A Systematic Approach to Host Interface Design for High-Speed Networks

Peter Steenkiste

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, Pennsylvania 15213-3891

Abstract

In recent years, networks with media rates of 100 Mbit/second or more have become widely available (FDDI, ATM, HIPPI, ..). However, many computer systems cannot make use of the available bandwidth because of the high overhead associated with network communication. In this paper we review the operations involved in communication over high-speed networks, and we describe optimizations of the network interface that improve network throughput. We also discuss how the payoff of the optimizations is influenced by features of the host software and architecture. This paper is based on our experience with the interfaces for the Nectar and Gigabit Nectar networks.

Keywords: network interfaces, high-speed networks, buffer management, memory hierarchy

This research was sponsored by the Defense Advanced Research Projects Agency (DOD) under contract number MDA972-90-C-0035, in part by the National Science Foundation and the Defense Advanced Research Projects Agency under Cooperative Agreement NCR-8919038 with the Corporation for National Research Initiatives.

1. Introduction

A network interface allows a computer system to send and receive packets over a network. It consists of a *network adaptor*, the hardware that connects the network medium with the host IO bus, and the *network software* on the host that handles application communication requests and manages the adaptor. For high-speed networks (100 Mbit/second and up), network throughput is typically limited by software overhead on the sending and receiving hosts. As a result, it is important to minimize the overhead associated with network communication. This is not only necessary to achieve good application-level latency and throughput, but will also result in a minimal number of cycles being lost for applications as a result of communication.

Communication overhead is influenced by several factors, including: the communication protocol used, the programming interface used by the application, and the architecture of the network adaptor. This paper describes how these different factors influence communication performance and under what conditions hardware support on the adaptor can result in improved performance.

The paper is organized as follows. We first describe the organization of a typical network interface (Section 2) and discuss performance considerations for interfaces to high-speed networks (Sections 3 and 4). We then look at how the network interface should be organized to support efficient communication. Section 5 considers software optimizations that apply to simple network adaptors, and we show how a more powerful adaptor can improve communication performance, given an appropriate organization of host software in Sections 6 through 10. We conclude with a short review of existing adaptors for high-speed networks (Section 11). Table 1 summarizes some network acronyms and terms used in this paper.

2. Traditional network interface

Most Unix workstations use networking code that is based on Berkeley UNIX™. Applications communicate over the network using the socket interface and the internet protocols (TCP or UDP over IP). We briefly review the operations involved in sending and receiving data over a network in this environment; Figure 1 shows the software involved organized following the OSI layered reference model.

On a send, the socket layer copies the data from the user's address space into system buffers and invokes transport and network protocols. If the user requested a reliable byte stream protocol, TCP over IP will be used, and the protocol processing will include packetization, error handling, end-to-end flow control, routing and congestion control. A best-effort protocol such as UDP over IP is simpler and performs only a subset of these tasks. When the protocol processing is finished, one or more packets are handed to the datalink layer, which will transmit them over the network. A similar sequence of operations is performed on the receive side, but in the reverse order. Specifi-

OSI reference model	Open Systems Interconnect reference model: organizes the network interface as a set of 7 layers, going from the application layer to the physical layer. Each layer implements a specific set of functions, and layers are separated by well defined interfaces.
Protocol	Set of rules and conventions used in the conversation between peer layers on the sending and receiving host, e.g. a transport protocol.
Packet-based network	Class of network in which communication is based on packets: variable-sized blocks of data. Networks typically place a limit on the packet size, called the maximum transfer unit (MTU). The MTU tends to grow with the network speed.
Cell-based network	Class of network in which communication is based on cells: fixed-sized, usually small, blocks of data.
ATM	Asynchronous Transfer Mode: cell-based network technology developed for telecommunications, but now also used as the basis for data networks. Cells are 53 bytes: 5 bytes of header and 48 bytes of payload.
FDDI	Fiber Distributed Data Interface: 100 Mbit/second fiber-optic local area network based on a ring topology.
HIPPI	High-Performance Parallel Interface: point-to-point channel with a bandwidth of 800 or 1600 Mbit/second. HIPPI can be turned into a network by the use of HIPPI crossbar switches.
IP	Internet protocol: protocol that support the transmission of data over a collection of networks. Corresponds to the network layer (layer 3) in the OSI model.
UDP	User Datagram Protocol: protocol that supports best-effort delivery of packets, i.e. does not retransmit corrupted or lost packets. UDP is an example of a transport protocol (layer 4).
TCP	Transmission Control Protocol: protocol that supports communication based on a reliable byte-stream. TCP is an example of a transport protocol (layer 4).

Table 1: Acronyms and terms

cally, the datalink layer places incoming packets in system buffers, called receive buffers, and after protocol processing has been performed, the data is copied into the user's address space by the sockets layer as part of the application's receive call.

The error handling performed by reliable transport protocols has a significant impact on how the data is handled by the protocol stack. Protocols typically use an end-to-end checksum to verify data integrity and timeouts to detect lost packets. Errors are detected by calculating a checksum over the data on both the transmit and receive side; the receiver ignores the packet if the checksum it calculates is different from the one inserted in the packet by the sender. Checksumming requires that the TCP protocol code on both the sending and the receiving host read all the data. To detect lost packets, the sending host starts a timer every time it sends a packet. If this timer expires before the packet has been acknowledged, the sender assumes the packet was lost or corrupted, and retransmits the packet. This means that the sender has to keep a copy of all transmitted data until it is acknowledged by the receiver. The copy of the data in the system buffers, also called the retransmit buffer, is used for that purpose.

