A Host Interface Architecture for High-Speed Networks

Peter A. Steenkiste[a], Brian D. Zill[a], H.T. Kung[a], Steven J. Schlick[a], Jim Hughes[b], Bob Kowalski[b], and John Mullaney[b]

[a] School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213-3890, USA

[b] Network Systems Corporation, 7600 Boone Avenue North, Brooklyn Park, MN 55428, USA

Abstract

This paper describes a new host interface architecture for high-speed networks operating at 800 of Mbit/second or higher rates. The architecture is targeted to achieve several 100s of Mbit/second *application-to-application* performance for a wide range of host architectures. The architecture achieves the goal by providing a streamlined execution environment for the entire path between host application and network interface. In particular, a "Communication Accelerator Block" (CAB) is used to minimize data copies, reduce host interrupts, support DMA and hardware checksumming, and control network access.

This host architecture is applicable to a large class of hosts with high-speed I/O busses. Two implementations for the 800 Mbit/second HIPPI network are under development. One is for a distributed-memory supercomputer (iWarp) and the other is for a high-performance workstation (DECstation 5000). We describe and justify both implementations.

Keyword Codes: B.4.1; C.2.1
Keywords: Data Communications Devices; Network Architecture and Design

1 Introduction

Recent advances in network technology have made it feasible to build high-speed networks using links operating at 100s of Mbit/second or higher rates. HIPPI networks based on the ANSI High-Performance Parallel Interface (HIPPI) protocol [1] are an example. HIPPI supports a data rate of 800 Mbit/second or 1.6 Gbit/second and almost all commercially available supercomputers have a HIPPI interface. As a result, HIPPI networks have become popular in supercomputing centers. In addition to HIPPI, there are a number of high-speed network standards in various stages of development by standards bodies. These include ATM (Asynchronous Transfer Mode) [2] and Fibre Channel [3].

As network speeds increase, it is important that host interface speeds increase proportionally, so that applications can benefit from the increased network performance. Several recent developments should simplify the task of building host interfaces that can operate at high rates. First, most computer systems, including many workstations, have I/O busses with raw hardware capacity of 100 MByte/second or more. Second, existing transport protocols, in particular Transmission Control Protocol/Internet Protocol (TCP/IP), can be implemented efficiently [4, 5]. Finally, special-purpose high-speed circuits, such as the AMCC HIPPI chip set, can be used to handle low-level, time-critical network interface operations.

However, these elements do not automatically translate into good network performance for applications. The problem is that the host interface involves several interacting functions such as data movement, protocol processing and the operating system, and it is necessary to take a global, end-to-end view in the design of the network interface to achieve good throughput and latency. Optimizing individual functions is not sufficient.

We have designed a host-network interface architecture optimized to achieve high application-to-application throughput. Our interface architecture is based on a *Communication Accelerator Block (CAB)* that provides support for key communication operations. The CAB is a network interface *architecture* that can be used for a wide range of hosts, as opposed to an *implementation* for a specific host. Two CAB implementations for HIPPI networks are under development. One is for the iWarp parallel machine [6] and the other one is for the DEC workstation using the TURBOchannel bus [7]. These two CAB implementations should allow applications to use a high percentage of the 100 MByte/second available on HIPPI. The interfaces will be used in the context of the Gigabit Nectar testbed at Carnegie Mellon University [8]. The goal of the testbed is to distribute large scientific applications across a number of computers connected by a high-speed network. The network traffic will consist of both small control messages for which latency is important, and large data transfers, for which throughput is critical.

In the remainder of the paper we first discuss the requirements for the host interface (Section 2). We then present the motivation and hardware and software architecture of the CAB-based interface (Section 3) and the design decisions for the two CAB implementations (Sections 4 and 5). We conclude with a comparison with earlier work.

2 Requirements for a Network Interface Design

In local area networks, throughput and latency is typically limited by overhead on the sending and receiving systems, i.e. it is limited by CPU or memory bandwidth resource constraints on the hosts. This means that the efficiency of the host-network interface plays a central role. Consuming fewer CPU and bus cycles will not only make it possible to communicate at higher rates since the communication bottleneck has been reduced, but for the same communication load more cycles will be available for the application. Efficiency is critical both for applications whose only task is communication (e.g. ftp) and for applications that are communication intensive, but for which communication is not the main task.

Since host architecture has a big impact on communication performance, we considered the communication bottlenecks for different classes of computer systems. A first class consists of workstations, currently characterized by one or a few CPUs, and a memory bandwidth of a few 100 MByte/second. Existing network interfaces typically allow these workstations to achieve a throughput of a few MByte/second, without leaving any cycles to the application. Some projects have been successful at improving throughput over FDDI, but these efforts concentrate on achieving up to 100 MBit/second for workstations only [9]. This communication performance is not adequate for many applications [10].

General-purpose supercomputers such as Cray also have a small number of processors accessing a shared memory. They have however a very high memory and computing bandwidth and they have I/O subsystems to manage I/O devices with minimal involvement from the CPU. These resources allow them to communicate at near gigabit rates while using only a fraction of their computing resources [5].

