Application-Allocated I/O Buffering with System-Allocated Performance

José Carlos Brustoloni Peter Steenkiste

August 1996

CMU-CS-96-169

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

An abridged version of this paper appears as "Effects of Buffering Semantics on I/O Performance" in *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI'96)*, USENIX, Seattle, Washington, October 28-31, 1996, pp. 277-291, and in *Operating Systems Review*, SIGOPS, ACM, vol. 30, Special Issue, 1996, pp. 277-291.

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.

Keywords: Buffering, Copy Avoidance, I/O, Data Communications, Performance Analysis, Operating Systems, ATM Networks

Abstract

We present a novel taxonomy that classifies I/O data passing between applications and operating system along three dimensions: buffer allocation scheme, guaranteed integrity, and optimization. We contribute new optimizations – *input-disabled pageout*, *transient output copy-on-write*, and *input alignment* – that are used in a novel buffering semantics, *emulated copy*. We implemented an I/O framework, Genie, that allows applications to select any semantics in the taxonomy. Using Genie for end-to-end communication over an ATM network, we found that, compared to other semantics, only copy had sharply inferior performance. All other semantics performed quite similarly, contradicting the expectation that emulated copy, being application-allocated and strong-integrity (as is copy semantics), should have considerably worse performance than those of move (system-allocated) or share (weak-integrity) semantics. We analyzed end-to-end latencies in terms of the costs of primitive data passing operations and modeled how those costs scale with CPU, memory, and network speeds. The analysis suggests that current trends tend to intensify the observed performance clustering. We conclude that I/O interfaces with copy semantics, such as that of Unix, can be transparently converted to emulated copy and thus achieve performance approaching the best in the taxonomy.

1. Introduction

Most workstation operating systems continue to use I/O buffering schemes with *copy* semantics, similar to that of Unix [14]. In such schemes, the system inputs or outputs data only through system buffers. On an output call, the system *copies* data from application buffers to system buffers and, conversely, on input completion, the system *copies* data from system buffers to application buffers.

The relative cost of the memory accesses necessary for such copying has increased dramatically since the 1970's, when Unix was introduced. Access times for DRAMs, the almost universal option for workstation main memory, have been improving by roughly only 50% each decade [11]. In contrast, CPU performance has been improving from 50 to 100% per year [11], and local area network (LAN) point-to-point bandwidth, as shown in Table 1, has been increasing by roughly an order of magnitude each decade. Today, LAN bandwidth sometimes actually *exceeds* main memory bandwidth.

Data passing between applications and operating system, therefore, has become a major bottleneck in workstation I/O performance [16]. Emerging I/O-intensive applications, such as multimedia, parallel file systems, and supercomputing on clusters of workstations, among others, demand elimination of this bottleneck.

Several previous works have proposed improving data passing efficiency by making I/O operations have *move* [22] or *share* [5, 2, 9] semantics. Move semantics avoids data copying by using typically much cheaper virtual memory (VM) manipulations. On output with move semantics, the system *removes* the region containing the application data from the application address space. The application buffer *becomes* the system output buffer, and its pages carry the data without copying, being simply unmapped. Conversely, on input with move semantics, the system inputs data into a system buffer and, after input completion, maps it to a freshly allocated region in the application address space. The system buffer *becomes* the application input buffer and its pages carry the data, again, without copying, being simply mapped. Although efficient, these manipulations also imply that applications can neither access output data after output nor determine the location or layout of input data. This restrictive application programming interface (API) may have prevented move semantics from gaining widespread use.

Share semantics eliminates data copying by performing I/O *in-place*, that is, directly to or from application buffers, without distinct intermediate system buffers. Application buffers are simply *promoted* to double as system buffers during I/O. This promotion may require VM manipulations such as *wiring* the buffer to guarantee that it remains in physical memory. Share semantics can use the same API as copy semantics, but offers lower integrity guarantees on application I/O buffers and may require special hardware support.

We present a novel taxonomy of I/O data passing semantics that extends previous works so as to permit a clear characterization of associated software and hardware tradeoffs. The taxonomy identifies a fourth basic semantics for I/O data passing, *weak move*, and associates with each basic semantics the possible optimizations.

This work contributes two new techniques for safety and correctness of in-place I/O: *I/O-deferred page deallocation* and *input-disabled copy-on-write*. We also introduce four new optimizations: *input-disabled pageout*, *region hiding*, *transient output copy-on-write* (TCOW), and *input alignment*. Input-disabled pageout improves the performance of in-place I/O by making it unnecessary to wire

LAN	Year introduced	Bandwidth (Mbps)
Token ring	1972	1, 4, or 16
Ethernet	1976	3 or 10
FDDI	1987	100
ATM	1989	155, 622, or 2488
HIPPI	1992	800 or 1600

Table 1: Approximate year of introduction and point-to-point bandwidth of several popular LANs

the application buffer during I/O, in the traditional sense of removing its pages from lists where the pageout daemon might find them. Region hiding avoids region allocation and removal costs in the *emulated move* semantics. TCOW and input alignment eliminate data copying in an optimized semantics, *emulated copy*, that offers the same API and integrity guarantees as those of copy semantics and thus can, unlike other semantics in the taxonomy, replace copy semantics without requiring any changes in existing applications.

We implemented a new I/O framework, Genie, that allows applications to select any semantics in the taxonomy. Using Genie for communication between PCs and AlphaStations over an ATM network at 155 Mbps, we found, rather surprisingly, that all semantics other than copy performed quite similarly. Only copy semantics had sharply inferior performance.

We analyzed end-to-end latencies in terms of the costs of primitive data passing operations required by each semantics and modeled how those costs scale with CPU, memory, and network speeds. The analysis suggests that, if current trends are maintained, performance differences between semantics other than copy will further decrease, and the performance gap between copy and the other semantics will widen.

The main conclusion of this work is that good I/O performance does not require radical redesign of the I/O API. Existing interfaces with copy semantics can be transparently converted to emulated copy semantics and thus achieve performance approaching the best obtainable with any semantics in the taxonomy.

The rest of this paper is organized as follows. Section 2 presents our taxonomy. Sections 3, 4, and 5 describe techniques for in-place, emulated move, and emulated copy I/O. Section 6 describes the implementation of each data passing semantics in terms of primitive data passing operations. Section 7 shows our experimental results, and Section 8 analyzes them. Section 9 discusses related work, and Section 10 presents our conclusions.

2. Taxonomy of data passing semantics

Data can be passed between applications and operating system according to different semantics. We classify buffering semantics, as shown in Figure 1, in three dimensions: buffer allocation scheme, guaranteed integrity, and level of optimization. The following subsections discuss each dimension in turn.

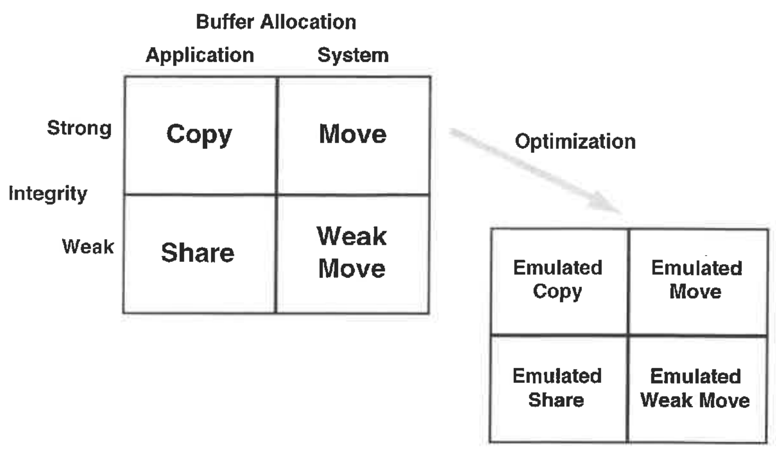

Figure 1: Taxonomy of data passing semantics

2.1. Buffer allocation scheme

In *application-allocated* buffering the *application* determines the location of its input buffers on input and retains access to its output buffers after output. This is the case of the *copy* semantics of Unix I/O [14]. In *system-allocated* buffering, on the contrary, the *system* automatically allocates application input buffers on input and deallocates application output buffers on output, and applications cannot or should not access their output buffers after output. System-allocated buffering is typified by DASH's *move* semantics [22].

Application-allocated and system-allocated buffering require different APIs. The main difference is on input: In the application-allocated API, the application passes the location of its input buffers to the system, while in the system-allocated API, the system returns to the application the location of newly allocated application input buffers. The system-allocated API also includes calls to allocate or deallocate I/O buffers. Applications with balanced amounts of input and output may be able to avoid explicit buffer allocation and deallocation by using buffers implicitly allocated in input operations for subsequent output operations.

System-allocated I/O buffers can be implemented as regions that are marked *moved in* when accessible by the application. Regions that are not system-allocated are marked *unmovable*. Output with system-allocated semantics is only allowed on *moved in* regions because the resulting deallocation might open inconsistent gaps in *unmovable* regions, such as the heap or the stack.

Compared to application-allocated buffering, system-allocated buffering imposes fewer constraints on the system and thus may be more easily optimized, but whether it results in better end-to-end performance depends on the application. Applications that require access to output buffers after output or that are sensitive to the layout of data, e.g. those using data structures such as arrays, may not be able to use system-allocated semantics without copying between system-allocated I/O buffers and application data structures. This user-level copying may defeat performance advantages of system-allocated semantics.

2.2. Guaranteed integrity

For *strong-integrity* buffering the system guarantees that it will output the data contained in the application output buffer at the time of output invocation, unaffected by subsequent overwriting by the application, and that the application will not observe application input buffers in incomplete or erroneous states during input or after failed input operations. For *weak-integrity* buffering the system makes no such guarantees.

Strong integrity is typified by copy and move semantics, both of which guarantee integrity by physically inputting or outputting data only through system buffers inaccessible by the application.

Weak integrity allows I/O to be performed *in-place*, directly into or from buffers mapped to the application. The application can access these buffers during I/O and, consequently, can corrupt data being output or observe input data in inconsistent states.

