

General Data Streaming *

Frank W. Miller
Pete Keleher
Dept. of Computer Science
Univ. of Maryland, College Park
{fwmiller,keleher}@cs.umd.edu

Satish K. Tripathi
College of Engineering
Univ. of California, Riverside
tripathi@engr.ucr.edu

Abstract

This work presents a new I/O system design and implementation targeted at applications that perform data streaming. The approach yields true zero-copy transfers between I/O devices in many instances. We give a general characterization of I/O elements and provide a framework that allows analysis of the potential for zero-copy transfers. Finally, we describe the design, implementation, and performance of a prototype I/O system in a real-time, embeddable, 32-bit operating system whose design is based on the presented analysis to minimize data copying.

1 Introduction

A variety of computer applications display a behavior termed *data streaming* where data is repetitively moved from one input/output (I/O) device to another. This work increases performance using mechanisms that couple I/O devices together tightly within the operating system kernel. In current systems, data copies decouple the transfer characteristics of I/O devices. However, data copies have a detrimental effect on performance. The design and implementation of streaming presented here eliminates data copies by passing data between I/O devices by reference.

The proposed design is quite general, applying to I/O systems for a wide range of operating system types, from embedded systems to personal (PC) class and workstation class computers. The initial implementation has been performed in an embeddable, 32-bit, real-time operating system. While predictability is the primary goal for the design of embedded, real-time operating systems, such systems are also generally resource constrained. The elimination of data copies frees both CPU and memory resources that can be used for other

purposes. As such, the performance gains achieved using the techniques presented here should be of great value to real-time operating system designers.

This work makes several contributions. 1) A small set of general parameters that may be used to characterize any I/O channels to be used as stream endpoints is given. 2) A general analysis of data streaming based on an orthogonal matching of these parameters is provided. 3) A set of algorithms designed to minimize data copies for each of the analytical cases is presented. 4) A large set of performance results taken from the prototype operating system that enumerate a representative set of I/O channel pairings and uniformly show throughput improvements are presented.

The rest of the paper is organized as follows. Section 2 provides motivation and context in related work. Section 3 presents a set of parameters for characterizing I/O channels and a general analysis of data streaming. Section 4 presents a set of algorithms which provide optimized streaming based on the analysis in Section 3. Section 5 provides a performance evaluation of a representative sample of an orthogonal matching of I/O channels. Finally, Section 6 discusses some observations and current work.

2 Related Work

The major motivation for the design and implementation presented here is an explosion in the number of different types of peripheral I/O devices that can be attached to modern day computer systems and the desire to support their data transfers as efficiently as possible. In embedded real-time systems, there is an almost infinite array of I/O devices in use as sensor/effector equipment. With respect to PCs and workstations, it was not so long ago that such a system consisted of a CPU, memory, disk, keyboard, display, and perhaps an Ethernet adapter. More recently, cameras, scanners, CD-ROMs and DVDs, microphones, sound

*Partial funding provided by NSF Grant CCR 9318933, IBM, and ASSERT grant from DOD.

cards with their associated amplifiers and/or speakers, all types of pointing devices, joysticks, projectors, modems, ISDN, all types of network adapters ({regular, fast, really fast} Ethernet, FDDI, ATM, Frame Relay, cable modems, etc.), and more are available and in widespread use.

Data streaming can occur at a variety of levels. Architectures such as I₂O [8] allow for *peer-to-peer streaming* across a backplane bus. The major advantage is performance but it comes at a price. System complexity and cost are significantly higher since code to manage the organization of data being moved across the backplane must be pushed into the devices and coordinated with the main system. For this reason, much of the work in data streaming has focused on *main memory streaming* where data passes through main memory on its way between peripheral devices.

In main memory streaming, the goal is to copy the data while it is in memory as few times as possible, with zero copy the ultimate goal. Much of the recent work in streaming has focused on the problem of *cross address space* copies. Brustolini and Steenkiste's *Genie* and Pai's work with *IO-lite* [14] which is based on Druschel's *fbufs* [4] are recent examples of the use of page remapping to move data residing in main memory across protection boundaries while avoiding data copies in streaming applications.

Reducing cross address space copies is important, however, in embedded systems, a significant number of applications do not make use of virtual memory addressing. Further, the cross address space works do not address the resulting situation where data transfers between I/O devices are now tightly coupled. The analysis and mechanism presented in this work complements the solutions proposed to address the cross address space problem, i.e. it simply assumes that data is brought into memory from one device and sent out another. It works regardless of how many times the data is moved across address space boundaries.

