Software Support for Outboard Buffering and Checksumming

Karl Kleinpaste, Peter Steenkiste, Brian Zill*
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Data copying and checksumming are the most expensive operations when doing high-bandwidth network IO 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 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 changes that were made to a BSD protocol stack to make use of a network adaptor that supports outboard buffering and checksumming. Our goal is not only to achieve "single copy" communication for application that use sockets, but to also have efficient communication for in-kernel applications and for applications using other networks. Performance measurements show that for large reads and writes the single-copy path through the stack is significantly more efficient than the original implementation.

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 [3, 17], and for large packets, this overhead is amortized over a lot of data. However, the per-byte 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 mainly the per-byte costs that make high speed communication over networks expensive and that ultimately limit throughput as the network bandwidth increases.

The per-byte overhead can be minimized 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 adaptor, 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 [19], and several projects have proposed or implemented network adaptors that include these features. [11, 20, 5, 8]. Alternatively, it is possible to use APIs with share semantics [6, 7, 2].

Many applications use sockets for communication, and as a result it is worthwhile to look at how they can be supported efficiently. The adaptor hardware and the host software support needed for a single-copy host interface for sockets have been widely described in the literature. However, implementing the software in an existing OS, and have it interoperate correctly with existing devices and applications turns out to be surprisingly complicated. In this paper we describe our implementation of a single-copy stack in the DEC OSF/1 operating system running on an Alpha workstation, using the Gigabit Nectar network adaptor.

The remainder of this paper is organized as follows. We first briefly describe the Gigabit Nectar adaptor architecture and the software requirements for using the adaptor effectively (Section 2). In Sections 3, 4, and 5 we describe the implementation of a single-copy path in a BSD protocol stack. We look at the applicability of the changes for other network interfaces in Section 6, and present performance results in Section 7. We conclude in Section 8.

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.

* Brian Zill is currently with Microsoft

To be presented at SIGCOMM'95

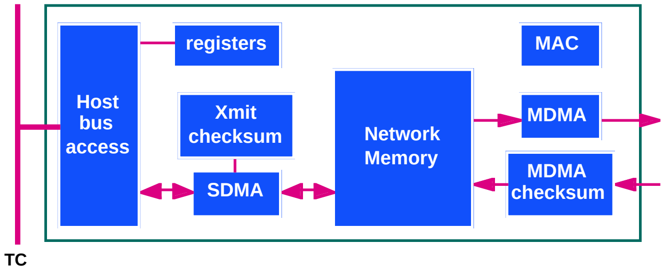

Figure 1: CAB adaptor architecture

2 Gigabit Nectar network adaptor

We briefly describe the architecture of the Gigabit Nectar adaptor, which is called CAB for Communication Acceleration Board, and the impact it has on the host software.

2.1 Adaptor architecture

Figure 1 shows a block diagram of the Gigabit Nectar CAB. The core of the adaptor is a memory used for outboard buffering of packets (network memory). The 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. The SDMA engine has a scatter/gather capability so it can collect the packet header and data from different buffers; user data will typically also be spread out over multiple VM pages that are not adjacent in memory. All DMA engines can operate at the same time, and they use time sharing to access network memory. The register file is used to queue host requests and return CAB responses. The host interface implements the bus protocol for a specific IO bus, in our case the Turbochannel.

The most natural place to calculate the checksum is while the data is transferred to or from the network. This is however not possible on transmit since TCP and UDP place the checksum in the header of the packet. As a result, the transmit checksum is calculated when the data flows into network memory, and it is placed in the header by the CAB in a location that is specified by the host as part of the SDMA request. On receive, the checksum is calculated when the data flows from the network into network memory, so that it is available to the host as soon as the message is available. Although this organization requires two checksum engines instead of one, it is desirable since it allows hosts to process packets as soon as they are received.

Media access control is performed by hardware on the CAB, under control of the host. This component of the CAB is network-specific. Our implementation is for HIPPI [9], which has a line rate of 100 MByte/second. The simplest MAC algorithm for a switch-based network is to send packets in FIFO order. However, this does not make good use of the network bandwidth because of the Head of Line (HOL) problem: if the destination of the packet at the head of the queue is busy, the node cannot send, even if the destinations of other packets are reachable. Analysis shows that one can utilize at most 58% of the network bandwidth, assuming random traffic [10]. The CAB uses multiple “logical channels”, queues of packets with different destinations, to get around this problem [20].