Special-purpose supercomputers such as iWarp [6] and the Connection Machine [11] have a very different architecture. Although these systems have a lot of computing power, the computing cycles are spread out over a large number of relatively slow processors, and they are not suited to support communication over general-purpose networks. A single cell (for iWarp), or a front-end (for CM) can do the protocol processing, but the resulting network performance will match the speed of a single processor, and will not be sufficient for the entire system. The issue is the efficiency of the network interface: can we optimize the interface so that a single processor can manage the network communication for a parallel machine?

Our goal is to define a "Communication Acceleration Block" that can support efficient communication on a variety of architectures. Specifically, this architecture must have the following properties:

1. *High-throughput, while leaving sufficient computing resources to the application.* The goal is to demonstrate that applications on high-performance workstations can achieve several 100 Mbits/second end-to-end bandwidth. It is not acceptable to devote most of the CPU and memory resources of a host to network related activities, so the network interface should use the resources of the host as efficiently as possible, and brute-force solutions that might work for supercomputers should be avoided. The exact performance will depend on the capabilities of the host.

2. *Modular architecture.* The portions of the architecture that depend on specific host busses (such as TURBOchannel) and network interfaces (such as HIPPI), should be contained in separate modules. By replacing these modules, other hosts and networks can be supported. For example, by using different host interfacing modules, the CAB architecture can interface with the TURBOchannel or to an iWarp parallel machine.

3. *Inherently low-cost architecture.* The host interface should cost only a small fraction of the host itself. It is essential that eventually the host interface can be cheaply implemented using ASICs, similar to existing Ethernet or FDDI controller chips. Early implementations may be more expensive, but the interface architecture should be amenable to low-cost ASIC implementation.

4. *Use of standards.* We concentrate on the implementation of the TCP and UDP internet protocols since they are widely used and have been shown to work at high transfer rates [5]. We use UNIX sockets as the primary communication interface for portability reasons. We also want to better understand how protocol and interface features influence the performance and complexity of the host-network interface, and other interfaces that are more appropriate for network-based multicomputer applications will be developed in parallel or on top of sockets.

3 The Host-Network Interface Architecture

Many papers have been published that report measurements of the overheads associated with communicating over networks [12, 4, 13, 14, 15, 16]. Even though it is difficult to compare these results because the measurements are made for different architectures, protocols, communication interfaces, and benchmarks, there is a common pattern: there is no single source of overhead. The time spent on sending and receiving data is distributed over several operations such as

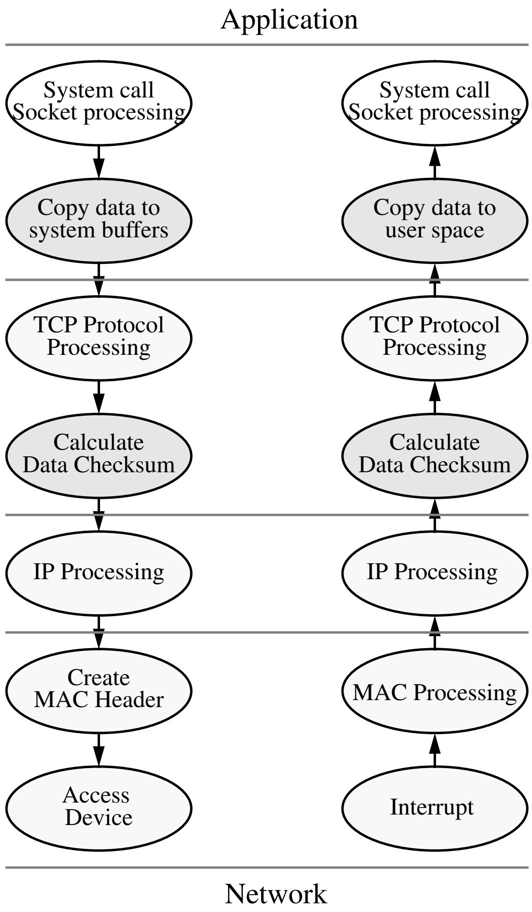

Figure 1: Network data processing overheads

copying data, buffer management, protocol processing, interrupt handling and system calls, and different overheads dominate depending on the circumstances (e.g. packet size). 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.

Figure 1 shows the operations involved in sending and receiving data over a network using the socket interface. These operations fall in different categories. First, there are overheads associated with every application write (socket call – white), and with every packet sent over the network (TCP, IP, physical layer protocol processing and interrupt handling – light grey); these operations involve mainly CPU processing. There is also overhead that scales with the number of bytes sent (copying and checksumming – dark grey); this overhead is largely limited by memory bandwidth. In the remainder of this section we first look at how we can minimize both types of overhead. We then present the CAB architecture, and we describe how the CAB is seen and used by the host.

3.1 Optimizing per-byte operations

As networks get faster, data copying and checksumming will become the dominating overheads, both because the other overheads are amortized over larger packets and because these operations make heavy use of a critical resource: the memory bus. Figure 2 shows the dataflow when sending a message using a traditional host interface; receives follow the inverse

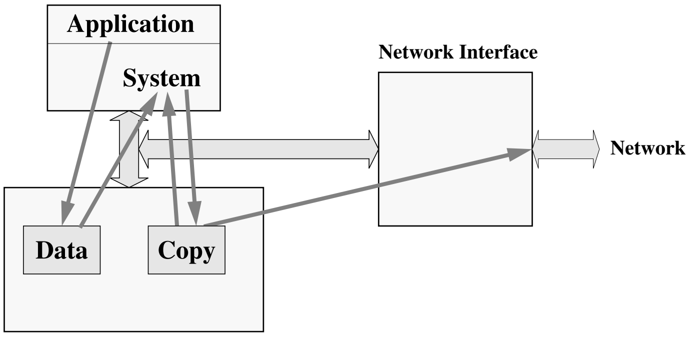

Figure 2: Dataflow in traditional network interface

path. The dashed line is the checksum calculation. There are a total of five bus transfers for every word sent. On some hosts there is an additional CPU copy to move the data between "system buffers" and "device buffers", which results in two more bus transfers.