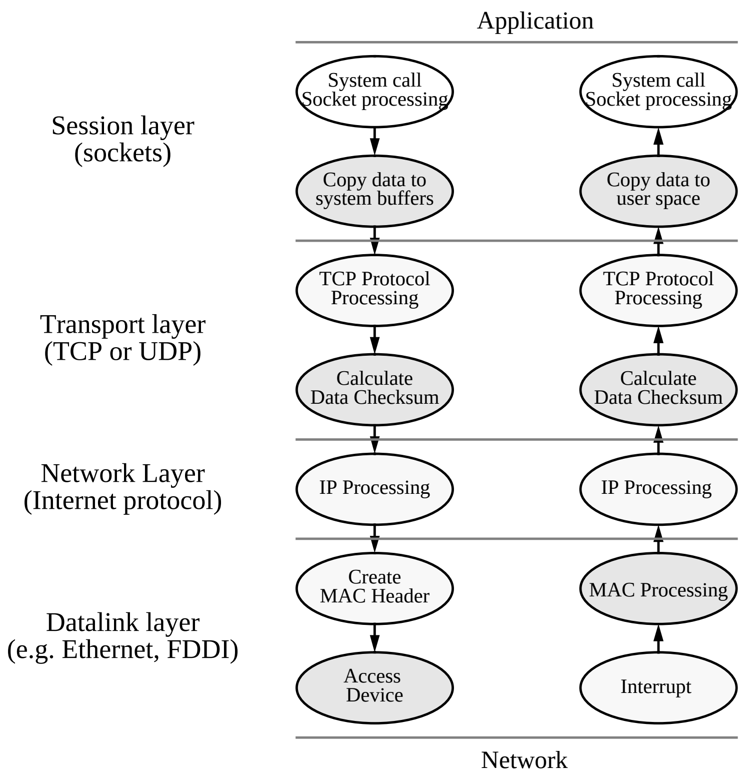

Figure 1. Communication software structure

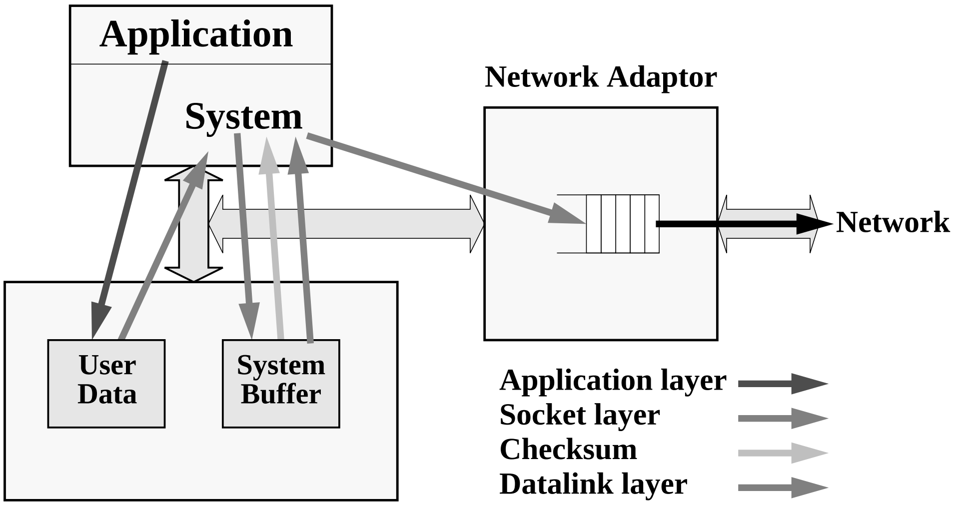

Figure 2. Data transfers in traditional network interface

Figure 2 summarizes the data transfers that are performed as part of a send. The application writes the message into a buffer in its address space. The socket code copies the data into a system buffer, and the transport protocol reads the data for the purpose of calculating the checksum. Finally, the datalink layer copies the data to the network adaptor. In total, the data crosses the memory bus six times, including the first write by the application. On some implementations, there is an additional CPU copy to coalesce data in multiple ‘‘system buffers’’ into a single ‘‘device buffers’’, which adds two more bus transfers. Receives follow the inverse path.

From the perspective of the host, the network adaptor looks like a small buffer plus some control registers. Packets that are written into the buffer are transmitted by the adaptor hardware, and incoming packets have to be copied from the buffer to host memory. In the simplest case, the buffer is organized as a FIFO. The size of the buffer is constrained by the following two factors. First, networks typically have requirements on how quickly bytes must be placed on the wire (for example: no idle time between bytes), and these requirements can be met only by having a certain amount of data on the adaptor before transmission starts. Second, increasing the buffer size potentially reduces the frequency with which the host has to interact with the adaptor; this will in general improve performance since interactions often cause host interrupts, which creates overhead.

3. Communication overhead

The overhead associated with network communication has been analyzed in several papers [1, 10]. Although it is difficult to compare the results because the measurements are made for different architectures, protocols, programming interfaces, and benchmarks, there is a common pattern: there is no single dominating source of overhead. The conclusion is that implementing an efficient network interface involves looking at all the functions in the network interface, and not just a single function such as, for example, protocol processing.

The distribution of packet sizes observed on networks is bimodal: it is a mix of small packets (header plus a few tens of data bytes) and maximum-size packets. The overheads associated with both types of packets are quite different. The operations associated with sending and receiving small packets falls in 4 classes [11]: transport protocol processing, context switching (thread/process switching and interrupt handling), datalink protocol processing (dealing with the network adaptor), and buffer management. Studies have shown that each class contributes significantly to the communication cost. As an example, Table 2 breaks up the time it takes to send and receive a one-word message (4 bytes) using different communication protocols over the Nectar network [11]. For large packets, the same set of operations has to be performed, but the cost of copying the data starts to dominate. Figure 3 shows how the communication cost grows with the message size. The cost of copying the data dominates the per-packet processing cost for packet sizes over 1500 bytes (Figure 3).

Different overheads scale differently with CPU and memory speed. First, there are overheads associated with every application send and receive operation (socket call -- white elipses in Figure 1), and with every packet sent or received over the network (TCP, IP, physical layer protocol processing and interrupt handling -- light grey elipses); these operations involve mainly CPU processing. Other overheads scale with the number of bytes sent (copying and

	Datagram	Request-Response	Reliable Message	UDP over IP
Datalink protocol	32 (33%)	36 (23%)	65 (44%)	37 (16%)
Transport protocol	6 (6%)	25 (16%)	31 (24%)	50 (21%)
Buffer management	30 (31%)	52 (34%)	27 (21%)	60 (26%)
Context switching	29 (30%)	41 (27%)	13 (10%)	87 (37%)
Total	97	154	127	234

Table 2: Breakup send and receive overhead for a one word message in microseconds (% of total latency)

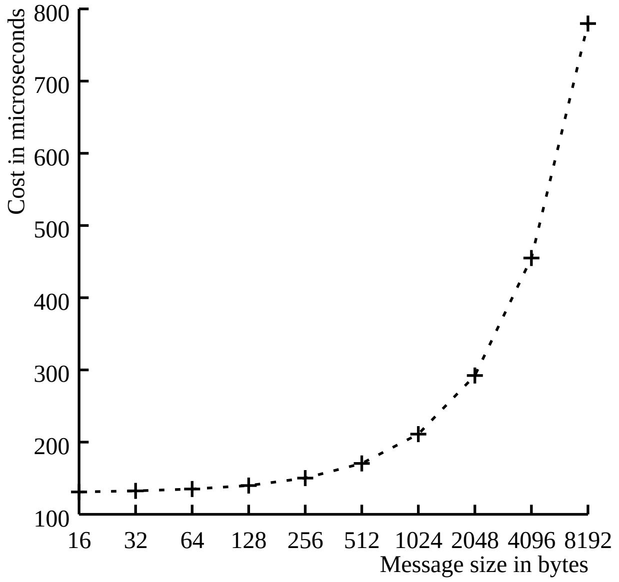

Figure 3. Message overhead on Nectar for the reliable message protocol

checksumming -- dark grey elipses); these operations are largely limited by memory bandwidth.

