Design, Implementation, and Evaluation of a Single-Copy Protocol Stack

Peter Steenkiste
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
e-mail: steenkiste@cs.cmu.edu
Tel: 412 268 3261

January 20, 1998

Abstract

Data copying and checksumming are the most expensive operations on hosts performing high-bandwidth network I/O over a high-speed network. Under some conditions, outboard buffering and checksumming can eliminate accesses to the data, thus making communication less expensive and faster. One of the scenarios in which outboard buffering and checksumming pays off is the common case of applications accessing the network using the Berkeley sockets interface and the Internet protocol stack. In this paper we describe the host software for a host interface with outboard buffering and checksumming support. The platform used is DEC Alpha workstations with a Turbochannel I/O bus and running the DEC OSF/1 operating system. Our implementation does not only achieve "single copy" communication for applications that use sockets, but it also interoperates efficiently with in-kernel applications and other network devices. Measurements show that for large reads and writes the single-copy path through the stack is five to seven times more efficient than the traditional implementation for large writes. We also present a detailed analysis of the measurements using a simple I/O model.

Keywords: zero-copy protocol implementation, TCP/IP, outboard buffering, communication API.

To Appear in "Software - Practice and Experience"

1 Introduction

For bulk data transfer over high-speed networks, the sending and receiving hosts typically form the bottleneck, and it is important to minimize the communication overhead to achieve high application-level throughput. The communication cost can be broken up in per-packet and per-byte costs. The per-packet cost can be optimized [1, 2], and for large packets, this overhead is amortized over a lot of data. However, the cost of per-byte operations such data copying or checksumming cost is not reduced by increasing the packet size. Moreover, the per-byte cost depends strongly on the memory bandwidth, which over time has not increased as quickly as CPU speed. As a result, it is often the per-byte costs that make high speed communication over networks expensive and that ultimately limit throughput as network bandwidth increases.

The per-byte overhead can be optimized by minimizing the number of times the data is accessed by the host CPU on its path through the network interface. The ideal scenario is a single-copy architecture in which the data is copied exactly once. For example for transmit, the data is copied from where it was placed by the application directly to the network adapter, and the checksum is calculated during that copy. In contrast, most host interfaces in use today copy the data two or three times before it reaches the network. For Application Programming Interfaces (APIs) with copy semantics (e.g. sockets), the single-copy architecture might require outboard buffering and checksum support [3], and several projects have proposed or implemented network adapters that include these features [4, 5, 6, 25]. Alternatively, it is possible to eliminate copying by using APIs with other semantics [8, 9, 10, 11], or by using copy semantics in a restricted way [12]. A detailed study of the impact of API semantics on data passing performance can be found in [13].

Many applications use BSD sockets and the Internet protocols for communication, and as a result it is worthwhile to look at how they can be supported efficiently. In this paper we describe and evaluate the software support needed assuming a network adapter that provides support for outboard buffering and checksumming. We also present a detailed performance study, which allows us to quantify both the speedups and additional overheads associated with our single-copy approach. Our implementation is for a HIPPI network and DEC workstations using the Turbochannel I/O bus [14] and running the DEC OSF/1 operating system. The interface was used in the Nectar gigabit testbed.

The remainder of the paper is organized as follows. We first describe the host interface architecture and the implementation of a single copy protocol stack. We then present performance measurements and their analysis, and we conclude with a discussion on related work and a paper summary.

2 Host-Network Interface Architecture

We discuss how we reduce communication overhead for applications that use BSD sockets and Internet protocols, and we describe a network adapter architecture that supports these optimizations.

2.1 Optimizing communication

The time spent on sending and receiving data is distributed over several operations including per-packet operations (protocol processing, interrupt handling, buffer management), and per-byte op-

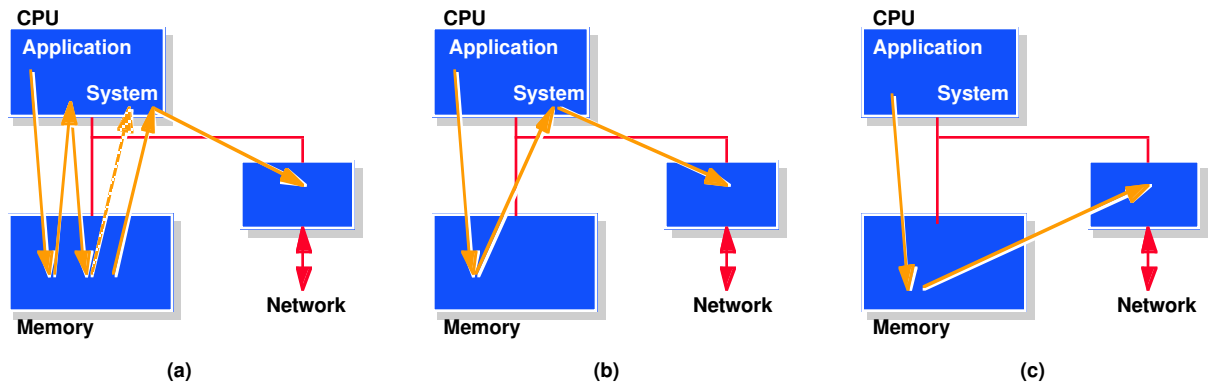

Figure 1: Dataflow in (a) a traditional network interface, (b) a network interface with outboard buffering and (c) a network interface with outboard buffering, DMA, and checksumming support

erations (copying data, checksumming). As networks get faster, data copying and checksumming become the dominating overheads, both because the other overheads are amortized over larger packets and because per-byte operations stress a critical resource (the memory bus), so we first look at per-byte operations. Figure 1(a) shows the dataflow when sending a message using a traditional host interface; receives follow the inverse path. The data is first copied from the user address space to kernel buffers. This copy is needed to implement the *copy semantics* of the socket interface: when the send call returns, the application can safely overwrite the data. The next per-byte operation is the TCP or UDP checksum calculation (dashed line), and finally, the data is copied to the network device. This adds up to a total of five bus transfers for every word sent. In some cases there is an additional CPU copy to move the data between "system buffers" and "device buffers", which results in two more bus transfers.

We can reduce the number of bus transfers by moving the system buffers that are used to buffer the data outboard, as is shown in Figure 1(b). The checksum is calculated while the data is copied. The number of data transfers has been reduced to two. This interface corresponds to the "WITLESS" interface proposed by Van Jacobson [4, 6]. Figure 1(c) shows how the number of data transfers can be further reduced to one by using DMA for the data transfer between host memory and the buffers on the network adapter. This is the minimum number with the socket interface. Checksumming is still done while copying the data, i.e. checksumming is done in hardware. Besides reducing the load on the bus, DMA has the advantage that it allows the use of burst transfers. This is necessary to get good throughput on today's high-speed I/O busses.

The reduction in bus transfers comes however at a cost. Network adapters with outbuffering are more complex than traditional "flow-through" adapters and may need a substantial amount of memory. The amount of buffer space needed for a TCP connection as retransmit buffer space on transmit and as a receive buffer on receive, is the product of its sustained transmit rate and the roundtrip time. Especially in a WAN environment, this can add up to a lot of memory that is dedicated to the network device. Since accesses to the memory are mostly sequential, DRAM or VRAM can be used.

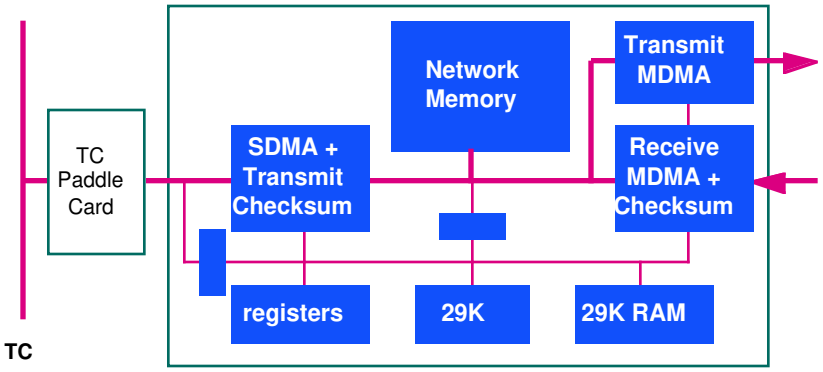

Figure 2: Workstation CAB adapter architecture

2.2 Gigabit Nectar network adapter

The Gigabit Nectar CAB (Communication Acceleration Block) provides outboard buffering, DMA, and checksumming support (Figure 1c). It was built using off-the-shelf components by NSC for DEC workstations with a Turbochannel I/O bus [14]. The workstation CAB sits in a separate box that connects to a Turbochannel paddle card through a cable. The paddle card uses the TcIA chip [15] to communicate over the Turbochannel.