Weak-integrity buffering can be application-allocated, which we call *share* semantics, or system-allocated, which we call *weak move* semantics. In weak move semantics, output buffers remain mapped to the application after output, but the application *should not* access them because the system may reuse them for subsequent input and their contents until then are indeterminate. This reuse can be implemented by what we call *region caching*: The system marks the region *weakly moved out* and enqueues it in the corresponding list, per address space, where the system can find it for later reuse. On a subsequent input, the system dequeues the region and marks it again *moved in*. Genie uses region caching for weak move as well as the optimized *emulated weak move* semantics. A similar caching technique is used for input buffering in the cached and cached volatile fbuf schemes [9].

2.3. Level of optimization

The buffer allocation scheme and guaranteed integrity together define what we call the *basic semantics* of a data passing scheme. The *semantics* is specified by the basic semantics plus the set of optimizations used. Some optimizations may depend on special conditions, and inclusion of this dimension in the taxonomy makes those conditions visible to programmers[1]. Contrary to the other two dimensions, which each had two discrete points, the optimization dimension admits a spectrum of possibilities, including many different from those presented here. We call Genie's optimized semantics *emulated* because Genie only uses optimizations that are transparent to applications and that do not require changes in programs written for the corresponding basic semantics. The rest of this subsection summarizes previously proposed optimizations, and the following three sections describe novel optimizations used in Genie.

Output with copy semantics can be performed in-place if the region containing the output data is made *copy-on-write* (COW) [17]. The system removes write permissions from all mappings of the pages in the region. An application's attempt to overwrite one such page will cause a VM fault. The system recovers from this fault by copying the page's data to a new page and mapping this new page to the same virtual address in the application address space, with writing enabled. Applications cannot overwrite output pages, which guarantees integrity, and copying only occurs

[1]For optimizations that are transparent to applications, however, our our use of the term *semantics* becomes non-standard; strictly speaking, the semantics in such cases would be what we call *basic semantics*.

if an application does attempt overwriting.

A page-level alternative to COW, reported to have better performance [1], is *sleep-on-write*: the system removes write permissions from all mappings of the output pages and marks the latter *busy* during output. An application's attempt to overwrite one such page will cause the application to fault and stall until output completes.

If the region containing the output data is system-allocated, yet another alternative to COW is *abort-on-write*. On output, the system marks the region *in output* and removes write permissions from the region and all mappings of its pages. An application's attempt to overwrite the latter will cause a protection violation exception and normally will abort the application. The region remains read-only until the application explicitly deallocates and again allocates it. When the application deallocates the I/O buffer, the system places the latter in a list, per address space, where it can be found for reuse. At output completion, the system marks the region *output completed*. At a subsequent I/O buffer allocation call, the system dequeues an *output completed* region, marks it *moved in*, reinstates write permissions, and returns the region to the application. A scheme similar to this is used for cached fbuf output [9].

Input with copy semantics can be optimized, if the application buffer is page-aligned and of length multiple of a page size, by *swapping* pages between system and application buffers. Swapping unmaps application buffer pages and then maps corresponding system buffer pages to the same virtual addresses. This optimization is present in IRIX and HP-UX systems.

I/O with share and weak move semantics can be optimized by requiring application I/O buffers to be located in special unpageable areas, which eliminates the need to wire the buffers during I/O [2].

3. Safe and efficient in-place I/O

Previous proposals have often restricted in-place I/O to special regions, such as *exposed buffer areas* [2] or *fbuf regions* [9]. In this section we describe how Genie makes in-place I/O safe and efficient regardless of data location.

3.1. I/O-deferred page deallocation

Wiring is sufficient to guarantee that a system buffer will remain in physical memory during I/O because only the pageout daemon might asynchronously deallocate system buffer pages. In the case of application buffers, however, wiring is insufficient because other events may also cause application memory deallocation. These events include normal or abnormal termination of the application and explicit memory deallocation by the (possibly malicious) application. If deallocated pages still have pending I/O when they are reused for a different process, there may be corruption of output data or of the other process's memory.

Genie makes in-place I/O safe by keeping counts of input and output references to each physical page in current input and output operations. Genie integrates in what we call *page referencing* the activities of preparing the descriptor with the physical addresses of an I/O request, verifying access rights, and updating reference counts.

Genie changes the system page deallocation routine to refrain from placing pages with nonzero input or output count in the list of free pages, whence they might be allocated to other processes. After completing an I/O operation, Genie *unreferences* pages by updating their reference counts. If a given page no longer has any input or output references, Genie verifies whether the page is still allocated to a memory object; if not, Genie assumes that the page was deallocated during I/O, and enqueues it in the list of free pages for reutilization. We call this scheme *I/O-deferred page deallocation* and use it for all in-place I/O.

3.2. Input-disabled pageout

Genie modifies the pageout daemon to refrain from paging out pages with nonzero input reference count. Pending input would modify these pages *after* pageout, making paged out data inconsistent. Moreover, the application that invoked the input is likely to access these pages after input, making them poor candidates for pageout. Genie allows the daemon to page out pages with zero input count normally, regardless of output reference count.

This optimization, *input-disabled pageout*, adds no overhead to page referencing and makes wiring unnecessary in Genie's emulated semantics, both on input and on output. Note that, unlike previous work [2], input-disabled pageout eliminates wiring costs without requiring data to be placed in unpageable buffer areas. Rather, data can be arbitrarily located, and there is no reduction in the amount of physical memory available to the rest of the system. Performance is the same with unpageable buffer areas or input-disabled pageout because the cost of either option is that of page referencing, which is necessary in both cases to guarantee safe deallocation of output pages.

3.3. Input-disabled COW

COW is frequently used to optimize interprocess communication or memory inheritance with copy semantics [17]. However, it may not implement copy semantics correctly if there is a pending in-place input operation in the region. Indeed, if the input is by DMA, the input will modify memory without generating any write faults, even though the pages are mapped read-only to the applications. Consequently, changes may be observed by processes other than the one that issued the input, and COW in this case actually implements share rather than copy semantics.

Genie guarantees correctness in this case by also monitoring the total number of input references to pages of each memory object in current input operations. Genie updates these counts at page referencing and unreferencing. Genie modifies the system COW set-up routine to perform a physical copy, instead of setting up COW, if any of the region's backing memory objects has nonzero input count.

The reverse case, when a region is marked COW before in-place input, does not require special handling, because input page referencing verifies write access rights, which will automatically fault-in a private, writable copy of the data.

4. Move emulation with region hiding

Genie uses *region hiding* to implement a new semantics, *emulated move*, that is compatible with move semantics but performs I/O in-place. On output with emulated move semantics, Genie removes read and write permissions to pages in the output region, marks the region *moved out*, and enqueues it for later reuse, as in region caching. Genie modifies the system VM fault routine to recover from faults only in *unmovable* or *moved in* regions. Attempts by the application to access the region after output will therefore cause unrecoverable VM faults, as would be the case if the region had actually been removed, but the region and its pages remain allocated to the application address space. On a subsequent input, Genie reuses the region, marks it *moved in*, and reinstates page read and write permissions.

5. Optimizations for copy emulation

5.1. TCOW

Conventional COW can be too expensive for output copy avoidance [1]. Page referencing, however, allows Genie to implement a specialized, page-level form of COW, TCOW, that is highly efficient for this purpose.

On output with emulated copy semantics, Genie simply references and removes write permissions from all mappings of the output pages. Attempts by applications to overwrite output pages will cause VM faults. Genie modifies VM fault processing as follows, in the case of write faults in regions for which the faulted application has write permissions, when the page is found in the top memory object backing the region [17]: If the output count of the page is nonzero, the system recovers from the fault by invalidating all mappings of the page, copying the contents of the page to a new page, swapping pages in the memory object, and mapping the new page to the same virtual address, with writing enabled; if the output count is zero by the time the write fault occurs, the system recovers by simply reenabling writing on the page (no copying). If the page is found but not in the top object, the fault is a conventional COW fault and is handled normally. Note that TCOW adds to page referencing only the cost of removing page write permissions, which is arguably the minimum necessary overhead for strong-integrity, safe in-place output.

TCOW differs from conventional COW in two ways. First, TCOW is transient — COW is conventionally long-term, while TCOW only lasts during output, which is when it is actually useful. Second, TCOW operates at page level instead of at region level. This allows TCOW to prevent the proliferation of regions on output and reduce the number of VM data structures that it needs to manipulate.

TCOW and sleep-on-write are both page-level techniques and perform very similarly for applications that do not overwrite output buffers during output. Both schemes add to the cost of removing write permissions only that of the same number of updates to fields (output reference count or busy bit, respectively) of the page data structure. TCOW offers the added benefits of supporting multiple concurrent output references to a given page and not stalling applications that do overwrite output buffers during output.

5.2. Input alignment

On input with emulated copy semantics, Genie inputs data into system buffers that start at the same page offsets and have the same lengths as the corresponding application buffers. Consequently, Genie can swap pages between system and application buffers even if application buffers are not page-aligned or have lengths that are not multiple of the page size. This scheme, *system input alignment*, goes against the traditional practice of allocating system buffers without regard to the alignment and length of application buffers. As illustrated in Figure 2, lack of alignment makes it impossible to swap pages.

For situations where the system is unable to align its buffers, Genie offers *application input alignment*, where the *application* aligns its buffers to system buffers. Genie includes an interface through which applications can query I/O modules (e.g. a protocol stack) about the preferred alignment and length of input buffers. The preferred alignment may be nonzero and the preferred length may be not equal to a multiple of the page size because of, for example, unstripped packet headers and network maximum transmission units.

Genie uses what we call *reverse copyout* to pass data in partially filled pages in aligned buffers. If data in the system page is shorter than the reverse copyout threshold, Genie simply copies it out, as shown for item 1 in Figure 2. If it is longer, however, Genie first completes the rest of the system page with corresponding data from the respective application page and then swaps the pages, as shown for items 3 and 4 in Figure 2. The threshold is set just above half the page size so as to minimize data copying.