2.2 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 and commands to DMA data into host memory.

Several features of the CAB have an impact on the structure of the networking software. First, to insure full bandwidth to the media, packets must start on a page boundary in CAB memory, and all but the last page must be full pages. 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 host software interacts with the CAB hardware in normal usage, 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 other factors 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 acknowledgement 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

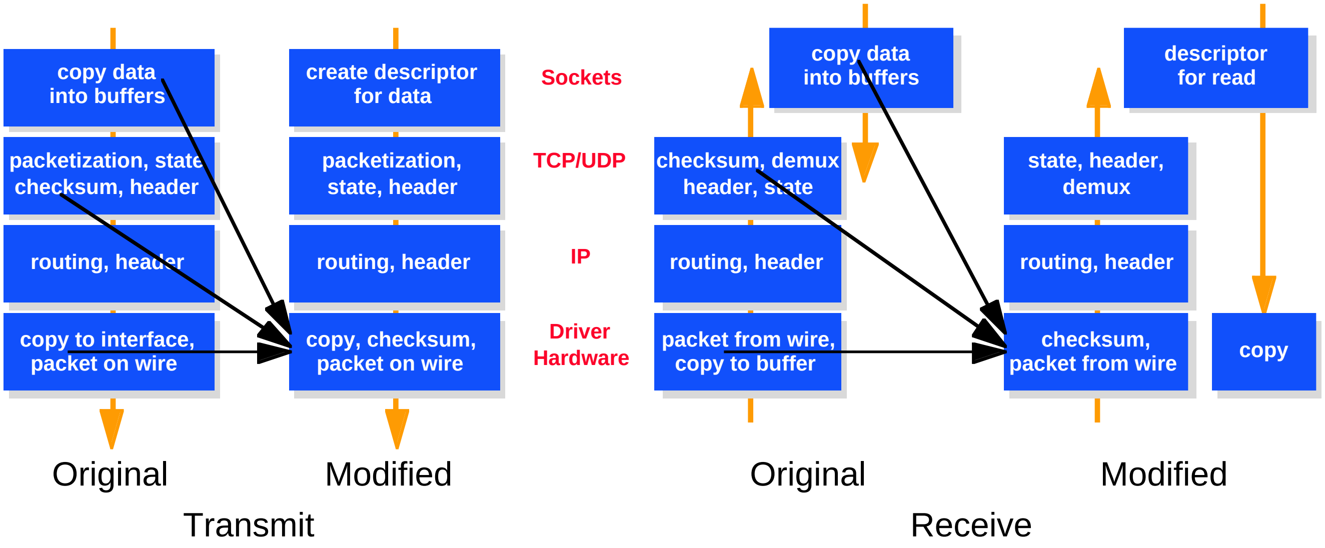

Figure 2: Software architecture

into user buffers. The last SDMA operation is flagged to generate an interrupt upon completion so that the user process can be scheduled.

3 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 [13]. 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 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 2 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.

We discuss how we implemented this software architecture in a Net2 BSD protocol stack, as it exists in DEC OSF/1 v2.0. In the next section we focus on the single-copy path through the stack. In Section 5 we look at correct and efficient interoperation with in-kernel applications and interfaces to other devices.

4 Single-copy path

When adding a single-copy path to the protocol stack, three important design decisions follow directly from the CAB architecture: single stack versus multiple stack implementation, implementation of the data descriptors, and checksum handling. We discuss these design issues in this section, 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.