Figure 2 shows a block diagram of the Gigabit Nectar CAB. At the core of the CAB is a "network memory" that stores incoming and outgoing packets (Figure 2). Network memory is implemented using DRAM and it feeds three DMA engines: one system DMA engine (SDMA) for data transfers to and from host memory, and two media DMA engines (MDMA) to move data to and from the network. All DMA engines can operate at the same time, using time sharing to access network memory. There is hardware support for the checksum calculation on both the SDMA path (outgoing packets) and MDMA path (incoming packets) [5]. The CAB is controlled using a 29K microprocessor, which has its own memory bus, separate from the network memory bus. The two busses are connected, allowing the host to access the register file, for example to write requests and read responses, and allowing the 29K to access network memory, for example to store the checksum in a packet header.

The 29K microprocessor is responsible for the interaction with the host, the management of buffers in network memory, the operation of the DMA engines, and media access control [5]. The microcode managing the SDMA engine is the bottleneck in the host interface. The reason is that the translation of host DMA requests into SDMA commands is subject to a large number of constraints. These constraints are imposed by the Turbochannel (e.g. Turbochannel burst should not cross 2KByte boundaries in host memory), TcIA (e.g. avoiding overflow of the 15 word combined data/command FIFO) and by the structure of network memory (e.g. starting addresses have to be a multiple of four words). The 29K meets these constraints by breaking up the pointer-length pairs provided by the host into multiple shorter bursts that can be handled correctly by the DMA hardware, and, in some cases, by copying a small number of words directly between host and network memory. These operations are expensive and limit throughput: while the CAB hardware was designed for bandwidths up to 300 Mbit/second, the microcode limits throughput to about 200 Mbit/second for 64KByte packets.

2.3 Host view

From the viewpoint of the host system software, the CAB is a large bank of memory accompanied by a means for transferring data into and out of that memory. The transmit half of the CAB also provides a set of commands for issuing media operations using data in the memory, while the receive side provides notification that new data has arrived in the memory from the media.

Several features of the CAB place constraints on the structure of the host software. For example, to insure full bandwidth to the media, packets must start on a page boundary in CAB memory. This, together with the fact that checksum calculation for Internet packet transmissions is performed during the transfer into CAB memory, dictates that individual packets are fully formed when they are transferred to the CAB.

To illustrate how the CAB can be used to support single-copy communication, we present a walk-through of a typical send and receive (with copy semantics). To handle a send, the system first examines the size of the message and the state of the TCP connection, and determines how many packets will be needed on the media. It then creates the headers in kernel space and issues SDMA requests to the CAB, one per packet. The CAB transfers the data from the user's address space to the CAB network memory using DMA. In most cases, i.e. if the TCP window is open, an MDMA request to perform the actual media transfer can be issued at the same time, freeing the processor from any further involvement with individual packets. Only the final packet's SDMA request needs to be flagged to interrupt the host upon completion, so that the user process can be scheduled. No interrupt is needed to flag the end of MDMA of TCP packets, since the TCP acknowledgment will confirm that the data was sent.

Upon receiving a packet from the network, the CAB automatically DMAs the first L words of the packet into *auto-DMA buffers*, i.e. preallocated buffers in host memory. The value L can be selected by the host. The CAB then interrupts the host, which performs protocol processing. For TCP and UDP, only the packet's header needs to be examined as the data checksum has already been calculated by the hardware. The packet is then logically queued for the appropriate user process. A user receive is handled by issuing one or more SDMA operations to copy the data out of network memory into user buffers. The last SDMA operation is flagged to generate an interrupt upon completion so that the user process can be scheduled.

3 Single-copy protocol stack

We discuss an implementation of a single-copy stack in an OSF/1 v2.0 kernel running on a DEC Alpha 3000/400.

3.1 Implementation in a BSD stack

To make the most efficient use of the CAB, data should be transferred directly from user space to CAB memory and vice-versa. This model is different from that found in Berkeley Unix operating systems, where data is channeled through the system's network buffer pool [16]. The difference in the models, together with the restriction that data in CAB memory should be formatted into complete packets, means that decisions about partitioning of user data into packets must be made

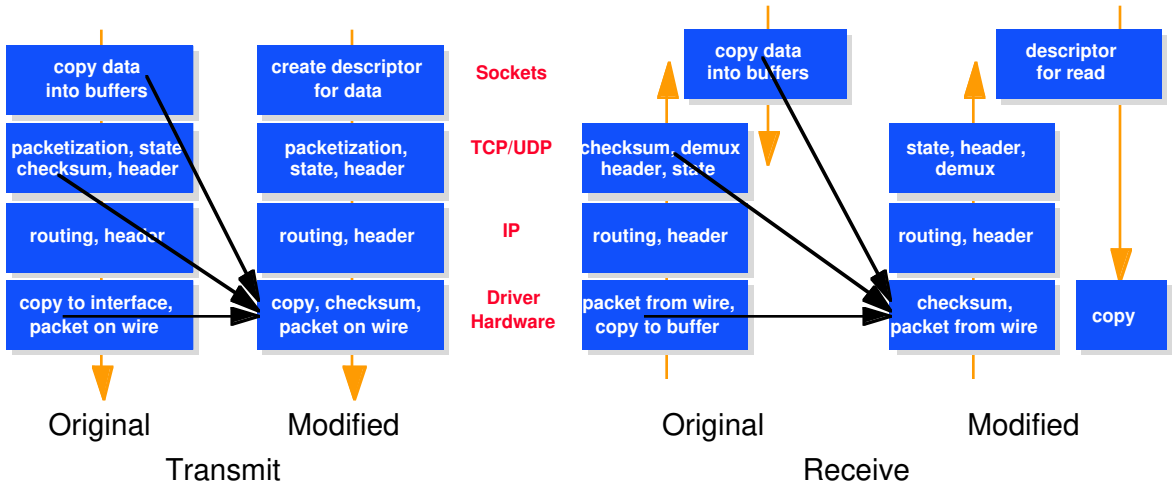

Figure 3: Software architecture

before the data is transferred out of user space. This requires that some of the functionality in the "layered" protocol stack be moved.

There are many ways of doing this reorganization, but the least disruptive solution is to maintain the existing protocol stack structure and to pass data descriptors representing the data through the stack instead of kernel buffers holding the data. Formatting operations on data, i.e. packetization, are done "symbolically" on the descriptor and not by copying the data. All data-touching operations are combined into a single operation that is performed in the driver. Figure 3 shows the control flow (grey arrows) through an original and a modified stack: the black arrows show how the per-byte operations are moved to the driver and hardware. To move the checksum calculation, information about the checksum calculation is associated with the data descriptor for the packet, thus allowing the checksum to be set up or used in the transport layer, but calculated in the driver. To support this software organization, the network device driver has to provide routines to transfer packets between host and network memory, *copy_in* and *copy_out*, besides the traditional *input* and *output* routines.

The implementation of our single-copy stack is driven by three important design choices: single stack versus multiple stack implementation, implementation of the data descriptors, and checksum handling. We discuss these design issues next, and we also look at the implications on the host software of the use of DMA and of the data alignment constraints imposed by the CAB.