We can reduce the number of bus transfers by moving the system buffers that are used to buffer the data outboard, as is shown in Figure 3. The checksum is calculated while the data is copied. The number of data transfers has been reduced to three. This interface corresponds to the "WITLESS" interface proposed by Van Jacobson [17]. Besides using the bus more efficiently, outboard buffering also allows packets to be sent over the network at the full media rate, independent of the speed of the internal host bus.

Figure 4 shows how the number of data transfers can be further reduced by using DMA for the data transfer between main memory and the buffers on the CAB. This is the minimum number with the socket interface. Checksumming is still done while copying the data, i.e. checksumming is done in hardware. Besides reducing the load on the bus, DMA has the advantage that it allows the use of burst transfers. This is necessary to get good throughput on today's high-speed I/O busses. For example, the DEC TURBOchannel throughput is about 11.1 MByte/second for single word transfers, but 76.0 MByte/second for 32 word transfers. However, on some systems, DMA adds enough overhead that it is sometimes more attractive to copy and checksum the data using the CPU (see Section 5).

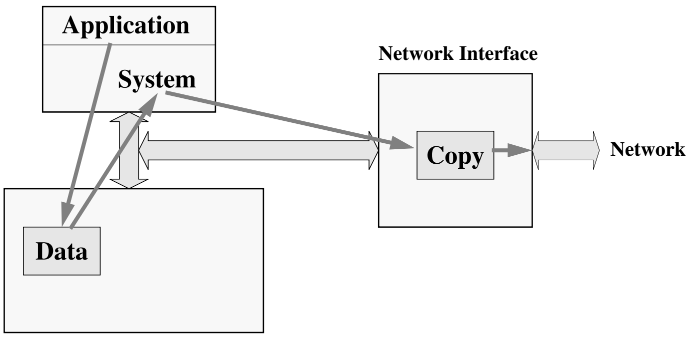

Figure 3: Dataflow in network interface with outboard buffering

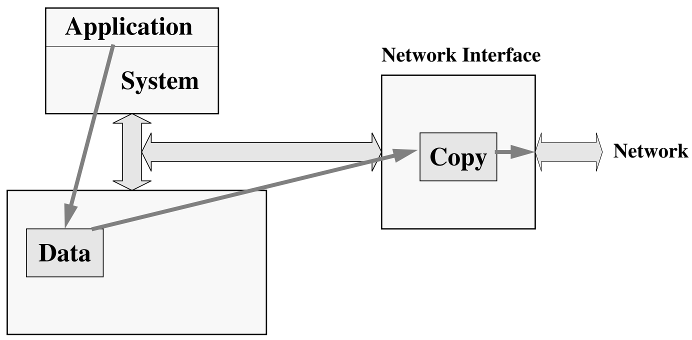

Figure 4: Dataflow in network interface with DMA

3.2 Optimizing per-packet operations

The CAB should have support for Media Access Control (MAC), as is the case for well established network interfaces such as Ethernet and FDDI, so that the host does not have to be involved in negotiating access to the network for every packet. This would be interrupt-intensive (interrupt based interface) or CPU-intensive (based on polling).

The remaining network interface functions on the host are TCP and IP protocol processing, including the creation of the TCP and IP headers. Measurements for optimized protocol implementation show that the combined cost of protocol processing on the send and receive side is about 200 instructions [4], or about 10 microseconds on a 20 MIPS workstation.

The main (likely only) benefit of moving protocol processing outboard is that it potentially frees up cycles for the application. The most obvious drawback is that the network interface becomes more complicated and expensive since it requires a high-performance general-purpose CPU (with matching memory system) for protocol processing. A second drawback is that the host and network interface have to share state and the host-interface pair must be viewed as a multiprocessor. Earlier experiments [18, 16] show that this can make interactions between the host and CAB considerably more complex and expensive compared with a master-slave model. Given these drawbacks and the limited advantages, we decided to perform protocol processing on the host, and to make the CAB a pure slave.

Given the high cost of crossing the I/O bus and of synchronization (e.g. interrupts), the CAB architecture minimizes the number of host-CAB interface interactions and their complexity. The host can request a small set of operations from the CAB, and for each operation, it can specify whether it should be interrupted when the operation is finished. The CAB generates a return tag for every request it finishes, but it only interrupts the host if requested. The host processes all accumulated return tags every time it is interrupted. This limits the number of interrupts to one per user write on transmit, and at most one per packet and one per user read on receive.

3.3 The CAB Architecture

Figure 5 shows a block diagram of the CAB architecture. The CAB consists of a transmit and a receive half. The core of each half is a memory used for outboard buffering of packets (network memory). Each memory has two ports, each running at 100 MByte/second. Network memory can for example be implemented using VRAM, with the serial access port at the network side. Data is transfered between main memory and network memory using system DMA (SDMA) and between network memory and the network using media DMA (MDMA).

