Copy Emulation in Checksummed, Multiple-Packet Communication

José Carlos Brustoloni and Peter Steenkiste
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
jcb@cs.cmu.edu, prs@cs.cmu.edu

Abstract

Data copying can be a bottleneck in end-to-end communication over high-speed networks. Emulated copy is an alternative I/O data passing scheme that preserves the API and integrity guarantees of copying but avoids the latter using VM manipulations – transient output copy-on-write (TCOW), input alignment, *and* page swapping*. We characterize and evaluate the support necessary in network adapters for emulated copy in checksummed, multiple-packet communication. Our experiments on an ATM network show that: (1) Emulated copy gives performance better than that of copying even without hardware checksumming support; (2) TCOW improves multiple-packet output performance without any hardware support or changes in applications; (3) Page swapping provides additional similar improvements on multiple-packet input if there is input alignment, which requires either hardware support* (early-demultiplexed/system-aligned buffering) *or changes in applications* (pooled/application-aligned buffering)*; and (4) The performance of application-aligned buffering is largely unaffected by* header/data splitting*, a common optimization. We propose a new optimization,* buffer snap-off*, that extends system-aligned buffering to the general case of arbitrary, unmatched data transfer and application input buffer lengths.*

1 Introduction

Workstation operating systems, such as Unix [13], typically perform physical input/output (I/O) only on system buffers. The system copies data from application buffers to system buffers before output and conversely from system buffers to application buffers after input. Copying decouples applications from physical I/O and gives applications total control over the layout, allocation, and deallocation of their I/O buffers.

Despite the flexibility that it affords, copying has the drawback of burdening host memory with read and write accesses for each I/O data word. Unfortunately, workstation main memory bandwidth has been and is expected to continue increasing more slowly than the point-to-point bandwidth of communication networks [10]. This has made copying between system and application buffers a bottleneck [16] in end-to-end communication over high-speed networks. ATM, for example, is commercially available at transmission rates of 622 Mbps, capable of challenging the memory system of even top-of-the-line workstations.

We recently proposed a novel I/O data passing scheme, *emulated copy*, that uses virtual memory (VM) manipulations rather than copying to pass data between applications and operating system [3]. Because VM manipulations do not actually touch data, they are relatively insensitive to low main memory bandwidth. Emulated copy uses *transient output copy-on-write* (TCOW) to avoid copying in the common output case, where the application does not overwrite its output buffers while output is taking place. On input, *input alignment* and *page swapping* pass data without copying and without corrupting application buffers with incomplete or erroneous data. Contrary to previous copy avoidance schemes, emulated copy preserves the application programming interface (API) and integrity guarantees of copying and does not restrict application buffer location, alignment, or length.

This paper investigates whether emulated copy gives performance better than that of copying in networking applications where data needs to be checksummed and fragmented into packets. Both checksumming and fragmentation may reduce the performance advantage of emulated copy. Checksumming can be integrated with copying at very little cost [6], whereas a similar integration is not possible with emulated copy. Fragmentation may make input alignment difficult because data is in general received in packets of arbitrary length, each containing header and trailer that do not contain application data and that therefore need to be stripped.

The impact of checksumming can be eliminated if the network adapter itself checksums packets. We verify experimentally, however, that emulated copy gives performance better than that of copying even without such hardware support. Because our VM manipulations are very cheap, the additional cost of reading the data for checksumming gives a total cost that is still much less than that of copying, which requires reading and writing the data.

Likewise, fragmentation hurdles can be re-

To appear in *Proc. INFOCOM'97*, IEEE, April 1997, Kobe, Japan. 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.

duced by hardware support in the network adapter. *Header/data splitting*, a feature found in several current adapters, enables page swapping if packet header lengths are fixed, application input buffers are page-aligned, and both application input buffers and packet data have lengths that are multiples of the page size. We generalize these conditions in what we call *application-aligned* buffering, where applications lay out their input buffers according to the *preferred alignment and length* informed by the system. We show experimentally that emulated copy with application-aligned buffering gives performance that is much better than that of copying and is largely unaffected by hardware support (header/data splitting).

Moreover, we demonstrate experimentally that *early demultiplexing*, a feature in some form present in most ATM adapters, can be exploited in *system-aligned* buffering to input data into system buffers with the same alignment and length as the corresponding application buffers. This enables page swapping even if application input buffers have arbitrary lay-out, provided that their lengths match those of the corresponding (single or multiple-packet) data transfers.

We contribute a novel optimization, *buffer snap-off*, that extends system-aligned buffering to the general case of arbitrary, unmatched data transfer and application input buffer lengths.

The rest of this paper is organized as follows. The next section describes in greater detail the emulated copy VM manipulations. Section 3 discusses network adapter support for input alignment and checksumming. Section 4 highlights desirable characteristics of the application programming interface. Section 5 presents our experimental results. Section 6 discusses related work, and Section 7 gives our conclusions.

2 Emulated copy

Applications that use I/O interfaces such as BSD Unix *sockets* [13], which copy data between application and system buffers, expect the system to maintain a certain *buffering semantics*. For a copy avoidance scheme to be compatible with these applications, it must implement the same semantics precisely [3]. In this section, we describe how our scheme, *emulated copy*, avoids data copying while fully preserving copy semantics.

