fault isolation. *Proc. 14 SOSP*, pages 203–216, December 1993.