4.1 Single versus multiple stacks

Besides the CAB, hosts will typically also have to support other interfaces ,i.e. 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 3a). 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

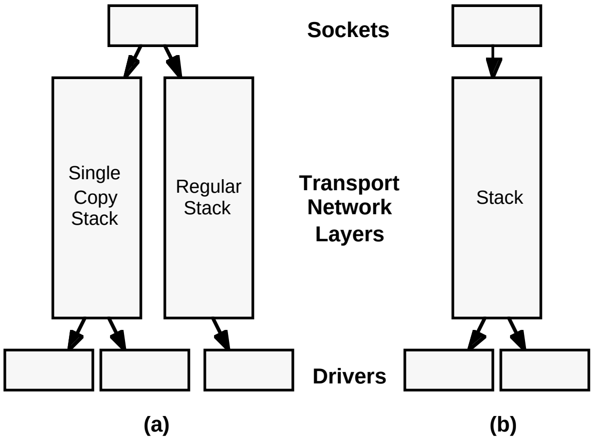

Figure 3: Single versus multiple stacks

interfaces, including “single copy” and traditional interfaces (Figure 3b). 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 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 Section 5.

4.2 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 adaptor. 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 (Section 2.2), 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 UIO mbufs (transmit) and to recognize 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 UIO mbufs are similar to pbufs [12]. 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.

4.3 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 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 [14]). 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).

4.4 DMA

DMA is used on many devices to achieve high throughput over burst-oriented buses. 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.

4.4.1 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 Mach [16] 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 [21] for the target address space. The invoking process can be the kernel, or in the case of a Mach 3.0 microkernel [16], the Unix server. Using these features, it is possible to do address translation and pinning of pages in the application address space in the driver, i.e. DMA support is localized.

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, and to perform the VM operations in the driver would require scheduling the application process for every packet. This implementation would not only be complicated but also inefficient. 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 (Section 7). 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 [6]).

4.4.2 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 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 the 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 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.

4.4.3 Optimization based on packet size

The tradeoff between programmed IO and DMA are well understood (e.g. [15]). 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 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.

4.5 Alignment issues

The CAB DMA engines place several restrictions on the source and destination addresses in host and CAB memory, lengths, and burst size combinations that can be supported. These restrictions stem both from features of the TcIA chip [4] that is used as the interface to the Turbochannel, and from the architecture of network memory, which was designed for high-bandwidth streaming of data. Most of these restrictions can be worked around by the host and CAB software, and only have an impact on the efficiency of the DMA. However, 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. 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 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

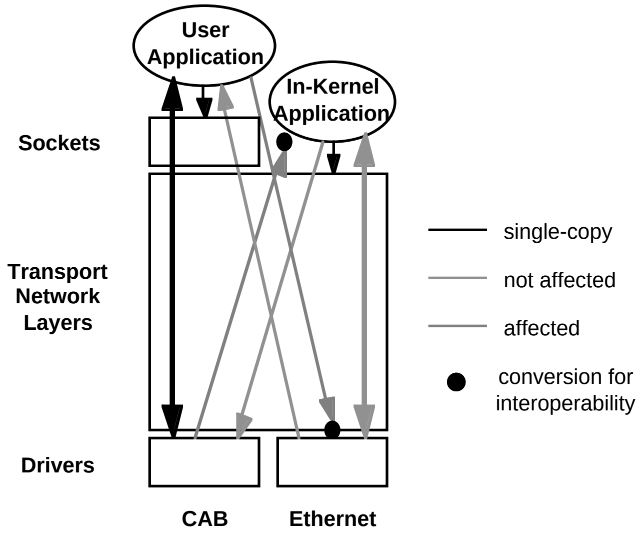

Figure 4: Paths through the protocol stack

writes to be rare.

5 Other devices and in-kernel applications

The previous section 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 4 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 4): 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 4), 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 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 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.

6 Applicability to other systems

An important question is how closely the single-path optimizations are tied to the details of the CAB architecture. [19] presents a taxonomy of host interfaces as a function of

		No Outboard Buffering			Packet Buffering			Outboard Buffering		
API	Checksum	PIO	DMA	DMA + Checksum	PIO	DMA	DMA + Checksum	PIO	DMA	DMA + Checksum
Copy	Header	Copy_C PIO	Copy_C DMA	Copy_C DMA	Copy PIO_C	Copy_C DMA	Copy_C DMA	PIO_C	Read_C DMA	DMA_C
Copy	Trailer	Copy_C PIO	Copy_C DMA	Copy_C DMA	Copy PIO_C	Copy_C DMA	Copy_C DMA	PIO_C	Read_C DMA	DMA_C
Shared	Header	Read_C PIO	Read_C DMA	Read_C DMA	PIO_C	Read_C DMA	DMA_C	PIO_C	Read_C DMA	DMA_C
Shared	Trailer	PIO_C	Read_C DMA	DMA_C	PIO_C	Read_C DMA	DMA_C	PIO_C	Read_C DMA	DMA_C