Copying from application to system buffers when an application triggers an output operation guarantees that the application cannot affect output data by overwriting its output buffers during output. Overwriting may happen because the application reuses its buffers for subsequent computation.

Emulated copy more efficiently achieves the same output behavior by using a novel VM manipulation, TCOW, instead of copying, when data is longer than a configurable threshold (between 0 and the page size). TCOW keeps counts of how many times each physical page is referenced in pending output operations. If the output count of a physical page is zero before a given output operation, TCOW removes write permissions from the page table entries of all VM mappings of that page. Consequently, VM hardware will cause any application that attempts to overwrite that page to receive a page fault. TCOW recovers from that exception, if the output count is still non-zero, by copying the contents of the output page to a new page and replacing the output page by this new page in the application buffer VM mapping; or if the output count is zero, by simply reinstating write permissions in the respective page table entry. Because copy-on-write only happens while output is taking place, we call this scheme *transient*. In the common case of applications that do not overwrite output buffers during output, no copying happens at all.

TCOW relies on *I/O-deferred page deallocation* to guarantee that output pages cannot be freed during output. Pages can normally be freed by explicit deallocation by the application, application termination, or pageout; reuse of a page with outstanding output following one of these events might corrupt output data. I/O-deferred page deallocation prevents such corruption by modifying the system's page deallocation routine to refrain from placing pages with non-zero I/O reference count in the list of free pages, whence they might be reused. The I/O subsystem frees each such page when the last I/O operation referencing the page completes.

On input, copying from system buffers to application buffers only after successful completion of the operation guarantees that application buffers will not be corrupted by incomplete or erroneous input data. To be fully compatible, a copy avoidance scheme must make the same integrity guarantees.

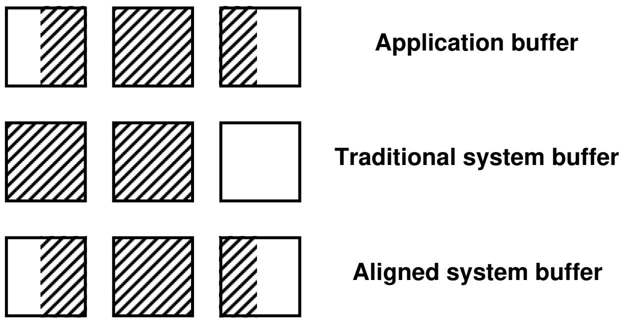

Figure 1: Input alignment makes it possible to swap pages.

Emulated copy uses *input alignment* to minimize copying on input without loss of application buffer integrity. Emulated copy inputs data into system buffers with the same page offsets and lengths as the corresponding application buffers, as shown in Figure 1. When the input completes, emulated copy transfers data to the application by *swapping pages* between application and system buffers (that is, it replaces physical pages previously mapped to the application buffer with the pages from the system buffer). Page swapping is a VM manipulation that typically can be implemented at a fraction of the cost of the equivalent copy.

Emulated copy handles partially filled pages in aligned system buffers as follows. If the data to be transferred is sufficiently short, it is more efficient to simply copy it out to the application buffer. If it is long, however, it costs less to complete the page with corresponding data from the application buffer, by what we call *reverse copy-out*, and swap pages. Assuming that the cost of swapping is s, the cost of copying is cl, where l is the length of the data copied, the data lengths in the pages in the application and system buffers are u and k, respectively, $m = \min(u, k)$, and p is the page size, copying to the application buffer costs less than swapping pages if:

$$cm < s + c(p - m) + c(k - m)$$

where $(p - m)$ is the length of data that needs to be copied from the application to the system page before swapping, and $(k - m)$ is the length of system data, beyond the end of application data, that needs to be saved for subsequent input. If $u = k$, this relation reduces to:

$$u < t = \frac{\frac{s}{c} + p}{2}$$

where t is a configurable threshold in the system, usually only slightly greater than $p/2$.

Applications can achieve input alignment by using *application-aligned buffering*. We define an interface through which applications can inquire the *preferred buffer alignment and length* for input from any given I/O module (e.g., a protocol stack). Using this interface, applications can lay out their input buffers accordingly and thereby avoid input copying whether or not there is hardware support for input alignment.

Application-aligned buffering is similar to the optimizations of IRIX, HP-UX, and Solaris [5], which replace copying by page swapping when application input buffers are page-aligned and of length multiple of the page size. Unlike those optimizations, however, application-aligned buffering does not require hardware support, as explained in Sections 3.1 and 3.2.

Unfortunately, application-aligned buffering cannot support application input buffers of arbitrary alignment and length, such as those of many existing applications written for I/O interfaces with copy semantics. To support this general case, emulated copy uses an alternate technique, *system-aligned buffering*. This optimization presupposes that applications inform the alignment and length of their input buffers to the system *before* physical input, using, for example, the API described in Section 4. The system uses that information to input data into system buffers with the same alignment and length. This customization requires hardware support, as discussed in Sections 3.3 and 3.4.