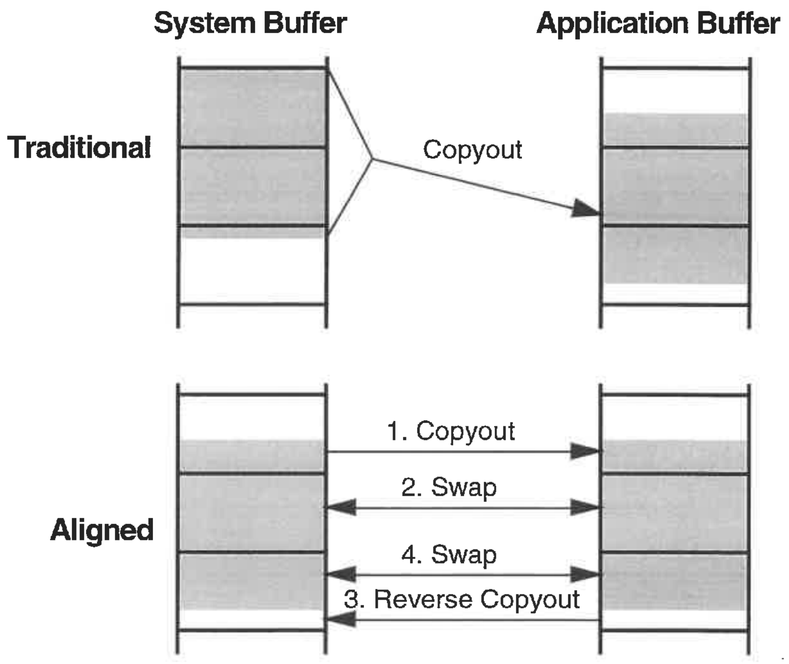

Figure 2: Input alignment makes it possible to swap pages.

6. Data passing implementation in Genie

This section describes, in terms of primitive data passing operations, how Genie implements each semantics for data passing between user-level applications and I/O modules (such as protocol stacks and drivers) in the kernel. The following subsections discuss output and input in turn. We will use these detailed descriptions to analyze experimental results in the next two sections.

Both on output and on input, Genie takes advantage of the fact that copy semantics often is very efficient for short data. If data is shorter than configurable thresholds, Genie automatically converts emulated copy or emulated share semantics into copy semantics.

6.1. Output

Output data passing involves two processing stages: *prepare*, when the application invokes the output operation, and *dispose*, when output completes and the system is ready to return control to the application. The operations used for output are summarized in Table 2, where "read-only" means "remove write permissions", and "invalidate" means "remove all access permissions" from all mappings (VM page table entries) corresponding to a given physical page.

	Prepare	Dispose
Copy	Allocate system buffer. Copyin output data.	Deallocate system buffer.
Emulated copy	Reference application pages. Read-only application pages.	Unreference application pages.
Share	Reference application pages. Wire region.	Unwire region. Unreference application pages.
Emulated share	Reference application pages.	Unreference application pages.
Move	Reference application pages. Wire region. Mark region *moving out*. Invalidate application pages.	Unwire region. Unreference application pages. Remove region.
Emulated move	Reference application pages. Mark region *moving out*. Invalidate application pages.	Unreference application pages. Mark region *moved out* and enqueue.
Weak move	Reference application pages. Wire region. Mark region *moving out*.	Unwire region. Unreference application pages. Mark region *weakly moved out* and enqueue.
Emulated weak move	Reference application pages. Mark region *moving out*.	Unreference application pages. Mark region *weakly moved out* and enqueue.

Table 2: Operations for data output from application to kernel

Note that Genie does not remove a system-allocated region until dispose time in order to guarantee that the corresponding virtual addresses will not be reassigned during I/O, thus allowing graceful recovery in case of error.

6.2. Input

Input data passing involves three processing stages: *prepare*, when the application invokes the input operation; *ready*, when the device needs buffering; and *dispose*, when the input operation is complete and the system is ready to return control to the application.

Input processing must match the type of input buffering used by the device controller. Genie distinguishes three types of device input buffering: *early demultiplexed*, *pooled in-host*, and *outboard*, as described in the following.

6.2.1. Input with early demultiplexing

With early demultiplexed input buffering, buffers reside in host main memory and are specified by multiple (pointer, length) pairs. The device controller keeps separate input buffer lists per input request or connection and inputs data directly into a buffer from the appropriate list.

Early demultiplexing enables in-place or system-aligned buffering *if* the application informs the system about the location of its buffers *before* physical input. This condition is trivially met when physical input is synchronous to application requests. In data communication, however, input may occur before solicited by the application, and location information must be provided either by the receiver, using a preposted, possibly asynchronous input operation, or by the sender, using a field in the packet header [4, 19] or implicitly by the connection used [15]. If location information is not available to the system, copy avoidance is still possible by application-aligned or system-allocated buffering.

Table 3 summarizes the operations used for preposted input with early demultiplexing.

	Prepare	Ready	Dispose
Copy		Allocate system buffer.	Copyout input data. Deallocate system buffer.
Emulated copy		Allocate aligned buffer.	Swap pages. Deallocate aligned buffer.
Share	Reference application pages. Wire region.		Unwire region. Unreference application pages.
Emulated share	Reference application pages.		Unreference application pages.
Move		Allocate system buffer.	Create region. Zero-complete system pages and fill region. Map region and mark *moved in*.
Emulated move	Dequeue *moved out* region, mark region *moving in*, and reference application pages.		Check region, unreference application pages, reinstate page accesses, and mark region *moved in*.
Weak move	Dequeue *weakly moved out* region, mark region *moving in*, and reference application pages. Wire region.		Check region. Unwire region. Unreference application pages and mark region *moved in*.
Emulated weak move	Dequeue *weakly moved out* region, mark region *moving in*, and reference application pages.		Check region, unreference application pages, and mark region *moved in*.

Table 3: Operations for data input with early demultiplexing

Note that, to maintain protection, move semantics has to clear the unused portions of a system buffer before mapping it to the application. If the semantics is emulated move, weak move, or emulated weak move and at prepare time no suitable cached region can be found in the appropriate queue, Genie allocates a new region and marks it *moving in*. For the same three semantics, Genie checks, at dispose time, that the cached region prepared and used for input is still present in the application address space. If it was removed (advertently or not) by the application, Genie maps the corresponding pages to a new region, guaranteeing that the location information returned to the application correctly points to the input data.

6.2.2. Input with pooled buffering

With pooled in-host buffering, the device controller allocates input buffers from a pool of fixed-size buffers (commonly pages) in host main memory without considering the corresponding input request or connection. This is still the most popular of the input buffering schemes, although early demultiplexing is becoming more common with the advent of ATM networks.

Pooled buffering does not allow in-place or system-aligned buffering, and copy avoidance is possible only with application-aligned or system-allocated buffering.

Genie always prepares application input buffers according to Table 3, enabling early demultiplexing. Actual input may use pooled buffering, however, either because the device does not support early demultiplexing or because the application did not inform the location of its input buffers before physical input. For pooled buffering, the ready-time and dispose-time operations are those shown in Table 4. At ready time the I/O module inputs data into *overlay buffers* allocated from a private pool of pages in main memory. At dispose time, Genie passes data from overlay buffers to application buffers and deallocates the overlay buffers, returning them to the respective I/O module pool for reuse.

	Ready	Dispose
Copy	Allocate overlay buffer. Overlay buffer.	Copyout input data. Deallocate overlay buffer.
Emulated copy	Allocate overlay buffer. Overlay buffer.	If aligned, swap pages, else copy out. Deallocate overlay buffer.
Share	Allocate overlay buffer. Overlay buffer.	Unwire region. Unreference application pages. If aligned, swap pages, else copy out. Deallocate overlay buffer.
Emulated share	Allocate overlay buffer. Overlay buffer.	Unreference application pages. If aligned, swap pages, else copy out. Deallocate overlay buffer.
Move	Allocate overlay buffer. Overlay buffer.	Create region. Zero-complete overlay pages, fill region and refill overlay buffer. Map region and mark *moved in.* Deallocate overlay buffer.
Emulated move Emulated weak move	Allocate overlay buffer. Overlay buffer.	Check region. Unreference application pages. Swap pages. Mark region *moved in.* Deallocate overlay buffer.
Weak move	Allocate overlay buffer. Overlay buffer.	Check region. Unwire region. Unreference application pages. Swap pages. Mark region *moved in.* Deallocate overlay buffer.

Table 4: Ready and dispose-time operations for input with pooled buffering

Note that in the case of move semantics, Genie maps overlay pages to the application and therefore needs to refill the overlay buffer with the same number of newly-allocated pages to avoid pool depletion.

6.2.3. Input with outboard buffering

With outboard buffering, the device controller allocates input buffers from a pool in outboard memory. Data in outboard buffers can be transferred directly into application buffers after successful input completion, providing strong integrity regardless of the semantics of application buffers and even if the system was not previously informed of the location of application buffers.

However, outboard buffering can also add complexity and cost to the controller. In the context of network adapters, early demultiplexed and pooled in-host buffering are examples of "cut-through" architectures, while outboard buffering has a "store-and-forward" architecture that typically imposes higher latency.

If the device uses outboard buffers for input, Genie alters the operations in Table 3 as follows: For all semantics other than emulated copy, at ready time, after the operations in the table, start DMA into host memory; and at dispose time, after the operations in the table, deallocate the outboard buffer. For emulated copy semantics, no buffer is allocated at ready time, and at dispose time, the system references the application pages, DMAs data from the outboard buffer to the application buffer, unreferences the application pages, and deallocates the outboard buffer. Consequently, with outboard buffering, emulated copy is implemented much as emulated share semantics.