3.2 Single versus multiple stacks

Hosts typically also have interfaces other than the CAB, e.g. other networks, loopback interface, ... There are two very different approaches to dealing with multiple interfaces. First, add a new "single copy" protocol stack to the system (Figure 4a). The new stack operates in parallel with the original stack, and the appropriate stack is selected based on which interface is used for communication. Alternatively, one can modify the existing stack to support all interfaces, including "single copy" and traditional interfaces (Figure 4b). With the first strategy, the single-copy architecture can be implemented as new stack, but it has the disadvantage that two parallel stacks have to be

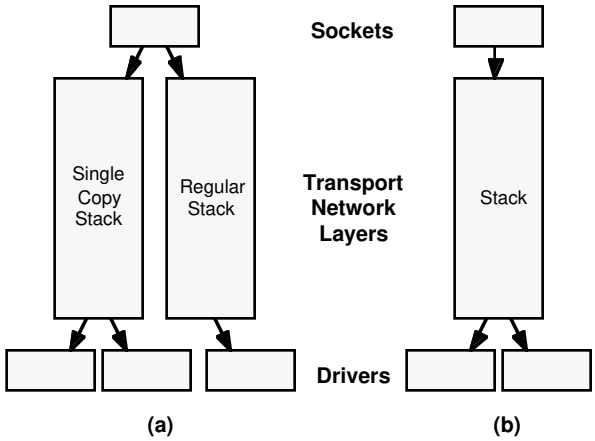

Figure 4: Single versus multiple stacks

maintained. In the second case, we expect more complicated changes to an existing stack, but only a single stack has to be maintained.

We opted for a single stack implementation for the following two functional reasons:

- On transmit, it is difficult to determine reliably what interface will be used at the socket level, since the interface selection is done in the network layer. Although it is possible to determine what interface will be used, certainly for connection-oriented protocols such at TCP, it is possible for the interface that is used for a given destination to change over time. This would require a "stack switch", which would be complicated to implement.

- It is not practical to keep the two stacks separate. Consider routing packets between interfaces that use different stacks: routing relies on a single stack, at least up to the network layer. A slightly more subtle example has to do with optimizing data transfers. Copy avoidance only pays off for large transfers; for small transfers, copying and potentially coalescing the data is simpler and more efficient. Since a single connection has to support both short and long reads/writes, the "single copy" stack will have to support both a single-copy and a traditional multiple-copy path, i.e. building a single stack makes more sense.

In the following sections we describe the "single copy" path through the stack in more detail. We address the issue of interoperability with other network devices and applications in the next section.

3.3 Data descriptors

With a single stack implementation, data can flow through the stack in three different formats:

- data in kernel buffers, i.e. traditional mbufs.

- data in user space: this format is used in the transmit stack, before the data is transferred to the adapter. It is also used on receive to describe the memory area specified in a read call.

- data in outboard buffers: this data shows up both in the transmit stack (e.g. retransmit buffers) and in the receive stack (large packets coming in through the CAB). Because of the characteristics of the CAB, this data is formated as packets.

In our implementation, all data formats are represented by mbufs, with the latter two formats relying on the *external mbuf* mechanism that was added to 4.3 BSD. External mbufs make it possible to store data in buffers that are managed separately from the regular pool of kernel mbufs. We created two new mbuf types: one to represent data stored in the user's address space (M_UIO mbuf) and another to represent data stored in network memory (M_WCAB mbuf). Both mbuf types include a new data structure called *uiowCABhdr* to store information about the checksum location and the task that issued the read or write (used for notification). The M_WCAB mbuf also includes a *wCAB* structure with an identifier for the packet in network memory, the packet checksum and information on how much of the outboard data is valid. The M_UIO mbuf type also includes a *uio* structure that describes the read/write memory area in the user's address space.

On transmit, the format of the data changes as it travels through the stack. After a write, the data travels down the stack as regular or M_UIO mbufs, depending on the data size, and the mbuf type is changed to M_WCAB after the data has been copied outboard. The M_WCAB mbufs can be retransmitted by TCP and are freed when the data is acknowledged. On receive, data travels up the stack as regular or M_WCAB mbufs, depending on whether the packet fits in an auto-DMA buffer or not, and it is freed after the data is copied into user space.

An important result of working inside the mbuf framework is that most of the changes related to copy optimization are hidden inside the macros and functions that operate on mbufs, and few changes had to be made to the transport and network layers in the stack. The changes to the stack were limited to:

- The socket code was changed to create M_UIO mbufs (transmit) and to recognize M_WCAB mbufs (receive).
- In the TCP layer, the code that copies a packet's worth of data into an mbuf chain to be handed to the driver was replaced by code that searches the transmit queue for a block of data at a specific offset. Note that this search routine has to operate on a list that includes mbufs of different types, including M_WCAB mbufs in the case of retransmit.
- The checksum routines were modified to use outboard checksumming (next section).

An alternative to using the existing mbuf framework would have been to defined a new data structure to represent the different data formats. However, this would have required more substantial changes to the code, and the data structure would have ended up being fairly similar to external mbufs. In some ways, our M_UIO mbufs are similar to pbufs [17]. Since we rely heavily on the external mbuf format, adding a single-copy path to a stack without external mbufs (i.e. pre-4.3 mbufs) would have been more complicated.

3.4 Checksumming

The CAB has hardware to calculate the IP checksum for both incoming and outgoing packets, but it does not "speak" IP, i.e. IP headers, including the IP header checksum, have to be provided by the host.

On transmit, the host routine that normally calculates the checksum instead collects information that will be needed by the CAB to do the checksum calculation in hardware. The checksum routine places the offset of the checksum field and the number of (four byte) words S that should be skipped

by the checksum engine in the M_UIO mbuf that describes the packet data. It also places a seed that represents the checksum of the first S words of the packet in the checksum field of the packet. This allows the CAB to set up the DMA engine, and to combine the calculated checksum with the seed in the header to obtain the complete checksum after the SDMA operation is done.

The host sets the value of S to the length of all the headers (HIPPI and IP), i.e. the hardware calculates the checksum over the user data, and the host is responsible for the fields in the header (the TCP header and pseudo-header [18]). While there are many other choices for S, this selection has the advantage that it works correctly when retransmitting data. Specifically, when retransmitting, the host provides a new header, with a new checksum seed. The CAB DMAs the new header on top of the old header and adds in the checksum of the body of the packet, which it had saved from when the packet was transferred the first time.

On receive, the CAB hardware starts calculating the checksum in at a fixed offset in the packet. The offset is selectable by the host software and is set to 20 words in our implementation, i.e. the HIPPI and IP header are skipped. The checksum is passed up the stack together with the first 176 words of the packet (data size of the mbuf). The checksum calculation routine of TCP/UDP adjusts the checksum calculated by the CAB by adding/subtracting the fields of the TCP header and pseudo-header, and then compares it with the checksum in the header.

A final detail of the checksum support is related to the difference between TCP and UDP checksums. The hardware always calculates a "TCP checksum", i.e. a ones-complement add. For UDP packets, this creates the risk that the checksum will add up to 0, in which case it would have to be changed to 0xffff to avoid confusion with the case that no checksum is specified. In practice this is not an issue, since a ones-complement add can only be 0 if all terms are 0, which will never happen for a checksum since some of the header fields included are guaranteed to be non-zero (e.g. the address fields).

3.5 DMA

DMA is used on many devices to achieve high throughput over burst-oriented busses. In general, the device driver is responsible for the management of the DMA engine and for setting up transfers between kernel buffers and the device. What makes the DMA required for the single-copy stack different is that the DMA engine transfers data directly between the application's address space and the device. In this section we discuss how this influences the host software.

3.5.1 Pinning and address translation

The use of DMA requires pinning/unpinning of pages and virtual-physical address translation. Both of these tasks are normally performed by the device driver, and when DMAing to or from kernel buffers, this means that the driver performs VM operations on pages in its address space. In contrast, when the DMA is to or from an application address space, the driver (or kernel) has to perform VM operations on pages in a different address space. Unfortunately, operating systems do not support this in a uniform way. We briefly describe implementations for the Mach 3.0 microkernel [19] and DEC OSF/1, and compare them in terms of complexity and performance.