The primary influence on the I/O system design has been the streaming work performed at UCSD [5, 6]. Fall's proof-of-concept when coupling I/O from similar channels together within a BSD-variant operating system kernel provided performance improvements that are echoed here. However, his approach has been expanded in this work to encompass orthogonal pairing between any I/O elements in the system. The central issue arising from this extension is the need to deal with the dislike characteristics of tightly coupled I/O channels.

case 1	$T^{in} = \text{fix} \wedge T^{out} = \text{fix} \wedge B^{in} = B^{out}$
	$T^{in} = \text{fix} \wedge T^{out} = \text{var} \wedge B^{in} = B^{out}$
	$T^{in} = \text{fix} \wedge T^{out} = \text{var} \wedge B^{in} < B^{out}$
	$T^{in} = \text{var} \wedge T^{out} = \text{var} \wedge B^{in} = B^{out}$
	$T^{in} = \text{var} \wedge T^{out} = \text{var} \wedge B^{in} < B^{out}$
case 2	$T^{in} = \text{fix} \wedge T^{out} = \text{var} \wedge B^{in} > B^{out}$
	$T^{in} = \text{var} \wedge T^{out} = \text{var} \wedge B^{in} > B^{out}$
case 3	$T^{in} = \text{fix} \wedge T^{out} = \text{fix} \wedge B^{in} > B^{out}$
case 4	$T^{in} = \text{fix} \wedge T^{out} = \text{fix} \wedge B^{in} < B^{out}$
	$T^{in} = \text{var} \wedge T^{out} = \text{fix} \wedge B^{in} = B^{out}$
	$T^{in} = \text{var} \wedge T^{out} = \text{fix} \wedge B^{in} < B^{out}$
	$T^{in} = \text{var} \wedge T^{out} = \text{fix} \wedge B^{in} > B^{out}$

Table 1. Static Parameter Combinations

3 Streaming Analysis

Define a device as any hardware that performs data movement, typically, between a peripheral and main memory. A device driver is defined as the lowest level software that controls a device. A device and either an *input channel* or an *output channel* specify a stream endpoint, either a *source* or a *sink*.

Each device has a value, B , associated with it that corresponds to the *maximum* contiguous block of memory required for a data transfer. Define b to be the amount of data moved during some transfer. If the transfer size associated with a device must be constant and equal to its block size, i.e. $b = B$ for all transfers, the device is said to require *fixed block transfers*. Hard disks and CD-ROM drives are examples of fixed block devices. If the block size can vary up to the maximum, i.e. $b \leq B$, the device is said to allow *variable block transfers*. Network adapters and video cards are examples of variable block devices.

We characterize each I/O channel with two parameters, the *block type*, T , which can take on the values **fix** (fixed) or **var** (variable) and the *maximum block size*, B . Both parameters are static, i.e. they do not vary from transfer to transfer.

Figure 1 is a list of all possible combinations that result when the parameters for source and sink channels are coupled. These twelve cases reduce to four that provide algorithmic opportunities to optimize throughput for streaming.

We define a *stream* as a half-duplex channel between a source and sink. The source provides a sequence of M transfers where each transfer has a size b^{in} . The stream delivers to the sink a sequence of N transfers each with size b^{out} . Consider the following analysis of generalized streaming using the two static parameters to maximize throughput.

Case 1 ($T^{in} = \text{fix} \wedge T^{out} = \text{fix} \wedge B^{in} = B^{out}$) \vee
 $(T^{out} = \text{var} \wedge B^{in} \leq B^{out})$

In this first case, either both the source and sink have the same fixed block type or the output block type is variable and the input block size is smaller than the output block size. Input blocks can be output directly. We have $N = M$ and no data copies are required since the input buffer can always be output directly. An illustration of this streaming case is given in case 1 of Figure 1.

Case 2 $T^{out} = \mathbf{var} \wedge B^{in} > B^{out}$

In this case, the output block type is variable and the input block size is larger than the output block size. The stream divides the input block into $\lfloor \frac{B^{in}}{B^{out}} \rfloor$ output blocks of size B^{out} and a single output block holding the remainder, $b^{out} = B^{in} - \lfloor \frac{B^{in}}{B^{out}} \rfloor B^{out}$. Note that the solid lines indicate that *reference buffers* are used to generate multiple output blocks for each input block. An illustration of this streaming case is given in case 2 of Figure 1.

Case 3 $T^{in} = \mathbf{fix} \wedge T^{out} = \mathbf{fix} \wedge B^{in} > B^{out}$

In this case, both the input and output block types are fixed and the input block size is greater than the output block size. The stream divides input blocks into smaller blocks for output. The number of output blocks required to empty an input block is $\frac{B^{in}}{B^{out}}$ which may be non-integral. Since output blocks must be filled, they may contain fractions of input blocks. For every $\frac{L}{B^{out}}$ output blocks, $\frac{L}{B^{in}} - 1$ copies of input data to an output block must be performed. The segments of data framed by dashed lines must be copied from an input buffer to an output buffer. An illustration of this streaming case is given in case 3 of Figure 1.