PIO PIO_C	programmed IO (with checksum)
DMA DMA_C	direct memory access (with checksum)
Copy_C	copy with checksum
Read_C	checksum calculation
▬ ▬	two copy architecture
▪ ▪ ▪ ▪	copy plus checksum
other	single copy architecture

Table 1: Host interface taxonomy

three parameters: the API to the application (copy or share semantics), the characteristics of the transport-level checksum (placed in header or in trailer), and the architecture of the adaptor. The latter covers data movement support (programmed IO versus DMA), data checksumming support, and nature of the data buffering (outboard buffering, no buffering, or single packet buffering that allows insertion of a checksum in the header). The paper shows how the minimum number of bus transfers that are performed as part of an IO operation is a direct function of these three host interface features.

Table 1 summarizes some of the results: its entries list the nature of the copy operations and memory accesses that have to be performed for the different host interface classes. The types of data accesses are: programmed IO (PIO), direct memory access (DMA), and memory-memory copy (COPY), all of which can be combined with a checksum calculation (PIO_C, DMA_C, and COPY_C), and checksum calculation (Read_C). The most efficient interfaces are single copy interfaces: they perform one copy of the data using programmed IO or DMA, and the checksum calculation takes place during this transfer. Some interfaces require a separate read of the data to calculate the checksum (dotted box), either because the checksum calculation cannot be merged with the copy (i.e. DMA only support), or because the checksum cannot be inserted in the packet header if it is. Finally, some interfaces require an extra memory-memory copy to implement the copy semantics of the API without outboard buffering support (dashed box). Note that in some cases this can be a "logical" copy, i.e. using remapping or copy on write.

The top entry in the last column has been the focus of this paper: user and in-kernel applications communicating through the CAB. However, the software implementation will also apply to all scenarios in the outboard buffering case, with the exception that the checksum optimization is only needed for the single-copy scenarios, i.e. when the checksum can be handled during the device copy. Similar mechanisms are needed for the single-copy and copy+read scenarios for the other adaptor architectures, although the lack of outboard buffering and an API with sharing semantics will simplify the implementation.

Another example of a single-copy interface is the Gigabit Nectar HIPPI interface [18], which has hardware support for protocol processing similar to that of the provided by the CAB. The mechanisms it implements are very similar to the ones we described, although they were implemented in a proprietary runtime system, and not in a Unix environment, and the iWarp interface uses an API with sharing semantics

7 Performance

We compare the performance of a single-copy stack with that of an unmodified stack.

7.1 Implementation and measurements

The single-copy stack was implemented in an OSF/1 v2.0 kernel running on a DEC Alpha 3000/400 with 64 MByte of memory. The OSF/1 protocol stack is based on Net2 BSD and also supports TCP window scaling [1]. The network device used is the CAB [20] and the Maximum Transmission Unit (MTU) is 32 KBytes. For all tests, the TCP window size is 512 KBytes. The implementation of the single-copy stack currently supports user-level and in-kernel applications communicating through Ethernet (Section 5) and user-level applications communicating through the CAB.

While the CAB hardware is designed for bandwidths up to 300 Mbit/second, the microcode currently limits throughput to less than half of that. The bottleneck is the transfer of data across the Turbochannel, which is managed by the TcIA chip [4]. The TcIA architecture and the way the chip is used on the CAB make it very hard to pipeline the DMA engines and to use large burst sizes (larger than 8 words), both