In Mach, the required VM operations can be performed on a different address by specifying the appropriate Mach port [20] for the target address space. In the case of a Mach 3.0 microkernel [19], this allows the device driver, which resides in the Unix server, to perform the address translation

and pinning of the pages in the application address space, i.e. DMA support is localized in the device driver.

In DEC OSF/1, as in most Unix systems, performing the VM operations on application pages requires a context that is only present when that application process is scheduled. Since packet transmission is often triggered by kernel events (e.g. arrival of a window update), the application context will in general not be present. Restoring the application context and performing the VM operations in the device driver is simply not practical. Our solution is to have the socket layer, which is always executed in the application context, map the data to be sent into kernel space, so that it can be handled in an appropriate way by the driver. This mapping is performed incrementally, one "socket buffer" worth at a time, as data is handed down to the transport layer. The equivalent problem does not exist on receive, since the *copy_in* function is initiated by the socket layer, which has the appropriate context.

Implementing the mapping in the socket layer (OSF/1) is less attractive than performing it in the driver (Mach). Not only is it more complicated since the support for the DMA device is no longer localized to the driver, but it also adds mapping overhead when using other devices, i.e. if the stack is not used in single-copy mode. Note that the mapping would also be needed if PIO were used on the CAB, since the PIO operation in the driver would also need access to the application address space.

Pinning and mapping increases the cost of each DMA operation. For applications that reuse the same set of buffers repeatedly, this overhead can be avoided by keeping the buffers pinned and mapped so the overhead is amortized over several IO operations; buffers can be unpinned lazily, thus limiting the number of pages that an application can have pinned at one time. Even though the API still has copy semantics, performance will be best if a limited set of buffers are used for communication, e.g. the usage of the API has share semantics (e.g [9]).

3.5.2 Synchronization

The copy semantics of the socket interfaces requires that the application is only allowed to continue after a copy has been made of the data (transmit), or after the incoming data is available (receive). Application wakeup requires synchronization between the driver, which controls the DMA, and the socket layer, which controls the application. This synchronization is implemented by including in the M_UIO mbuf the socket buffer pointer, which can be used by the driver to wake up the application; this is done as part of the end-of-DMA interrupt handling.

Since data is DMAed one packet at a time, large writes will be broken up in a number of DMAs. To make sure the process is woken up at the right time, the socket layer keeps track of the outstanding DMAs using a UIO counter. It is incremented every time a packet is separated from the M_UIO mbuf describing the data in user space, and it is decremented every time the driver receives an end-of-DMA notification. Note that an interrupt is only needed for the last DMA of a write; all other end-of-DMA notifications can be handled at that time. The receive side has a similar structure: the UIO count is used to keep track of outstanding DMA requests, and the application process is rescheduled after all DMA operations of incoming packets have finished.

Once the host has issued a DMA operation to the CAB, it cannot be be canceled. This means that when a read or write operation is interrupted (say because of an alarm signal), the process will only be allowed to restart after all outstanding DMA operations have completed.

3.5.3 Optimization based on packet size

The tradeoff between programmed IO and DMA are well understood (e.g. [21]). PIO has typically a higher per-byte overhead while DMA has a higher per-transfer overhead; as a result PIO is typically more efficient for short transfers, and DMA for longer transfers. Since the CAB only supports DMA, this is not an issue. However, there is a similar tradeoff: data can be transferred using DMA between user space and the device, or using a memory-memory copy followed by DMA between kernel space and the device. The former is more efficient for large transfers, while the latter should be used for short transfers. On transmit, the socket layer can optimize the transfer by formatting the data either as a M_UIO or as a regular mbuf, depending on the size of the write. On receive, the size of the auto-DMA buffer determines the size of the smallest packet for which copy-avoidance is used.

3.6 Alignment issues

The CAB SDMA engine places several restrictions on the source and destination addresses in host and CAB memory, lengths, and burst size combinations that can be supported. While most of these restrictions can be worked around by the host and CAB software, the restriction that starting addresses in host memory have to be (32 bit) word aligned, cannot be hidden from the user, since we are DMAing directly in and out of the user's address space. Note that this is a fairly common restriction for I/O devices. As a result, the single-copy path can only be used for read/write requests that are word aligned, and the traditional path is used for unaligned accesses, i.e. data is copied through kernel buffers.

Note that on transmit, it is possible to align the bulk of the data by first sending a short packet. For example, if a write starts at an address that is a 16 bit boundary (but not a 32 bit boundary), we can send a first packet of 16 bits, which will have to be copied, but the remainder of the data can be DMAed since it is now word aligned. This might pay off for very large writes, although we have not implemented this optimization. This flexibility does not exist on receive.

The net effect is that unaligned reads and writes will run slower, but that is the case for many operations in computer systems. Performance conscious programmers should do aligned reads and writes. Since compilers and *malloc()* always align the data structures they allocate, we expect unaligned reads and writes to be rare.

4 Interoperability with other devices and in-kernel applications

The discussion up to this point has focussed on applications using sockets to transmit or receive through the CAB. Two more important cases have to be supported:

- Besides regular user applications, many in-kernel applications make use of the network. They include IO intensive applications such as file servers, and applications with low bandwidth requirements such as ICMP. They use TCP or UDP over IP, or raw IP.
- Hosts often have network interfaces other than the CAB, and these interfaces typically do not support single-copy communication.

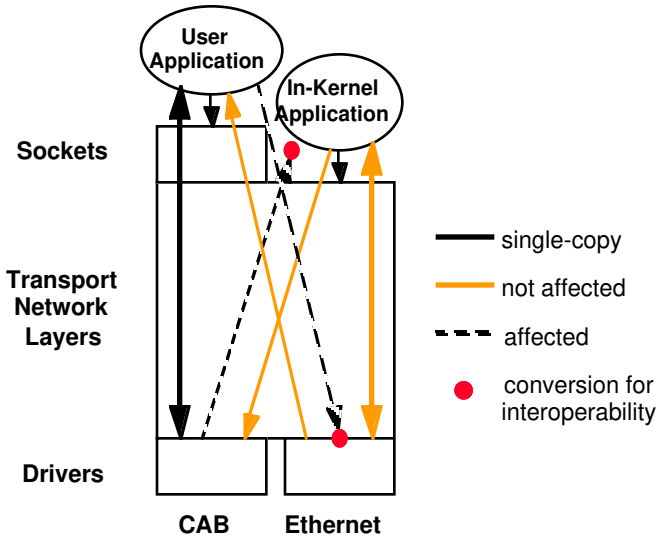

Figure 5: Paths through the protocol stack

Figure 5 shows the different paths through the protocol stack. The single-copy path described in this paper is shown in black. Given the nature of the changes to the protocol stack described in the previous section, in-kernel applications communicating through existing interfaces should not be affected (thick grey arrow in Figure 5): both the applications and the drivers directly exchange mbufs with the protocol stack, which still handles "regular" mbufs.

However, in-kernel applications communicating through the CAB and applications using the socket interface communicating through existing interface might create problems (thin arrows in Figure 5), both as a result of the new mbuf types and as a result of the asynchronous DMA copies. Making these paths work should not require modifying in-kernel applications or drivers for existing devices. Not only would this significantly increase the amount of code that has to be modified and maintained, but in many cases it is impossible because applications are distributed in binary form only, i.e. even recompilation is not an option.

We look at the transmit and receive paths for both in-kernel applications and drivers for existing devices, resulting in four scenarios:

- *transmit through existing devices*: packets containing M_UIO mbufs representing data in the user address space will not be recognized by the device driver. The solution is to add a thin layer of code at the entry point to the driver to convert M_UIO mbufs into regular mbufs using a memory-memory copy. Note that this does not increase the number of copies compared with a regular stack: a copy has merely been delayed.

- *receive from existing device*: existing devices will only create regular mbufs, which are still supported by the modified stack, i.e. this case is handled automatically.