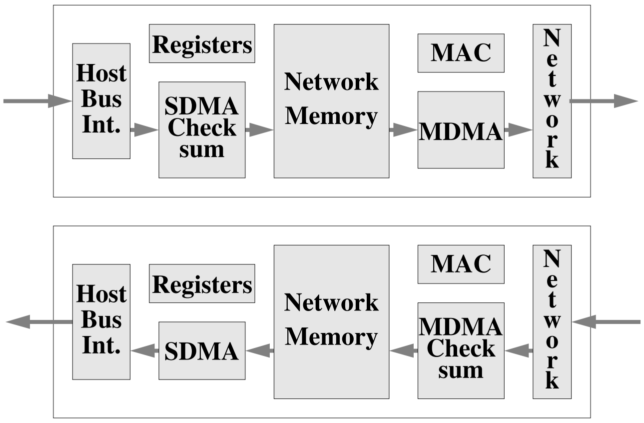

Figure 5: Block diagram generic network interface

The most natural place to calculate the checksum on transmit is while the data is placed on the network. This is however not possible since TCP and UDP place the checksum in the header of the packet. As a result, the 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.

Media access control is performed by hardware on the CAB, under control of the host. We concentrate on MAC support for switch-based networks, specifically HIPPI networks. The simplest MAC algorithm for a switch-based network is to send packets in FIFO order. If the destination is busy, i.e. one or more links between the source and the destination is being used for another connection, the sender waits until the destination becomes free (called camp-on). This simple algorithm 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 has shown that one can utilize at most 58% of the network bandwidth, assuming random traffic [19].

MAC on the CAB is based on multiple "logical channels", queues of packets with different destinations. The CAB attempts to send a packet from each queue in round-robin fashion. If a destination is busy, the CAB moves to the next queue, which holds packets with a different destination. The exact MAC algorithm is controled by the host through retry frequency and timeout parameters for each logical channels. The host can also specify that camp-on should be used after a number of tries to improve the chances that packets to a busy destination will get through (i.e. balance fairness versus throughput).

The register files on both the transmit and receive half of the CAB are used to queue host requests and return tags. The host interface implements the bus protocol for the specific host. Depending on implementation considerations such as the speed of the bus, the transmit and receive halves can have their own bus interface, or they can share a bus interface.

The CAB architecture is a general model for a network interface. Although the details of the host interface, checksum and MAC blocks depend on the specific host, protocol and network, the architecture should apply to a wide range of hosts, protocols and networks. In Sections 4 and 5 we describe implementation for this architecture for the iWarp parallel machine and for the DECstation 5000 workstation, two very different systems.

3.4 Host View of Network Interface

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

Several features of the CAB 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 should be fully formed when they are transfered to the CAB.

To make the most efficient use of this interface, data should be transferred directly from user space to CAB memory and vice-versa. This model is different from that currently found in Berkeley Unix operating systems, where data is channeled through the system's network buffer pool [20]. 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 transfered out of user space. This means that instead of a conventional "layered" protocol stack implementation (Figure 1) where decisions about packet formation are the sole domain of the transport protocol, some of that functionality must now be shared at a higher level.

To illustrate how host software interacts with the CAB hardware in normal usage, we present a walk-through of a typical read and write. To handle a user write, the system first examines the size of the write and other factors to determine how many packets will be needed on the media, and then it 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. In most cases, i.e. if the TCP window is open, a 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 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 read is handled by issuing one or more SDMA operations to copy the data out of the interface memory. The last SDMA operation is flagged to generate an interrupt upon completion so that the user process can be scheduled.

4 The iWarp CAB Implementation

The first implementation of the CAB architecture is for iWarp, a distributed memory parallel computer. Each iWarp cell has a CPU, memory, and four pathways supporting communication with neighboring cells. The cells are interconnected as a torus [6]. An iWarp array communicates with other computer systems by inserting "interface boards" in the back loops of the torus (Figure 6). Interface boards are, besides being linked into the iWarp interconnect, also connected to an external bus or network, which allows them to forward data between iWarp and the outside world.

The iWarp-Nectar interface board, or HIPPI Interface Board (HIB), consists of two iWarp

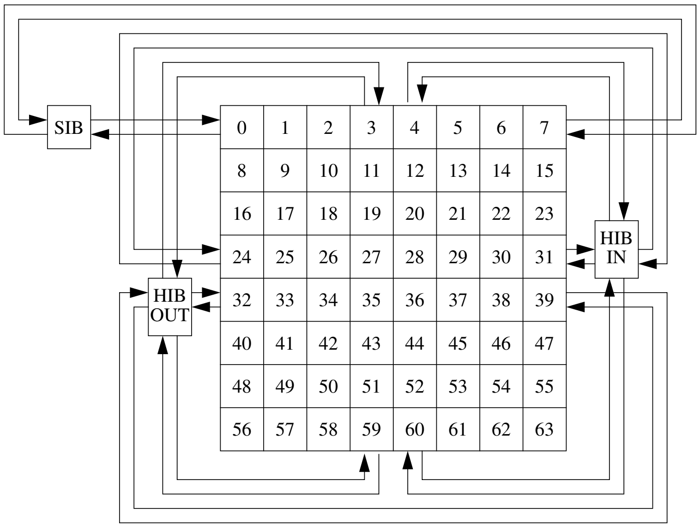

Figure 6: Connecting the network interface to iWarp

interface cells and a CAB (Figure 7). Each interface cell is linked into the iWarp torus independently through four iWarp communication pathways, as is shown in Figure 6 for an 8 by 8 iWarp array. The two iWarp cells play the role of host on the network. They are responsible for TCP and UDP protocol processing and for distributing data to, and collecting data from, the cells in the iWarp array.

