Evaluation of Data Passing and Scheduling Avoidance *

José Carlos Brustoloni
jcb@cs.cmu.edu
+1 412 268 3046

Peter Steenkiste
prs@cs.cmu.edu
+1 412 268 3261

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15221

Abstract

Multimedia applications often transfer data between devices or output the same data to multiple devices at very high rates, and therefore require an efficient I/O subsystem. *Data copying* and *context switching* have long been identified as sources of I/O inefficiency, and consequently there have been many proposals of: (1) *Copy avoidance* techniques, which pass data between application and system buffers without copying; (2) *Data passing avoidance* techniques, which allow applications to perform multiple I/O operations on system buffers, without passing data to or from application buffers; and (3) *Scheduling avoidance* techniques, which allow multiple I/O operations on system buffers to proceed at interrupt level, without scheduling. Adoption of such optimizations remains limited because most require major changes in the API or structure of existing operating systems. We recently proposed, however, new copy avoidance techniques, *emulated copy* and *emulated share*, which, unlike previous work, require no such changes. In our experiments, our copy avoidance optimizations gave performance improvements almost as good as or better than those of data passing avoidance, and scheduling avoidance provided only modest performance improvements over priority scheduling. Our results suggest that the I/O performance of conventional operating systems can be greatly improved without the major changes in API or organization commonly advocated.

1 Introduction

Multimedia applications often transfer data between devices or output the same data to multiple devices at very high rates. Therefore, the operating system must provide an efficient I/O subsystem to adequately support such applications.

Data copying and *context switching* have long been identified as sources of I/O inefficiency. Accordingly, many *copy avoidance* [6, 2, 3] and *data passing avoidance* [17, 9, 1, 18, 21] techniques have been proposed for eliminating data copying, and several *scheduling avoidance* schemes have been proposed for eliminating context switching [9, 21]. We discuss these optimizations in turn.

1.1 Copy avoidance

Most workstation operating systems, including Unix [13] and Windows NT [5], use copying to pass I/O data between *application buffers* (pageable buffers accessible by an application at user level) and *system buffers* (unpageable buffers used for physical I/O).

Copying allows operating systems to provide a flexible application programming interface (API), where applications themselves allocate, lay out, reuse, and deallocate their I/O buffers. However, the relative cost of copying has been quickly increasing because improvements in memory bandwidth have been typically much slower than those in the performance of processors [12] or devices such as LANs [3]. Copying has become a major bottleneck in workstation I/O performance [16].

Copy avoidance[1] techniques can be used to pass I/O data between application and system buffers without copying. Unfortunately, most previously proposed such techniques (e.g., [6, 2]) require major changes in the API of conventional systems and thus have not been widely adopted.

In a recent paper, however, we proposed novel copy avoidance schemes, *emulated copy* and *emulated share* [3], which can preserve the API of copying and therefore could easily be adopted in conventional operating systems. In our experiments, *emulated share* provided the best performance among a very broad range of copy avoidance options (including interfaces with *share* or *move* semantics), and *emulated copy* was almost as good [3].

In emulated share I/O, application buffers double as system buffers during I/O; I/O occurs *in-place*, directly between application buffers and physical devices. Compared to copying, emulated share offers the same API, but with lower application buffer integrity guarantees. Using emulated share rather than copying I/O, applications can, after triggering an output, affect physical output data by overwriting output

*This research is sponsored in part by the Advanced Research Projects Agency under contract number DABT63-93-C-0054 and in part by the Advanced Research Projects Agency under contract number F19628-92-C-0116.

[1] In this paper, we will use "copy avoidance" to abbreviate "copy-avoidance data passing".

buffers; and read input buffers in inconsistent or erroneous states during physical input or after failed input operations.

Unlike emulated share I/O, emulated copy I/O uses virtual memory (VM) manipulations (described in Section 2.2.2) to preserve not only the API, but also the application buffer integrity guarantees of copying. Emulated copy can therefore *transparently* replace data copying.

1.2 Data passing avoidance

Although copy avoidance techniques can make data passing efficient, many multimedia, I/O server, and multicasting applications may not require all I/O data to be passed between system and application buffers. Such applications might benefit from *data passing avoidance*, where data passing is reduced or eliminated, rather than merely made more efficient.

For example, multimedia or I/O server applications that transfer large amounts of data between kernel-level I/O modules (e.g., networking module and video digitizer/compression card), without processing that data, can eliminate data passing to and from application buffers by replacing the latter with *kernel* or *outboard* buffers, i.e., system buffers in host or outboard memory, respectively, that applications can specify for multiple I/O operations but cannot access from user level, as shown in Figure 1.

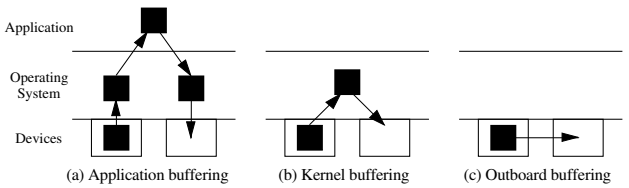

Figure 1: Kernel (*b*) and outboard (*c*) buffering allow multimedia applications to transfer data between devices without the overhead of data passing to and from application buffers (*a*).

Multicast or mirroring applications, which output the same data to multiple kernel-level I/O modules, can reduce data passing by passing data from application buffers to kernel buffers once, and then using those kernel buffers for multiple output operations, as shown in Figure 2.

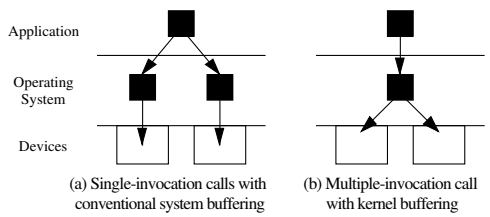

Figure 2: Multicast with kernel buffering reduces data passing between application and system buffers.