Case 4 $(T^{in} = \mathbf{fix} \wedge T^{out} = \mathbf{fix} \wedge B^{in} < B^{out}) \vee (T^{in} = \mathbf{var} \wedge T^{out} = \mathbf{fix})$

In the final case, either both the input and output block types are fixed and the input block size is smaller than the output block type or the input block type is variable and the output block type is fixed. In both situations, blocks must be aggregated to fill the larger output blocks. The number of input blocks that fill an output block is given by $\frac{B^{out}}{B^{in}}$. If this result is non-integral, input blocks must be split.

Define $L = lcm(B^{in}, B^{out})$ to the least common multiple of B^{in} and B^{out} . Observe that for every $\frac{L}{B^{in}}$ input blocks, $\frac{L}{B^{out}}$ blocks will be output. Furthermore, for every $\frac{L}{B^{out}}$ output blocks,

one copy of data from an input block to an output block can be avoided. An illustration of this streaming case is given in case 4 of Figure 1.

4 Streaming Algorithms

A set of algorithms whose goal is to provide the efficient streaming presented in the previous section is now given. These algorithms make assumptions about the design of the I/O system. First, access to all I/O elements is performed through a common applications programming interface (API). Second, a common buffer representation is used by all system I/O elements.

The *Roadrunner* I/O architecture utilizes a common interface to all I/O elements. This is accomplished using a global file system (or **fs**) layer which is supported by specific file system implementations. This architecture is based on work from several sources. The use of the file system name space to address various elements of the system is reminiscent of Plan 9/Inferno [15]. The internal implementation is based primarily on the **vnodes** [10, 16, 17] architecture present in 4.4BSD [11] (among others) and the stackable architectures proposed by Heidemann and Popek [7].

The most significant design element is the use of a common buffer representation by all I/O elements. Figure 2 illustrates the design of the common buffer representation and Table ?? lists the routines used by the kernel to manage them. Buffers have two parts, a descriptor and an optional data block. The fields of the descriptor are partitioned into those used for referencing data in a data block and those used for caching. The **data** field points at the beginning of an allocated data block. The **size** field contains the length of the allocated block. The **start** and **len** fields mark where useful data is contained in the allocated block. The **pos** field is used to provide an offset into the start of the useful data. The **devno** (or device number), and **blkno** (or block number), fields are used for caching. These two fields are combined in a hash function that allows quick insertion and retrieval of buffers into and out of hash table managed buffer pools. I/O elements use *move semantics* when managing buffers, i.e. a buffer only resides in the global cache when it is *not* owned by some file descriptor.

The algorithm presented here conceptually “overlays” a stream of input buffers onto a stream of output buffers. The overall structure of this algorithm is given in Figure 3.