7. Experimental comparison of data passing semantics

This section reports end-to-end latencies and I/O processing times for datagram communication measured using the various buffering semantics and early demultiplexed or pooled input buffering. We ran our experiments on computers of the types shown in Table 5, connected by the Credit Net ATM network [13] at OC-3 (155 Mbps) rates. All figures in this section refer to Micron P166 PCs. Results for the other platforms were similar and are summarized in Section 8.1. The Credit Net network adapter transfers data between main memory and the physical medium by burst-mode DMA over the PCI I/O bus. We used the NetBSD 1.1 operating system augmented with an implementation of the Genie interface, through which applications accessed the network. We measured latencies by capturing the value of the CPU on-chip cycle counter at appropriate points in the code. All measurements were made on warm caches and are the averages of five runs. We report primarily latencies rather than throughput to simplify analysis.

	Micron P166	Gateway P5-90	DEC AlphaStation 255/233
CPU	Pentium 166 MHz	Pentium 90 MHz	21064A 233 MHz
Integer rating	4.52	< 2.88	< 3.48
L1-cache	8 KBI + 8 KBD, 3560 Mbps	8 KBI + 8 KBD, 1910 Mbps	16 KBI + 16 KBD, 2860 Mbps
L2-cache	256 KB, 486 Mbps	256 KB, 244 Mbps	1 MB, 1366 Mbps
Memory	32 MB, 4 KB page, 351 Mbps	32 MB, 4 KB page, 146 Mbps	64 MB, 8 KB page, 350 Mbps

Table 5: Characteristics of the computers used in the experiments. The integer rating used for the Micron P166 is the listed SPECint95 of the Dell XPS 166. The rating taken as upper bound for the Gateway P5-90 is the listed SPECint95 of the Dell XPS 90, which has a bigger and faster L2-cache. The rating taken as upper bound for the AlphaStation is its listed SPECint_base95 because the version of NetBSD used on it could not be compiled with optimizations. The cache and memory copy bandwidths listed are the peak values we observed using a *bcopy* benchmark at user level.

Figure 3[2] shows latencies using early demultiplexing. We measured latencies for increasing datagram lengths equal to a multiple of the page size, up to 60 KB, the largest such multiple allowed by ATM AAL5. For these datagrams, copy semantics gave much higher latency than did any of the other semantics, which pass data using VM manipulations instead of copying. The differences between semantics other than copy were small. The most striking difference was that between copy and emulated copy semantics, which offer the same API and integrity guarantees.

[2] In all figures in this section, we list the semantics in the legend in the same order as the respective curves. For fine discrimination between curves that appear cluttered in the figure, please refer to Table 9.

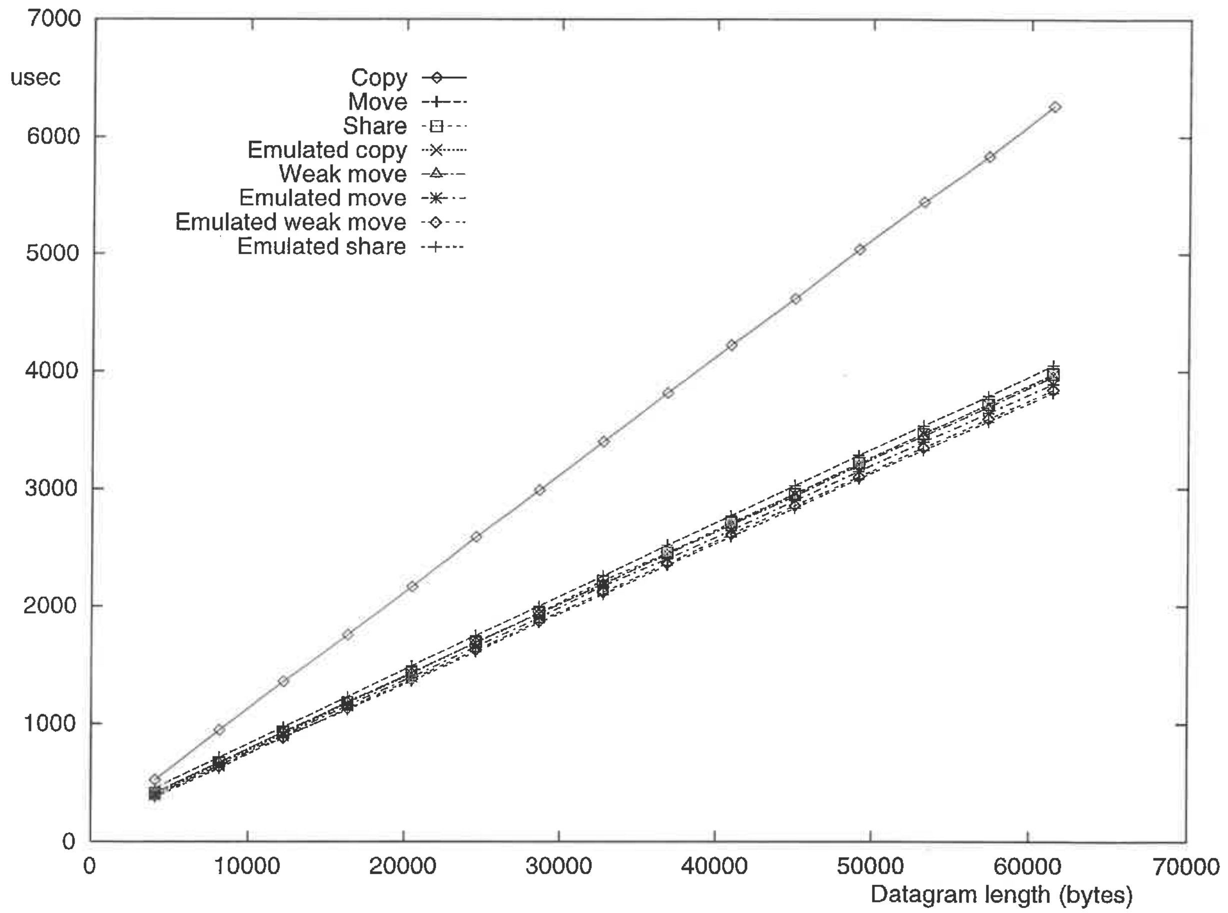

Figure 3: End-to-end latency with early demultiplexing. Semantics other than copy have similar performances.

Using TCOW and input alignment, emulated copy reduced latencies for 60 KB datagrams by 37%. For all data lengths, emulated copy also gave lower latency than those of move and share semantics. This advantage is due to the use of input-disabled pageout instead of region wiring in emulated copy semantics. Emulated move semantics gave slightly lower latencies than those of emulated copy semantics because it simply invalidates and reinstates page table entries instead of fully swapping pages, which also requires updating the respective memory object. Latency was still lower, but only slightly, with the emulated weak-integrity semantics, which do not require page table updates. The equivalent throughput for single 60 KB datagrams was 78 Mbps for copy, 121 Mbps for move, 124 Mbps for share, emulated copy, and weak move, 126 Mbps for emulated move, 128 Mbps for emulated weak move, and 129 Mbps for emulated share semantics.

We instrumented the idle loop in the operating system scheduler to measure CPU idle time while performing the experiment of Figure 3. Subtracting idle times from end-to-end latencies, we obtained the I/O processing times ($t_{I/O}$) shown in Figure 4. We derive the relationship between $t_{I/O}$ and system throughput in Section 8.2. I/O processing times were much higher for copy semantics than for any other semantics. The I/O processing for an exchange of 60 KB datagrams occupied the CPU during 3224 μsec with copy, 986 μsec with move, 952 μsec with share, 931 μsec with weak move, 825 μsec with emulated copy, 753 μsec with emulated move, 688 μsec with emulated weak move, and 644 μsec with emulated share semantics.

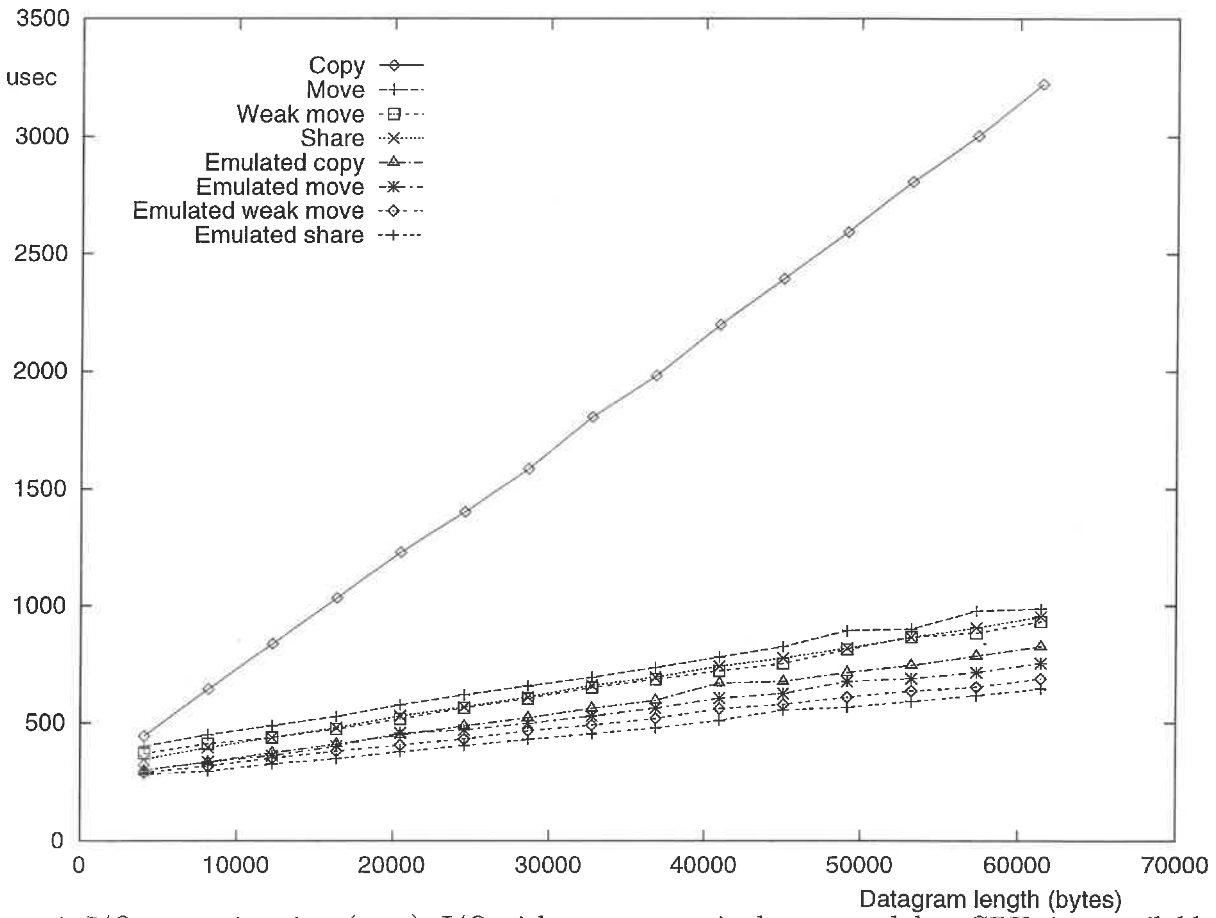

Figure 4: I/O processing time ($t_{I/O}$). I/O with copy semantics leaves much less CPU time available for application processing.