To transmit data over the HIPPI network, cells in the iWarp array send data over the iWarp interconnect to the "HIB OUT" interface cell. The interface cell combines the data, stores it in staging memory as a single data stream and issues a write call to invoke the TCP/IP protocol. The TCP/IP protocol stack does the protocol processing, using the CAB to DMA the data from staging memory into network memory, calculate the checksum and transmit the data over the HIPPI network. The inverse process is used to receive data. The HIB can also be used as a raw HIPPI interface, for example to send data to a framebuffer.

The motivation for using two cells on the network interface is to provide enough bandwidth between the network and the iWarp array. An iWarp cell has a bandwidth of 160 MByte/second to its memory system (Figure 7); this bandwidth is shared by the data stream and program execution. Since HIPPI has a bandwidth of 100 MByte/second in each direction, a single iWarp cell would clearly not be able to substain the peak HIPPI bandwidth. A two-cell architecture doubles the available memory bandwidth, but requires that TCP protocol processing is distributed between the transmit and receive cells. A shared memory is provided to simplify the sharing of the TCP state (open connections, open window, unacknowledged data, etc.).

The iWarp Nectar interface physically consists of a transmit and receive board, plus a board that contains the HIPPI interfaces (necessary to provide space for connectors). All boards are VME boards, but only use VME for power and ground. Most of the board space is used for the iWarp components, memory and support chips. Network memory consists of two banks of 2 MBytes of VRAM (can be upgraded to 8 Mbyte each). Each staging memory (see Figure 7) consists of 128 KBytes of dual ported static RAM. The transmit board is currently operational, and the full interface is scheduled for completion in December 1992.

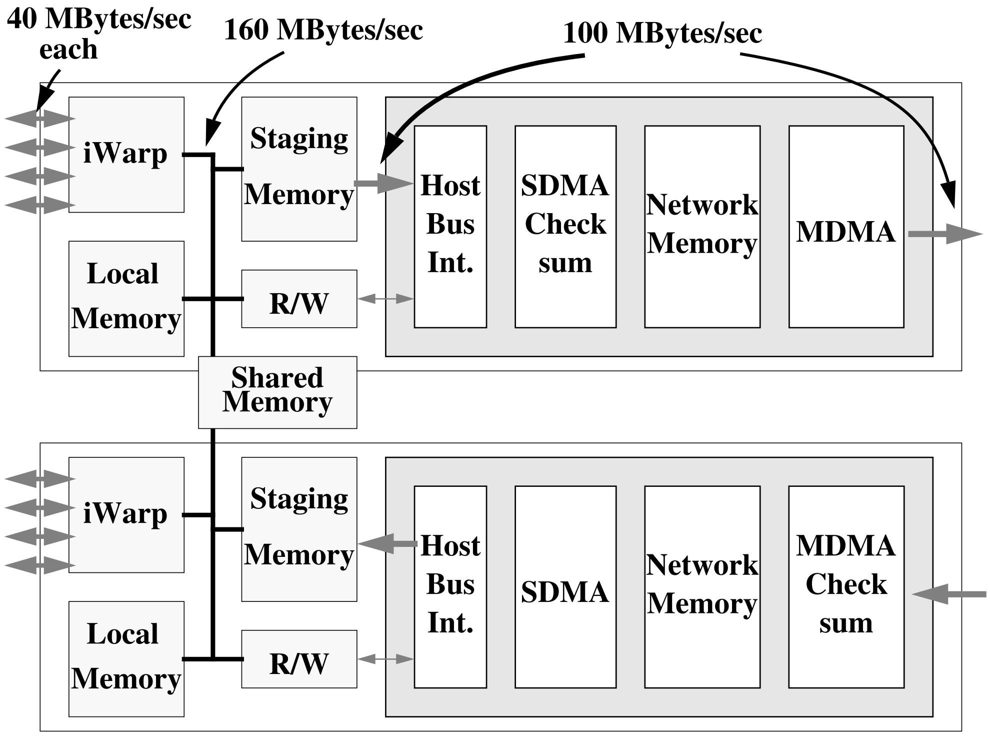

Figure 7: Block diagram iWarp network interface

5 The Workstation CAB Implementation

5.1 Target Machine

The CAB architecture is being implemented for the DEC workstations using the TURBOchannel I/O bus [7]; the initial platform will be the DEC 5000/200. The workstation interface is a single board implementation with a shared bus interface for transmit and receive. The interface is being built using off-the-shelf components and will be operational in Fall 1992.

The DEC workstation was selected as the target for our network interface after a careful comparison of the DEC, HP, IBM and Sun workstations. One interesting result was that all four classes of workstations have similar I/O subsystems: all are based on 100 MByte/second I/O busses that rely heavily on burst transfers for high sustained throughput. TURBOchannel was chosen because it is an open, simple bus architecture and because there is research ongoing at CMU in the operating systems area, mainly based on DEC workstations, that we want to use. Based on this similarity in I/O architecture, we expect that our results for the TURBOchannel should largely carry over to other workstations.