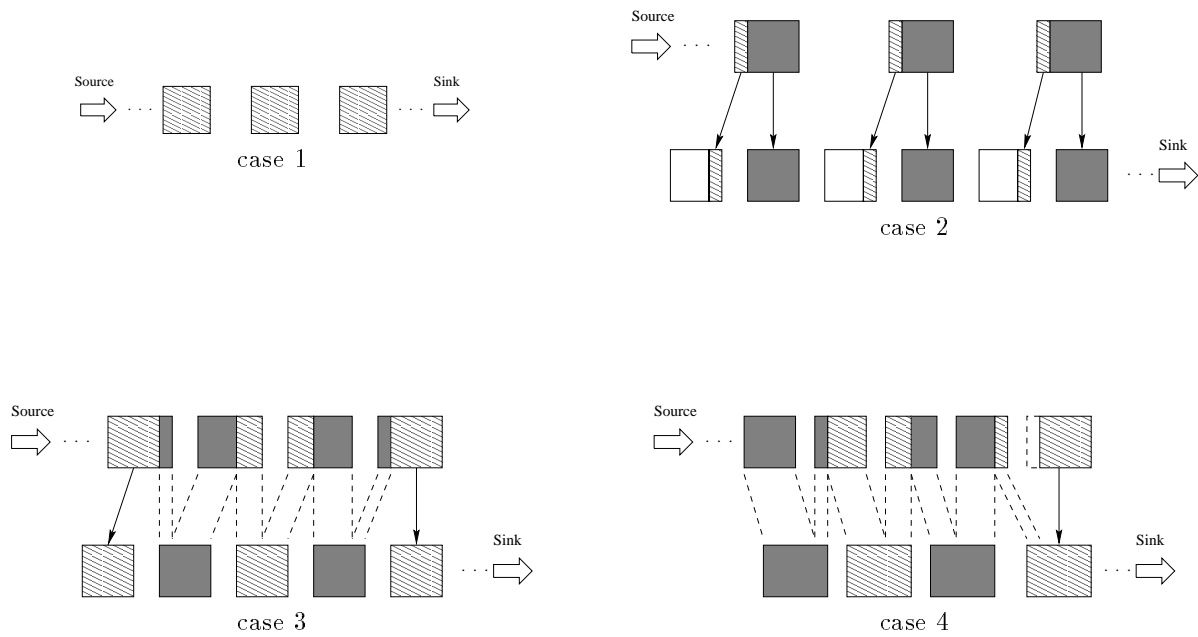


Figure 1. The set of streaming cases derived from the static parameter combinations.

```

int stream(int in, int out)
{
    int infilesz;
    int inblktype, outblktype;
    int inbufsize, outbufsize;

    Get channel paramters

    if ((inblktype == FIX && outblktype == FIX && inbufsize == outbufsize) ||
        (outblktype == VAR && inbufsize <= outbufsize)) {
        Case 1 stream
    } else if (outblktype == VAR && inbufsize > outbufsize) {
        Case 2 stream
    } else if (inblktype == FIX && outblktype == FIX && inbufsize > outbufsize) {
        Case 3 stream
    } else if ((inblktype == FIX && outblktype == FIX && inbufsize < outbufsize) ||
        (inblktype == VAR && outblktype == FIX)) {
        Case 4 stream
    }
}

```

Figure 3. The overall structure of the overlay algorithm

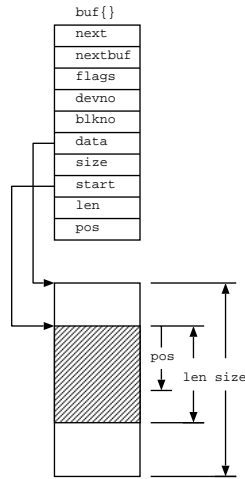


Figure 2. Common buffer representation.

Routine	Description
bread()	Retrieve cached buffer
bput()	Enter buffer in cache
balloc()	Blocking get buffer descriptor <i>only</i>
_balloc()	Non-blocking get buffer descriptor <i>only</i>
bget()	Blocking get buffer with data block
_bget()	Non-blocking get buffer with data block
bfree()	Release buffer to the free list
brrel()	Enter buffer in cache or release to free list

Table 2. Common Buffer Management Routines

4.1 Case 1

This simplest cases occur when either the input and output block types are fixed and the input and output buffer sizes are equal or the output block type is variable, and the input buffer size is less or equal to the output buffer size. In these instances, input buffers can be output directly using the simple algorithm in Figure 4. This algorithm implements exactly the behavior specified in case 1 of Figure 1. The system call, `read(in, b)`, reads data from the input channel, `in`, into the buffer, `b`. The system call, `write(out, b)`, writes data to the output channel, `out`, using the buffer, `b`.¹ Since input buffers are output directly, no data copies are required.

4.2 Case 2

This case occurs when the output block type is variable and the input block size is greater than the output block size. In these instances, the algorithm given in Figure 5 can be executed. It's operation matches exactly the analysis given in case 2 of Figure 1.

The algorithm repetitively fetches a buffer, `bin`, from the input channel and progresses through its data sending it to the output channel. The buffer, `bout` is a *reference buffer*, i.e. its descriptor is used to progress through areas of data in `bin`. The maximum block size for `bout` is used until the end of the data has been reached in `bin`. In this last output transfer, the amount of data sent will be `inbufsize mod outbufsize`. Since all output buffers are passed by reference, no data copies are required.

4.3 Case 3

This case occurs when the input and output block types are both fixed and the input buffer size is greater than the output buffer size. The algorithm given in Figure 6 is based on the analysis in case 3 of Figure 1 but does not match its operation exactly.

The key observation made when developing this algorithm is that it is very likely that the input buffer size is a multiple of the output buffer size. If this is so, it is desirable to have the algorithm behave similarly to case 2, i.e. passing partitions of the input blocks by reference. Only when the input block size is not a multiple of the output block size should copying be performed. Fortunately, the case where a multiple is present is common. An example is when a file is copied

¹The specifications for `read()` and `write()` have been condensed for clarity in this presentation.

```

int n;
for (n = 0; n < infilesize;) {
    b = bget(inbufsize);
    read(in, b);
    write(out, b);
    n += inbufsize;
}

```

Figure 4. The case 1 stream algorithm

```

int n;
for (n = 0; n < infilesize;) {
    if (bin == NULL) {
        bin = bget(inbufsize);
        read(in, bin);
    }
    bout = balloc();
    bout->start = bin->start + bin->pos;
    if (bin->len - bin->pos >= outbufsize) {
        bout->len = outbufsize;
        n += outbufsize;
        bin->pos += outbufsize;
    } else {
        bout->len = bin->len - bin->pos;
        n += bin->len - bin->pos;
        brel(bin);
        bin = NULL;
    }
    write(out, bout);
}

```

Figure 5. The case 2 stream algorithm

between disk file systems that utilize a different block size.