4. Performance Considerations

Network communication places a high burden on the resources of the host. As a result, the peak application-to-application network throughput that workstations can support is often only a few tens of megabits per second, even if the network bandwidth is 100 Mbit/second (FDDI) or more (ATM and HIPPI).

As networks get faster, data copying and checksumming, the per-byte operations, become increasingly the dominating overheads for two reasons. First, as networks get faster, larger packets are used, so the per-packet overheads are amortized over larger packets. Second, on many computer systems the memory system is the most performance critical part of the system, and the per-byte operations make heavy use of this resource. Moreover, the data accesses performed during network operations have very low locality, so memory system optimizations such as caching are not very effective [4].

We showed earlier how data crosses the memory bus five times in the network interface (Figure 2). These transfers will limit network throughput if the network bandwidth is a non-negligible fraction of the memory bandwidth. Given the memory bandwidth of current workstations (one to a few hundreds of megabytes per second), the memory bandwidth of the host will become the bottleneck for network bandwidths of 100 Mbit/second and up.

Another network performance measure is the latency for short packets, i.e. the time it takes for the message to travel from an application on one system to an application on another system. For small packets, per-packet and per-send/per-receive overheads dominate the total communication cost since the cost of copying the data is small. For small packets it is actually often advantageous to copy the data, since sharing buffers between different parts of the protocol stack increases the buffer management cost [7]. Latency is reduced by optimizing the per-packet operations, as is discussed in a number of papers in the literature [10, 11]. Because of space constraints we will not further discuss the network interface design issues that affect latency.

In the remainder of this paper we look at techniques that make the network interface more efficient for large packets, i.e. we focus on increasing throughput. They include optimized implementations of the networking software, and the use of more powerful network adaptors. Software solutions typically address the issue of redundant data transfers. Hardware solutions fall in three area: DMA support for more efficient data movement, buffer space on the adaptor to eliminate copy operations, and protocol support on the adaptor. We concentrate on high-speed networks, i.e. media rates of 100 Mbit/second and above, and assume a reliable protocol is used (i.e. the sender must keep a copy of the data for retransmission, and the sender and receiver have to calculate a checksum).

5. Software Optimizations

Efficient implementations of the network interface for high-speed networks center around optimizing the data transfers, and the programming interface used by the application determines how difficult this is.

The minimal adaptor used in the traditional network interface (Section 2) leaves relatively little room for optimization, given the constraints imposed by the socket interface. With the socket interface, the application specifies the area in which messages are built, and operations block until that area can be safely reused. This requires the system software to make a retransmit copy of the data as part of the send call; otherwise the application would have to remain blocked until the data had been acknowledged. On receive, the application specifies where the data should be placed, so data must be copied to a specific location as part of the receive operation. We will call communication primitives with these characteristics *immediate* primitives. Even with these constraints it is however possible to optimize the checksum calculation by calculating the checksum while the data is being copied, as is shown in Figure 4. This eliminates one access to the data, and often the checksum is free since the computation can be scheduled in the load and store delay slots on many high-performance CPUs.

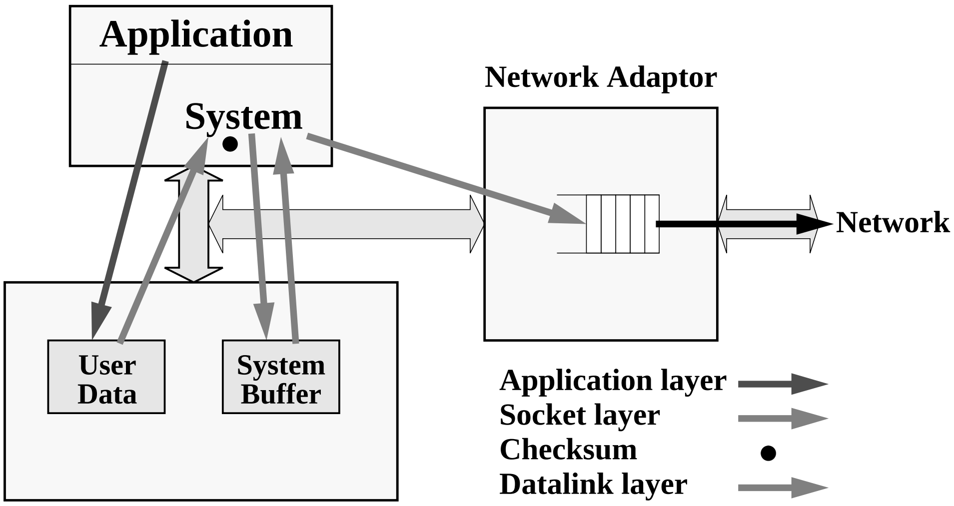

Figure 4. Data transfers with checksum calculated during data copy

Eliminating more data transfers requires modification of the application programming interface. With *buffered* communication primitives, the application builds and receives messages in buffers that are shared with the system. It releases control of the buffer as part of a send call, and a receive call returns a pointer to a buffer that has to be freed or reused for transmission after the message has been consumed. Buffered communication primitives allow us to eliminate one of the two data copy operations on both transmit and receive. Since on transmit, ownership of the data buffer is transferred to the system, that buffer can be used as a retransmit buffer, and there is no need to make a copy of the message (Figure 5). On receive, the application is given a pointer to the location of the data, so there is no requirement to copy the message to a specific location in the user's address space. Buffered primitives are used in a number of systems, including Nectar [2] and the Firefly [10].

For protocols that store the checksum in the header, such at TCP, buffered sends will require a separate pass over the data to calculate the checksum, as shown in Figure 5. Although it is possible to calculate the checksum during the copy operation to the adaptor, it is not possible to insert it in the header of the packet if the buffer on the adaptor is a FIFO. This separate pass over the data can be avoided for protocols that place the checksum in a packet trailer (such as XTP), or if there is at least one packet worth of random-access buffering on the adaptor (see next section). Checksumming does not require a separate pass over the data on receive.