- *in-kernel applications transmit*: data is specified as a chain of regular (or cluster) mbufs, which are still handled by the modified stack. One potential problem is that the structure of the mbufs might not be acceptable to the CAB driver, specifically, they might not be able to accommodate the larger mbuf headers that are needed when the conversion to WCAB mbufs takes place. This is handled by simply checking the format, and changing it if needed. Note

that since the communication API of in-kernel applications often has share semantics, with the mbufs being the shared buffers, we automatically get single-copy communication with the CAB: the data is copied once using DMA, and the checksum is calculated during that checksum.

- *in-kernel applications receive*: WCAB mbufs, which are passed up the stack by the CAB driver, will not be handled correctly by existing in-kernel applications. The solution is obvious: convert them to regular mbufs before they enter the application. The fact that the copy has to be done using DMA, i.e. asynchronously, adds some complexity since the application has to resynchronize with the driver when the DMA terminates. An additional concern is packet reordering, specifically the reordering of (long) packets that require DMA and (short) packets that do not require DMA. Although maintaining packet order is not a strict requirement (IP does not guarantee in order delivery), frequent reordering of packets could confuse clients or reduce their performance, for example by triggering retransmits.

5 Implementation complexity

The single-copy path was implemented in the DEC OSF/1 v2.0 kernel. The OSF/1 protocol stack is based on Net2 BSD and also supports TCP window scaling [22]. The implementation of the single-copy stack supports user-level and in-kernel applications communicating through Ethernet and user-level applications communicating through the CAB.

The implementation of the single-copy stack in an OSF/1 kernel has three components: a device driver for the CAB device, modifications to support the M_UIO and M_CAB mbuf types, and stubs to deal with in-kernel applications and other network devices. The CAB device driver is similar to that of other network devices, except that it also provides routines to transfer packets between host and network memory, *copy_in* and *copy_out*, besides the traditional *input* and *output* routines. Most of the changes dealing with supporting the new mbuf types are in macros that manipulate mbufs or mbuf chains. Ignoring the CAB driver, which is about 4500 lines of code, a total of about 1900 lines were changed or added to the protocol stack (counted using diff, i.e. includes comments).

Most of the complexity of the code changes was in three areas. First, since the same mbuf traverses the entire stack, the question of who is in charge of the mbuf at any given point is critical for correctness. This is not an issue in the traditional stack since mbufs are copied, so each entity has its own copy and can operate on it and free it without concern of what happens elsewhere in the code. The second source of complexity is that M_WCAB mbufs have both external and local data associated with them, adding complexity to the management of pointer, length and offset fields. The local data consists of the headers on the transmit (and retransmit) path, and the auto-DMA area on the receive path. Finally, correct handling of the kernel uio structures is complex since accesses are destructive, and the structure is accessed several times, e.g. every time when a data block is pinned or unpinned, which happens in socket buffer size increments.

6 Performance evaluation

We compare the performance of a single-copy stack with that of an unmodified stack. We look at both throughput and latency. The systems used are DEC Alpha 3000/400 workstations with 64

MByte of memory. For all experiments, the Maximum Transmission Unit (MTU) is 64 KBytes and the TCP window size is set to 512 KBytes.

6.1 Throughput measurements

6.1.1 Design of experiments

What protocol stack is used affects the efficiency of the communication, i.e. how much overhead does communication introduce. Depending on the specific adapter and host, the stack might also have an impact on the throughput. For this reason, we will use both *throughput* and *system utilization* as performance measures. Throughput is measured using *ttcp*, which measures user process to user process throughput. Estimating the utilization accurately is more difficult. The CPU utilization of *ttcp* is not a good indicator, since certain communication overheads (e.g. ACK handling and any transmits it triggers) are not charged to the process for which the action is performed (*ttcp* in our case), but to the process that happens to be active when the interrupt takes place. To solve this problem, we ran a compute-bound low-priority process called *util* at the same time as ttcp on both the sending and the receiving node. The *util* program is started up and killed by *ttcp* and uses any cycles that are not used by *ttcp*, i.e. it can be viewed as a user program doing useful work while communication is taking place. When calculating the utilization due to communication, we charge any system time accumulated by *util* to *ttcp*. The presence of the *util* program has some impact on the throughput, but it is limited to a few percent.

When using this method, we discovered that the sum of the CPU times charged to *util* and *ttcp* does not add up to the elapsed time of the tests. Consistently, about 7-8% of the time is unaccounted for. This time is likely spent in various background processes, including the idle process, and we will assume that it should be charged proportionally to *util* and *ttcp*, so our estimate for CPU utilization to support communication is calculated as:

$$utilization = \frac{tccp(user) + ttcp(sys) + util(sys)}{tccp(user) + ttcp(sys) + util(sys) + util(user)}$$

6.1.2 Experimental results

Figures 6(a) and (b) show the throughput and utilization as a function of the read/write size. The utilization results are for the sender, but the results on the receiver are similar. Figure 6(a) includes the throughput for raw HIPPI reads and writes. The raw HIPPI throughput test generates well-formed packets that can be handled efficiently by the microcode, so the raw HIPPI results represent the highest throughput one can expect for a given packet size. For the measurements using the modified stack we always used the single-copy path (i.e. it does not fall back to "regular" path for small writes as described in Section "DMA") and we do not coalesce the M_UIO mbufs generated by multiple writes into a single packet.

The throughput results show that for small writes (16 KByte and less) the original stack provides a higher throughput than the single-copy stack. This is a result both of a higher communication efficiency (see below) and the lack of coalescing in the single-copy stack. For larger reads and writes the modified stack has a higher throughput: the peak throughput is 176 Mbit/second for the modified stack versus 122 Mbit/second for the unmodified stack. The utilization measurements

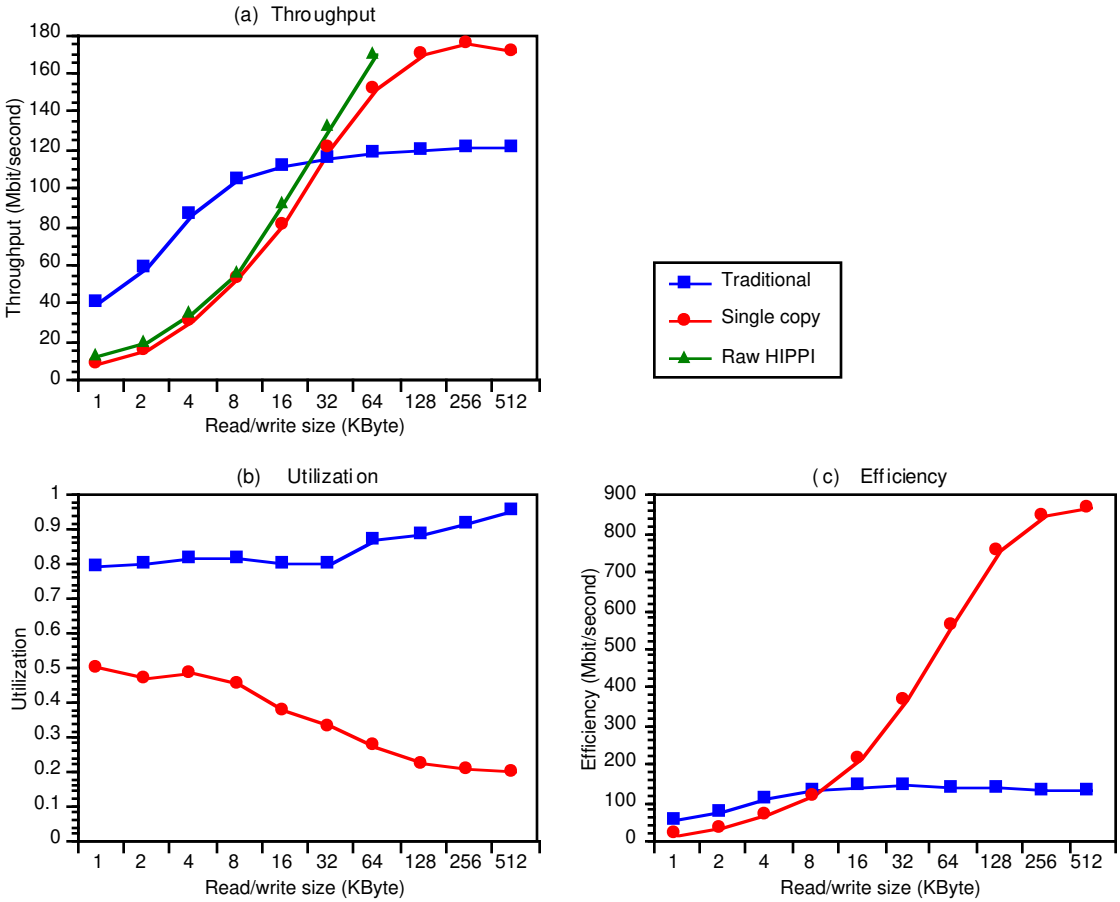

Figure 6: Throughput, utilization and efficiency as a function of read/write size

shows that the modified stack uses fewer CPU cycles to provide the higher throughput. The unmodified stack uses over 90% of the CPU, i.e. data copying and checksumming by the CPU limits throughput, while the modified stack uses as little as 20% of the CPU, i.e. even at 170 Mbit/second there are still many CPU cycles left for use by applications.