The algorithm again repetitively reads blocks from input channel. In this algorithm case, however, a separate inner **for** loop is used to progress through the data in each input block. There are four cases that occur as it progresses through data in each input block. They are defined by the guards on the inner and outer **if** statements.

The first two cases occur when no output block is currently being filled. In the first case, the amount of data in the input block is greater than or equal to the size of an output block. A reference buffer is defined and sent out the output channel. The second case occurs when the amount of data left in the input block is less than the size of an output buffer. The remainder of the input block is copied into a new output buffer and the algorithm proceeds to the next input block.

The second two cases occur when there is an as yet unfilled output block that was carried over. This can happen as a result of the second or fourth cases. In the third case, there is enough data in the input block to fill the remainder of the output block. The data is copied and the filled output block is sent to the output channel. The fourth case occurs when the data remaining in the input block will not fill the carried over output buffer. This action taken is similar to case 2.

If the input buffer size is a multiple of the output buffer size, the first case is the only one that is ever executed and no data copies are required. If however, the input buffer size is not a multiple of the output buffer size, the situation in case 2 of Figure 1 occurs, i.e. a copy is avoided only when the least common multiple of the two buffer sizes occurs.

4.4 Case 4

This case occurs when the input and output block types are both fixed and the input buffer size is less than the output buffer size or the input block type is variable and the output block type is fixed. The algorithm given in Figure 7 is based on the analysis in case 4 of Figure 1 but does not match its operation exactly.

In this case, one copy must always be performed. For the first part of this case, it is necessary to aggregate smaller input blocks into larger output blocks. In the second part, since it is impractical to expect that the the data in variable input blocks will have repetitive sizes, all the variable input data must be aggregated into the fixed output buffers.

This case also proceeds by looping over input buffers. During each iteration, however, it is possible that an input or output buffer may have been carried

```

int n;
for (n = 0; n < infilesize;) {
    bin = bget(inbufsize);
    read(in, bin);
    while (bin->pos < bin->len)
        if (bout == NULL) {
            if (bin->len - bin->pos >= outbufsize) {
                bout = balloc();
                bout->start = bin->start + bin->pos;
                bout->len = outbufsize;
                write(out, bout);
                n += outbufsize;
                bin->pos += outbufsize;
            } else {
                bout = bget(outbufsize);
                bcopy(bout->start,
                    bin->start + bin->pos,
                    bin->len - bin->pos);
                bout->len = outbufsize;
                bout->pos = bin->len - bin->pos;
                n += bin->len - bin->pos;
                break;
            }
        } else {
            if (bin->len - bin->pos >= bout->len - bout->pos) {
                bcopy(bout->start + bout->pos,
                    bin->start + bin->pos,
                    bout->len - bout->pos);
                n += bout->len - bout->pos;
                bin->pos += bout->len - bout->pos;
                write(out, bout);
            } else {
                bcopy(bout->start + bout->pos,
                    bin->start + bin->pos,
                    bin->len - bin->pos);
                bout->pos += bin->len - bin->pos;
                n += bin->len - bin->pos;
                break;
            }
        }
    }
    brel(bin);
}

```

Figure 6. The case 3 stream algorithm

over. If not, the required buffer is allocated. If there is enough input data to fill the output buffer, the data is copied and the buffer is sent to the output channel. The input buffer carries over. Otherwise, the output data is copied and the output buffer is carried over.

5 Performance

Performance measurements, especially apples to apples comparisons are difficult in this work for a number of reasons. Probably most important is that we are not aware of any other systems with an I/O subsystem designed to do generalized coupling of I/O elements within the kernel in the manner being described here. The approach that has been taken is to provide two sets of measurements for a subset of an orthogonal coupling of a representative set of I/O endpoints. The first set *simulates* the environment found in most current systems, i.e. it uses `read()/write()` system calls to perform streaming transfers. The second set uses the appropriate streaming algorithm.