Although data passing avoidance can be expected to provide greater I/O performance improvements than those of copy avoidance, it is also likely to require significant changes in the I/O API, and particularly in the system buffer allocation and naming schemes. Conventionally, I/O calls always pass all I/O data between explicit application buffers and implicit system buffers, and each system buffer is automatically deallocated when the single logical I/O operation that uses it terminates. Kernel and outboard buffers, however, must survive across multiple logical I/O operations. Data passing avoidance APIs may:

1. Let the application explicitly allocate and deallocate system buffers, name them using capabilities in I/O calls, and move data between application and system buffers. This is the solution used in the *container shipping* [17] facility.
2. Let the application specify, in an I/O call, an input and an output operation using the same implicit system buffer, automatically deallocated at the end of the call. This is the solution used in the *splice* [9] interface.
3. Allow the application to run as an operating system *extension* at kernel level, so that the application can directly access kernel buffers and submit them for multiple I/O operations. This is the solution used in extensible kernels, such as SPIN [1] and VINO [18], and facilities such as ASHs [8, 21].

Possibly because data passing avoidance schemes require unconventional I/O APIs, they have not been widely deployed.

1.3 Scheduling avoidance

Data passing avoidance makes it possible to use a given kernel or outboard buffer in multiple logical I/O operations. An important question is how to schedule these multiple operations. For example, after a kernel or outboard buffer is filled with data from a video digitizer/compression card, that buffer may be submitted to a network driver for data transmission. The invocation of the network driver may occur in the context of:

1. a regularly scheduled process (e.g., the application itself), as in the *container shipping* interface [17];
2. a callout or software interrupt, as in the *splice* interface [9]; or
3. the interrupt that signals completion of the previous I/O operation (e.g., video capture and compression), as in the ASHs facility [21].

The latter two alternatives are examples of *scheduling avoidance* techniques, which, by eliminating context switching, may improve I/O performance. However, such unscheduled processing may also disrupt system scheduling.

1.4 Contributions and organization

Copy avoidance has important advantages over data passing avoidance: compatibility with conventional APIs and operating system organizations, and ability to improve the performance of all I/O, rather than only I/O of the types shown in Figures 1

and 2. However, experimental comparisons between copy avoidance and data passing avoidance have been missing in the literature.

This paper contributes such comparisons. We show that copy avoidance can provide performance improvements almost as good as or better than those of data passing avoidance.

Additionally, we experimentally evaluate scheduling avoidance. Our measurements show that multiple-priority schedulers, such as that of Unix, can introduce much less latency than reported in previous work [8, 21]. For situations where such latency is significant, we contribute a novel technique, *limited hijacking*, that enables interrupt-level application-specific processing without jeopardizing system scheduling.

Our results suggest that the the I/O performance of conventional operating systems can be greatly improved without the radical modifications in API or operating system structure often proposed.

The rest of this paper is organized as follows. Section 2 gives an overview of our implementation; details are described in the Appendix. Section 3 shows our experimental results, and Section 4 analyzes them. Section 5 discusses related work, and Section 6 presents our conclusions.

2 Implementation in Genie

This section describes, at a high level, the implementation of I/O data passing and scheduling optimizations in Genie, our experimental I/O framework [3]. A more detailed description of the framework can be found in the Appendix.

2.1 Overview of the Genie API

Applications can specify, in a Genie call, invocations to single or multiple *I/O module entries*, as illustrated in Figure 3. I/O modules are, for example, device drivers, protocol stacks, or file systems, and each of their entries implements a different function, e.g., to input or output data. Multiple-invocation calls can be repeated for a specified number of iterations.

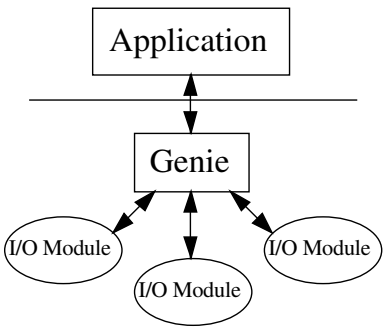

Figure 3: Genie API.

Genie can execute calls *synchronously* or *asynchronously*. In the former case, control returns to the application only when the call completes; in the latter, Genie immediately returns a *handle*, which the application can later use to synchronize with call completion or to abort the call.

Applications must specify the buffers used in a Genie call and select among those the ones to be used in each I/O module invocation. Applications can choose the *data passing semantics* of each buffer, as explained in the next section.

2.2 I/O data passing

Genie *prepares* and *disposes* the buffers specified in a call at the beginning and end of the call, respectively, and *readies* input buffers when requested by an I/O module. Tables 1 and 2 summarize the implementation of the data passing semantics used in this paper.

2.2.1 Emulated share

Genie inputs or outputs data to or from emulated share buffers *in-place*, i.e., directly to or from application buffers, without distinct intermediate system buffers. To make in-place I/O safe, Genie keeps counts of references to each physical page in current I/O operations, and:

1. Modifies the system's page deallocation routine to refrain from placing pages with nonzero I/O reference count in the list of free pages; Genie frees each such page when the last I/O operation referencing it completes (*I/O-deferred page deallocation*). This guarantees that even if a process explicitly deallocates one of its memory regions or terminates (normally or otherwise), its pages with outstanding in-place I/O will not be reallocated to another region or process and thereby be corrupted; and

2. Modifies the pageout daemon to refrain from paging out physical pages with nonzero input reference count (*input-disabled pageout*), since pending input would be likely to make paged out data inconsistent.

