Exploiting In-Kernel Data Paths to Improve I/O Throughput and CPU Availability

Kevin Fall (kfall@ucsd.edu)
Joseph Pasquale (pasquale@ucsd.edu)

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

Abstract

We present the motivation, design, implementation, and performance evaluation of a UNIX kernel mechanism capable of establishing fast in-kernel data pathways between I/O objects. A new system call, *splice()* moves data asynchronously and without user-process intervention to and from I/O objects specified by file descriptors. Performance measurements indicate improved I/O throughput and increased CPU availability attributable to reduced context switch and data copying overhead.

Nov, 1992

1. Introduction

Improved computer hardware has enabled the development of complex applications with enormous I/O demands. Providing adequate performance for such applications poses a significant challenge to the operating systems community, especially with the anticipated 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 in the I/O system structure may limit achievable performance by introducing unnecessary overheads.

I/O Intensive applications are those applications moving a large amount of data (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 (or to avoid memory altogether whenever possible), and separating I/O control from I/O data transfer [Pas92].

With UNIX being the primary operating system available for most scientific and high performance computing platforms today, evaluation and improvement of UNIX system performance when exposed to I/O intensive workloads is important to ensure a viable future execution environment. This study outlines UNIX I/O system modifications aimed at improving the performance of such applications.

2. Design Goals

Our goal in modifying the I/O system is to demonstrate improved data throughput and increased CPU availability with asynchronous operation between devices or files without adversely affecting the standard UNIX I/O architecture and interface. An attractive strategy for achieving these goals is to decouple process execution from I/O data flow by introducing a new system call based on the following design principles:

- Avoid unnecessary data copying
- Provide asynchronous operation
- Support concurrent I/O operations

A new system call `splice()` achieves these goals with two unique features. First, the call has no buffer interface as do the UNIX `read()` and `write()` system calls because data is not moved to and from user space. Although we could have opted for a shared-memory interface as several others have suggested for efficient I/O to user space, we wished to avoid the memory interface entirely. A memory-based interface implies a need to bring the required data into the system's main memory in quanta and format dictated by the virtual memory hardware of the local machine. With splice, data could reside entirely beyond the reach of a process, perhaps outside the machine's main memory.

Second, the call operates asynchronously in a fashion similar to the asynchronous I/O calls present in several current versions of UNIX, and Windows NT []. A calling process may continue user-mode execution while I/O is proceeding between objects. The process may regain control of the splice execution periodically be carefully adjusting the transfer size parameter (described below), but does not need to create threads or call specialized ‘‘async I/O’’ functions as several systems require.

An old-style telephone operator ‘‘patching together’’ two communicating parties is an appropriate analogy for the operation of splice. Splice may also be thought of as providing the ‘‘reverse’’ capability of the original Streams IPC pseudoterminal (PT) [Rit84] or the streams-based pipe implementation in 8th Edition UNIX [PrR85]. The PT in Ritchie’s streams and pipes in 8th Edition provide IPC by cross-connecting file descriptors within the kernel. Splice, in contrast, provides the cross-connection of devices within the kernel as specified by the calling process.

3. Interface

Splice takes two UNIX file descriptors and an integer size as arguments. The file descriptors specify the source and sink of I/O data, respectively. They may refer to files, character device special files, or sockets. The size parameter specifies the number of bytes to be moved between the source and sink files; a special value indicates the splice should execute until an end of file condition is reached or the operation is interrupted by the caller. The splice operates asynchronously if either of the file descriptors have the FASYNC flag enabled, as set by a call to `fcntl()`. A calling program can opt to catch SIGIO to detect the completion of an asynchronous splice.

4. Example

The following example illustrates the use of splice in an application which plays back a digitized movie from a file:

```
      ...

int audiofile, videofile;     /* digital audio/video files */
int audio_dev, video_dev;     /* output dacs */

audiofile = open("movie.audio", O_RDONLY);
videofile = open("movie.video", O_RDONLY);

audio_dev = open("/dev/speaker", O_WRONLY);
video_dev = open("/dev/video_dac", O_WRONLY);

fcntl(audiofile, F_SETFL, FASYNC); /* set async operation */

/* copy the audio information; return immediately */
splice(audiofile, audio_dev, SPLICE_EOF);

/* loop, delivering one frame every timer interval */
setitimer(ITIMER_REAL, &inter_frame_time);
do {
    rval = splice(videofile, video_dev, sizeof(video_frame));
    pause();      /* wait for timer to go off */
            /* it will reload automatically */
} while (rval > 0);
```

```
        . . .
```