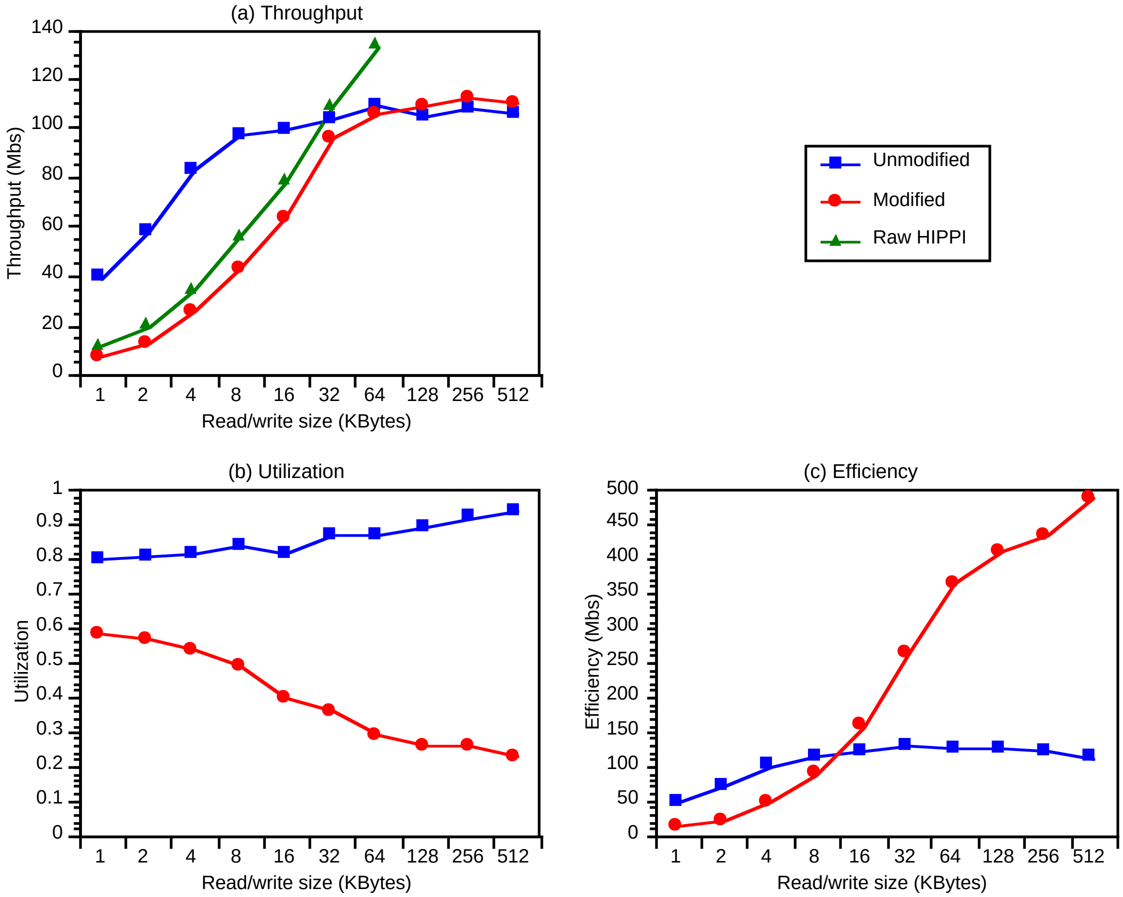

Figure 5: Throughput, utilization and efficiency as a function of read/write size

of which are needed to achieve high throughput. The current microcode does not yet program the SDMA engines in an optimal way. An additional source of overhead is dealing with alignment constraints imposed by the TcIA and network memory, which often requires the use of short burst to properly align larger transfers. We are continuing the optimization of the microcode.

Which protocol stack is used will affect the efficiency of the communication, i.e. how much overhead does communication introduce, and depending on the specific adaptor 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*.

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 charged proportionally to *util* and *ttcp*, so our estimate for CPU utilization to support communication is calculated as:

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

7.2 Experimental results

Figures 5(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 5(a) includes the throughput for raw HIPPI reads and writes. The raw HIPPI throughput test generates well-formed packets