2.2.2 Emulated copy

Genie outputs data from emulated copy buffers also in-place. To guarantee that applications cannot affect physical output data by overwriting output pages during output, as would be the case in output with copy semantics, Genie makes application output pages read-only. Additionally, Genie modifies the system's write fault routine as follows: if the page's output reference count is nonzero, the system copies the output page to a new page and maps the new page to the same virtual address, with writing enabled; otherwise, the system simply reenables writing on the page. Because copying may only occur during output, we call this scheme *transient output copy-on-write (TCOW)*.

Input into emulated copy buffers cannot be performed in-place, since that would allow applications to see input buffers in inconsistent states, unlike input with copy semantics. Instead, Genie inputs data into system buffers with the same page offset and length as the corresponding application buffers (*input alignment*). When physical input successfully completes,

Semantics	Prepare	Dispose
Copy	Allocate system buffer. Copyin output data.	Deallocate system buffer.
Emulated copy	Reference application pages. Read-only application pages.	Unreference application pages.
Emulated share	Reference application pages.	Unreference application pages.
Kernel	Allocate system buffer.	Deallocate system buffer.
Outboard	Invoke lender outboard allocation entry.	Invoke lender outboard deallocation entry.

Table 1: Data passing operations for output buffers.

Semantics	Prepare	Ready	Dispose
Copy		Allocate system buffer.	Copyout input data. Deallocate system buffer.
Emulated copy		Allocate aligned buffer.	Swap pages. Deallocate aligned buffer.
Emulated share	Reference application pages.		Unreference application pages.
Kernel		Allocate system buffer.	Deallocate system buffer.
Outboard	Invoke lender outboard allocation entry.		Invoke lender outboard deallocation entry.

Table 2: Data passing operations for input buffers.

Genie swaps pages between the application and system buffers, i.e., unmaps pages from the application buffer and maps pages from the system buffer to the same virtual addresses. Genie handles partially filled system buffer pages as follows: if the data length is below a threshold, Genie simply copies it out; otherwise, Genie completes the page with corresponding data from the respective page in the application buffer (*reverse copyout*) and then swaps pages.

2.2.3 Kernel and outboard buffering

Kernel and outboard buffering are defined as data passing semantics in addition to those defined in our previous work [3].

Applications must specify the *lender* of each outboard buffer. The lender is the I/O module (e.g., a device driver) that allocates and deallocates buffers in the corresponding outboard memory. Lenders must implement *outboard buffer allocate* and *deallocate* entries. Genie invokes these entries automatically when preparing or disposing outboard buffers, respectively.

2.2.4 Buffer management

Genie enforces *per-process* and *global* limits on the amount of system buffering in outstanding I/O operations. If, at prepare time, Genie determines that one of these limits would be exceeded, Genie will: (1) If the call is synchronous, enqueue the call for retry, up to a configurable number of times; or (2) If the call is asynchronous, return an error code to the caller.

To prevent system buffer leaks at process termination, Genie maintains per-process lists of outstanding I/O calls. At process termination, Genie automatically aborts the process's pending I/O calls.

Additionally, to guarantee that all allocated system buffers will eventually be reclaimed, Genie enforces a configurable maximum timeout value for I/O calls.

2.3 I/O scheduling

Applications can specify that processing of multiple invocations in a Genie call be:

1. *Sequential:* Next invocation processed after completion of the current invocation (useful for I/O servers);
2. *Periodic:* Similar to sequential, with possible delay between iterations to satisfy a specified period (useful for digital audio and video);
3. *Parallel:* Invocations of each iteration made in a specified order, but completed asynchronously in arbitrary order (useful for multicast); or
4. *Selective:* Similar to parallel, but each iteration completes when any of its invocations completes (as in Unix `select`).

Processing of an invocation may complete at interrupt level. In such circumstances, immediate processing of the next invocation, without scheduling [21], implicitly assigns to that invocation priority higher than that of any application processing[2]. We call such scheduling avoidance *hijacking*; it eliminates context switching latencies, but may also disrupt system scheduling.

To prevent such disruption, Genie supports only *limited hijacking*: Scheduling can be avoided only for very short processing[3]. Genie monitors the *accumulated processing time* of each call. This time is the sum of measured invocation processing times, reset to zero when the application is rescheduled or there is a delay between periodic iterations. Each I/O module entry has associated *fixed* and *variable* costs. Genie estimates the processing time of an invocation by adding the respective entry's fixed cost and the product of the

[2] Also true of processing as callout or software interrupt [9].

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

respective entry's variable cost and the total length of the buffers specified for the invocation. Genie avoids scheduling if the sum of the accumulated processing time and next invocation's processing time estimate is less than the *hijack limit*, a system constant much lower than the scheduling quantum. Otherwise, Genie awakes the application, and processing of the call continues when the application is rescheduled.

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

3 Experimental results

This section reports Genie's performance for I/O of the types shown in Figures 1 and 2 on a Micron PC with a 166 MHz Pentium processor and 32 MB of main memory. We used the NetBSD 1.1 operating system augmented with an implementation of Genie. Our experiments performed I/O on two devices: (1) The CreditNet OC-3 (155 Mbps) ATM network interface card, which transfers data between host memory and network by burst-mode DMA over the PCI bus with automatic CRC and segmentation and reassembly; and (2) The Cyclone board, which contains an Intel 960 processor, 2 MB of memory, and a bridge with DMA engine between the board's bus and the host's PCI bus. The system was otherwise idle unless explicitly noted. We measured time intervals by recording the value of the Pentium on-chip cycle counter at appropriate points in the code. We report averages of three consecutive runs after a "warm-up" run.