The above code segment illustrates the use of splice to transfer a complete file (e.g. an audio file) as well as a partial file (e.g. frames of a video file). For the audio example, the splice moves digital audio samples asynchronously from the audio data file to the output DAC (D-to-A converter). The program assumes the audio DAC driver converts and delivers audio at the appropriate playback rate to match the recording rate in the file. Several audio device interfaces (e.g. Sun's `/dev/audio`) operate in this fashion. For video, the program assumes a video device capable of displaying frames at a maximum rate faster than the recording rate of the source file. That is, a full-speed splice between the source file and the display device would play video too quickly to match the corresponding audio. Slowing the splice transfer rate is achieved by ensuring the FASYNC property is not set, and adjusting the size parameter to specify a limited transfer quantum (e.g. the size of a single frame for video). The calling process retains control of the transfer rate by making splice requests at appropriate intervals. A video ‘‘fast forward’’ or ‘‘slow motion’’ could be affected by adjusting the interval timer value. Moreover, splice requires no buffer handling by the user program, and provides support for multiple simultaneous I/O operations.

5. Implementation

Splice is currently implemented as a system call under Ultrix 4.2A, and has been tested on a DecStation 5000/200 and DecStation 5000/240. The code comprises about 3000 lines of C source code (including comments), and increases the kernel's object size by about 10% (to 1.9MB).

5.1. Background

The current implementation of splice supports file-to-file splices between files residing on local disk storage devices, socket-to-socket splices for the UDP transport protocol, and framebuffer-to-socket splices for sending graphical images and video. For brevity, this discussion outlines only the portions of the implementation relevant to the 4.2BSD-based filesystem. The splice implementation requires 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 paper assumes basic familiarity with these functions. They are discussed in more detail in [LMK89]. The filesystem design is discussed in [LMK89].

5.2. 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 (Ultrix terminology for generic filesystem node). A special *splice descriptor* 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 calling process context to be available. The entire list of all physical block numbers comprising the source file is determined by successive calls to `bmap().` The list of physical blocks is stored in a dynamically allocated table in the splice descriptor. 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. At this point, all information necessary to proceed with an asynchronous data transfer has been stored in the splice descriptor, and user-mode execution of the calling process may be resumed.

5.2.1. 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, provide most of the needed functionality. The physical block number is retrieved by indexing into the table in the splice descriptor 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 preceeding 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.

5.2.2. Write-Side Operation

The write side of splice is called 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. Avoiding lock-step behavior by introducing the asynchrony provided by the callout list 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 splice descriptor and logical block number a buffer's data is associated with. 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 splice descriptor 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, but rather only sets the `b_bcount` field in the new buffer header to the requested size. The data 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. We thus avoid copying between cache buffers. 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.

5.2.3. Flow Control

Flow control for splice cannot be achieved by causing the calling program to block; in any case, the caller is not directly responsible for initiating read or write requests, 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 splice descriptor 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 five additional reads. These values are adequate to prevent both the source from being underutilized and the destination from being overwhelmed.

6. Experiments

We performed several experiments to measure the effectiveness of splice. The goal of these experiments is to demonstrate improvement in CPU availability and I/O system throughput due to reduced copying and switching overheads when using splice rather than user-level read/write sytem calls to transfer data between files.

6.1. Configuration

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

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

Because we wanted to test splice with very fast devices, we also implemented a RAM disk. 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. The ram disk driver uses 16MB of statically allocated memory from the kernel's BSS region. The system we measured was running with 5MB of free core.

6.2. CPU Availability Test

We accomplish the goals of measuring CPU availability and throughput 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 take to complete. To measure changes in CPU availability, we compare the amount of time required for the test program to complete the same number of operations in the CP and SCP environments. To measure device-to-device file I/O throughput, we ensured a read cache cold start condition by performing large file I/Os through the buffer cache before taking measurements. We ensured write-through behavior for the cache in the case of writes 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.

CPU Availability Factors (Copying 8 MB File)				
Disk Type	F_{cp} Slowdown	F_{scp} Slowdown	I Improvement	SCP Percentage Improvement
RAM	2.0	1.2	1.7	70%
RZ58	1.6	1.2	1.3	30%
RZ56	1.6	1.3	1.2	20%

Table 1

Table 1 shows the relative performance effect of CP and SCP for copying an 8MB file on disks with different performance characteristics. Alternative sizes for the file were statistically indistinguishible from the 8MB representative case listed above. Column one lists the type of disk being used, including the two SCSI disks described above and the RAM disk driver. Column two shows the slowdown experienced by our test program in the CP environment. Column three indicates the slowdown experienced in the SCP environment. Column four indicates the improvement factor experienced by SCP over CP as a function of disk type. Finally, column five indicates the percentage CPU availability improvement achieved by SCP over CP.