Figure 5 shows short datagram latencies with early demultiplexing. Move semantics gave by far the highest latency for short datagrams because it has to zero the part of the page not occupied by data. Emulated move semantics gave much lower latencies because it performs I/O in place, using region hiding, and therefore does not need to zero the remainder of the page. Copy semantics gave the lowest (145 μsec, tied with emulated share semantics) but also the most rapidly rising latency because of the high incremental cost of copying.

We set output thresholds so that Genie automatically converted to copy semantics output of data shorter than 1666 bytes with emulated copy or 280 bytes with emulated share semantics. We set the reverse copyout threshold at 2178 bytes. (Performance is only moderately sensitive to these settings; we empirically determined these values to give good results.) With these settings, emulated copy semantics had about the same latency as that of copy semantics for data up to half page long; above that, reverse copyout and swapping significantly reduced the latency of emulated copy relative to that of copy semantics. Emulated share had, for all data lengths, the lowest latency, because its data passing overhead consists solely of page referencing. The difference between latencies with emulated copy and emulated share semantics was maximal at half page size: 325 vs. 254 μsec. Weak move and emulated weak move semantics gave slightly higher latencies than those of share and emulated share, respectively, because of region caching costs avoided in application-allocated semantics. The slightly higher latency of emulated move semantics relative to that of emulated weak move semantics is due to region hiding. The higher latencies of share

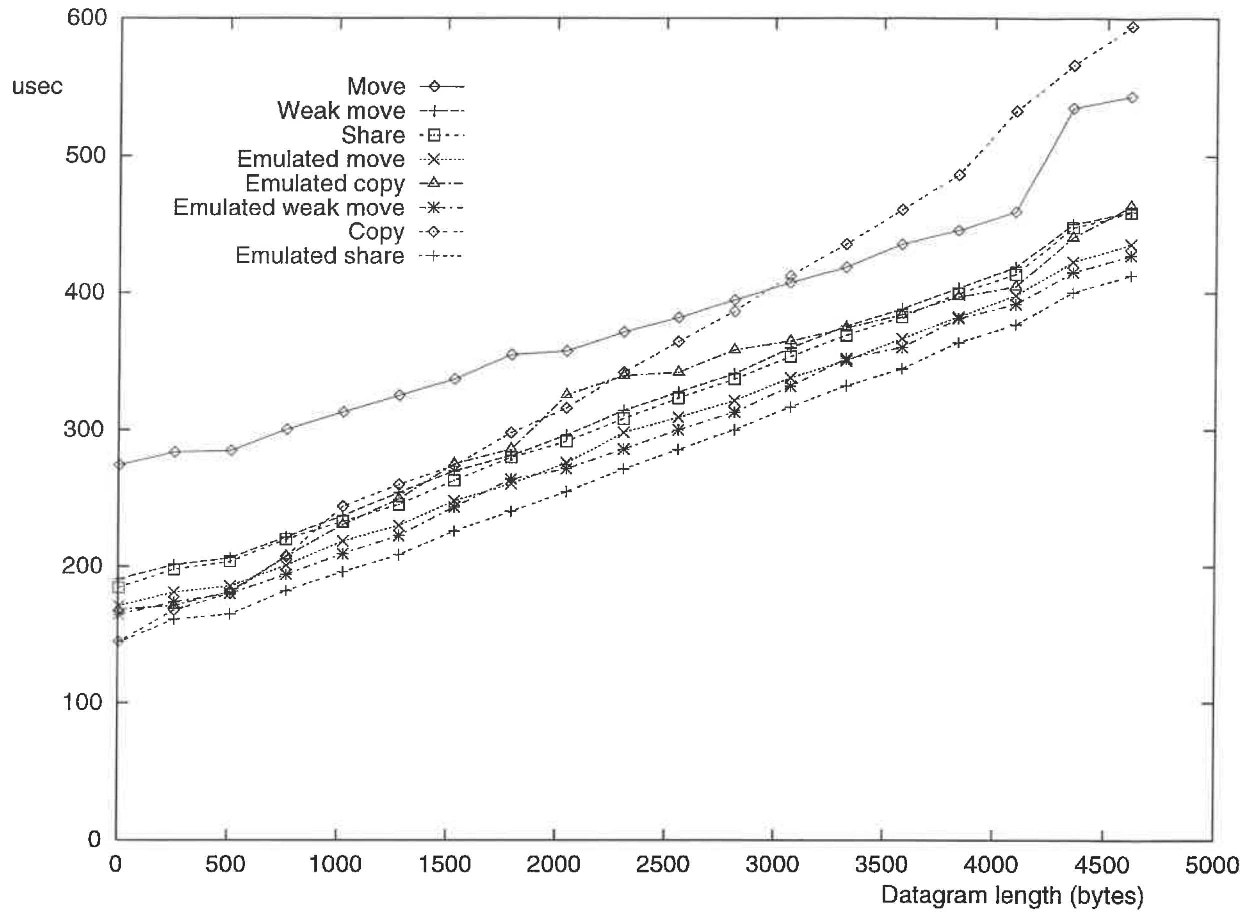

Figure 5: End-to-end latency for short datagrams with early demultiplexing. Using reverse copyout, emulated copy avoids copying more than about half a page.

and weak move semantics relative to their emulated counterparts are due to region wiring and unwiring, which cost about 35 μsec for the first page and become unnecessary in the emulated semantics because of the input-disabled pageout optimization.

Figure 6 shows latencies with pooled input buffering and application-aligned application buffers. Copy and emulated copy have latencies only very slightly higher than the respective latencies with early demultiplexing, corresponding to the same operations plus buffer overlay overhead. Share, move, and weak move semantics have higher latencies than those of emulated copy because of region wiring and unwiring. All other semantics have latencies very close to that of emulated copy. For single 60 KB datagrams, the equivalent throughput is 77 Mbps for copy, 120 Mbps for share, move, and weak move, 123 Mbps for emulated move, emulated copy and emulated weak move, and 124 Mbps for emulated share semantics.

Figure 7 shows latencies with pooled input buffering and unaligned application buffers. In this case, emulated copy, share, and emulated share semantics require data copying on input, whereas the other semantics are unaffected. This figure clearly shows the impact of data copying, splitting the semantics into a group with no copies (system-allocated semantics), another with two copies (copy semantics, with one copy at the sender and another at the receiver), and the remaining group, between the other two, with one copy. For single 60 KB datagrams, the equivalent throughput is 77 Mbps for copy semantics, around 92 Mbps for the other application-allocated semantics, and

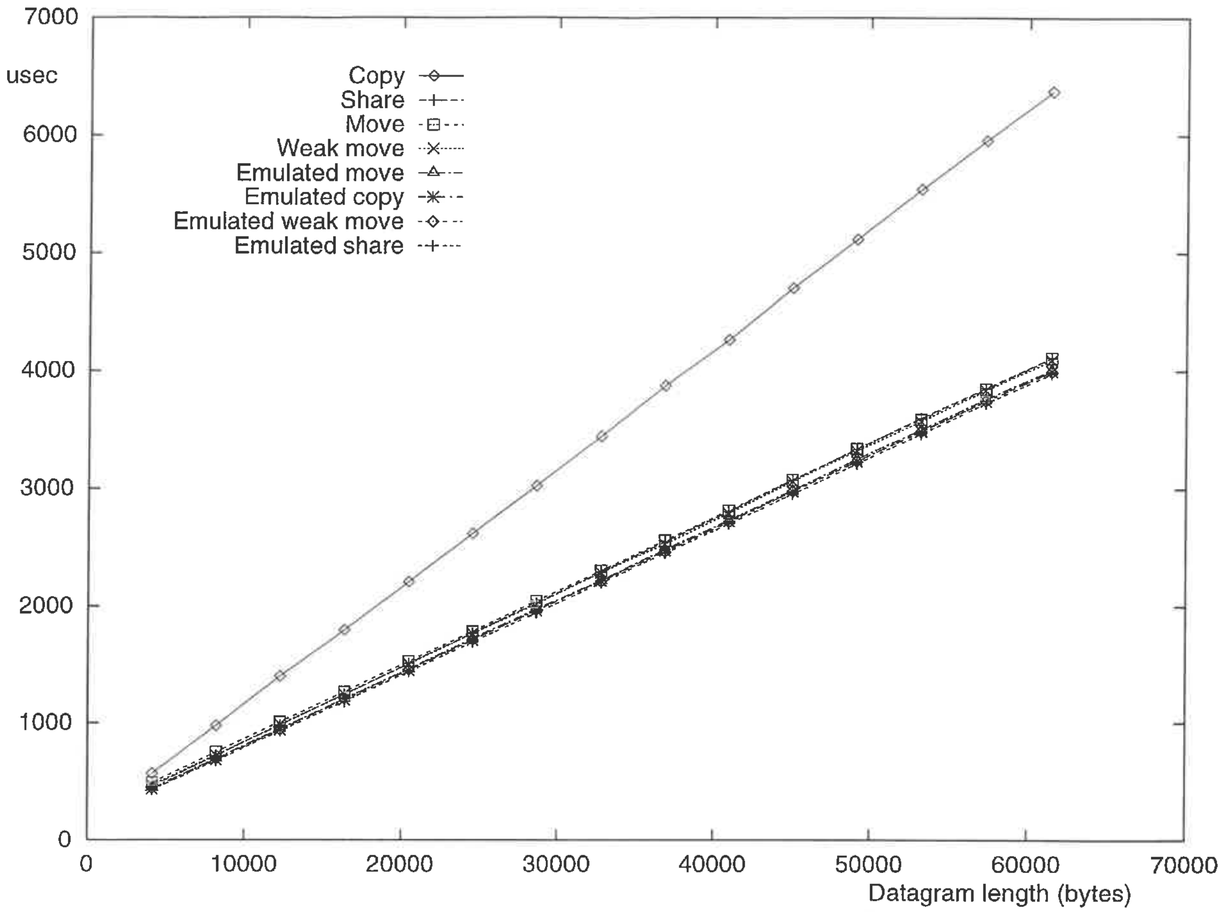

Figure 6: End-to-end latency with application-aligned pooled input buffering. If there is alignment, non-copy application-allocated semantics give performances similar to those of system-allocated semantics.