5.2 Processor Cache and Virtual memory

On workstations, the use of DMA to transfer data between network memory and host memory is made more complicated by the presence of a cache and virtual memory. Because of these extra overheads, it might sometimes be more efficient to use the CPU to copy packet between user and system space (programmed I/O – grey arrow in Figure 8).

DMA can create inconsistencies between the cache and main memory, resulting in wrong

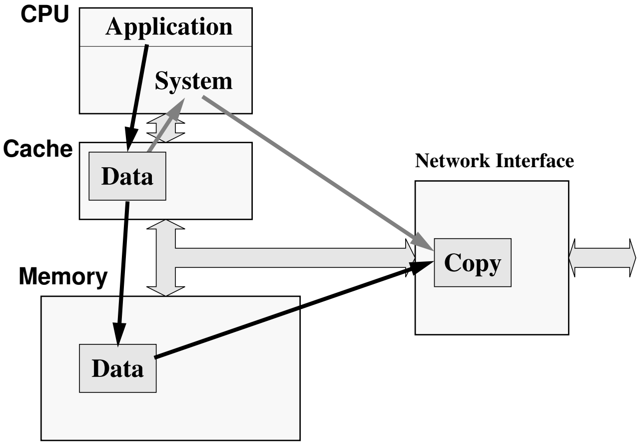

Figure 8: DMA-cache interaction

data being sent over the network or being read by the application. Hosts can avoid this problem by flushing the data to memory before transferring it using DMA on transmit (black arrows in Figure 8), and by invalidating the data in the cache before DMAing on receive. Write-through caches only require cache invalidation on receive, which can be performed in parallel with the data transfer from the device to main memory. For performance reasons, most workstations move towards write-back caches where both cache flushing and invalidation add overhead to the DMA. Fortunately, these operations make efficient use of the bus since they use bursts to transfer cache lines to main memory.

On workstations, user pages have to be wired in memory to insure that they are not paged out while the DMA is in progress. This cost, however, is acceptable and becomes smaller as the CPU speed increases. For example, the combined cost of a page wire and unwire on a 15 MIPS DECstation 5000/200 running Mach 2.6 is 134 microseconds while the time to transfer a 16 KByte page from device to memory is 262 microseconds (using a 64 byte burst size). While this reduces the throughput from 62.4 Mbytes/second to 41.3 Mbytes/second, it is still much faster than the maximum CPU copy rate (11.1 Mbytes/second). This is one area where moving away from sockets can help. If the application builds messages in wired-down buffers (of limited size), the overhead is eliminated. This is similar to mailboxes in Nectar [18].

Figure 9 compares the transmit and receive throughput that can be obtained across Turbo Channel on the DEC 5000/200 with programmed I/O and with DMA using a block size of 16 words. The overheads for cache invalidation (the DEC 5000/200 has a write-through cache) and page wiring and unwiring have been included. For small read and write sizes, programmed I/O is faster. The exact crossover point between programmed I/O and DMA is hard to calculculate, because it depends not only on the hardware features of the system, but also on how much data is in the cache at the time of the read or write and the cost of cache pollution.

DMA is faster, even when considering the high overheads, because it can use burst transfers that make much more efficient use of the bus than the single word transfers used in programmed I/O. On other workstation architectures (such as the HP Snake and the IBM RS/6000) the CPU can also burst data across the bus. This functionality is intended for use with graphics devices, but can also be used for the network interface. Although the I/O bus throughput that can be obtained using the CPU is typically still lower than the raw DMA rate, it makes programmed I/O more attractive, and it increases the crossover point between the two.

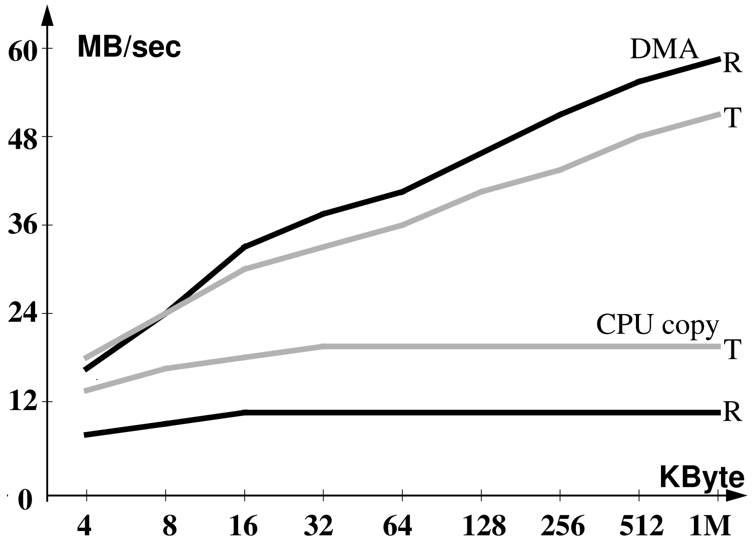

Figure 9: Transmit (T) and receive (R) transfer rates across TURBOchannel

5.3 Performance Estimates

To validate the design of the CAB for workstations, we traced where the time is spent when sending and receiving data on a DECstation 5000/200 running Mach 2.6, using existing interfaces to Ethernet and FDDI. The networking code consists primarily of Tahoe BSD TCP/IP code with an optimized checksum calculation routine [21]. It does not include optimizations such as header prediction, but we are in the process of including these. We use a special TURBOchannel-based timer board with a 40 nanosecond resolution clock to measure the overhead of different network related operations.