3 Network adapter support

This section describes network adapter features relevant to input alignment and checksumming, and highlights software techniques that compensate the eventual lack of hardware support.

3.1 Pooled in-host input buffering

Network adapters with *pooled* in-host input buffering receive packets into buffers allocated from a list of *buffer segments* shared among multiple *reception ports*. Buffer segments reside in host memory and are specified by (pointer, length) pairs (scatter/gather). The host enqueues segments at the tail of the buffer segment list when preparing for reception, and the adapter dequeues segments from the head of the list when receiving packets. Reception ports are the objects from which applications receive data; they are identified in packet headers and can be, for example, ATM virtual connections or TCP ports.

In hosts with physically addressed DMA, the length of each segment is such that the segment does not cross page boundaries; commonly, each segment is a page. In hosts with virtually addressed DMA, the length of each segment is normally equal to the network maximum transmission unit (MTU); to simplify emulated copy, all segments are page-aligned. In either case, at the end of a packet the adapter drops the remainder of the current segment, so that storage of the next packet starts at the next segment.

Pooled input buffering does not support input alignment. The latter is possible, however, if applications use application-aligned buffering. If: (1) Packet headers and trailers have fixed lengths; (2) Data lengths of all packets in a data transfer have a known length L ($\leq$ MTU minus header and trailer lengths), except possibly for the last packet, which may have data length of at most L; and (3) Either: (a) For each data transfer, the length of application input buffering matches that of the data transfer; or (b) All data transfers and application input buffers have lengths that are integral multiples of L, although not necessarily equal; then copying can be avoided by setting the preferred alignment equal to the header length and the preferred length equal to L.

3.2 Header/data splitting

A refinement of pooled input buffering is to have the adapter use separate *header* and *data/trailer* buffer segment lists, shared among multiple reception ports, as depicted in Figure 2. The initial portion of a packet is stored in a segment allocated from the header list, while the remainder of the packet is stored in segments allocated from the data/trailer list [12]. If segments in the header list have length equal to that of packet headers, this scheme normalizes the preferred buffer alignment to page boundaries.

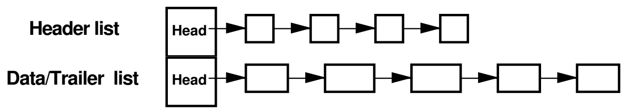

Figure 2: Separate header and data/trailer buffer lists

Solaris zero-copy TCP [5] uses this technique with L set to the largest integral multiple of the page size not greater than the MTU minus TCP/IP header length. Input copying is thereby avoided if packet headers do not contain options (so that TCP/IP headers have fixed length), application input buffers are page-aligned, and both application input buffers and data transfers have lengths that are integral multiples of the page size.

3.3 Early demultiplexing

Network adapters with *early demultiplexed* input buffering maintain *separate* buffer segment lists per reception port. When receiving the beginning of a new packet, the adapter demultiplexes the packet header to determine which input buffer list to use for reception of the packet [1], as shown in Figure 3. [1]

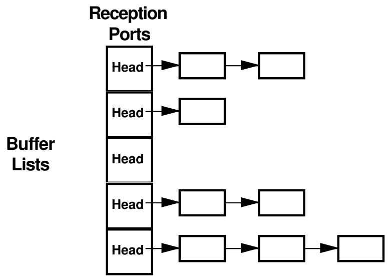

Figure 3: Early demultiplexing

Early demultiplexing supports system-aligned buffering: The system can enqueue, at the buffer segment list of each reception port, segments with alignment and length matching those of the respective application buffers. Each segment may have different alignment or length, and consequently, it must be possible for the host to *reclaim* (dequeue) specific buffer segments in case of exceptions such as timeout of the corresponding input request.

If packet header, data, and trailer lengths are predictable by the receiving system (not necessarily fixed), proper alignment of input data can be maintained by interspersing header and trailer segments at the appropriate points in the buffer list of the respective reception port. Header and trailer segments allow the remaining data segments to end up concatenated in virtual memory in spite of data packetization.

The length of data transfers may be known by previous arrangement between the end applications. Packet header, data, and trailer lengths are then predictable if the header and trailer lengths are fixed or easily computable and the protocol used follows some simple deterministic rule, such as sending all packets except possibly the last one with length equal to the MTU. The MTU may be known because it is fixed (for example, in a LAN serving a cluster of workstations) or configurable (for example, using the TCP MSS option [13]). In general, the MTU can be estimated by what we call *MTU tracking*, that is, monitoring the maximum length of packets received at the given reception port. Knowledge of the total data transfer length is necessary to avoid enqueueing more segments than necessary to hold the data.

[1] ATM adapters that reassemble packets from ATM cells must implement at least a primitive form of early demultiplexing, keeping track of the different segments being reassembled per reception port, but possibly allocating segments from a pooled (rather than per-port) buffer segment list.

3.4 Buffer snap-off

Early demultiplexing does not by itself support buffer boundaries. Consequently, it enables system-aligned buffering only if data transfers and application input buffering have matching lengths. We propose a novel technique, *buffer snap-off*, that removes this restriction.