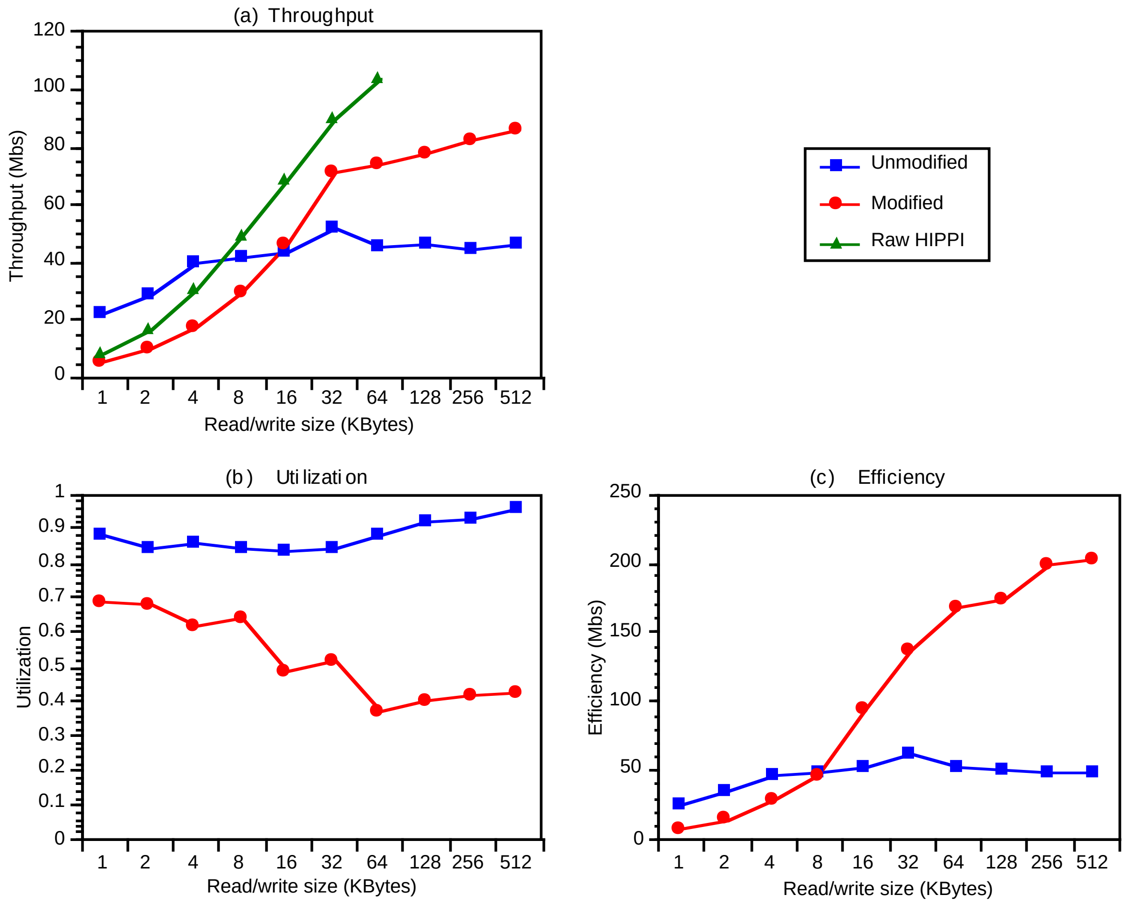

Figure 6: Throughput, utilization and efficiency on an Alpha 3000/300

that can be handled very efficiently by the microcode, so the raw HIPPI results represent the highest throughput one can expect for a given packet size. Note that the measurements for the modified stack always use the single-copy path (i.e. it does not fall back to "regular" path for small writes as described in Section 4.4.3) and does not coalesce the M_UIO mbufs generated by multiple writes into a single packet.

The throughput results show that for small writes the single-copy stack is faster than the original stack. This is a result both of a higher efficiency (see below) and the lack of coalescing in the single-copy stack. For larger reads and writes the two stacks give similar throughputs. The utilization result show however that the modified stack uses fewer CPU cycles to provide that throughput, i.e. it leaves more cycles to 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 5(a) and (b). Figure 5(c) shows the efficiency for the both implementations of the stack. We see that the single-copy stack is significantly 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 for writes of 16 KByte and higher. Note that the efficiency is a rough estimate of the communication throughput that the host can sustain, ignoring limitations imposed by the network or the adaptor.