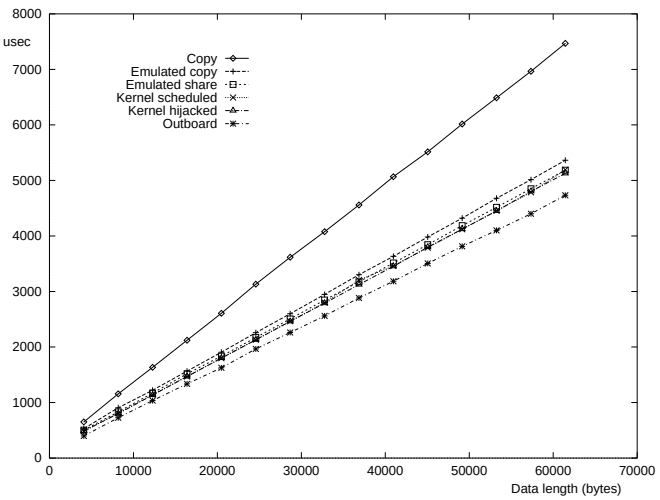

Figure 4: Total latency of data transfer from Cyclone board to ATM network.

Figure 4[4] shows the total latency of transfers of data with length equal to increasing multiples of the page size (4 KB) from the Cyclone board to the ATM network (as shown in Figure 1) as AAL5 packets, without checksumming. For copy, emulated copy, and emulated share semantics, we used separate synchronous single-invocation input and output calls. For kernel semantics, we used single synchronous multiple-invocation calls with sequential invocations for input and output; we manipulated the hijack limit to achieve scheduled or hijacked processing. For outboard semantics, we used a synchronous single-invocation output call, and specified the outboard buffer lender to be the Cyclone board. For 60 KB data, relative to copying, emulated copy reduced latency by 28%, and emulated share and kernel buffering reduced latency by 31%. Outboard buffering provided modest additional latency reduction.

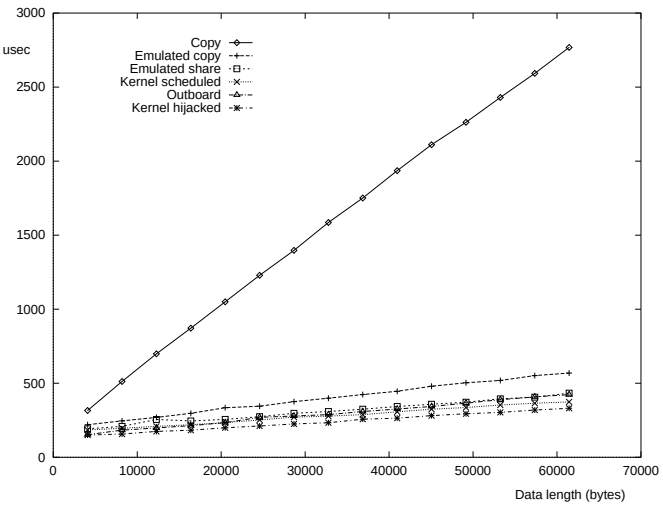

Figure 5: I/O processing time ($t_{I/O}$) of data transfer from Cyclone board to ATM network.

We instrumented the scheduler idle loop to determine how long the processor was idle during the measurements of Figure 4. Subtracting idle times from latencies, we obtained the *I/O processing times* ($t_{I/O}$) shown in Figure 5. We analyze the relationship between $t_{I/O}$ and throughput in the next section.

Figure 6 shows the total latency for output of the same data (as shown in Figure 2) to Cyclone board and ATM network, in that order. We used separate asynchronous single-invocation calls or single synchronous multiple-invocation calls with parallel invocations for output to each device. Latencies differed little between emulated copy and emulated share because of an optimization: Genie now avoids making output pages that are already read-only again read-only (Table 1). Given that we made measurements for output of increasing lengths of the same buffer, Genie avoided most read-only operations. Relative to copying, copy avoidance alone (single-invocation emulated copy) reduced latency by 43%, more than did data passing avoidance alone (multiple-invocation

[4] In all figures, the legend is listed in the same order as the corresponding curves.

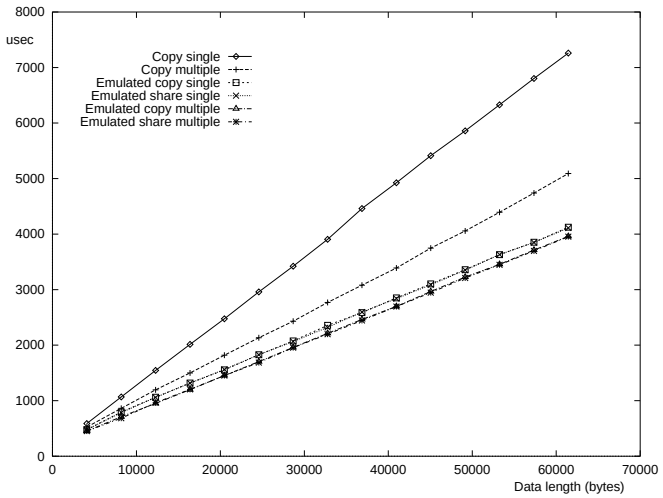

Figure 6: Total latency to output the same application data to Cyclone board and ATM network.