Buffer snap-off assumes that headers and trailers have lengths h and t that are either fixed or computable on-the-fly by the adapter. The adapter maintains separate header/trailer and data buffer lists. The header/trailer list can be shared among multiple reception ports and can be further split into separate header and trailer lists. Segments in the header/trailer list have length greater than or equal to the largest allowed h and t. The first h bytes of a packet are stored in a segment allocated from the header/trailer list, the last t bytes are stored in another segment from the header/trailer list, and the remaining data is stored in segments allocated from the data buffer list (*header/data/trailer splitting*). The adapter maintains separate data buffer lists per reception port. Data buffer lists can be represented by hierarchical lists of buffer segments, or more simply by lists of buffer segments, where each segment contains an end-of-buffer (EOB) bit set only at the last segment of the respective buffer. These alternatives are shown in Figure 4.

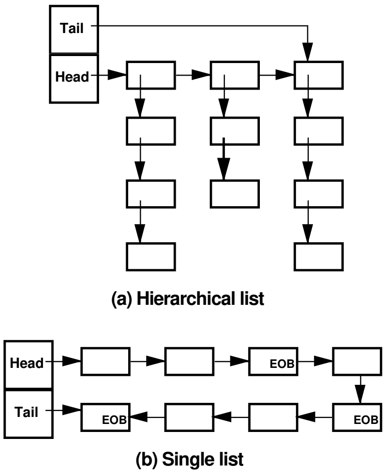

Figure 4: Data buffer representation alternatives

We define two snap-off modes, selectable per reception port: *single-packet*, suitable for datagram service, and *multiple-packet*, especially suitable for byte-stream service, as shown in Figure 5.

In single-packet mode, on end-of-packet, the adapter should dequeue and deallocate the remaining segments of the current data buffer and signal completion of the input operation. Conversely, if the packet data is longer than the current data buffer, the adapter should drop the packet.

In multiple-packet mode, on the contrary, on end-of-packet, the adapter should *snap off* the remainder of the current data buffer segment and place it at

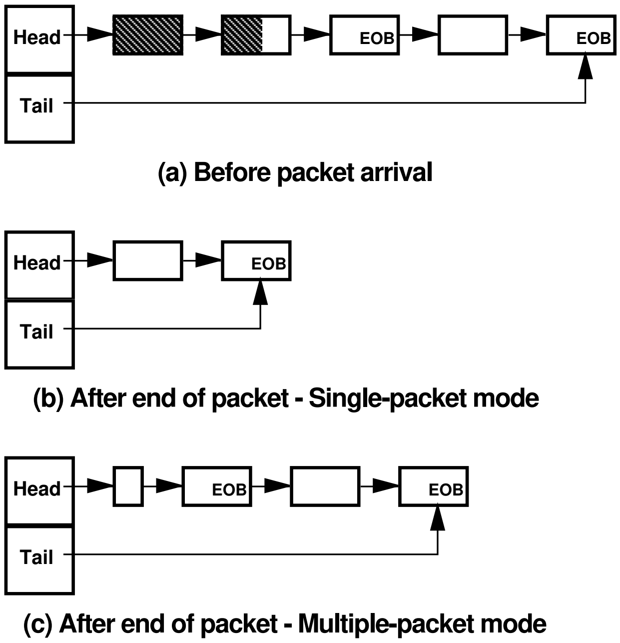

Figure 5: Buffer snap-off. Data from an incoming packet occupies the shaded area in (a). The rest of the buffer is then dequeued (b) or snapped off (c).

the head of the data buffer list of the current reception port. This way, the data of successive packets for the same reception port will be concatenated in the system input buffer, as required, even if packet data lengths are not integral multiples of the page size (as, for example, in maximally-sized Ethernet, FDDI, or ATM AAL5 packets). Input requests in this mode remain pending until the corresponding input data buffering is completely filled. Packets may span multiple buffers; end of the data buffering of an input request simply marks the completion of the request, deferred until successful end of the last packet using it. Out-of-order reception (not possible in ATM) is dealt with as an exception, in software, by copying.

Existing adapters in general do not implement buffer snap-off and instead simply drop the remainder of the current buffer segment at the end of a packet. This behavior jeopardizes copy avoidance, because: (1) In single-packet mode, if the current segment at the end-of-packet is not the last segment in the current buffer, storage for the next packet will start with the alignment of the next segment, which is not the first segment of the next buffer; and (2) In multiple-packet mode, the unfilled remainder of the current segment creates a gap in system buffers with respect to application buffers, causing data in subsequent segments to become misaligned and therefore require copying.

If the adapter supports early demultiplexing with header/data splitting per reception port, headers have fixed length, and there are no trailers (as in TCP/IP), host software may be able to approximate snap-off by dequeuing remaining segments of the current buffer (single-packet mode) or enqueuing the remainder segment at the head of the buffer list (multiple-packet mode). This software approximation does not work, however, if packets are received back-to-back, which doesn't give the host opportunity to fix the buffer list before arrival of the next packet.

3.5 Checksumming

Network adapter hardware may checksum data while it is transferred (by DMA or other block move mechanism) into or out of host memory, thereby relieving the host processor (and memory) of this task. Note that in emulated copy the adapter inputs data not into application buffers but rather into aligned system buffers. Consequently, if the checksum is incorrect, the previous contents of application buffers can be left intact, thus fully preserving copy semantics.