Buffered primitives place a larger burden on the user than immediate primitives such as sockets. On transmit, the application has to build the message in an area that is specified by the system and it can no longer use the data after the send. Data is received in buffers that were allocated by the system and application-level messages might not be

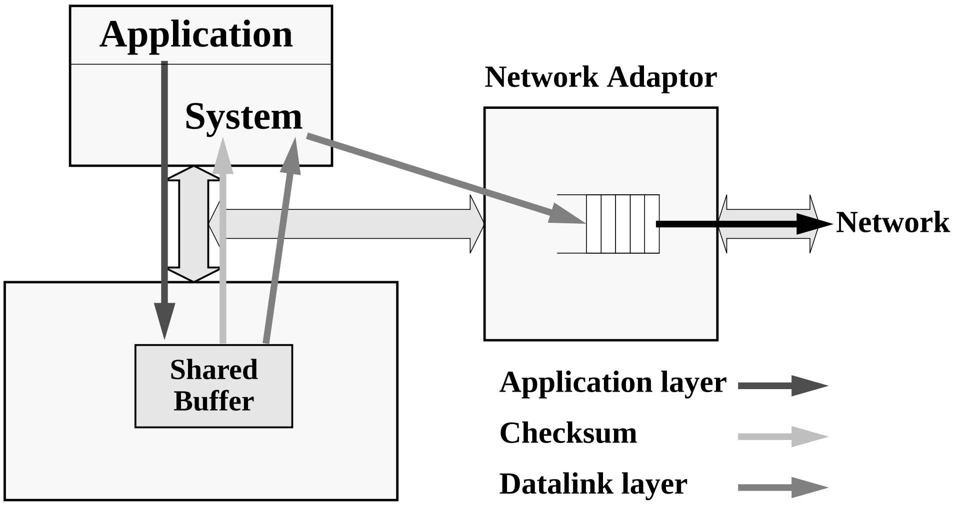

Figure 5. Data transfers with shared buffer interface to application

contiguous in memory since they might have been sent using multiple packets that were placed in separate, non-contiguous buffers. It is exactly this reduced flexibility for the user, and extra flexibility for the system, that allows a more efficient network interface.

It is possible to approximate the benefits of buffered primitives using a socket-like application programming interface. On transmit, the copy of the message in the application space is used as retransmit copy. This is achieved by changing the status of the virtual memory pages containing the message to unwritable; they are made writable once the data has been acknowledged. If the application tries to overwrite the data before it is acknowledged, the data has to be copied, or the process can be blocked. On receive, the data component of incoming packets are placed in separate pages, starting at the beginning of the page. If applications issue receives using an address that corresponds to a page boundary, data can be made available without copying by remapping the data. However, if a non-page aligned address is specified, the data has to be copied. The effectiveness of these techniques depends not only on the details of the system, e.g. cost of page remapping, but also on the behavior of the application. In general, these techniques will be most effective if the programmer understands the optimization that is being performed, i.e. the programmer knows a shared buffer interface is being emulated.

6. DMA

A first way to enhance the minimal adaptor described earlier is to add a direct memory access (DMA) engine to the adaptor to move data directly between host memory and the buffer on the adaptor without involvement by the host CPU. This has several advantages compared with having the CPU copy the data (programmed IO or PIO):

- DMA eliminates one transfer over the bus compared with PIO, as shown for the socket interface in Figure 6.
- Most high speed busses depend heavily on the use of large burst transfers to achieve high throughput.

Figure 7 shows, for example, the throughput of the DEC Turbochannel as a function of the burst size. DMA engines can easily use large bursts, so they can achieve high throughput. CPUs are often limited to moving single words or small bursts, thus limiting the network throughput that can be achieved.

- DMA can happen in parallel with computation on the CPU. How much overlap can be achieved depends on the available memory bandwidth, and on the cache hit rate of the application.

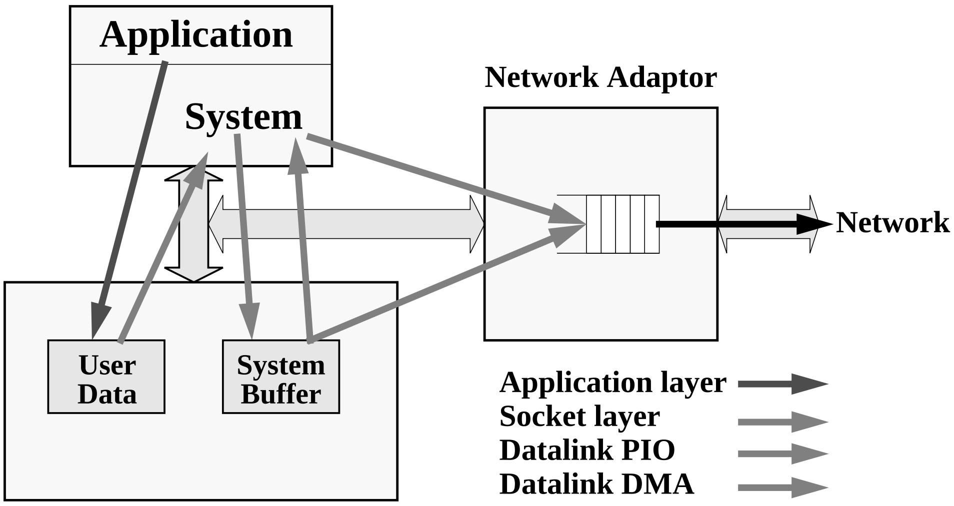

Figure 6. Data transfers with DMA

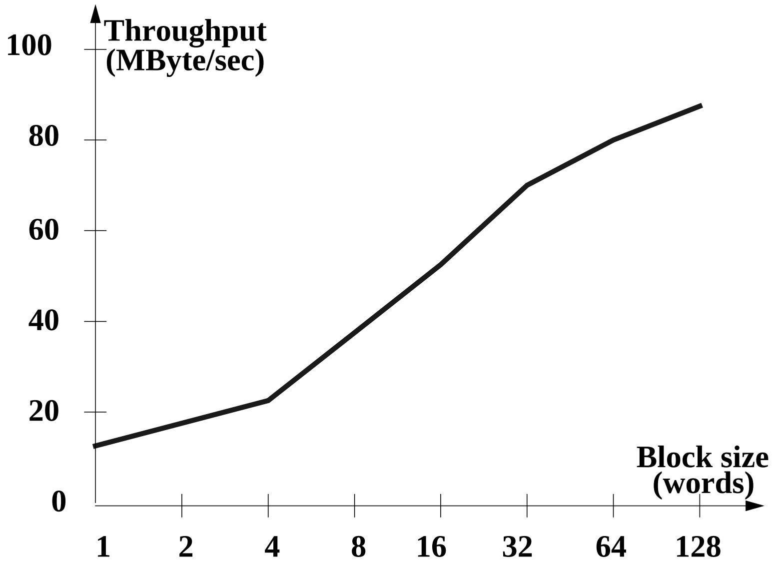

Figure 7. Throughput of Turbochannel for different burst sizes

DMA can achieve a higher peak throughput than programmed IO, but several overheads are associated with it:

- The use of DMA can result in incorrect data being transferred as a result of inconsistencies between the cache and memory, as illustrated in Figure 8): the DMA engine fetches old data from memory if the new data only appears in the cache (systems with write-back cache only). The software has to provide consistency by flushing the data to memory before starting the DMA (grey arrow). A similar problem occurs on receive, and cache invalidation is required to guarantee consistency. Flushing and invalidation are expensive operations and can reduce throughput significantly. Fortunately, newer workstations (DEC Alpha, SparcStation 10) often guarantee consistency between the cache and the memory in hardware. This is a result of the fact that CPU chips are being designed to work as part of multiprocessors, which require hardware cache consistency.