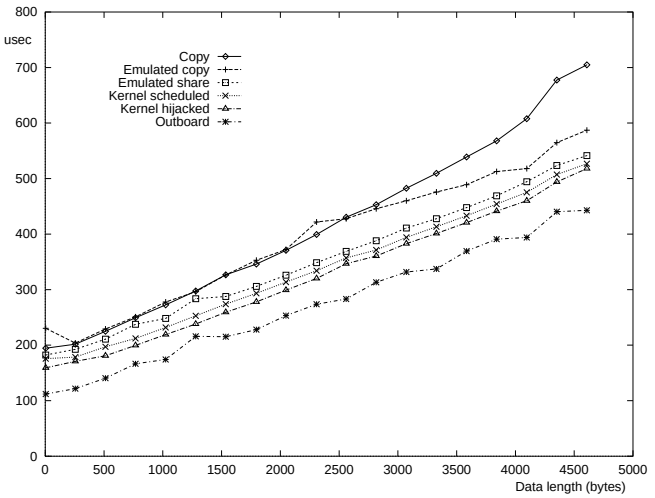

Figure 8: Latency of data transfer from Cyclone board to ATM network on otherwise idle system.

copy), 30%. Combined copy avoidance and data passing avoidance (multiple-invocation emulated copy) reduced latency by 46%.

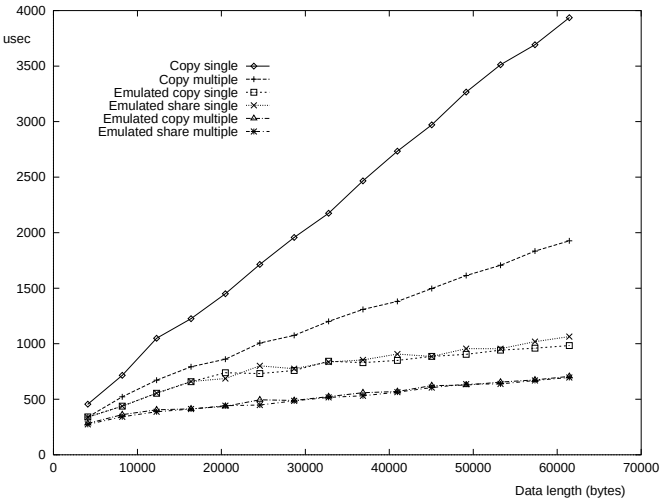

Figure 7: I/O processing time ($t_{I/O}$) to output the same application data to Cyclone board and ATM network.

Subtracting idle times from the latencies of Figure 6, we obtained the *I/O processing times* ($t_{I/O}$) shown in Figure 7. Note that, for long data, copy avoidance alone (single-invocation emulated copy) reduced $t_{I/O}$ (improving throughput) much more than data passing avoidance alone (multiple-invocation copy) did, and that copy avoidance and data passing avoidance combined synergistically in multiple-invocation emulated copy.

Figure 8 shows the total latency of short data transfers between the same devices as in Figure 4. The fairly constant and small difference between latencies of kernel semantics with scheduled or hijacked processing corresponds to the cost of rescheduling (16 μs). Note that NetBSD avoids context switching when the same process is running before and after rescheduling, even after an idle period (guaranteed in Figure 8 because there were no other runnable processes). Latencies were slightly higher with emulated share than with scheduled kernel semantics because of an extra system call and page referencing/unreferencing costs in the emulated share case. Latencies were considerably lower with emulated copy than with copy semantics for data longer than about half a page because of Genie's *reverse copyout* optimization (Section 2.2.2).

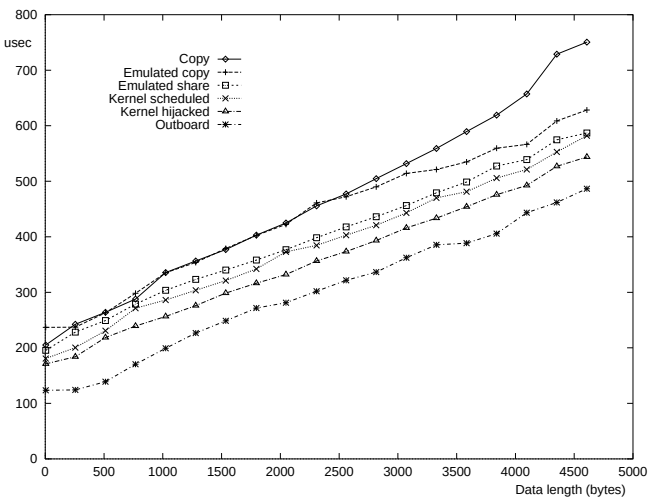

Figure 9: Latency of data transfer from Cyclone to ATM network with ten concurrent compute-bound processes.

We repeated the experiments of Figure 8 with the host running ten concurrent compute-bound processes. When returning to user mode after an interrupt, exception, or system call, NetBSD resched-

ules the CPU if a higher-priority process is runnable. Compute-bound processes in NetBSD and other Unix-derived systems attain low scheduling priority as they accumulate running time, whereas I/O-bound processes, such as our benchmark, receive high priority. We consequently expected our latency measurements to be affected by only modest context switching overheads. This was not true in our first measurements because NetBSD did not guarantee strict priority scheduling. Rescheduling leaves interrupts enabled, after which events may make another higher-priority process runnable *before* the CPU returns to user mode. In such cases, the current lower priority process may be allowed to run until the next timer interrupt (we measured spurious latencies of up to 10 ms). We eliminated this priority inversion by, after rescheduling and immediately before returning to user mode, disabling interrupts and again checking if rescheduling is needed; if not, interrupts are atomically reenabled as part of the return from interrupt instruction. This modification guarantees that whenever the processor is in user mode, it is running the highest-priority process. Figure 9 shows the latencies after the modification. Note that differences between Figures 8 and 9 are quite small. Each rescheduling plus context switching cost on average 28 μs.

4 Analysis

This section analyses the latencies reported in Figure 4 and discusses the relationship between throughput and the I/O processing times reported in Figure 5.

4.1 Latency

We instrumented the code that is executed in the experiments reported in Figure 4 so as to measure the latencies spent in each section of the code. We performed least-squares linear fit on latencies that correlated well with data length and took averages of nearly-constant latencies. Table 3 shows the resulting latencies.