If checksums are transmitted in packet headers, as is the case in TCP/IP, hardware output checksumming requires packets to be staged in outboard buffers. Such staging allows the checksum field in the header to be corrected after data transfer from host memory and before transmission to the network. Given that output data is copied to outboard buffers, TCOW is unnecessary. On the other hand, outboard buffering may require a non-trivial amount of memory in network adapters. A low-cost alternative may be to use hardware checksumming only for *input*, which does not require outboard memory, and for output use TCOW and checksumming by the host processor.

4 Application programming interface

Emulated copy can be applied to any I/O interface with copy semantics. Certain features of the API can, however, expand the number of cases where copy avoidance occurs. This section highlights such features, as implemented in Genie, a new, optimized I/O framework [3].

To enable system-aligned buffering, the application must inform the alignment and length of its input buffers to the system *before* physical input of the data. Genie allows applications to invoke I/O operations synchronously or asynchronously. The latter case can be used to *pre-post* buffers that are eventually going to be needed for input. The API includes calls to poll or abort asynchronous operations.

In I/O interfaces such as sockets, input operations complete when any non-null amount of data is available, not necessarily filling the application buffer. If the application expects to receive data in the remainder of an input buffer, it can submit a new input operation. This does not generate a race condition with respect to the arrival of a new packet because reception occurs asynchronously in system buffers, without any alignment constraints. This decoupling is made possible by copying.

The same approach may not be appropriate for emulated copy. When an application invokes an input operation, Genie *prepares* system buffers with the correct alignment. When the input operation completes and control is transferred back to the application, Genie *disposes* the system buffers by page swapping and loses any further alignment or length information.

If the application had to submit a new input operation for the remainder of an input buffer, there would be a time interval during which system-aligned buffering would not be prepared for newly arrived packets.

The solution used in Genie is to allow *partial delivery* of data to the application. When the application polls the status of a pending multiple-packet input operation, Genie swaps pages filled with previously received packets, but keeps the remainder of the input buffering prepared for further input. In multiple-packet mode, an input operation completes only when the corresponding input buffering is completely full, unless the application or a timeout aborts the operation before then.

5 Evaluation on an ATM network

This section compares experimental latencies for checksummed or multiple-packet communication between applications running on different hosts connected by the CreditNet ATM network at 155 Mbps using copying or emulated copy I/O data passing at the end hosts. Applications accessed the network via an implementation of Genie on the NetBSD 1.1 operating system. We used Micron P166 (166 MHz Pentium) computers with 32 MB of main memory. All measurements presented here are the averages of five runs (except where noted) on otherwise idle hosts and network; variances were small. Immediately before the measurements, the benchmark application zero-filled all buffers and communicated 4 KB in each direction so as to warm the instruction and data caches. The benchmark application then measured round-trip times, using the Pentium on-chip cycle counter, for data lengths equal to increasing multiples of the page size (4 KB). Reported latencies are equal to the round-trip times divided by two. In this section, all graphs have legends listed in the same order as the corresponding curves.

The CreditNet network adapter supports pooled and early demultiplexed input buffering and header/data splitting [12]. It has only limited support for data/trailer splitting (it automatically removes ATM AAL5 trailers (depadding) only if the PDU length modulo 48 is not greater than 40) and does not support buffer snap-off or checksumming.

Figure 6 shows the single-packet latency for early demultiplexed communication using copying or emulated copy and with or without IP checksumming by the host processor. Application buffers were page-aligned. The throughput for single 60 KB datagrams with or without checksumming was, using copying, 71 or 78 Mbps, and using emulated copy, 95 or 119 Mbps, respectively. The penalty of checksumming is lower when using copying because copying brings data into the cache, which emulated copy doesn't. The penalty of checksumming with copying can be further reduced on output by integrating the checksum and copy in a single step [6], so that each data word is read and written only once (in Figure 6, copying and checksumming were performed separately, so that each data word was read twice and written once). The same optimization would be incorrect for input because the application buffer would be overwritten with erroneous data when the checksum fails, violating the integrity guarantees of copy semantics.

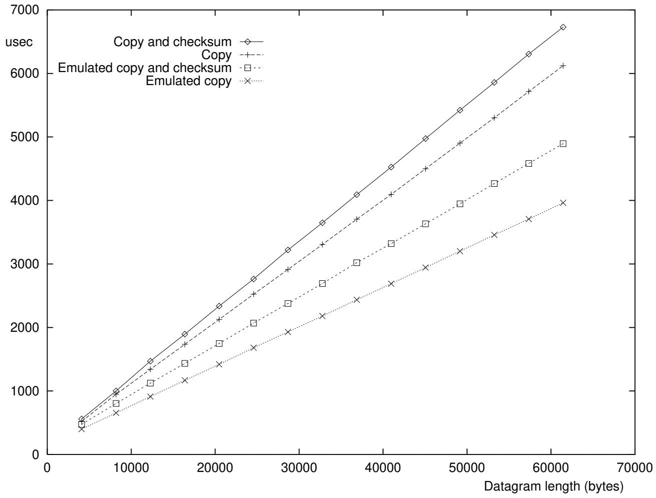

Figure 6: End-to-end latency for single packets. Emulated copy is cheaper than copying even if the host processor also has to read the data for checksumming.