121 Mbps for the system-allocated semantics.

Figure 7 may give the impression that system-allocated semantics are intrinsically more efficient than application-allocated semantics if pooled buffering is used. However, if an application is insensitive to data layout enough to use system-allocated semantics, then in principle that application can also align its buffers to system buffers, and then emulated copy, emulated share, and system-allocated semantics give very similar performance, as shown in Figure 6. If, on the contrary, an application *is* sensitive to data layout, it would require application-level copies between system-allocated I/O buffers and application data structures. The total number of copies (possibly one copy on output, if the application needs to retain access to the same or other data on the same pages, plus one copy on input) is then at best the same as if emulated copy or emulated share semantics were used (one copy on input only). In those cases, system-allocated semantics may actually give worse end-to-end performance than application-allocated semantics do.

We do not show results with outboard buffering because of limitations in the hardware used. However, we expect that, compared to early demultiplexing, the staging of data at an outboard buffer will add an equal amount of latency to all semantics except emulated copy. Because of the special way the latter is handled, we expect it to have performance even closer to that of emulated share semantics.

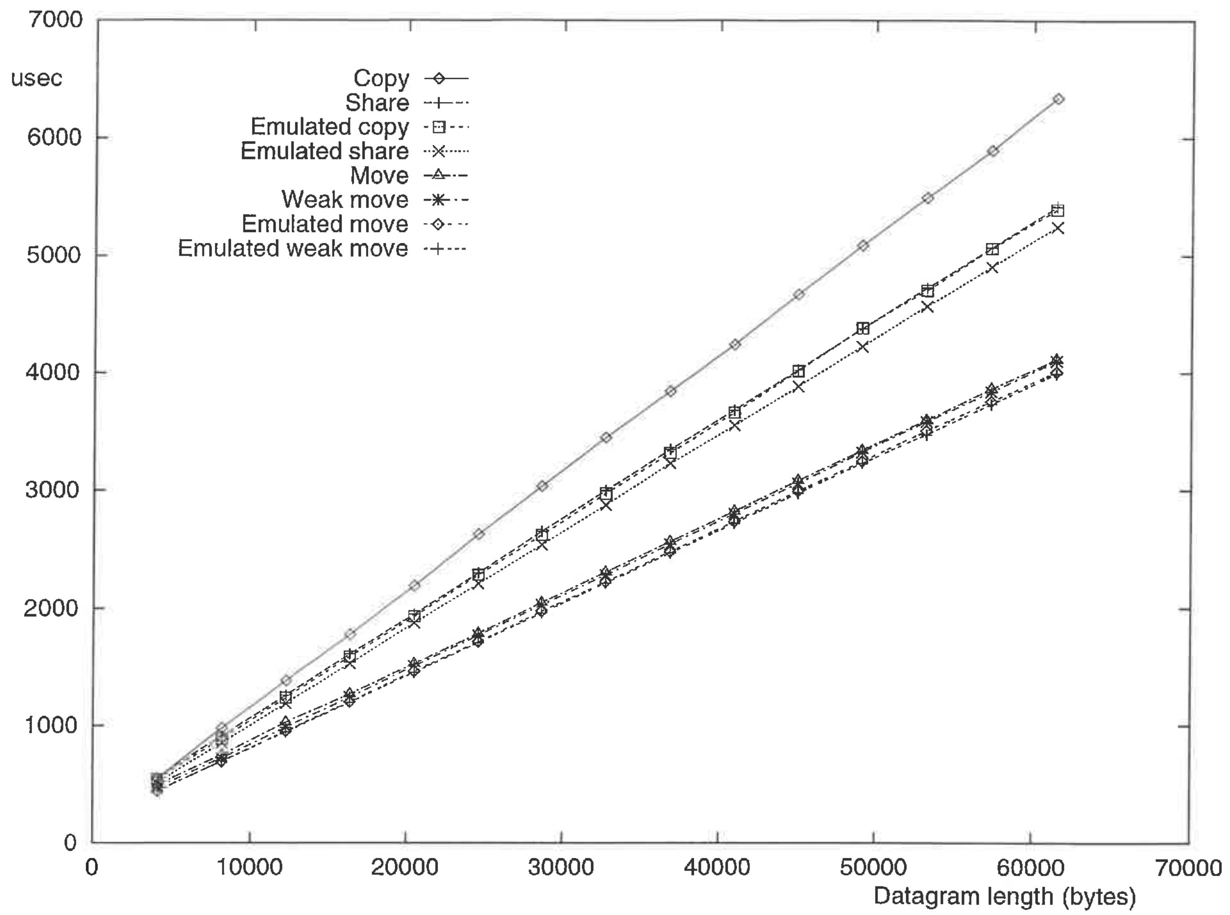

Figure 7: End-to-end latency with unaligned pooled input buffering. Without alignment, non-copy application-allocated semantics require copying at the receiver.

8. Analysis

This section analyzes the effects of I/O data passing semantics on end-to-end latency and system throughput, and discusses the sensitivity of our results to current trends in CPU, memory, and network speeds.

8.1. End-to-end latency

In this subsection, we analyze the empirical end-to-end latencies in terms of the costs of primitive data passing operations and model how those costs scale with CPU, memory, and network speeds.

End-to-end latencies can be *broken down* into the sum of a *base* latency and *data passing* latencies at the sender and receiver. The base latency captures end-to-end costs that are independent of the particular buffering semantics or input buffering scheme used, such as crossing the application-kernel boundary and incurring driver, device, network, and interrupt latencies. Data passing latencies, on the contrary, depend on the semantics and input buffering scheme used. We take the base latency to be equal to the end-to-end latency of emulated share semantics with early demultiplexing, reduced by the costs of referencing and unreferencing application I/O buffers.

Only *prepare* time data passing operations at the sender contribute to end-to-end latency, because dispose-time operations overlap with network latencies and latencies at the receiver. Conversely, with early demultiplexing, prepare and ready time operations at the receiver overlap with latencies at the sender and in the network, and only the *dispose* time operations at the receiver contribute to end-to-end latency. With pooled or outboard buffering, only *ready* time and *dispose* time operations at the receiver contribute to end-to-end latency.

We directly measured the latencies of primitive data passing operations by instrumenting the Genie code. We added instructions at appropriate points in the code to record the value of the CPU on-chip cycle counter. We recorded the time intervals for each operation and datagram length when performing the experiments reported in Figures 3, 6, and 7, taking the averages of five runs. We then performed a least-squares linear fit on each operation latency versus datagram length. We obtained excellent correlation except in cases of constant or very small latencies. We averaged the fitted equations of each operation latency over the semantics and input buffering schemes where the operation is used. We show the results for each platform in Tables 6, 7, and 8, respectively. Note that *copyin* cost less than *copyout* because our experiments were on warm caches. On output (*copyin*), data can be read from the cache, while on input (*copyout*) it has to be read from memory. The *copyin* cost is actually nonlinear because the L1-cache has much higher bandwidth than that of the L2-cache. This causes a negative y-intercept in the corresponding linear fit.

Operation	Latency	Operation	Latency
Base	$0.0598\ B + 130$	Swap	$0.00163\ B + 15$
Copyin	$0.0180\ B - 3$	Copyout	$0.0220\ B + 15$
Reference	$0.000363\ B + 5$	Unreference	$0.000100\ B + 2$
Wire	$0.00141\ B + 18$	Unwire	$0.000237\ B + 10$
Read only	$0.000367\ B + 2$	Region create	24
Invalidate	$0.000373\ B + 2$	Region fill	$0.000398\ B + 9$
Region mark out	3	Region fill & overlay refill	$0.000716\ B + 11$
Overlay allocate	7	Region map	$0.000474\ B + 6$
Overlay	7	Region check, unreference, reinstate, mark in	$0.000507\ B + 11$
Overlay deallocate	$0.000344\ B + 12$		
Region check	5	Region check, unreference, mark in	$0.000194\ B + 6$
Region mark in	1		

Table 6: Costs of primitive data passing operations on the Micron P166 computer, in μsec. B is the data length in bytes.

Operation	Latency	Operation	Latency
Base	$0.0718\ B + 179$	Swap	$0.00285\ B + 27$
Copyin	$0.0440\ B - 63$	Copyout	$0.0535\ B + 23$
Reference	$0.000649\ B + 10$	Unreference	$0.000186\ B + 3$
Wire	$0.00256\ B + 33$	Unwire	$0.000440\ B + 18$
Read only	$0.000648\ B + 3$	Region create	43
Invalidate	$0.000707\ B + 3$	Region fill	$0.000714\ B + 15$
Region mark out	5	Region fill & overlay refill	$0.00132\ B + 20$
Overlay allocate	17	Region map	$0.000835\ B + 12$
Overlay	15	Region check, unreference, reinstate, mark in	$0.000972\ B + 20$
Overlay deallocate	$0.000543\ B + 22$		
Region check	9	Region check, unreference, mark in	$0.000327\ B + 11$
Region mark in	2		

Table 7: Costs of primitive data passing operations on the Gateway P5-90 computer, in μsec. B is the data length in bytes.