Table 4 compares, for each data passing scheme, the total latency estimated by adding the corresponding breakdown latencies, shown in Table 3, with the least-squares linear fit of the actual total latencies reported in Figure 4. The good agreement between predicted and actual total latencies suggests that our latency breakdown model is quite accurate for the data lengths considered, which are multiples of the page length.

Note that the enter and leave latencies for multiple-invocation calls (c, d) are higher than those for single-invocation calls (a, b) because of the larger number of parameters in the multiple-invocation case. The delay between sequential invocations in multiple-invocation calls (e, which includes limited hijacking calculations) is, however, quite low.

Table 3 reveals why our total latency measurements for long data were so similar for emulated share, emulated copy, and hijacked or scheduled kernel buffering: all breakdown latencies other than copyin (p) and copyout (q) are strongly dominated by device latencies (j and l). I/O processing times (Figures 5 and 7) are considerably more sensitive to data passing scheme than are total latencies (Figures 4 and 6) because device latencies contribute to the latter but not to the former.

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

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

4.2 Throughput

To analyze the relationship between the throughput θ and the I/O processing time $t_{I/O}$, we assume that the system runs a single application that handles multiple *work units* concurrently. Each unit corresponds to data of length L and requires total CPU time t_{APP} for application processing and $t_{I/O}$ for I/O processing. We also assume that the physical I/O subsystem can support a maximum throughput θ_{PHYS}, considering device, controller, and I/O bus bandwidths. Then:

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

where $\theta = \theta_{PHYS}$ if the physical I/O subsystem is saturated, and $\theta = L/(t_{APP} + t_{I/O})$ if the CPU is saturated (i.e., there are enough concurrent work units to eliminate CPU idle time). Therefore:

1. If $t_{APP} \gg t_{I/O}$ (computing takes much more CPU time than I/O does), then θ is essentially unaffected by the data passing scheme.
2. Conversely, if $t_{APP} \ll t_{I/O}$ (I/O-intensive application, e.g., I/O server), then:
   (a) If $\theta_{PHYS} \leq L/t_{I/O}$ (saturated physical I/O subsystem), then θ is also independent of the data passing scheme.
   (b) If $\theta_{PHYS} > L/t_{I/O}$ (saturated CPU), then $\theta = L/t_{I/O}$.

Figure 5 suggests that, for non-copy data passing schemes, even application processing no more complex than copying the data may give $t_{APP} \gg t_{I/O}$ and the same θ for all schemes.

Assuming, on the contrary, $t_{APP} \ll t_{I/O}$ (I/O server), Table 5 indicates the minimum value of θ_{PHYS} ($L/t_{I/O}$) necessary to saturate the CPU.

The bandwidth of the OC-3 ATM card limits θ_{PHYS} to about 135 Mbps. According to the values in Table 5, such θ_{PHYS} will not saturate the CPU for any data passing scheme other than copy. Therefore, θ will be 135 Mbps for all non-copy data passing schemes, but only 104 Mbps for copy and $L = 4$ KB.

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

Table 3: Latency breakdown for device-to-device data transfers, in μs. B is the data length in bytes.

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

Table 4: Estimated and actual total latencies for device-to-device data transfers, in μs, on an otherwise idle host. If other processes are active, h should replace g. B is the data length in bytes.

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

Table 5: I/O processing time ($t_{I/O}$, in μs) and throughput for CPU saturation ($L/t_{I/O}$, in Mbps) in device-to-device data transfers.

Table 5 shows that higher-bandwidth (perhaps parallel) devices and higher L can considerably increase θ, especially with non-copy data passing schemes, up to the PCI bus limit, 800 Mbps. For L = 4 KB, replacement of copy with emulated share semantics, which is possible in conventional APIs, can, using faster devices, improve θ by up to 64%; the hijacked kernel scheme, which requires introduction of a new API, can further improve θ by up to another 27%. For L = 60 KB, however, θ is insensitive to data passing scheme, as long as the latter is non-copy, because of the upper bound set by the I/O bus.

5 Related Work

Genie implements an API that is equivalent to the superset of the APIs of several previous works. For example, the APIs of most conventional operating systems, such as Unix [13] and Windows NT [5], can be considered special cases of Genie single-invocation calls where the data passing semantics is always copy. Likewise, the *splice* interface [9] can be considered a special case of a multiple-invocation Genie call where the number of invocations is always two, the first invocation is always to an input entry, the second invocation is always to an output entry, and kernel buffering is used.

On the other hand, Genie implements a more restrictive model of kernel extensibility than do systems such as SPIN [1] or VINO [18]. Genie multiple-invocation calls allow applications to combine the services of multiple I/O servers in application-specific ways; however, in-kernel I/O servers are themselves pre-existing and trusted. Our extensibility model is similar to that of Unix shells, which allow combining the functionality of multiple programs. SPIN and VINO, on the contrary, allow applications to install quite general application-specific programs as untrusted kernel extensions. Such generality does come at a price, however: to make execution of untrusted extensions safe, SPIN requires extensions to be written in a type-safe language, and VINO has to encapsulate extensions for software fault isolation. These requirements have been reported to make code from 10% to 150% slower [19]. Installation and tearing down of extensions are also probably less expensive in Genie (Table 3, c and d) than in SPIN and VINO, which require extension code to be compiled, checked, or sandboxed, dynamically linked, and downloaded into the kernel for installation, and unlinked and deallocated

for tearing down.

The combination of services of multiple I/O servers in Genie multiple-invocation calls is quite similar to that achieved in Scout *paths* [15]. Genie allows, however, any I/O servers to be invoked in a run-time call, whereas Scout paths are constrained to build-time *router graphs*.