The curves for "copy" and "copy and checksum" in the figure can be taken, respectively, as the lower and upper bound of the latency using integrated copy and checksum. The figure shows that the cost of emulated copy plus checksumming is substantially less than that lower bound. This is because VM manipulations are very cheap and checksumming only requires one memory access, to read data, while copying requires two memory accesses, to read and write data.

Figure 7 shows multiple-packet latencies with pooled input buffering, eight-byte packet headers (SNAP), and no transport-level flow control. All curves other than the top two use emulated copy. The bottom two curves indicate that, if the application input buffer is aligned according to the preferred offset and the MTU is equal to 8 KB (two pages), the throughput for 60 KB with or without header/data splitting is 122 or 119 Mbps, respectively. Performance is slightly worse without header/data splitting because of the overhead of reverse copy-out at the beginning and copy-out at the end of each packet. The throughput for 60 KB decreases to 93 Mbps if the MTU used is 9180 bytes (the default MTU for IP over ATM [17]) and the application input buffer starts with the preferred offset but is contiguous. Only the data in the first two pages ends up aligned, and the rest must be copied. (Note that the first two pages of *each* packet could be aligned and swapped if the application used non-contiguous input buffers, with each segment conforming to the preferred offset and length.) The throughput for 60 KB decreases further to 90 Mbps if the application input buffer already starts misaligned, so that all data needs to be copied, even if the MTU is a multiple of the page size (8 KB). Note that even in this last case, emulated copy still gives much better performance than does copying. This advantage is due to TCOW's

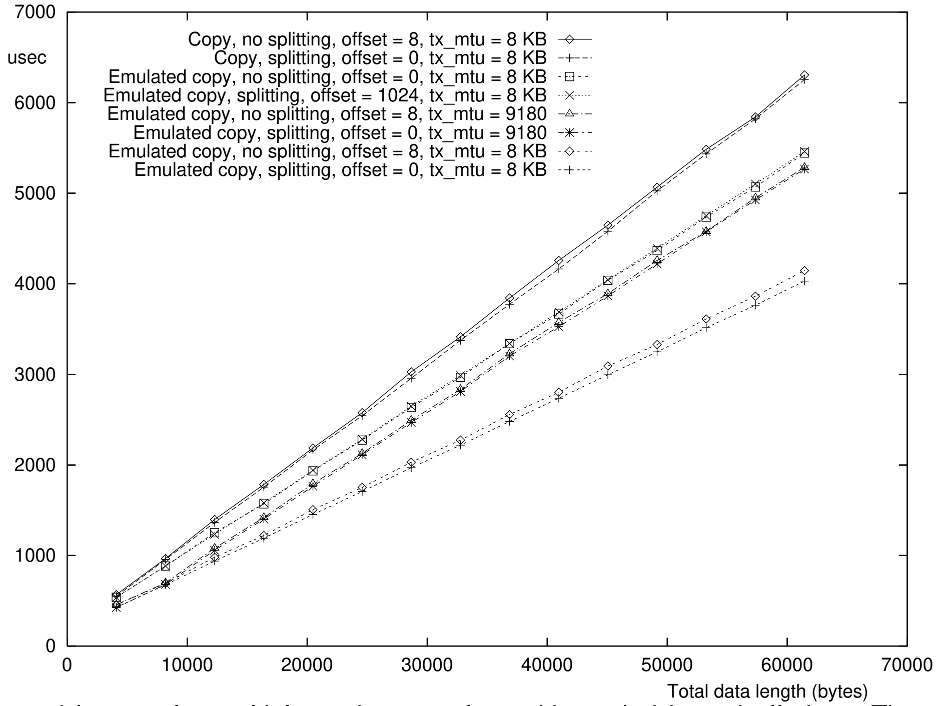

Figure 7: End-to-end latency for multiple-packet transfers with pooled input buffering. The performance of application-aligned buffering is not substantially affected by the presence or absence of header/data splitting.

copy avoidance on output. The top two curves indicate that the throughput for 60 KB using copying is 78 Mbps. We found that this value is relatively insensitive to header/data splitting, application input buffer alignment, and MTU.

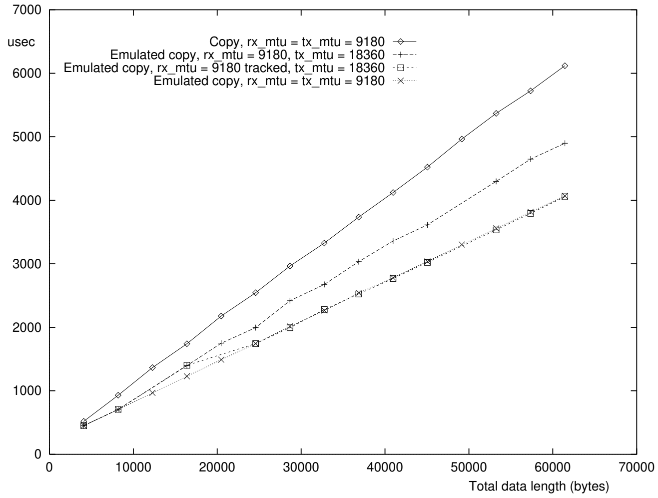

Figure 8: End-to-end latency for multiple-packet transfers with early demultiplexing. Copying is avoided regardless of application input buffer alignment and MTU. The MTU can be quickly estimated by MTU tracking.