Operation	Latency	Operation	Latency
Base	0.0710 B + 235	Swap	0.00443 B + 12
Copyin	0.00974 B - 5	Copyout	0.0182 B + 1
Reference	0.000347 B + 8	Unreference	0.000244 B + 6
Wire	0.00175 B + 23	Unwire	0.000450 B + 25
Read only	0.000275 B + 4	Region create	43
Invalidate	0.00141 B + 4	Region fill	0.000428 B + 8
Region mark out	5	Region fill & overlay refill	0.000730 B + 10
Overlay allocate	17	Region map	0.00175 B + 3
Overlay	11	Region check, unreference, reinstate, mark in	0.00167 B + 16
Overlay deallocate	0.000336 B + 28		
Region check	7	Region check, unreference, mark in	0.000227 B + 12
Region mark in	2		

Table 8: Costs of primitive data passing operations on the AlphaStation 255/233 computer, in μsec. B is the data length in bytes.

Semantics		Early demultiplexing	Application-aligned pooled	Unaligned pooled
Copy	E	0.0997 B + 141	0.100 B + 166	0.100 B + 166
	A	0.0998 B + 125	0.101 B + 139	0.101 B + 144
Emulated copy	E	0.0621 B + 153	0.0625 B + 178	0.0828 B + 177
	A	0.0622 B + 150	0.0622 B + 175	0.0848 B + 195
Share	E	0.0619 B + 165	0.0637 B + 204	0.0841 B + 203
	A	0.0621 B + 162	0.0638 B + 197	0.0846 B + 219
Emulated share	E	0.0602 B + 137	0.0621 B + 175	0.0825 B + 175
	A	0.0600 B + 137	0.0619 B + 167	0.0824 B + 178
Move	E	0.0628 B + 197	0.0634 B + 224	0.0634 B + 224
	A	0.0626 B + 202	0.0631 B + 234	0.0631 B + 234
Emulated move	E	0.0610 B + 151	0.0625 B + 185	0.0625 B + 185
	A	0.0609 B + 150	0.0623 B + 183	0.0623 B + 183
Weak move	E	0.0620 B + 173	0.0637 B + 212	0.0637 B + 212
	A	0.0615 B + 170	0.0633 B + 206	0.0633 B + 206
Emulated weak move	E	0.0603 B + 144	0.0621 B + 183	0.0621 B + 183
	A	0.0602 B + 143	0.0619 B + 184	0.0619 B + 184

Table 9: Estimated (E) and actual (A) end-to-end latencies on Micron P166 computers, in μsec. B is the data length in bytes.

Taking the values from Table 6, we added, for each semantics, the base latency, the costs of the respective output prepare-time operations indicated in Table 2, and the costs of the respective input dispose-time operations indicated in Table 3, obtaining an estimate of the respective end-to-end latency with early demultiplexing. We show these estimates in Table 9, along with the least-squares linear fit of the actual end-to-end latencies from Figure 3. Adding base latency, the costs of output prepare-time operations, and the costs of input ready-time and dispose-time operations indicated in Table 4, for each semantics, we obtained estimates of the respective end-to-end latencies with pooled buffering and application-aligned or unaligned application buffers. We show these estimates in Table 9, along with the least squares linear fit of the actual end-to-end latencies from Figures 6 and 7. The good fit between estimated and actual latencies suggests that our breakdown model is accurate for the datagram lengths considered, which are multiples of the page size (additional terms would increase accuracy for intermediate lengths, but also make the model more complicated).

Using the breakdown model, the end-to-end latency when sender and receiver use different semantics can be expected to be equal to the sum of the base latency plus sender-side latencies of the semantics used by the sender plus receiver-side latencies of the semantics used by the receiver.

The breakdown model can be extended into a *scaling* model that takes into account CPU, memory, and network speeds. To a first approximation: (1) The multiplicative factor of the base latency is *network-dominated* and equal to the inverse of the net network transmission rate, subject to adapter and I/O bus bandwidth limitations; (2) The fixed term of the base latency is equal to the sum of I/O bus, device, and network latencies, plus a term corresponding to fixed operating system overhead, which scales inversely to CPU speed; (3) The *copyout* multiplicative factor is *memory-dominated* and equal to the inverse of the main memory copy bandwidth, and the associated fixed term can be ignored; (4) The *copyin* multiplicative factor is *cache-dominated* and may vary between the inverse of the copy bandwidth of the L2-cache and the inverse of the copy bandwidth of main memory, depending on data and cache sizes and cache associativity and locality. The fixed term can be ignored; (5) All other parameters are *CPU-dominated* and scale inversely to to CPU speed, as estimated by an appropriate integer benchmark, such as SPECint95 [3].

	Micron P166		Gateway P5-90		AlphaStation 255/233	
Type of Parameter	Estimated	Actual	Estimated	Actual	Estimated	Actual
Network-dominated	> 0.0570	0.0598	> 0.0570	0.0718	> 0.0570	0.0710
Memory-dominated	0.0228	0.0220	0.0548	0.0535	0.0229	0.0182
Cache-dominated	$> 0.0165, < 0.0228$	0.0180	$> 0.0328, < 0.0548$	0.0440	$> 0.00586, < 0.0229$	0.00974

Table 10: Data passing costs estimated according to net network transmission rate and memory and L2-cache copy bandwidths are consistent with the actual values.

We used data from Tables 5, 6, 7, and 8 to verify the scaling model. Table 10 shows the verification of (1), (3), and (4) for each platform. Table 11 shows the verification of (3), (4), and (5) across platforms. Agreement between estimated and actual scaling was quite good for the Gateway P5-90, which has the same architecture as the base case. In the AlphaStation, CPU-dominated ratios had geometric means consistent with the model but variances that were much higher than those of the Gateway P5-90. This could be expected, given that the AlphaStation has a substantially different architecture.

	Gateway P5-90				AlphaStation 255/233			
Type of Parameter	Estimated	GM	Min	Max	Estimated	GM	Min	Max
Memory-dominated	2.40	2.43	2.43	2.43	1.00	0.83	0.83	0.83
Cache-dominated	$> 1.44, < 3.33$	2.46	2.46	2.46	$> 0.26, < 1.39$	0.54	0.54	0.54
CPU-dominated multiplicative factor	> 1.57	1.79	1.58	1.92	> 1.30	1.64	0.75	3.77
CPU-dominated fixed term	> 1.57	1.83	1.53	2.59	> 1.30	1.54	0.47	3.74

Table 11: Scaling of data passing costs relative to the Micron P166. "GM", "Min", and "Max" are the geometric mean, minimum, and maximum values of the ratios of parameters of each given type.

A comparison between Tables 6 and 9 using the scaling model explains the clustering in Figures 3, 6, and 7. Network-dominated costs strongly dominate CPU-dominated costs (making performance differences between non-copy semantics relatively minor), but not memory and cache-dominated costs (penalizing copy semantics).

Extrapolating based on the scaling model, if CPU speeds continue to increase faster than transmission rates, as is the current trend, the performance differences between semantics other than

[3]The cost of page table updates may scale otherwise between processors of different architecture, causing the cost of the *read-only*, *invalidate*, *swap*, *region map*, and *reinstate* operations to diverge from this model. Page table updates are particularly costly in multiprocessors, where the semantics that do not use these operations – weak (with early demultiplexing) and copy – may have some advantage.

copy will tend to decrease, and if CPU speeds continue to increase faster than main memory bandwidth, the performance difference between copy and other semantics will increase. At OC-12 (622 Mbps) rates, the scaling model predicts that the end-to-end throughput for single 60 KB datagrams with early demultiplexing on the Micron P166 PCs will be close to 140 Mbps with copy semantics, 404 Mbps with emulated copy semantics, 463 Mbps with emulated share semantics, or 380 Mbps with move semantics, giving emulated copy almost three times better performance than that of copy semantics.

8.2. System throughput

In this subsection, we derive the relationship between the I/O processing time $t_{I/O}$ (Figure 4) and the system throughput θ.

To simplify analysis, 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 capacity.

If the system has no idle CPU time (possibly because of multiple concurrent work units), then $\theta = L/(t_{APP} + t_{I/O})$. Conversely, if the physical I/O subsystem is saturated, then $\theta = \theta_{PHYS}$. In general, the system can support a maximum throughput[4]:

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

Consequently:

1. If $t_{APP} \gg t_{I/O}$ (application performs much more computation than I/O), then θ is essentially independent of buffering semantics.
2. Conversely, if $t_{APP} \ll t_{I/O}$ (I/O-intensive application, e.g. I/O server), then:
   (a) If $t_{I/O} \leq L/\theta_{PHYS}$ (saturated physical I/O subsystem), then θ is also independent of buffering semantics; or
   (b) If $t_{I/O} > L/\theta_{PHYS}$ (saturated CPU), then $\theta = L/t_{I/O}$. In this case, using the data from Figure 4, we can estimate that, on that system, for $L = 60$ KB, emulated copy and emulated share semantics can increase system throughput with respect to that of copy semantics by 291% and 401%, respectively. (Note that the improvement will be less if the the CPU saturates with copy semantics but not with non-copy semantics.) The system throughput with emulated share semantics is up to 28% greater than that with emulated copy semantics.

[4]Note that the throughput for single datagrams calculated in the previous sections does not involve multiple concurrent work units, may saturate neither CPU nor physical I/O subsystem, and therefore may be less than the system throughput derived here.