Figure 6 shows the throughput, utilization and efficiency for an Alpha 3000/300LX, a 125 MHz system with a half speed Turbochannel. This system is only about half as powerful as the Alpha 3000/400, and as a result, the more efficient single-copy stack results in higher communication throughput.

Note that for both systems, the efficiency of the unmodified stack is slightly higher for intermediate read/write sizes (64KByte) than for the larger sizes. We believe that this is a cache effect, i.e. with smaller writes, there is some reuse of data in the cache. On the Alpha 3000/4000, we also observed that reducing the TCP window increases efficiency slightly, even though the throughput is lower. This is probably also a cache effect.

Operation	Cost
Pin	$35 + 29 * n$
Unpin	$48 + 3.9 * n$
Map	$6 + 4.5 * n$

Table 2: Cost in microseconds of virtual memory operations as a function of the number of pages n

7.3 Analysis

Using estimates of the per-byte, per-page, and per-packet overheads on an Alpha 3000/400, we can estimate the expected efficiency of both stacks on that system. These estimates can be used to clarify the measurements.

In the unmodified stack, data that is sent by the application is copied once (socket layer) and read once (during the checksum calculation) by the CPU. For large writes, we expect no locality for the copy (user space to kernel buffers) and little locality for the read. We can estimate the cost of these operations by repeatedly copying/reading regions of an appropriate size; the size of the region will determine the degree of locality in the cache. Copies of a 1 MBytes (no locality) run at 350 Mbit/second, while a read of a 512 KByte region (window size) runs at 630 Mbit/seconds. The per-packet overhead was measured at about 300 microsecond per packet. These estimates add up to a an efficiency of 180 Mbit/second, which is somewhat high, but still reasonably close to the measured efficiency (Figure 5).

We can make a similar estimate for the single-copy stack. The copy and checksum overheads have been replaced by the overheads to pin, unpin and map the host memory pages holding the send and receive buffers. Using a microsecond timer on the CAB, we measured the overheads of these operations on the Alpha 3000/400 - the mean results are shown in Table 2. Using these measurements and a 300 microsecond per-packet overhead, the efficiency of the modified stack for 32 KBytes packets is 490 Mbit/second, which is very close to the measured values in Figure 5.

The only difference between the two stacks is the per-byte/per-page overhead. For the original stack, the estimated per-byte cost accounts for 80% of the overhead, while for the single-copy stack, this number drops to 43%. This means that for the single-copy stack, the per-packet overhead for 32 KByte packets is now more significant than the per-byte cost.

8 Conclusion

We 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 a the actual data. The format change is required because we do not or cannot modify the code of these applications or drivers to deal with the new mbuf types.

Acknowledgements

We would like to thank B.J. Kowalski and Somvay Boualouang from Network Systems Corporation. They worked very hard on providing us with a reliable CAB. We would also like to thank Kevin Cooney, who improved both the performance and stability of the microcode, and Garth Gibson, who made the Alpha 3000/300 available on short notice. Finally, we are very grateful to Jeffrey Mogul for his help in improving the quality of the paper.

References

[1] D. Borman, R. Braden, and V. Jacobson. Tcp extensions for high performance. Request for Comments 1323, May 1992.

[2] 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.

[3] 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.

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

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

[6] 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.

[7] 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.

[8] 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.

[9] Ken Hardwick. Hippi world – the switch is the network. In *Thirty Seventh IEEE Computer Society International Conference*, pages 234–238. IEEE, February 1992.

[10] M. G. Hluchyj and M.J. Karol. Queueing in high-performance packet switching. *IEEE Journal on Selected Areas in Communication*, 6(9):1587–1597, December 1988.

[11] Van Jacobson. Efficient protocol implementation. ACM '90 SIGCOMM tutorial, September 1990.

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

[13] 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.

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

[15] K.K. Ramakrishnan. Performance considerations in designing network interfaces. *IEEE Journal on Selected Areas in Communication*, 11(2):203–219, February 1993.

[16] 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.

[17] 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.

[18] Peter 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 21th Annual International Symposium on Computer Architecture*. IEEE, May 1994.

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

[20] 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.

[21] Linda Walmer and Mary Thompson. *A Programmer's Guide to the Mach System Calls*. Carnegie Mellon University, 1989.