All of the performance measurements presented were made on the same machine, an IBM PC/350, that includes a 75 MHz Pentium with 256 Kbyte second level cache, 16 Mbytes of memory, an 800 Mbyte EIDE hard

disk, and an SMC EtherPower 10BaseT PCI Ethernet adapter. While this machine is dated compared to those currently available, it is important to note that the absolute figures in these measurements are not as important as the *relative* results, those using the streaming interface versus those using the traditional `read()/write()` interface.

A list of the file systems and devices used as stream endpoints is given in Figure 3. The `/dev/null` device is accessed through the `devfs` file system. It provides a variable block type with a maximum block size of 4096 bytes. The UDP and TCP file systems provides endpoints utilizing the DEC tulip based PCI Ethernet adapter. The UDP endpoint uses a variable block type with a maximum of 1446 bytes The TCP endpoint uses a variable block type with a maximum of 4096 bytes. Two DOS compatible file systems are used that reside on the same hard disk in different partitions. Both have a fixed block type with one having a block size of 2048 bytes and the other a block size of 4096 bytes.

Table 4 shows how these endpoints can be combined for streaming. There are two types of cells in this matrix. Those cells that contain only a single number give the streaming case exercised when the corresponding endpoints are combined. For example, when the

```

int n;
for (n = 0; n < infilesize;) {
    if (bin == NULL) {
        bin = bget(inbufsize);
        read(in, bin);
    }
    if (bout == NULL) {
        bout = bget(outbufsize);
        bout->len = outbufsize;
    }
    if (bin->len - bin->pos >= bout->len - bout->pos) {
        bcopy(bout->start + bout->pos,
              bin->start + bin->pos,
              bout->len - bout->pos);
        n += bout->len - bout->pos;
        bin->pos += bout->len - bout->pos;
        write(out, bout);
    } else {
        bcopy(bout->start + bout->pos,
              bin->start + bin->pos,
              bin->len - bin->pos);
        n += bin->len - bin->pos;
        bout->pos += bin->len - bin->pos;
        brel(bin);
        bin = NULL;
    }
}

```

Figure 7. The case 4 stream algorithm

Endpoint	File System/Device	Blk Type	Max. Size
1	devfs/null	var	4096
2	udpfs/tulip	var	1446
3	tcpfs/tulip	var	4096
4	dosfs/wd disk	fix	2048
5	dosfs/wd disk	fix	4096

Table 3. Stream endpoints used in measurements

DOS file system with block size 2048 bytes is used a source and the UDP endpoint is used a sink, the case 2 streaming algorithm applies. Those cells that contain two numbers separated by a slash give the streaming case and the figure number of measurements presented for that endpoint combination. For example, when the DOS file system with block size 2048 bytes is used as a source and the TCP endpoint is used as a sink, the case 1 streaming algorithm applies and measurements taken for this case are presented in Figure 13.

Figure 8 shows the baseline performance of the IDE disk in the test machine. Because disk I/O transfers take so long compared to memory movements, speedups are heavily dependent on the use of caching. The improvements are good when the cache is in play, i.e. the file, or portions of it, reside in the cache. The improvements are modest when disk I/O dominates the transfers. Consider the points for a 1 MB cache when a 512 KB file is read. The streaming transfers are accomplished at a rate of 30.5 Mbps while the `read()/write()` transfers yield a throughput of 22.6 Mbps. Streaming yields a 35% improvement in this case. However, if the same cache size is used

		Destination				
		1	2	3	4	5
Source	1	1	2	1	4/9	4
	2	1	4	1	4	4
	3	1	4	4	4	4
	4	1/8	2	1/13	1/10	4/12
	5	1	2	1	3/11	1

Table 4. Endpoint combinations for measurements. Cells containing a single number give the streaming case used for the endpoint combination. Cells containing two numbers separated by a slash give the streaming case and the figure number for measurements taken for the endpoint combination.

to transfer a 1 MB file, the best throughput for a streaming transfer is 1.9 Mbps versus 1.7 Mbps for the `read()/write()` transfers. Streaming yields only a 6% improvement here.

Figure 9 illustrates the effect of the overall cache size for a given data block size during file writes. Since delayed writes are occurring, disk operations do not come into play until the file size exceeds the cache size. When the file fits into the cache, speedups are in the neighborhood of 79%. When the file size exceeds the cache size, `bget()` calls must flush buffers, i.e. write the dirty contents to disk, before filling them with new data. This results in the significantly reduced speedup of approximately 5%.