- When DMA is used to transfer data to and from the user's address space, the pages that hold the data have to be locked in memory for the duration of the transfer. Per-operation locking is not needed for DMA access to dedicated system or shared user-system buffers since they can be locked permanently, so DMA access to these areas is less expensive.
- With DMA, data copying happens asynchronously. This adds overhead since the CPU and adaptor have to synchronize at the end of the DMA, typically using an interrupt.

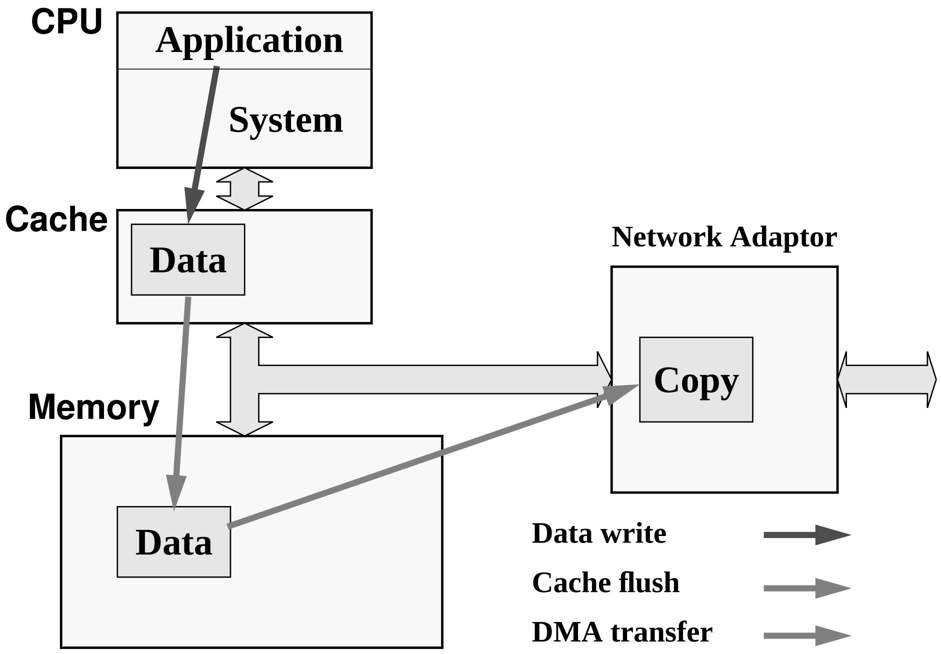

Figure 8. Interactions between the cache and DMA

The tradeoff between using DMA and programmed IO depends heavily on characteristics of the IO bus and the host architecture and software. As an example, Figure 9 shows the bandwidths over a Turbochannel on a DEC DS5000/200 for different block sizes using DMA and PIO; the numbers include the overhead of dealing with the cache and page locking/unlocking. The DEC 5000 does not have hardware memory-cache consistency (favoring programmed IO), and does not allow burst mode access to the Turbochannel from the CPU (favoring DMA). For large block sizes DMA is needed to achieve high throughput, but for smaller block sizes programmed IO is more efficient. The throughput for transmit and receive are different, mainly because writing across a bus is usually slightly faster than reading, and the direction of the access is different for transmit and receive; the direction of the access is also different for DMA and PIO. A detailed case study on the tradeoffs between the use DMA and PIO for an FDDI interface is presented in [9].

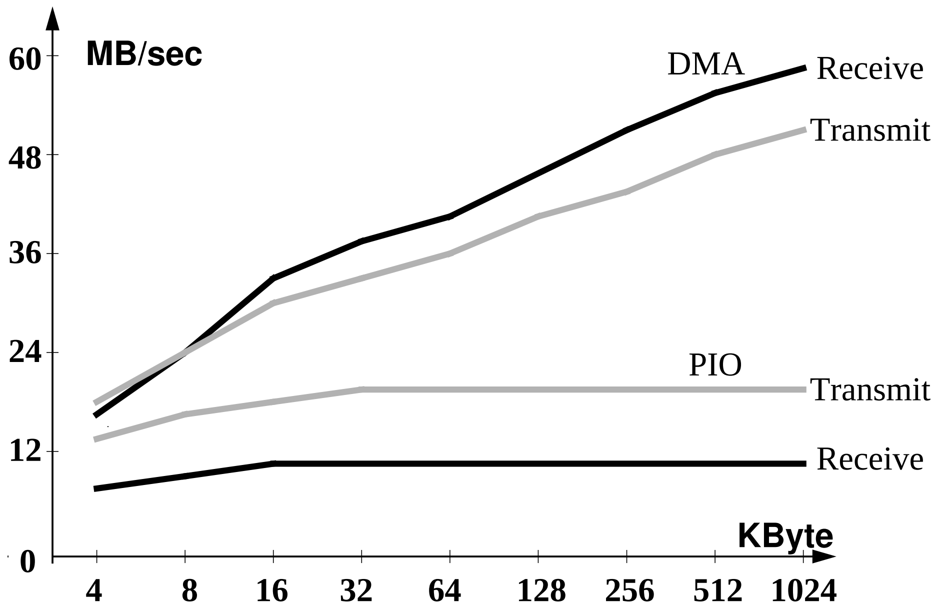

Figure 9. Transmit and receive transfer rates across TURBOchannel using DMA and programmed IO

7. Outboard buffering

A second form of hardware support on the adaptor is buffer space that is larger and more flexible than the speed matching FIFO in the minimal adaptor described in Section 2. As a first step one can allocate space for a few packets. We call this packet buffering. Packet buffering can be beneficial for a number of reasons:

- If the transport protocol used stores the checksum in the header, this will allow the checksum to be inserted in the header of the packet, after being calculated as the packet is copied to the adaptor. At least one packet's worth of buffering is needed, independent of whether the copy and the checksum are done in software or hardware.
- For some networks it is important that the adaptor has access to a few packets so that it can schedule packet transmission based on the conditions in the network. The simplest example is a switch-based network, where packet scheduling would be driven by the availability of output ports on the switch. This requires that the adaptor has access to a number of packets with different destinations.