To better evaluate the overhead we define the communication *efficiency* as how many Mbit/second of communication can be supported if the full CPU were utilized for communication, i.e. the ratio of the throughput and efficiency graphs in Figure 6(a) and (b). Figure 6(c) shows the efficiency for both implementations of the stack. We see that the single-copy stack is five to seven times more efficient than the unmodified stack for large writes, but less efficient for small writes. The cross over point is between 8 and 16 KByte, indicating that the single-copy stack will pay off, i.e. be more efficient, for writes of 16 KByte and higher. Note that the efficiency is a rough estimate of the communication throughput that the host can sustain, assuming throughput is not limited by the network or the adapter.

Finally, throughput results are often very sensitive to the alignment of the application buffers. This is especially the case when page remapping is used to implement the socket copy semantics. Figure 7 shows the throughput for 128KB reads and writes for different alignments. We took measurements for alignments ranging from 0 to 8192 bytes (the page size) in increments of 64 bytes (measurement points are not shown to make the graph readable). We observe that the throughput

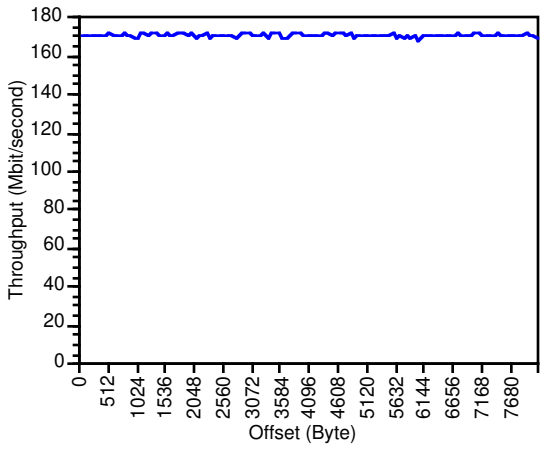

Figure 7: Throughput for different buffer alignments (128 KByte read/write size)

Symbol	Meaning	Unit
C	Communication cost	$\mu second/KByte$
B	Per-Byte cost	$\mu second/KByte$
p	Packet size	$KByte/packet$
P	Per-packet cost	$\mu second/packet$
w	Write size	$KByte/write$
W	Per-write cost	$\mu second/write$

Table 1: Symbols in analysis of communication efficiency

for non-page aligned reads/writes is almost identical to that of aligned reads/writes.

6.1.3 Analysis of the communication efficiency

The measurements reported in the previous section show that for large messages, the single-copy stack is much more efficient than the traditional stack. In this section we would like to validate the claim that this increase in efficiency is the result of copy avoidance, i.e. it is caused by a reduction of the per-byte costs. Ideally we would this by measuring the per-write, per-packet, and per-byte costs in the two stacks, but unfortunately this not practical. While there are some code segments that can easily be classified as "per-byte" or "per-packet", e.g. bcopy and checksumming, a lot of code is hard to classify. Many routines, e.g. buffer management, have both per-packet (or per-write) and per-byte components that are hard to separate. Moreover, functions may sometimes be called in response to per-packet or per-message events in ways that are hard to trace, e.g. events triggered by interrupts. This makes the classification of costs based on low-level tracing impractical, and we will instead estimate the per-byte, per-packet, and per-message costs by fitting a simple model of the communication cost to the efficiency measurements of the previous section. We will also present measurements of the cost of some specific tasks, both as a validity check for the model and to clarify its results.

In order to better understand the performance difference between the traditional and single-

copy stack, we develop a simple model for the cost associated with communication. We can model the overhead of sending data as the sum of costs associated with the handling of writes, packets and bytes. Using the notation explained in Table 1 the cost C is:

$$C = \frac{W}{w} + \frac{P}{p} + B$$

Given the changes we made in the protocol stack, we expect the same per-packet cost in both stacks, but different per-byte and per-write costs. This results in five model parameters: P, W_{Trad}, W_{Single}, B_{Trad}, and B_{Single}. Note that the model parameters do not translate directly into the cost of specific operations in the protocol stack, since many operations have a per-write, per-packet, and per-byte component. For example, we expect the buffer management cost to have different components. Finally, some overheads may not scale in a perfectly linear way over the entire range of write sizes w, so the cost model is only a rough approximation.

We can simplify the model by replacing the packet size p by its actual value. For large writes, the packet size p will usually be the MTU (64KByte in our case) for both the traditional and single copy stack. For smaller writes, the two stacks use different packets sizes. Since we do not coalesce in the single-copy stack, the packet size is the write size: $p = w$. The traditional stack does coalesce, so the packet size may be larger. Although packets will not necessarily always be a full MTU, we will assume for the purposes of the model that they are. As a result of the differences in packet size, we have two flavors of the cost model. A first version with $p = w$ applies to the single-copy stack for write sizes of 64KByte or less. The second version with $p = MTU$ applies to the single-copy stack for write sizes of 64KByte or more, and to the traditional stack. In summary, our cost models are of the following form:

$$C = \alpha + \frac{\beta}{w} \quad \begin{array}{ll} with\ \ \alpha = B_{Trad} + \frac{P}{MTU}\ and\ \beta = W_{Trad} & traditional\ stack \\ with\ \alpha = B_{Single} + \frac{P}{MTU}\ and\ \beta = W_{Single} & single\ copy\ stack;\ w \geq MTU \\ with\ \ \alpha = B_{Single}\ and\ \beta = W_{Single} + P & single\ copy\ stack;\ w \leq MTU. \end{array}$$

The communication cost (in microseconds per Kbyte of data sent) is inverse-proportional to the communication efficiency, so we can do a linear least square fit of the cost equations shown above to (the inverse of) the measurements in Figure 6(c); the variable is the inverse of the write size w. For all three data sets the fit was very good, i.e. an R^2 of 0.96 or better. The least square fit for each data set gives a value for α and β, resulting in six equations for the model variables. Given that we have five model parameters, we have an overconstrained set of equations, i.e. six equations with five variables. There are many ways of dealing with this. The approach we selected is to identify the parameter that is least likely to be constant across the full range of w, and to break it up into multiple parameters. The parameter we selected is the per-byte cost in the single-copy stack. It is dominated by the page manipulation costs, and while this cost is proportional to the write size for large writes, it is constant for small writes. We will approximate this using two per-byte costs for the single-copy stack. We now have six model variables with six equations. The results are shown in Table 2.

Before we analyze the results, we would like to increase our confidence in the model. We do this by comparing the model estimates in Table 2 with direct measurements of some important sources of communication overhead that can easily be classified.