Using these measurements, we constructed graphs that show how much of the host memory bandwidth, and indirectly of the CPU cycles, is consumed by the different communication-related operations, and how many cycles are left for the application. Figure 10 shows the result for communication over FDDI. The horizontal axis shows the throughput and the vertical axis shows the percentage of the DS5000/200 CPU that is left for the application for a given throughput. We observe for example that using 100 % of the processor we can only drive FDDI at about 21 Mbits/second assuming 4500-byte packets. This result is in line with actual measurements. It is clear that even after optimizing the TCP protocol processing, the existing network interface will not allow us to communicate at the full FDDI media rate, let alone HIPPI rates.

Based on these measurements we can estimate the performance for a HIPPI network interface based on the CAB architecture. The result is shown in Figure 11. As expected, network throughput is limited by the bandwidth of the memory bus (peak of 100MByte/second). The graph shows that we should be be able to achieve about 300 Mbit/second throughput. This is 35 % of the total memory bandwidth available on the DEC 5000/200. These estimates do not include the overhead of wiring and unwiring pages (should reduce throughput), or optimizations of TCP/IP (should improve the results).

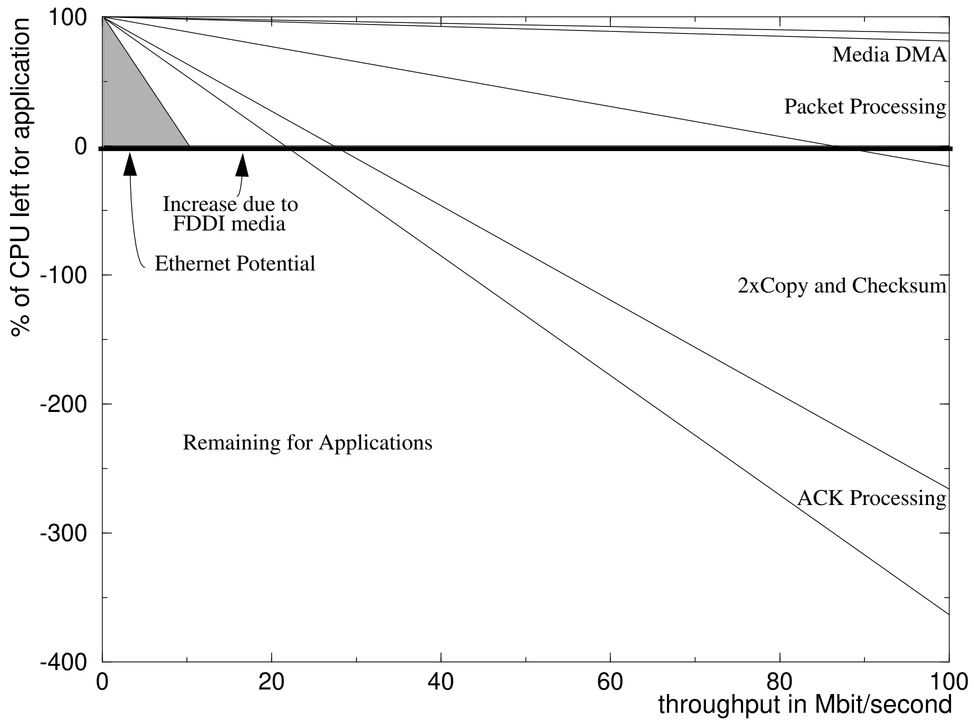

Figure 10: Overhead for Communication over FDDI

6 Comparison with earlier work

Several groups have worked on host-network interfaces that provide support for fast communication. The effort closest to ours is the “WITLESS” interface proposed by Van Jacobson (see Section 3). The main difference between the two approaches is that we plan to use DMA for large transfers, since it allows us to use the I/O bus more efficiently. This is important if the goal is to achieve throughputs of several 100 MBit/second. An interface based on this architecture, but with hardware support for checksumming, was built for the HP workstatation [9].

Several other efforts have moved at least some of the protocol processing outboard. The VMP network adaptor board (VMP-NAB) [22] supports the VMP request-response protocol and splits protocol processing between the host and the NAB. The host provides the NAB with a list of requests it is willing to accept, and responses it is expecting. This allows the NAB to process incoming packets independent from the host, and store them in preallocated buffers. This reduces the overhead on the host, but the network interface has to be able to interpret packet headers.

The Protocol Engine Inc. network interface [23] uses custom chips to do outboard protocol processing for XTP, TP4 and TCP/IP. One of the interface architectures has “intelligence” on board to manage multi-media communication independently from the host.

The Crossbar Interface (CBI) being developed by LANL is a store-and-forward protocol processor for switch-based networks based on HIPPI. It can be used as a Host Interface for systems that are unable to do protocol processing (such as the CM2) and as a Routing Interface between switches. It is intended to support high throughput for very large packets.