A more aggressive form of buffer support on the adaptor is called outboard buffering: the buffers on the adaptor are large enough to be used as retransmit and receive system buffers. Moving the system buffers onto the adaptor can eliminate the memory-memory copy required to move the data in and out of the user's address space with a socket interface (Figure 10 versus Figure 4). On transmit, the checksum is calculated during the copy, as before. On receive, the checksum is best done while the data flows from the network into the buffer on the adaptor, so that it is available immediately, and protocol processing does not have to wait until the data is copied into the user's address space. This requires calculating the checksum in hardware. Compared with buffering in host memory

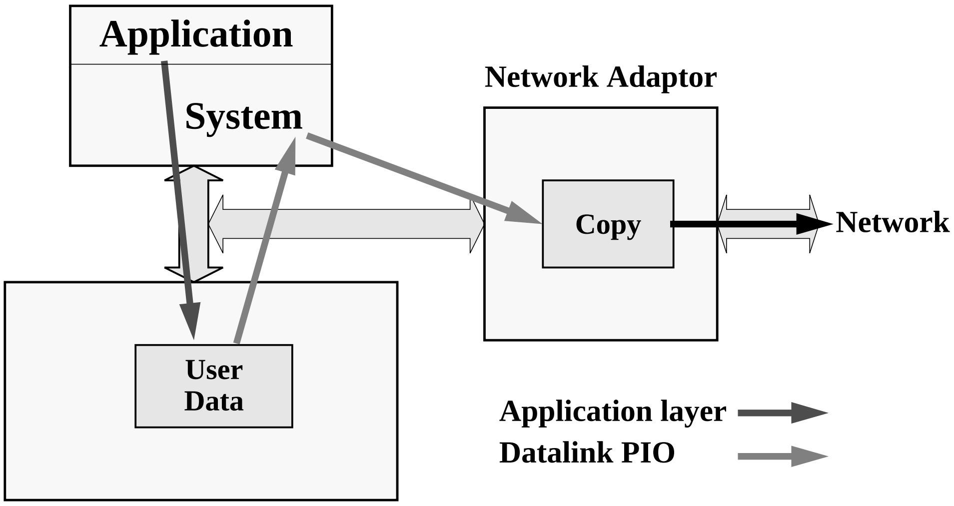

Figure 10. Data transfers with outboard buffering

(Figure 2), we have reduced the number of bus transfers by 2, both on transmit and receive. This interface corresponds to the ‘‘WITLESS’’ interface proposed by Van Jacobson.

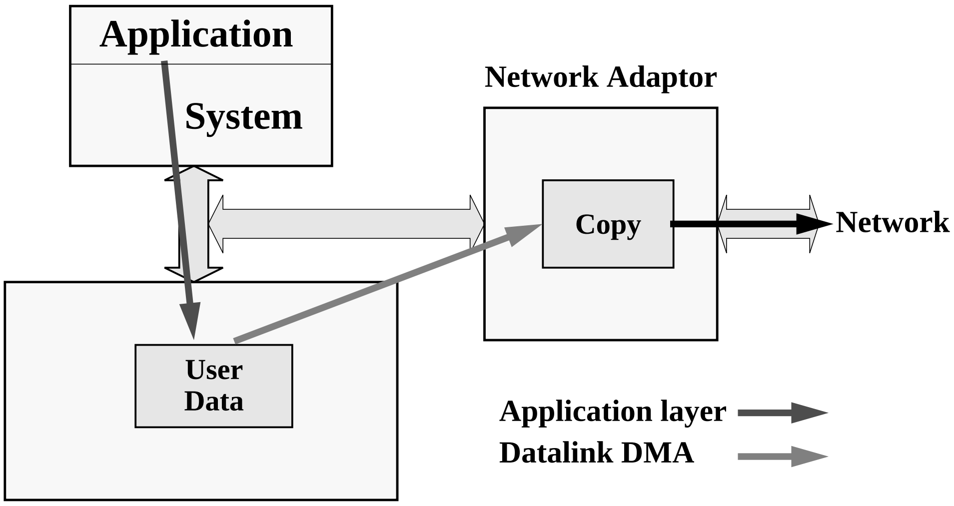

Figure 11. Data transfers with outboard buffering and DMA

If we combine outboard buffering with DMA (Figure 11), the number of transfers is reduced to two. If checksumming is done during the data copy, then hardware checksumming is now also required on transmit. The number of bus transfers is down to two, which is the minimum for a socket interface. For network speeds that are a high percentage of the memory bandwidth, reducing the number of bus transfers from six with a traditional software implementation to two with an adaptor that supports outboard buffering and DMA can have a significant impact on the throughput and efficiency of the network interface. Outboard buffering also reduces the number of times the host has to be interrupted. Although it is still necessary to notify the host on end-of-DMA so that the application can be rescheduled, it is not necessary to notify the host that packets have been transmitted, since the acknowledgement of the data by the receiver will implicitly indicate successful transmission [12].

Outboard buffering (with DMA) is a very effective optimization for applications that use immediate communication primitives. However, it does not reduce the number of bus transfers for applications that use buffered primitives, since the shared buffers already automatically provide retransmit and receive buffering. The exception is when the application is allowed to access the buffers on the adaptor directly, as is the case in Nectar [2]. The Nectar system supports outboard buffering and makes these buffers available through a shared buffer interface. This allows data to be sent and received with only a single transfer over the bus (Figure 12), which is the absolute minimum. However, this organization only performs well if programmed IO gives good performance compared with DMA, as is the case in that system.

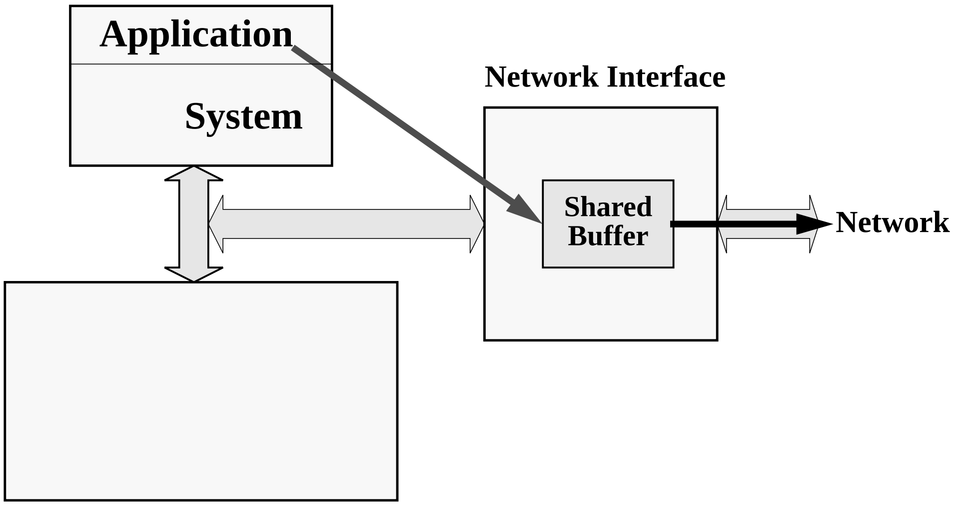

Figure 12. Data transfers with shared buffer on adaptor

8. Dataflow summary

Both the software and hardware optimization discussed in the previous sections had as a goal to reduce the number of data transfers across the bus, or to make those transfers more efficient. These factors were identified earlier as the key performance indicator for interfaces to high-speed networks. We observed that even with this simple performance measure, the best way of organizing the network interface and the benefits of hardware support depend on several factors: the programming interface used by the application, the placement of the checksum in the packet, the bus and memory architecture of the host, and to a lesser extent, the characteristics of the network. These factors jointly define the environment in which the network interface has to operate.