	Per-Write ($\mu second/write$)	Per-Packet ($\mu second/packet$)	Per-Byte ($\mu second/KByte$)
Traditional	103	99	56
Single-copy	391		7.6 (small) 6.8 (large)

Table 2: Estimated costs based on least square fit of model to measurements

Operation	Fixed cost	Per-page cost
Pin	35	29
Unpin	48	3.9
Map	6	4.5

Table 3: Cost in microseconds of virtual memory operations

To try to characterize the per-packet and per-write costs we will use measurements of the communication overhead for small packets, which should have very small per-byte overheads. The time spent in tcp_output, including time in IP and the device driver, is about 130 microseconds. This is mostly per-packet overhead, although there is some per-byte overhead, e.g. buffer management, and also some per-write TCP state management overhead included. There is a good match with the result in Table 2: 130 versus 99 microseconds. The time spent in the socket layer, e.g. before tcp_output is called and after it returns, is on average about 220 microseconds in the traditional stack and 780 microseconds in the modified stack. Intuitively one would expect these costs to be mostly per-write overheads, but we observe that the socket layer costs are about twice as high as the per-write costs estimated by the model. The reason is that some overhead grows with data size, e.g. the creation of data descriptors, and will be classified as per-byte costs by the model. Moreover, for large writes some of the code in the socket layer is executed multiple times, again causing some of the overhead to be classified as per-byte or per-packet overhead. As a result, only part of the socket-layer overhead shows up as a per-write cost in the model.

Per-byte costs include some overheads that are easy to measure, e.g. data copying, but they also include overheads that are distributed throughout the protocol stack. As a result, the per-byte costs cannot easily be measured directly, and we will limit ourselves to quantifying the larger sources of per-byte overhead. For the traditional stack, important per-byte costs are the copy of the data between the application address space and kernel buffers and the checksumming of the data. Measurements of *bcopy* and the TCP checksum routine show that their cost is 22.6 and 10.4 microseconds/KByte respectively; this accounts for about 60% of the cost in Table 2. For the single-copy stack, an easy to identify per-byte cost is the incremental overhead for pinning, unpinning and mapping pages (Table 3). Our measurements show that it accounts for more than 60% of the per-byte cost (Table 2). Examples of operations that contribute to the per-byte overhead in ways that are hard to quantify are managing mbufs, writing SDMA requests to the adapter, and

handling acknowledgements[1].

In this section we estimated the per-write, per-packet and per-byte overhead by fitting a simple performance model of communication costs to our measurements of communication efficiency. The measurements of the cost of specific operations should give us some confidence that these estimates are reasonably accurate. We discuss the impact of these costs in the next section.

6.1.4 Discussion

The results in Table 2 can be used to explain the difference in efficiency between the single-copy and traditional stack. First, the per-byte costs of the two stacks differ by a factor of eight, i.e. on today's architectures, page manipulation is substantially cheaper than copying data. This is exactly the reason why we implemented the single-copy architecture. Note that we expect this difference in the per-byte cost between the two architectures to grow over time, both as a result of further increases in the page size, and as a result of a widening of the gap between CPU speed and memory speed [13].

The factor of eight difference in per-byte costs translates into a factor of five or more difference in overall efficiency for larger writes. The reason is that the per-write and per-packets costs were not reduced by moving to a single-copy architecture. On the contrary, the per-write costs in the single-copy architecture are about 290 microseconds higher than in the traditional stack. This is a difference of almost a factor of four and it explains why the cross-over point between the throughput and efficiency graphs of the two stacks (Figure 6) is so high. A more careful analysis of the code path shows that there are several reasons for this sharp increase in the per-write cost. A first difference between the per-write costs for the two paths through the stack is the fixed cost of the pin and unpin calls (Table 3), which are only needed in the single-copy stack. It accounts for a third of the increase. These operations could be optimized, and more recent measurements for a different operating system (NetBSD) and faster computer systems show indeed much lower overheads for virtual memory management [13]. A second reason is additional book keeping in the single-copy stack: the single-copy stack does not only have to manage mbufs, but it also has to manage data structures that keep track of the buffers in application space so that they can be unpinned appropriately.

The most significant difference between the two stacks is that, with the modified stack, the sending process is put to sleep as part of the write call, waiting for the data to be DMAed out of its address space; after the SDMA is finished, an interrupt triggers the wakeup of the process. None of these costs are needed with the traditional stack: the process does not have to sleep and wakeup since the copy operation decouples the process from the activity of the network device, and no interrupt is needed at the end of the SDMA operation since kernel buffers can be reclaimed in a lazy fashion when there is other device activity. Note that, with the single copy stack, the interrupt is unavoidable, but the overhead of blocking and waking up the process may not be incurred by all applications. For example, applications that use asynchronous writes may never block and thus will have a smaller increase in the per-write cost.

While for the traditional stack the overhead for the transmission of a packet is strongly dominated by the per-byte cost, the costs are much more balanced for the single copy stack. For

[1] On an active connection, an acknowledgements/window update is typically sent for roughly every $2 * MTU$ data received. Since the MTU is fixed in our experiments, this means that the cost of sending and receiving acknowledgements grows proportionally to the number of bytes sent

example, for a 64KByte write, the per-byte cost accounts for 95% of the communication overhead with the traditional stack and for only 48% with the single-copy stack. For 512 KByte writes, the difference is 97% versus 75%. This means that further improvements in communication efficiency will have to include optimization of the per-packet and per-write operations. This would also improve the efficiency of smaller writes [23].

6.2 Latency

Another performance parameter is the latency for short messages. For 4 byte messages we measured a roundtrip latency using the unmodified TCP/IP stack of 1.0 millisecond over Ethernet and 2.7 milliseconds over HIPPI. The roundtrip latency goes up to 1.6 milliseconds (over Ethernet) and 3.5 milliseconds (over HIPPI) using the modified stack, but as we discussed in the previous section, the modified stack should only be used for large writes.

The reason for the higher latency over HIPPI than over Ethernet is the architecture of the CAB and not the difference in network type. Communication through the CAB is a three step process: the host hands a request to the CAB, the CAB DMAs the data into network memory and the data is transferred to the network link. Since the CAB is managed by software executing on the 29K processor, each step is performed in software, thus increasing latency. In contrast, over Ethernet there are only two steps (there are no separate SDMA and MDMA operations) and only the first step involves software. In short, the higher latency through the CAB is in part a result of having outboard buffering and in part the result of having an experimental prototype, since a production version would probably not use a microprocessor. The fact that a network device controlled by software increases latency was also observed in the Nectar system [2], the predecessor of Gigabit Nectar.

7 Related work

We review related work in the area of host interface design, focusing on single-copy architectures.

The Afterburner interface [6], built by HP Research Labs in Bristol, also uses outboard buffering and checksumming to achieve single-copy communication for sockets and IP protocols. Its predecessor, Medusa [24], has a similar architecture. Afterburner provides the host side of a network adapter and it can be used for different networks by adding a network-specific module, e.g. FDDI or ATM [25, 26]. While at a high-level the Afterburner and CAB architectures are similar, there are a number of significant differences. First, Afterburner uses programmed IO to transfer data to outboard memory, plus some reformating of outboard data is possible, thus relaxing the constraints on the host software. Second, in some versions of Afterburner, there is only a single checksum engine (on the path between host and outboard memory), so on receive, the checksum is only available after the data has been transferred to the application buffer. This adds some complexity in recovering from errors, plus it might be necessary to make a temporary copy of the data to avoid a TCP timeout, if the application does not post a receive early enough. Finally, the software architecture used for Afterburner is different, in part as a result of the different outboard memory structure. On transmit, the socket layer is responsible for the data copy between host memory and outboard memory, which means that it needs knowledge about what network will be used for each write. Moreover, since data is moved outboard before TCP processing is performed, there are some

constraints on how TCP can packetize the data stream. On receive, the two approaches used are more similar, although the use of programmed IO should result in a simpler implementation for Afterburner.

Another example of a single-copy interface is the Gigabit Nectar HIPPI interface for iWarp [27, **?**], which has hardware support for protocol processing similar to that of the provided by the workstation CAB. The organization of the protocol stack is similar to what we describe in this paper, although the implementation is in a proprietary runtime system, and not in a Unix environment.