Systems that allow applications to allocate kernel or outboard buffers must prevent applications from hoarding such buffers. Genie and *splice* [9] solve this problem by requiring operations on a given kernel or outboard buffer to occur only in the context of the I/O call where the buffer was allocated; the buffer is automatically deallocated when the call completes. *Container shipping* [17] and extensible kernels [1, 18], on the contrary, allow applications or extensions to hold kernel buffers outside the scope of an I/O call. The referenced papers do not clarify whether or how those systems prevent kernel buffer depletion.

We discuss hardware support for emulated copy in [4].

RMDP [10], EEVDF [20], and SFQ [11] are examples of scheduling policies that allow quality of service guarantees and may be of particular interest for multimedia applications.

Scheduling disruptions due to interrupt-level processing have been reported in several recent papers [14, 7, 15]. LRP [7] and Scout [15] demonstrate substantial throughput improvements under high receiver load by executing in the context of scheduled processes code that in the BSD Unix protocol stack is interrupt-driven.

ASHs have shown that hijacking can dramatically reduce latency [8, 21], in opposition to our results. The source of this discrepancy is that ASHs run on Aegis, which has a round-robin scheduler, whereas our measurements were on NetBSD, with a debugged implementation of the multiple-priority scheduling policy common to most Unix-derived systems.

6 Conclusions

Several different styles of I/O optimization proposed in recent works on operating system support for multimedia and extensible kernels can be brought together under the concepts of *data passing avoidance* and *scheduling avoidance*. Contrary to previous evaluations, which were biased against conventional abstractions by not optimizing them, we compared data passing and scheduling avoidance with copy avoidance and strict priority scheduling, optimizations compatible with conventional abstractions.

Our device-to-device data transfer experiments show that our copy avoidance techniques can give large throughput improvements. Smaller further improvements are possible with data passing avoidance. Our analysis suggests that throughput differences between data passing and copy avoidance may in many cases not be observable because, when I/O processing is not tied up by data copying, either application processing may take longer than I/O processing does, or the physical I/O subsystem may saturate. Copy avoidance has the advantage of not requiring new abstractions and API.

In our multicast experiments, copy avoidance alone gave *greater* performance improvements than those of data passing avoidance alone. Data passing and copy avoidance combined synergistically.

In our experiments, scheduling avoidance provided measurable but modest latency and throughput improvements. We demonstrated a novel technique, *limited hijacking*, that allows scheduling to be avoided selectively and therefore safely.

Acknowledgements

We thank Prashant Chandra for his help in experimental set-up, and George Necula and the anonymous referees for their comments on earlier drafts of this paper.

References

[1] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers and S. Eggers, "Extensibility, Safety and Performance in the SPIN Operating System", *Proc. 15th SOSP*, pp. 267-284, ACM, Dec. 1995.

[2] J. Brustoloni. "Exposed Buffering and Sub-Datagram Flow Control for ATM LANs", *Proc. 19th Conf. Local Computer Networks*, pp. 324-334, IEEE, Oct. 1994.

[3] J. Brustoloni and P. Steenkiste, "Effects of Buffering Semantics on I/O performance", *Proc. OSDI'96*, pp. 277-291, USENIX, Oct. 1996.

[4] J. Brustoloni and P. Steenkiste, "Copy Emulation in Checksummed, Multiple-Packet Communication", *Proc. INFOCOM'97*, IEEE, April 1997.

[5] H. Custer. "Inside Windows NT". Microsoft Press, 1993.

[6] P. Druschel and L. Peterson, "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility", *Proc. 14th SOSP*, pp. 189-202, ACM, Dec. 1993.

[7] P. Druschel and G. Banga, "Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems", *Proc. OSDI'96*, pp. 261-275, USENIX, Oct. 1996.

[8] D. Engler, M. Kaashoek and J. O'Toole Jr., "Exokernel: An Operating System Architecture for Application-Level Resource Management", *Proc. 15th SOSP*, pp. 251-266, ACM, Dec. 1995.

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

[10] R. Gopalakrishnan and G. Parulkar, "A Real-time Upcall Facility for Protocol Processing with QoS Guarantees", *Proc. 15th SOSP*, p. 231, ACM, Dec. 1995.

[11] P. Goyal, X. Guo and H. Vin, "A Hierarchical CPU Scheduler for Multimedia Operating Systems", *Proc. OSDI'96*, pp. 107-121, USENIX, Oct. 1996.

[12] J. Hennessy and D. Patterson. "Computer Architecture: A Quantitative Approach". Morgan Kaufmann Pub., San Mateo, CA, 1990.

[13] S. Leffler, M. McKusick, M. Karels and J. Quaterman. "The Design and Implementation of the 4.3BSD UNIX Operating System", Addison-Wesley Pub. Co., Reading, MA, 1989.

[14] J. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel", *Proc. 1966 Conf.*, pp. 99-111, USENIX, Jan. 1996.

[15] D. Mosberger and L. Peterson, "Making Paths Explicit in the Scout Operating System", *Proc. OSDI'96*, pp. 153-167, USENIX, Oct. 1996.

[16] J. Ousterhout, "Why aren't operating systems getting faster as fast as hardware?", *Proc. Summer 1990 Conf.*, pp. 247-256, USENIX, June 1990.

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

[18] M. Seltzer, Y. Endo, C. Small, K. Smith, "Dealing With Disaster: Surviving Misbehaved Kernel Extensions", *Proc. OSDI'96*, pp. 213-227, USENIX, Oct. 1996.

[19] C. Small and M. Seltzer, "A Comparison of OS Extension Technologies", *Proc. 1996 Conf.*, pp. 41-54, USENIX, Jan. 1996.