Table 3 summarizes the minimal number and the type of accesses across the memory and I/O bus performed as part of a transmit operation. The rows correspond to different constraints imposed by the host software (the application programming interface and the placement of the checksum) and the columns represent adaptor features. Network features might require certain adaptor features, e.g. some network technologies will require at least packet buffering. The symbols indicate the type of transfer:

R	CPU read from host memory (through the cache)
W	CPU write to host memory (through the cache)
P	CPU write across the IO bus (part of programmed IO)
D	DMA operation from host memory to adaptor

Table 3 represents only a partial picture of the communication overhead. First, the efficiency of the different types of transfers varies widely, and depends in most cases on whether the data is in the cache or in memory. Second, some overheads, for example page locking for DMA, are not shown, and finally, some of the copy operations could be avoided by using copy-on-write or remapping as discussed in Section 5.

Table 3 shows that hardware support pays off in many cases. The use of DMA reduces the number of bus transfers in almost all cases, and outboard buffering can reduce the number of transfers across the bus for most combinations of uses and architectures, compared with minimal outboard buffering. Specifically, outboard buffering eliminates two transfers for applications that use a socket-like interface, and almost always one transfer if buffered primitives are used, independent of whether programmed IO or DMA is used. Furthermore, comparing the number of bus transfers for different types of primitives under similar conditions shows that it does pay to use primitives that give the system more control over where data is placed and when transfers take place. Specifically, moving from immediate to buffered sends and receives generally reduces the number of bus transfers. Note also that DMA is less expensive with buffered primitives since page locking can often be avoided.

A number of simplifying assumptions have been made in our discussion. For example, we have ignored data format conversion for communication between architectures with different data representations and encryption. These operations are often performed in the application space but they could also benefit from integration with the network interface in the same way as we optimized checksumming.

Cell-based networks break all packets into a stream of small fixed-sized cells. An example is ATM, a standard originally defined for telecommunication, but is now also being used for data networks. The design issues for interfaces to cell-based networks are largely the same as for packet-based networks. The segmentation and reassembly of packets is usually done in hardware and has little impact on the issues discussed in this paper. One difference is that the unit of transmission for cell-based networks is very small (44-48 data bytes for ATM), so while packet-based networks typically require that the entire packet is stored on the adaptor before transmission can start,

		no outboard buffering			packet buffering			outboard buffering		
Application Programming Interface	Checksum	PIO	DMA	DMA + checksum	PIO	DMA	DMA + checksum	PIO	DMA	DMA + checksum
Immediate	Header	R+W+R+P 4	R+W+D 3	R+W+D 3	R+W+P 3	R+W+D 3	R+W+D 3	R+P 2	R+D 2	D 1
Immediate	Trailer	R+W+R+P 4	R+W+D 3	R+W+D 3	R+W+P 3	R+W+D 3	R+W+D 3	R+P 2	R+D 2	D 1
Buffered	Header	R+R+P 3	R+D 2	R+D 2	R+P 2	R+D 2	D 1	R+P 2	R+D 2	D 1
Buffered	Trailer	R+P 2	R+D 2	D 1	R+P 2	R+D 2	D 1	R+P 2	R+D 2	D 1

Table 3: Number and type of data transfers: Read, Write, Programmed IO, DMA

transmission over an ATM network can start with very little data. This is one of the reasons why ATM has the potential of providing low-latency communication. However, it makes network interface architectures that require storing entire packets on the adaptor, unattractive since it eliminates this benefit.

9. Protocol processing

Protocol processing is often blamed for the high cost of network communication, even though it is only one of several sources of overhead associated with communication [11]. Moving protocol processing to the adaptor can reduce communication overhead on the host and improve overall communication performance. Outboard support for protocol processing can be done at a number of levels:

1. Protocol processing is performed on the host. This is the most common implementation.
2. Protocol processing is performed on the host, but the network adaptor provides support for specific operations such as checksumming or separating the header and data.
3. The most aggressive option is to perform protocol processing on the network adaptor. The network adaptor can be very flexible and support a wide range of protocols, typically in software [2], or it can be hardwired for one specific protocol [6].

Outboard protocol processing, i.e. the most aggressive option, has a number of disadvantages. First, the network adaptor becomes more complex and expensive, especially if multiple protocols have to be supported. One of the reasons is that the engine performing protocol processing should be fast, preferably as fast as the host [2, 9]. Second, interactions between the host and the network adaptor can become complicated. For example, there is a need for flow control between the host and the adaptor, which can be complicated if the host and adaptor do not share memory.

Transport protocol processing (excluding the checksum) is relatively inexpensive. Measurements show that the overhead for optimized implementations of reliable protocols can be as low as 200 instructions [1]. Moving this function outboard is unlikely to be worthwhile, unless several other costs are also reduced as a result, or under special conditions, for example when connecting a dumb device to a network. However, providing support for specific operations such as checksumming can be beneficial. Note that protocol processing code typically needs access to the data only to calculate the checksum, so the decision on where to do protocol processing is largely independent from the design decisions on data movement discussed earlier.

10. Impact on host software

The different forms of hardware support require changes to the traditional protocol stack of Section 2 to be effective. These changes can affect the code in fairly significant ways. For example, a relatively simple feature, such as DMA, turns a simple copy loop into an asynchronous operation that includes descheduling the caller and rescheduling it when the DMA operation is complete. Such a change has a significant impact on the behavior of the network interface.

Performance modifications to the protocol stack typically combine operations that are logically part of different layers, thus violating a layered structuring of the protocol code. The simplest example is the calculation of the TCP checksum. When we calculate the checksum during the copy from application space into system buffers or from host memory to the adaptor, we are executing part of the transport protocol in the socket or datalink layers. Having to optimize across layers to get good performance is no surprise: people have observed before that the layered organization should be viewed as a model, but that for performance reasons, the implementation of the stack has to be more integrated.

Integrating protocol layers does not imply that the protocol stack implementation cannot be modular. For example, the transport protocol checksum can by calculated during the data transfer to the adaptor by having the datalink layer apply the checksum routine as it copies the data and insert the resulting checksum value in a specific location in the transport header. Both the checksum routine and the checksum location in the header can be supplied by the transport layer through a well-defined interface, thus maintaining a clean separation between the transport and datalink layers. A similar technique can be used when the checksum is calculated in hardware during the DMA transfer [12].