To a first approximation, t_{APP} scales inversely to CPU speed, as measured by an appropriate benchmark, such as SPECint95; $t_{I/O}$ scales inversely to the memory copy bandwidth, for copy semantics, or inversely to CPU speed, for non-copy semantics; and θ_{PHYS} scales proportionally to device speed. Therefore, we can project the effects of current trends in CPU, memory, and device speeds as follows:

1. For copy semantics, the relationship between t_{APP} and $t_{I/O}$, for applications with good cache locality, may tend to $t_{APP} \ll t_{I/O}$ (I/O-intensive application) because CPU speeds are increasing faster than memory bandwidth is. For applications with poor locality, t_{APP} may scale similarly to $t_{I/O}$. The relationship between $t_{I/O}$ and L/θ_{PHYS} tends to $t_{I/O} > L/\theta_{PHYS}$ (saturated CPU) for devices, such as high-speed networks, whose bandwidth is increasing faster than that of memory.

2. For non-copy semantics, t_{APP} tends to scale similarly to $t_{I/O}$. The relationship between $t_{I/O}$ and L/θ_{PHYS} tends to $t_{I/O} \leq L/\theta_{PHYS}$ (saturated physical I/O subsystem) because CPU speed is improving faster than device bandwidth is.

Consequently, also for system throughput, current trends tend to reduce performance differences between non-copy semantics, because of either relatively long-running applications or saturated physical I/O subsystem. For I/O-intensive applications, the tendency is to increase performance differences between copy and non-copy semantics, because of saturation of CPU/memory in the former and of the physical I/O subsystem in the latter.

9. Related work

Previous works on I/O buffering have typically focused on optimizations using a particular semantics, device, or application. We are not aware of other direct, controlled comparisons covering as broad a range of buffering options as we present here.

The technique discussed here for emulated copy input with outboard buffering is a generalization of those in [18, 8]. However, those works also use outboard buffering for copy elimination on output. We, on the contrary, use simple VM manipulations to avoid copying on output, which may simplify device interface design. Staging output through an outboard buffer may still be advantageous, however, if, for example, there is hardware to compute the TCP checksum (which goes in the packet header) while data is being DMAed from application buffer to outboard buffer (as in [18, 8]).

Fbufs [9] are system-allocated but are optimized with mixed semantics. Cached fbuf output has semantics similar to emulated copy, but requires wiring and uses abort-on-write instead of TCOW. Cached volatile fbuf output has semantics similar to share. Cached and cached volatile fbuf input have semantics similar to weak move but use read-only buffers that must be deallocated explicitly.

We assumed the use of DMA for I/O, which is necessary for high throughput in most current systems. Programmed I/O may make it unnecessary to wire application buffers, reducing latency, but subject to page faults, which could cause deadlock if the faulted process is itself a (possibly user-level) pager or is invoked by a pager. In our implementation, input-disabled pageout eliminates wiring costs safely and cheaply. Programmed I/O may have the advantage of simplifying the implementation of early demultiplexing [5, 2].

There have been proposals to reduce the penalty of copying by integrating it with other data touching operations, such as TCP checksumming [6]. Integration of checksumming on input has semantic implications: If checksumming is integrated with the copy from device or system buffer to application buffer, and the checksum is wrong, the application buffer will be overwritten with faulty data, and the semantics is actually share, not copy. If a system buffer is involved (i.e., not programmed I/O between device and application buffers), in our implementation, at least for long data, it costs less to pass the data by VM manipulation and then read it for checksumming than to read and write (one-step checksum and copy) the data [3].

We have assumed in this paper a one-to-one relationship between application buffers and network packets. This may hold in datagram communication, but is uncommon in byte streams. In the latter case, the sender fragments data into packets of variable length and adds a header and a trailer to each packet; the receiver has to strip headers and trailers and concatenate data of successive packets. In another paper [3], we show that early demultiplexing provides sufficient support for multiple-packet emulated copy *if* there is end-to-end agreement on the length of data transfers. We propose a novel hardware feature, *buffer snap-off* [3], that removes the latter restriction.

The work reported here concerns I/O data path (buffering) optimizations, but we would like to point out recent work on I/O control-path optimizations, such as bypassing the operating system (OS) [7, 10, 20] and separating control from data transfer [21, 15]. In terms of the analysis from the previous section, bypassing the OS reduces the constant term of the base latency (i.e., the latency for short data). Given that the wiring of application buffers requires OS intervention, OS bypassing may imply some form of either unpageable buffer areas or application-level programmed I/O between application buffers and device controller.

10. Conclusions

We introduced a new taxonomy that characterizes in a structured way a very broad and general range of options for I/O data passing between applications and operating systems. We described the implementation of data passing in Genie, a new I/O framework that allows applications to select any data passing semantics in the taxonomy. Using an implementation of Genie on NetBSD, we made direct, controlled evaluations and comparisons of the different buffering semantics. We identified the fundamental similarities and differences between the various semantics, assessed the impact of different architectural support found in device controllers, and investigated how performance scales with CPU and memory speeds.

The experiments demonstrated significant performance gains due to TCOW, input alignment, region hiding, and input-disabled pageout. The results indicate that large I/O performance improvements are possible in many Unix-derived operating systems by using emulated copy instead of copy semantics. This substitution does not require changes in applications because both semantics give the same integrity guarantees and can offer the same API.

At the I/O rates tested, some additional latency reduction is possible by using emulated share semantics, particularly at short data lengths. The scaling model predicts that larger relative improvements would occur when interfacing with faster devices or in shared-memory multiprocessors. Emulated share semantics can also considerably further improve the system throughput in CPU-bound I/O. Although emulated share semantics can offer the same API as copy semantics, the lower integrity guarantees may require changes in applications. We expect that few or no changes would be necessary in multi-programmed or distributed applications that already lock and checkpoint data.

Contrary to what might have been expected, we did not find system-allocated semantics to have any consistent advantage relative to application-allocated semantics, even for devices with pooled input buffering. In fact, we found that, for such devices, application-aligned, application-allocated semantics offer essentially the same performance as system-allocated semantics, but less restrictions: Applications can't choose the alignment or layout of their input buffers, but at least retain access to their output buffers. Because system-allocated semantics may require either substantial rework of pre-existing applications written for copy semantics or application-level copies that negate performance improvements, we believe that emulated copy and emulated share semantics – our two optimized application-allocated semantics – offer more practical and general approaches for I/O performance improvement.

Acknowledgements

We thank Prashant Chandra and Todd Mummert for their help in experimental set-up, and the anonymous referees and our OSDI paper "shepherd", Willy Zwaenepoel, for their many helpful suggestions.

References

[1] J. Barrera III. "A fast Mach network IPC implementation", in *Proc. USENIX Mach Symp.*, USENIX, Nov. 1991, pp. 1-11.

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

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

[4] G. Buzzard, D. Jacobson, M. Mackey, S. Marovitch and J. Wilkes. "An implementation of the Hamlyn sender-managed interface architecture", in *Proc. OSDI'96*, USENIX, Oct. 1996.

[5] D. Cheriton and W. Zwaenepoel. "The Distributed V Kernel and its Performance for Diskless Workstations", in *Proc. 9th SOSP*, ACM, Oct. 1983, pp. 129-140.

[6] D. Clark and D. Tennenhouse. "Architectural considerations for a new generation of protocols", in *Proc. SIGCOMM'90*, ACM, Sept. 1990, pp. 200-208.

[7] E. Cooper, P. Steenkiste, R. Sansom and B. Zill. "Protocol Implementation on the Nectar Communication Processor", in *Proc. SIGCOMM'90*, ACM, Sept. 1990, pp. 135-143.

[8] C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards and J. Lumley. "Afterburner", in *IEEE Network*, July 1993, pp. 36-43.

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

[10] P. Druschel, L. Peterson and B. Davie. "Experience with a High-Speed Network Adaptor: A Software Perspective", in *Proc. SIGCOMM'94*, ACM, Aug. 1994, pp. 2-13.

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

[12] K. Kleinpaste, P. Steenkiste and B. Zill. "Software Support for Outboard Buffering and Checksumming", in *Proc. SIGCOMM'95*, ACM, Aug. 1995, pp. 87-98.

[13] C. Kosak, D. Eckhardt, T. Mummert, P. Steenkiste and A. Fischer. "Buffer Management and Flow Control in the Credit Net ATM Host Interface", in *Proc. 20th Conf. Local Computer Networks*, IEEE, Oct. 1995, pp. 370-378.

[14] 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.

[15] T. Mummert, C. Kosak, P. Steenkiste and A. Fischer. "Fine Grain Parallel Communication on General Purpose LANs", in *Proc. 10th Intl. Conf. Supercomputing*, ACM, 1996, pp. 341-349.

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

[17] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black,W. Bolosky and J. Chew. "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures", in *Proc. 2nd ASPLOS*, ACM, Oct. 1987, pp. 31-39.

[18] P. Steenkiste, B. Zill, H. Kung, S. Schlick, J. Hughes, R. Kowalski and J. Mullaney. "A Host Interface Architecture for High-Speed Networks", in *Proc. 4th IFIP Conf. High Performance Networks*, IFIP, Dec. 1992, pp. A3 1-16.

[19] T. Stricker, J. Stichnoth, D. O'Hallaron and S. Hinrichs. "Decoupling Synchronization and Data Transfer in Message Passing Systems of Parallel Computers", in *Proc. Intl. Conf. Supercomputing*, ACM, July 1995, pp. 1-10.

[20] T. von Eicken, A. Basu, V. Buch and W. Vogels. "U-Net: A User-Level Network Interface for Parallel and Distributed Computing", in *Proc. 15th SOSP*, ACM, Dec. 1995, pp. 40-53.

[21] C. Thekkath, H. Levy and E. Lazowska. "Separating Data and Control Transfer in Distributed Operating Systems", in *Proc. 6th ASPLOS*, ACM, Oct. 1994, pp. 2-11.

[22] S.-Y. Tzou and D. Anderson. "The Performance of Message-passing using Restricted Virtual Memory Remapping", in *Software – Practice and Experience*, vol. 21(3), March 1991, pp. 251-267.