When contending with *cp* for the RZ56 or RZ58 trial, the test program executes at 60% of the IDLE rate. For the RAM trial, it executes at 50% of the IDLE rate. When contending with *scp*, however, the test program achieves 80% of the IDLE rate for the RAM and RZ58 cases, and 70% of the IDLE rate for the RZ56 case. Thus, processes will experience a 20 to 70 percent execution speed improvement when contending with splice-based copying versus read/write-based copying, depending on the device speeds.

6.3. Throughput Tests

The next table indicates the throughput characteristics of SCP and CP:

Mean Throughput Measurements (Copying 8 MB File)			
Disk Type	SCP Throughput (KB/s)	CP Throughput (KB/s)	%-Improvement with SCP
RAM	3,343	1,884	77%
RZ58	587	545	8%
RZ56	368	371	-1%

Table 2

Table 2 achievable throughput using *cp* vs. *scp* for copying files. For the throughput tests, we disabled the test program used to produce Table 1, so the figures in Table 2 represent maximum attainable throughput measures assuming an otherwise idle CPU. Column one again 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 over the CP rate attainable with SCP. Thus, splice-based copying can operate at 1.8 times the maximum throughput of read/write-based copying in the best case, which is a significant improvement in I/O performance. For real disks, the disk transfer time dominates the overall throughput measurement and the benefit of splice is minor.

6.4. 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 small modifications for non-blocking behavior, we have not made fundamental modifications to the buffering, scheduling, or block allocation strategies present in most UNIX systems. We plan to investigate these areas, as well as the performance of our SCSI device driver, with the expectation of higher performance.

7. Related Work

The work described here relates to the general problems of system overhead encountered with large throughput I/O loading. Dean and Armand [] explore the effect of microkernel-based operating system design on data movement performance, suggesting the desirability of including device manipulation code directly in user processes to avoid copying. Similar suggestions are made by Forin et. al in [FGB91], who suggest the mapping of device registers directly to user-level processes. Govindan and Anderson [] describe ‘‘memory-mapped streams’’ as a mechanism for moving continuous media data between address spaces using shared memory. These approaches require the interface of I/O devices to appear as memory objects, and must therefore be mappable to a process’ address space. Several architectures restrict the ability to map devices, especially to user address space. Furthermore, we believe the data transfer size granularity should be specified by the application, rather than being constrained by details of the VM hardware.

In an approach similar to ours, Pasieka et. al [PCM91] suggest the UNIX *ioctl* be used to pass handles between source and destination devices, referring to kernel-level data objects. Their scheme decouples data movement from the application but requires user process execution to effect a data transfer between devices.

8. Conclusions

This study suggests a viable and promising augmentation to the standard UNIX system interface for I/O intensive applications. The experimental results indicate a reduction in process-related overhead which contributes to improved performance, both in terms of throughput and CPU availability during I/O periods. The programmer interface is convenient for the class of applications wishing to move data unaltered from one device or file to another. We believe the class of I/O intensive applications to be a large one, including multimedia programs wishing to connect audio and video ‘‘streams’’ between devices and files.

References

[DEC90] Digital Equipment Corporation, ‘‘DECStation 5000/200 KN02 System Module Functional Specification, Rev 1.3’’, *Workstation Systems Engineering*, Palo Alto, CA, Aug, 1990.

[FGB91] A. Forin, D. Golub and B. Bershad, ‘‘An I/O System for Mach 3.0’’, *Proc. Usenix Mach Symposium*, Monterey, CA, Nov, 1991, 163-176.

[LMK89] S. J. Leffler, M. K. McKusick, M. J. Karels and J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley Publishing Company, 1989.

[PCM91] M. Pasieka, P. Crumley, A. Marks and A. Infortuna, ‘‘Distributed Multimedia: How Can the Necessary Data Rates be Supported?’’, *Proc. Usenix Summer Conference*, Nashville, TN, June, 1991, 169-182.

[Pas92] J. Pasquale, ‘‘I/O System Design for Intensive Multimedia I/O’’, *Proc. IEEE Workshop on Workstation Operating Systems*, Key Biscayne, FL, April 1992.

[PrR85] D. Presotto and D. Ritchie, ‘‘Interprocess Communication in the 8th Edition Unix System’’, *Proc. Usenix Winter Conference*, Dec 1985, 309-316.

[Rit84] D. M. Ritchie, ‘‘A Stream Input-Output System’’, *ATT Bell Laboratories Technical Journal 63*, 8 (October 1984), 1897-1910.

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