Figure 8 shows multiple-packet latencies with early demultiplexing, eight-byte packet headers, and no transport-level flow control. Application input buffers were offset from a page boundary by 1 KB. The throughput for 60 KB with copying was 80 Mbps and is relatively insensitive to the MTU. The throughput for 60 KB with emulated copy was 121 Mbps if the actual MTU (tx_mtu) was the same as that expected by the receiver (rx_mtu), 9180 bytes. The actual MTU can vary widely without affecting performance, as long as it is the same as that expected by the receiver. If the actual MTU is 18360, twice as long as that expected by the receiver, the throughput for 60 KB falls to 100 Mbps, because data beyond the first two pages ends up not aligned in system buffers and needs to be copied to the application buffer for delivery. The figure also shows latencies for the first run using MTU tracking. After data longer than the actual MTU has been received, MTU estimation converges to the actual value, and performance becomes the same as when the MTU is fixed and known.

Buffer snap-off extends the conditions under which system-aligned buffering occurs, but does not improve the performance in situations where system-aligned buffering already occurs. Consequently, we can estimate the performance with buffer snap-off to be roughly equal to the bottom line in Figure 8.

6 Related work

TCOW differs from the Mach copy-on-write (COW) scheme [18] by being transient (Mach COW is long-term) and page-level (Mach COW operates on whole regions and requires more VM data structures to be updated). Another page-level COW scheme is that of Solaris, but it differs in important aspects from TCOW. According to the description in [5], the Solaris scheme eagerly reinstates write permissions on output completion. Eager reinstatement can be incorrect if the status of the region containing the page changes after output start. The application may, for example, change region protection to read-only, deallocate the region, or fork another process that inherits the region by copy-on-write. Additionally, if more than one output operation references the same page, output operations other than the first one either require data copying or may output corrupted data after the first operation completes, as the application may then overwrite the page. Finally, [5] does not describe whether or how the Solaris scheme prevents the page from being freed during output if the application explicitly deallocates the page or terminates.

System-allocated semantics such as *move* [3] eliminate copying without any special hardware support. The *system* then determines the location and layout of application input buffers; system buffers can be simply mapped to application address spaces, without alignment constraints. On output, the system automatically deallocates application output buffers. Unfortunately, system-allocated semantics require an API different from that of copy and other *application-allocated* semantics [3]. The API makes system-allocated semantics incompatible with many existing applications, for example those written for the socket interface. New applications that require a particular data layout, e.g. those that use arrays, or that need to retain access to output buffers after output, may not be able to use system-allocated semantics without user-level copying, which negates any performance improvements. We have experimentally demonstrated that system-allocated semantics have performance that is similar to that of application-aligned buffering [3]. The latter option will be preferable in many cases, because it is compatible with the usual application-allocated API and imposes less restrictions on applications (which can't choose data layout, but at least retain access to output buffers after output).

Outboard buffering can eliminate copying while preserving copy semantics. Data in outboard buffers can be transferred from the adapter to application buffers, rather than to system buffers, eliminating copying on input without early demultiplexing or buffer snap-off and regardless of application buffer alignment and length and packet lengths. However, outboard buffering may also add latency, complexity, and cost to the adapter. Moreover, depending on when data is checksummed, outboard buffering may result in semantics other than copy and introduce considerable jitter in transport-level acknowledgements. The Medusa [2] and Afterburner [8] adapters, for example, checksum while transferring data from outboard buffers to application buffers. Consequently, the latter may be corrupted with incorrect data, making the semantics *share* (weak integrity) rather than copy (strong integrity) [3]. Additionally, those adapters require packet acknowledgements to be delayed until the receiving process is scheduled to run *and* the application decides to receive the data. This can introduce considerable variability in round-trip times (RTT), which may interfere with transport-level algorithms that depend on precise RTT estimates. By checksumming while receiving data from the network into outboard buffers, the Gigabit Nectar WCAB [11] adapter is able to both guarantee copy semantics and maintain the usual decoupling between transport-level acknowledgements and application processing.

The Nectar protocol processor [7] and, more recently, the Charisma [15] adapter allow applications to directly map outboard buffers, in a truly zero-copy arrangement that may be beneficial if the host processor can access outboard memory as efficiently as it can access host memory. Unfortunately, adapters normally connect to the host via an I/O bus. In most current architectures, this makes outboard memory uncacheable and imposes relatively high arbitration overheads for word-sized accesses by the host. Consequently, applications that actually access the I/O data (not the case in [15]) may very well run more slowly with the data outboard than in a more conventional arrangement, where I/O data is transferred to and from host memory. Compatibility may also be a problem because the zero-copy arrangement does not provide copy semantics. Additionally, outboard memory is normally not pageable, making the scheme particularly susceptible to resource management problems when confronted with malicious or buggy applications that hog outboard buffers.