Another host software consideration is that in practice, a single host will run applications that use different programming interfaces and possibly different protocols, so the architectural features we presented will not always pay off. In general, the extraneous features will not have a significant impact on the performance. For example, the operation of the adaptor will typically be pipelined with the operations on the host so they do not affect throughput. The features might add latency for short packets, for which performance considerations are very different from those for high throughput.

11. Comparison with other work

Table 4 shows the architectural features of a number of network adaptors for high speed networks (100Mbit/second or more) plus information on the nature of the host software that uses the adaptor; entries marked as - are unspecified. The table is ordered by when the interfaces were described in the literature. In most cases, the architectural decisions concerning DMA, buffering and protocol processing support on the adaptor follow from the intended use of the interface, as discussed in this paper:

- Most systems do not provide outboard protocol processing. Most of those that do, have a special reason: support for multi-media applications for which data flows to or from devices without host involvement (Protocol Engines), moving applications close to the network (Nectar), or support for systems unable to do protocol processing (LANL CBI).
- Outboard buffering is provided for most interfaces that emphasize socket-based communication (HP Medusa and Afterburner, Gigabit Nectar, and LANL CBI), or that make the outboard buffers directly available to applications through a shared buffer interface (Nectar). The systems in Table 4 that do not have outboard buffering provide a small buffer or FIFO to stage data between host memory and the network, as described in Section 2.
- Almost all interfaces provide DMA support for transfers between host memory and the outboard buffer

or queue. The reason is that the network media rates are a high percentage of the available bus bandwidth, so using the bus efficiently is important. On most systems, the software uses a combination of DMA and CPU copy, depending on the circumstances.

For most of these interfaces, the sustainable throughput (if published) is well below the network media rate, mostly as a result of limited memory bandwidth on the host. A meaningful comparison of the throughput is difficult because of differences in conditions that affect performance (packet size, protocol used, ..).

	Outboard Buffering	DMA	Protocol Processing	Transport Checksum	Application Interface	Bus Platform	Network Mbit/sec
Protocol Engines	Yes	Yes	XTP/IP	Yes	Immediate	multiple	up to 200
NAB [6]	No	Yes	VMTP	Yes	Buffered	VMP	100
Nectar [2]	Yes	Both	Yes	Yes	Buffered	VME	100
UltraNet	No	Yes	TP4 TCP	Yes	Immediate	VME/HIPPI	250-800
Bellcore ATM [5]	No	Yes	No	No	-	TurboChannel	622
U of Penn ATM [5]	No	Yes	No	No	-	MicroChannel	155
Fore Systems ATM [3]	No	No	No	No	-	SBus/TC	155
Medusa [5]	Yes	Yes	No	Yes	Immediate	HP Snake	100
Afterburner [8]	Yes	Yes	No	Yes	Immediate	HP Snake	100-800
LANL CBI	Yes	Yes	TCP/IP	Yes	-	HIPPI	800
Gigabit Nectar [12]	Yes	Yes	No	Yes	Immediate	TurboChannel	800
DEC FDDI [9]	No	Yes	No	No	Immediate	TurboChannel	100

Table 4: Overview of high-speed network interfaces

12. Conclusion

The application-to-application throughput over high-speed networks is often limited by communication overhead on the sending and receiving host, so it is critical to make communication processing as efficient as possible. In this paper we showed that the organization of an efficient network interface depends strongly on the external constraints placed on the interface: the nature of the buses and cache on the host, the programming interface used by the application for communication, and the placement of the checksum in the packet. Table 3 shows for example that for the same adaptor hardware, the minimal number of transfers across the bus can differ by a factor of three (one versus three), depending on the constraints. Powerful network adaptors can be used to improve network throughput. We presented the benefits of hardware support on the adaptor by gradually adding features to a minimal adaptor. Not surprisingly, we found that how well features pay off depends again on the hardware and software environment in which the network interface operates. The features and tradeoffs presented in this paper define a design space that should help designers in doing a systematic evaluation of design choices..

Acknowledgements

This paper based on the author's experience in the Nectar and Gigabit Nectar projects. It benefited from many discussions with members of both projects, including Jose Brustoloni, Ming-Jen Chan, Eric Cooper, Jim Hughes, Karl Kleinpaste, BJ Kowalski, HT Kung, Robert Sansom, and Brian Zill. The referees, Adam Beguelin, David Eckhardt, Allan Fisher, Thomas Gross and Dave O'Hallaron provided valuable comments on drafts of the paper.

References

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

2. Eric Cooper, Peter Steenkiste, Robert Sansom, and Brian Zill. Protocol Implementation on the Nectar Communication Processor. Proceedings of the SIGCOMM '90 Symposium on Communications Architectures and Protocols, ACM, Philadelphia, September, 1990, pp. 135-143.

3. Eric Cooper, Onat Menzilcioglu, Robert Sansom, and Francois Bitz. Host Interface Design for ATM LANs. Proceedings of the 16th Conference on Local Computer Networks, IEEE, October, 1991, pp. 247-258.

4. Peter Druschel, Mark B. Abbott, Michael A. Pagels, and Larry L. Peterson. "Network Subsystem Design". *IEEE Network Magazine 7,* 4 (July 1993), 8-17.

5. -. "Special Issue on Host Interfacing". *IEEE Journal on Selected Areas in Communication 11,* 2 (February 1993).

6. Hemant Kanakia and David R. Cheriton. The VMP Network Adaptor Board (NAB): High-Performance Network Communication for Multiprocessors. Proceedings of the SIGCOMM '88 Symposium on Communications Architectures and Protocols, ACM, August, 1988, pp. 175-187.

7. H.T. Kung, Robert Sansom, Steven Schlick, Peter Steenkiste, Matthieu Arnould, Francois J. Bitz, Fred Christianson, Eric C. Cooper, Onat Menzilcioglu, Denise Ombres, and Brian Zill. Network-Based Multicomputers: An Emerging Parallel Architecture. Proceedings of Supercomputing '91, IEEE, Albequerque, November, 1991, pp. 664-673.

8. -. "Special Issue on End-System Support for High-Speed Networks". *IEEE Network Magazine 7,* 4 (July 1993).

9. K.K. Ramakrishnan. "Performance Considerations in Designing Network Interfaces". *IEEE Journal on Selected Areas in Communication 11,* 2 (February 1993), 203-219.

10. Michael Schroeder and Michael Burrows. "Performance of Firefly RPC". *ACM Transactions on Computer Systems 8*, 1 (February 1990), 1-17.

11. Peter Steenkiste. Analyzing Communication Latency using the Nectar Communication Processor. Proceedings of the SIGCOMM '92 Symposium on Communications Architectures and Protocols, ACM, Baltimore, August, 1992, pp. 199-209.

12. 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. Proceedings of the 4th IFIP Conference on High Performance Networks, IFIP, Liege, Belgium, December, 1992, pp. A3 1-16.