The remaining sets of measurements provide results for applications that make use of the streaming algorithms. Figures 10, 11, and 12 give speedups for execution of the file copy program, `cp`, and Figure 13

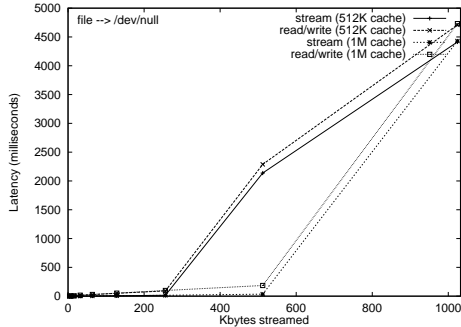


Figure 8. File read stream

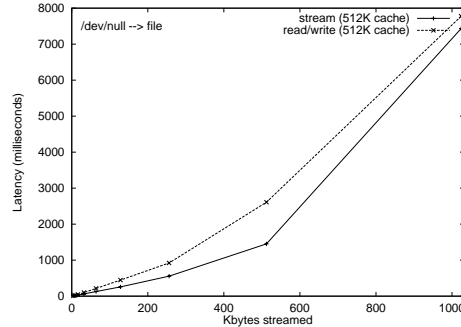


Figure 9. File write stream

gives the speedups associated with the *Roadrunner* web server, `httpd`.

In Figure 10, a copy from one place to another on the same file system is presented. Improvements here range from 33% for the case of a 1 MB file and 512 KB cache to 75% for a 512 KB file and 1 MB cache to 77% for a 1 MB file and 1 MB cache. In Figure 11, a file is copied from a file system with a buffer size of 4096 bytes to a file system with a buffer size of 2048 bytes. The performance improvements for the input buffer greater than the output buffer are the best of any of the algorithm cases. Consider the data points for the 256 KB cache size. The `read()/write()` performance is 1.25 Mbps for a 256 KB file, 608 Kbps for a 512 KB file, and 535 Kbps for a 1 MB file. The stream performance is 1.4 Mbps for a 256 KB file, 1.29 Mbps for a 512 KB file, and 1.2 Mbps for a 1 MB file. These throughputs yield 12.5%, 87%, and 124% speedups, respectively. In Figure 12, a file is copied from a file system with a buffer size of 2048 bytes to a file system with a buffer size of 4096 bytes. The speedups here, while uniform, are modest. This is to be expected since this stream uses case 4, where copies are always required.

There is an interesting dynamic here with the cached blocks in Figures 10 and 11. When `read()/write()` is used, a block is brought into the cache for the input channel and another block is entered in the cache for the output channel. When streaming is used, a single block is entered in the cache. The block is originally brought in for the input channel but is then passed by reference to the output channel. When this happens, the input channel reference goes away. This halves the cache utilization for the stream and explains why the performance of the 1 MB file and 1 MB cache hasn't yet dropped off as in the case of the 1 MB file and 512 KB cache. The tradeoff is that future references to the same input block will require another disk reference.

The speedups in Figure 13 vary depending on the retrieved file size. A best case speedup of 505% has been

observed when a file that is less than or equal to the TCP send buffer size resides in the global cache. More modest improvements of 26% are observed for large files. It should be noted that these results are particularly impressive given that in the web server, a modified XINU [2] implementation is being used. This implementation contains few of the optimizations present in BSD derived stacks, such as no-copy send and receive buffers, optimized checksums and header prediction.

6 Conclusion

We have presented a general framework that characterizes the need for copying in systems that perform data streaming. We have demonstrated analytically that true zero-copy data transfers can be used in many instances and described the performance of a prototype implementation in a real-time, embeddable, 32-bit operating system. We have shown that zero-copy transfers uniformly improve performance in our environment. It is important to note that these improvements are hardware independent. The I/O system design and streaming algorithms are characteristics of our operating system architecture and should provide speedups on any hardware platform on which they are used.

Our approach does have potential drawbacks. First, the exclusive use of *move* semantics eliminates the possibility of concurrent access to a single buffer, which reduces the number of expensive I/O operations performed. Work is currently under way to extend the common buffer design to allow concurrent access to data blocks. Second, performance improvements are greater when I/O transfer times are close to data copy times, and this is not generally the case for disk I/O.

Finally, we have implemented an entirely new I/O system from scratch in order to prototype this design approach. The I/O system was made small, fast, and inherently extensible by using a stackable `vnode`-based