Application device channels [9] allow applications to safely enqueue I/O requests directly in the adapter, bypassing the operating system. Such bypassing may reduce latency, but also requires application buffers to be located in unpageable host memory and to have share or weak move [3] rather than copy semantics.

Carter and Zwaenepoel [4] show how driver software can approximate early demultiplexing for *blast* protocols on adapters with pooled input buffering. Blast protocols transmit long data in *blasts*, that is, multiple back-to-back packets, the last of which is acknowledged by the receiver. In [4], the driver, on receiving the first packet in a blast of more than two packets, replaces data segments at the head of the buffer segment list by segments that point directly to the receiving application's memory beyond the data of the first two packets in the blast. Data of the first two packets is copied, while the rest of the data is input directly from the network into the corresponding application memory, if the blast is not interrupted by packets destined to other reception ports. Interruptions by an unrelated packet require copying the data of that and the remaining blast packets to their correct destinations. [4] gives experimental evidence that such interruptions are very rare in practice; however, they constitute a potential security leak. Additionally, the semantics implemented is share rather than copy. The security leak could be eliminated and the semantics could be made fully compatible with copying by making replacement segments point to aligned system buffers, and swapping pages as in our scheme.

O'Malley et al. [14] modify Carter and Zwaenepoel's scheme so that each blast is preceded by a *control packet* describing the blast, and a 150 μsec delay. The delay allows the receiving host to insert, at the head of the buffer segment list, appropriate header and data segments for all packets in the blast, so that blast data ends up concatenated in a system buffer. After receiving the last packet of the blast, the system buffer is mapped to a new region in the application address space. This scheme eliminates the security leak, as in our proposed modification, but implements move rather than copy semantics.

7 Conclusions

We described a set of VM manipulations – TCOW, input alignment, and page swapping – that make it possible to fully emulate copy semantics on output or input without actually copying data. We characterized network adapter features needed for copy emulation in checksummed, multiple-packet communication, and reported the performance on an adapter with partial support.

Our results show that emulated copy gives performance better than that of copying even if the adapter does not provide checksumming and the host processor must read the data to perform that task.

Our measurements indicate that TCOW eliminates output copying and substantially reduces end-to-end latency without any special hardware support or changes in applications.

Our experiments demonstrate that page swapping provides similar additional performance gains when there is input alignment. Input alignment requires either hardware support (early-demultiplexed/system-aligned buffering) or changes in applications (pooled/application-aligned buffering). We found that header/data splitting, an optimization found in several current adapters, does not provide an attractive cost/benefit ratio: Its generalization, application-aligned buffering, achieves essentially the same performance without hardware support. In contrast, early demultiplexing, a feature that is present at least in primitive form in most ATM cards, enables input alignment under substantially more general conditions. Buffer snap-off, a novel hardware optimization, extends system-aligned buffering to the general case of arbitrary, unmatched byte stream or datagram lengths and application input buffering alignment and length.

References

[1] M. Bailey, B. Gopal, M. Pagels and L. Peterson. "PATHFINDER: A Pattern-Based Packet Classifier", in *Proc. OSDI'94*, USENIX, Nov. 1994, pp. 115-124.

[2] D. Banks and M. Prudence. "A High-Performance Architecture for a PA-RISC Workstation", in *J. Selected Areas in Communications*, IEEE, vol. 11, no. 2, Feb. 1993, pp. 191-202.

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

[4] J. Carter and W. Zwaenepoel. "Optimistic Implementation of Bulk Data Transfer Protocols", in *Proc. SIGMETRICS'89*, ACM, May 1989, pp. 61-69.

[5] H. J. Chu. "Zero-Copy TCP in Solaris", in *Proc. Winter Tech. Conf.*, USENIX, Jan. 1996.

[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, B. Zill. "Protocol Implementation on the Nectar Communication Processor", in *Proc. SIGCOMM'90*, ACM, Sept. 1990, pp. 135-144.

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

[10] J. Hennessy and D. Patterson. "Computer Architecture: A Quantitative Approach", 2nd. ed., Morgan Kaufmann Pub., San Francisco, CA, 1996.

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

[12] C. Kosak, D. Eckhardt, T. Mummert, P. Steenkiste and Allan 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.

[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] S. O'Malley, M. Abbot, N. Hutchinson and L. Peterson. "A Transparent Blast Facility", in *Internetworking: Research and Experience*, vol. 1, 1990, pp. 57-75.

[15] B. Murphy, S. Zeadally and C. Adams. "An Analysis of Process and Memory Models to Support High-Speed Networking in a UNIX Environment", in *Proc. Winter Tech. Conf.*, USENIX, Jan. 1996.

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

[17] R. Atkinson. "Default IP MTU for use over ATM AAL5". RFC 1626, Naval Research Laboratory, May 1994.

[18] 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. ASPLOS-II*, ACM, Oct. 1987, pp. 31-39.