It is possible to achieve a single-copy architecture (with an extra data access for checksum calculation) for sockets without outboard memory by using virtual memory support. On transmit, pages are sent over the network directly from the application address space, and the application buffer is used as the retransmit copy. To avoid corruption of the data, the buffer is either write protected, or the application is blocked until the data is acknowledged. On receive, incoming data is stored in page aligned kernel buffers and the buffers are mapped in the application address space when a (page aligned) read is posted. These techniques make it possible to achieve single-copy communication, although there are usually severe constraints on the buffer alignment and length, e.g. buffers have to be page aligned. If these constraints are not met, the regular path through the stack is used, although this is not a fundamental constraint [13]. This approach is for example used on some SGI systems; they also use an outboard checksum unit to eliminate the data access for checksumming. With a single-copy stack based on outboard buffering, performance is less sensitive to the alignment and length of the application buffer, but more hardware support is needed.

Finally, many groups have implemented APIs with share and move semantics [8, 9, 10, 11]. By changing the semantics of the API it is sometimes possible to achieve single-copy communication without the need for outboard buffering. We are evaluating these techniques in the context of the Credit Net project [28, 13, 29].

8 Conclusion

We described a network adapter architecture and implementation that provides outboard buffering and checksumming in support of single-copy communication for applications using BSD sockets and IP protocols. We also described how a single-copy path can be added to a typical Unix protocol stack that provides a socket API and uses the TCP and UDP Internet protocols. The implementation uses the external mbuf mechanism to pass data descriptors through the stack, thus making it possible to combine all data-touching operations in the bottom of the stack. New mbuf types are used to describe data in the user's address space and in the outboard memory. Our measurements on a DEC Alpha workstation running OSF/1 v2.0 show that for large reads and writes, the single-copy stack is almost three times more efficient than the original stack.

One of the more difficult issues when implementing the single-copy path is maintaining compatibility with in-kernel applications and other network devices. We achieve this by using a single stack that supports both single-copy communication, plus communication based on traditional mbufs. When passing data into a driver for an existing device or into an in-kernel application, it is sometimes necessary to change the data format from an mbuf holding a descriptor to an mbuf holding the actual data. The format change is required because we do not want to or cannot modify the code of these applications or drivers to deal with the new mbuf types.

Acknowledgements

This research was supported by the Advanced Research Projects Agency/CSTO monitored by the Space and Naval Warfare Systems Command under contract N00039-93-C-0152.

We would like to thank B.J. Kowalski and Somvay Boualouang from Network Systems Corporation. They worked very hard on providing us with reliable CABs. We would also like to thank Karl Kleinpaste, Brian Zill and Kevin Cooney.

References

[1] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

[2] Peter Steenkiste. Analyzing communication latency using the nectar communication processor. In *Proceedings of the SIGCOMM '92 Symposium on Communications Architectures and Protocols*, pages 199–209, Baltimore, August 1992. ACM.

[3] Peter A. Steenkiste. A systematic approach to host interface design for high-speed networks. *IEEE Computer*, 26(3):47–57, March 1994.

[4] Van Jacobson. Efficient Protocol Implementation. ACM '90 SIGCOMM tutorial, September 1990.

[5] Peter A. Steenkiste, Brian D. Zill, H.T. Kung, Steven J. Schlick, Jim Hughes, Bob Kowalski, and John Mullaney. A host interface architecture for high-speed networks. In *Proceedings of the 4th IFIP Conference on High Performance Networks*, pages A3 1–16, Liege, Belgium, December 1992. IFIP, Elsevier.

[6] Chris Dalton, Greg Watson, David Banks, Costas Calamvokis, Aled Edwards, and John Lumley. Afterburner. *IEEE Network Magazine*, 7(4):36–43, July 1993.

[7] Aled Edwards, Greg Watson, John Lumley, David Banks, Costas Calamvokis, and Chris Dalton. User-space protocols deliver high performance to application on a low-cost Gb/s LAN. In *Proceedings of the SIGCOMM '94 Symposium on Communications Architectures and Protocols*, pages 14–23. ACM, August 1994.

[8] Eric Cooper, Peter Steenkiste, Robert Sansom, and Brian Zill. Protocol implementation on the nectar communication processor. In *Proceedings of the SIGCOMM '90 Symposium on Communications Architectures and Protocols*, pages 135–143, Philadelphia, September 1990. ACM.

[9] Peter Druschel and Larry Peterson. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the Fourteenth Symposium on Operating System Principles*, pages 189–202. ACM, December 1993.

[10] Peter Druschel, Larry Peterson, and Bruce Davie. Experience with a high-speed network adaptor: A software perspective. In *Proceedings of the SIGCOMM '94 Symposium on Communications Architectures and Protocols*, pages 2–13. ACM, August 1994.

[11] Jose Brustoloni. Exposed buffering and subdatagram flow control for ATM LANs. In *Proceedings of the 19th Conference on Local Computer Networks*, pages 324–334. IEEE, October 1994.

[12] Hsiao keng Chu. Zero-copy tcp in solaris. In *Proceedings of the Winter 1996 USENIX Conference*. Usenix, January 1996.

[13] Jose Brustoloni and Peter Steenkiste. The effects buffering semantics on i/o performance. In *Proceedings 2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 277–291. Usenix, October 1996.

[14] Digital Equipment Corporation. *TURBOchannel Overview*, April 1990.

[15] Jim Crapuchettes. *TURBOchannel Interface ASIC Functional Specification*. TRI/ADD Program, DEC, revision 0.6, preliminary edition, 1992.

[16] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[17] Van Jacobson. pbufs. Personal communication, October 1992.

[18] J. Postel. Transmission control protocol. Request for Comments 793, September 1981.

[19] Richard F. Rashid, Robert V. Baron, A. Forin, David B. Golub, Michael Jones, Daniel Julin, D. Orr, and R. Sanzi. Mach: A Foundation for Open Systems. In *Proceedings of the Second IEEE Workshop on Workstation Operating Systems*, pages 109–113, September 1989.

[20] Linda R. Walmer and Mary R. Thompson. *A Programmer's Guide to the Mach User Environment*. Computer Science Department, Carnegie Mellon University, November 1989.

[21] K.K. Ramakrishnan. Performance Considerations in Designing Network Interfaces. *IEEE Journal on Selected Areas in Communication*, 11(2):203–219, February 1993.

[22] D. Borman, R. Braden, and V. Jacobson. TCP Extensions for High Performance. Request for Comments 1323, May 1992.

[23] Jonathan Kay and Joseph Pasquale. The Importance of Non-Data Touching Processing Overheads in TCP/IP. In *Proceedings of the SIGCOMM '93 Symposium on Communications Architectures and Protocols*, pages 259–268. ACM, October 1993.

[24] David Banks and Michael Prudence. A High-Performance Network Interface for a PA-RISC Workstation. *IEEE Journal on Selected Areas in Communication*, 11(2):191–202, February 1993.

[25] Aled Edwards, Greg Watson, John Lumley, David Banks, Costas Calamvokis, and Chris Dalton. User-space protocols deliver high performance to applications on a low-cost gb/s lan. In *Proceedings of the SIGCOMM '94 Symposium on Communications Architectures and Protocols*, pages 14–23. ACM, August/September 1995.

[26] Aled Edwards and Steve Muir. Experience implementing a high-performance tcp in user-space. In *Proceedings of the SIGCOMM '95 Symposium on Communications Architectures and Protocols*, pages 196–205. ACM, August/September 1995.

[27] Peter A. Steenkiste, Michael Hemy, Todd Mummert, and Brian Zill. Architecture and evaluation of a high-speed networking subsystem for distributed-memory systems. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 154–163. IEEE, May 1994.

[28] Corey Kosak, David Eckhardt, Todd Mummert, Peter Steenkiste, and Allan Fisher. Buffer management and flow control in the credit net ATM host interface. In *Proceedings of the 20th Conference on Local Computer Networks*, pages 370–378, Minneapolis, October 1995. IEEE.

[29] Jose Brustoloni and Peter Steenkiste. Copy emulation in checksummed, multiple-packet communication. In *IEEE INFOCOM'97*, page To be presented. IEEE, April 1997.