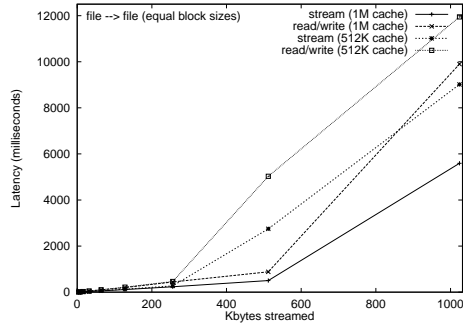


Figure 10. File copy stream, $B^{in} = B^{out}$

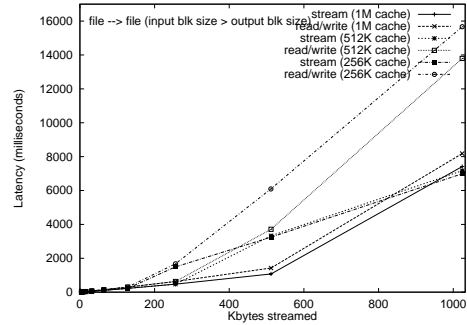


Figure 11. File copy stream, $B^{in} > B^{out}$

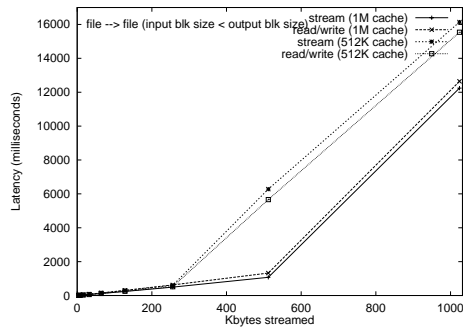


Figure 12. File copy stream, $B^{in} < B^{out}$

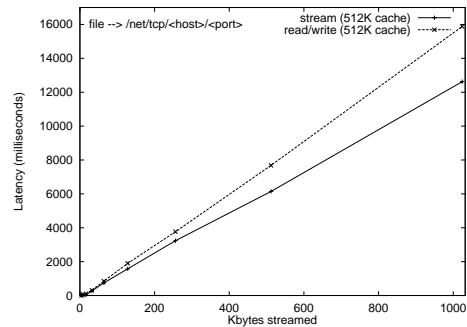


Figure 13. Web server stream

design. While the *Roadrunner* system is targeted at embedded system development, we do have a set of tools that allow self-hosted development. The authors are prepared to make the complete sources for the *Roadrunner* system available for further research.

References

- [1] Brustolini, J. and Steenkiste, P., "Effects of Buffering Semantics on I/O Performance", *Proc. of 2nd Symposium on Operating System Design and Implementation (OSDI)*, 1996.
- [2] Comer, D., *Internetworking with TCP/IP: Volume II*, Prentice-Hall, 1995.
- [3] Cornfed Systems, Inc., *Roadrunner Operating System Reference*, 1998.
- [4] Druschel, P., "Operating System Support for High-Speed Communication", *CACM*, 39, 9, 1996.
- [5] Fall, K. and Pasquale, J., "Improving Continuous-Media Playback Performance with In-Kernel Data Paths", *Proc. of the Intl. Conference on Multimedia Computing and Systems (ICMCS)*, 1994.
- [6] Fall, K., *A Peer-to-Peer I/O System in Support of I/O Intensive Workloads*, Ph.D. Dissertation, University of California, San Diego, 1994.
- [7] Heidemann, J. and Popek, G., "File-System Development with Stackable Layers", *ACM Transactions on Computer Systems*, 12, 1, 1994.
- [8] I₂O Special Interest Group, *Intelligent I/O for Enterprise Computing*, Presentation by I₂O SIG Member Representatives at Fall '97 COMDEX, 1997.
- [9] Intel Corp., *80386 Programmer's Guide*, 1994.
- [10] Kleinman, S., "Vnodes: An Architecture for Multiple File System Types in Sun UNIX", *Proc. of the Summer 1986 Conference*, USENIX, 1986.
- [11] McKusick, M., Bostic, K., Karels, M., and Quarterman, J., *The Design and Implementation of the 4.4BSD Operating System*, Addison-Wesley, 1996.
- [12] Miller, F. W. and Tripathi, S. K., "An Integrated Input/Output System for Kernel Data Streaming", *Proc. of the SPIE/ACM Multimedia Computing and Networking (MMCN)*, 1998.
- [13] Miller, F. W., *Input/Output System Design for Streaming*, Ph.D. Dissertation, University of Maryland, College Park, 1998.
- [14] Pai V., Druschel P., and Zwaenopoele, W., *IO-lite: A Unified I/O Buffering and Caching System*, TR97-294, Rice University, 1997.
- [15] Pike, R., Presotto, D., Thompson, K., and Trickey H., "Plan 9 from Bell Labs", *Plan 9: The Documents - Volume 2*, AT&T Bell Laboratories, 1995.
- [16] Rosenthal, D., "Evolving the Vnode Interface", *Proc. of the Summer 1990 Conference*, USENIX, 1990.
- [17] Skinner, G. and Wong, T., "Stacking Vnodes: A Progress Report", *Proc. of the Summer 1993 Conference*, USENIX, 1993.