[20] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and G. Plaxton, "A Proportional Share Resource Allocation for Real-Time, Time-Shared Systems", *Proc. 17th Real-Time Systems Symp.*, pp. 288-299, IEEE, Dec. 1996.

[21] D. Wallach, D. Engler and M. Kaashoek, "ASHs: Application-Specific Handlers for High-Performance Messaging", *Proc. SIGCOMM'96*, pp. 40-52, ACM, Aug. 1996.

A The Genie I/O Framework

A.1 Application programming interface

The prototype of Genie's single-invocation call is:

```
io_submit(module, entry,
          bufc, bufv,
          timeout, handlep)
```

where `module` specifies the I/O module to be invoked (for example, a file system, protocol stack, or device driver), `entry` specifies the entry point within that module (for example, to input or output data into or from submitted buffers, or to open or close a connection or file), `bufc` is the number of submitted buffers, `bufv` is a vector of pointers to the descriptors of the submitted buffers, `timeout` is the maximum time allowed for completion of the call, and `handlep` points to an identifier returned by Genie. The application can use the identifier to synchronize with call completion or to abort the call. A null `handlep` makes the call synchronous. `io_submit` returns the result of the call (for example, whether *success* or *pending*).

The prototype of the multiple-invocation call is:

```
io_multiple_submit(scheduling, iterations,
                   actual_iterationsp,
                   period,
                   invocs_number, invocs,
                   bufc, bufv, handlep)
```

where `scheduling` can be *sequential, periodic, parallel,* or *selective*, as explained in Section 2.3; `iterations` specifies how many times the call should be repeated; `actual_iterationsp` points to number of times the call was actually repeated (returned by Genie); `period` is the time interval between successive periodic iterations; `invocs_number` is the number of module invocations in each iteration; `invocs` specifies the module invocations of each iteration, in the desired order; and `bufc, bufv,` and `handlep` have the same meaning as in `io_submit`.

Module invocations of each iteration of a multiple-invocation call are specified by a vector of structures of the following type:

```
struct io_invocation {module, entry,
                      bufc, ind_bufv,
                      timeout, resultp}
```

where `module, entry, bufc,` and `timeout` have the same meanings as in `io_submit`, `ind_bufv` is a vector of indices into the `bufv` vector of the `io_multiple_submit` call, specifying the buffers submitted for the given invocation, and `resultp` points to the result of the invocation in the last iteration executed (returned by Genie).

Applications can synchronize with completion of an asynchronous call by calling:

```
io_sync(handle, timeout)
```

where `timeout` specifies the maximum time allowable for call completion. It `timeout` is null, `io_sync` simply polls the status of the call.

Applications can abort an asynchronous call by calling:

```
io_abort(handle)
```

A.2 Data passing

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

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

where `out_arg` points to an argument of length `out_arg_length` that is set by the application and should accompany the buffer, `in_arg` points to an argument of length `in_arg_length` that is set by the invoked I/O module and should be returned to the application, `first_sg` points to the first scatter-gather element in the buffer, `direction` specifies the direction in which data specified by the scatter-gather list should be passed to or from the application (*in* or *out*), when application-accessible, and `actual_lengthp` points to the data length actually input or output (returned by Genie).

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

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

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

1. *Application-accessible, application-allocated* (e.g., copy, emulated copy, or emulated share), in which case `location` gives the virtual address where data starts, set by the application;

2. *Application-accessible, system-allocated* (e.g., move), in which case `location` points to a structure specifying the location of the data. The structure contains the number of virtually contiguous data segments (set by the application), followed by (pointer, length) pairs specifying the successive data segments (set by the application on output and by the system on input); or

3. *Application-inaccessible* (i.e., kernel or outboard), in which case `location` specifies the *lender*, i.e., the I/O module that should be invoked to allocate and deallocate the given buffer element (a null lender denotes allocation and deallocation from Genie's own buffer pool).

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

Genie links a system buffer to each scatter-gather element of submitted buffers. In cases of in-place I/O, system buffers refer to the same physical memory as do the corresponding scatter-gather elements.

System buffers are denoted by lists of structures of type:

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

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

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

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

Within a multiple-invocation call, an input buffer can be converted to an output buffer, or vice-versa, by invocation to the *turn-around* entry of the special I/O module 0. Turn-around requires the buffer to have only elements with *symmetrical* semantics, that is, semantics whose prepare/ready and dispose operations are the same for input or output buffers (that is the case of emulated share, kernel, and outboard semantics).

A.3 I/O module interface

Genie invokes I/O modules according to the prototype:

```
module_invoke(entry, bufc, bufv,
              timeout, handle)
```

where `entry, bufc, bufv`, and `timeout` have the same meaning as in `io_submit`, and `handle` is an identifier that the module can use to notify Genie of asynchronous completion and that Genie can use to abort the invocation. I/O modules cannot block; if a given invocation cannot complete immediately, the module should enqueue information necessary for asynchronous completion, and return *pending*.

Genie defines a number of *reserved entries* that every I/O module must implement, including entries for module initialization, interrupt notification, kernel or outboard buffer allocation or deallocation, and aborting a given invocation. Such entries can be used by Genie but not directly by users. Modules are free to implement any number of other entries. Genie verifies that entries invoked in user calls are not reserved, but does not otherwise interpret entries or buffers submitted. Parameter checking is responsibility of the invoked modules.

Genie provides a run-time library of routines that I/O modules can use, for example, to verify or set user or process permissions, ready or overlay [3] input buffers, notify completion of an invocation, or make single- or multiple-invocation calls to other I/O modules. I/O modules cannot call external routines other than those in the library. I/O modules can have private pools of pageable and unpageable memory, but cannot access external global memory.