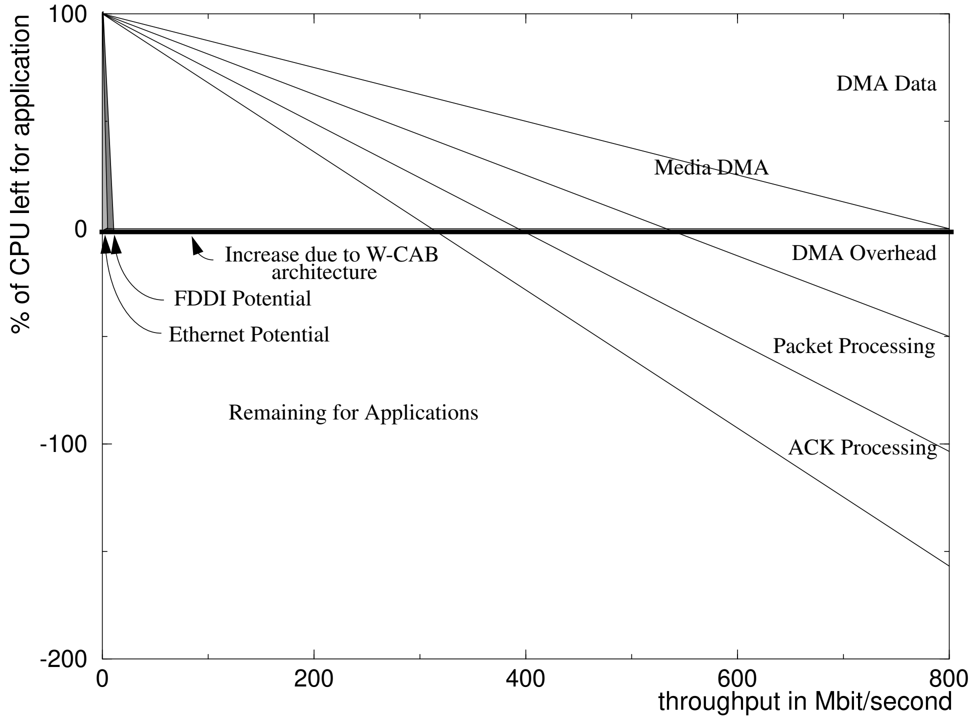

Figure 11: Estimated Overhead for Communication over HIPPI using CAB

The Ultra network interface does outboard protocol processing for TCP, TP4, and Ultra native mode [24]. It requires full synchronization between sender and receiver since the data is transfered directly from the user space on the sender to user space on the receiver, without any buffering on the network interface. For many applications, some buffering to decouple the sender and receiver is desirable. Ultra also had to modify the TCP/IP packet format by placing the checksum in a trailer.

The Nectar prototype system [25] provides a powerful outboard protocol engine. Besides performing protocol processing, it could also execute application code. By moving protocol processing outboard, communication was made very inexpensive, and application code on the network interface could access the network with very low latency, since it bypassed the slow I/O bus (a VME bus). Since Gigabit Nectar uses a much faster I/O bus and has been optimized for efficient protocol processing, protocol processing no longer has to be performed outboard to achieve efficient communication.

The current Connection Machines HIPPI interface has the problem that it does not allow protocol processing, i.e. only raw HIPPI can be used. Furthermore, all communication has to be managed by the front end, which communicates with the HIPPI interface over an Ethernet. Several organizations [26, 27] have been successful at using the interface to distributed applications across the Cray and CM, by implementing an RPC package that allows applications to set up communication on the HIPPI interface. We hope that the architecture used for the iWarp network interface will support both protocol processing and a more straightforward way of managing I/O.

7 Conclusion

The CAB network interface architecture describes in this papers provides several features that support efficient communication over high-speed networks:

1. *Efficient use of memory and bus bandwidth.* By providing outboard buffering and DMA, the CAB reduces the number of data transfers over the host memory bus. This is very different from outboard protocol processing [23].

2. *Free CPU cycles for applications.* By performing data checksums and media access control in hardware, the CAB frees the CPU for other tasks.

3. *Reduce interrupts.* By limiting the number of interrupts from the CAB to one per user write on transmit and one per packet and at most one per read on user read, host performance is no longer bounded by interrupt handling speed.

The two CAB implementations, for iWarp and DECstation 5000, described in this paper will be operational soon. There are a number of engineering applications which are being prepared to make serious use of these implementations.

8 Acknowledgements

The research was supported in part by the Defense Advanced Research Projects Agency (DOD) and monitored by DARPA/CMO under Contract MDA972-90-C-0035. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or of the U.S. Government.

9 References

1 ANSI, "High-Performance Parallel Interface - Mechanical, Electrical and Signalling Protocol Specification (HIPPI-PH)." ANSI X3.183-1991, 1991.

2 M. de Prycker, "Asynchronous Transfer Mode." Ellis Harwood, 1991.

3 ANSI, "Fibre Channel Physical and Signaling Interface (FC-PH)." Working draft proposed American National Standard for Information Systems, 1992.

4 D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An Analysis of TCP Processing Overhead," *IEEE Communications Magazine*, vol. 27, pp. 23–29, June 1989.

5 A. Nicholson, J. Golio, D. A. Borman, J. Young, and W. Roiger, "High Speed Networking at Cray Research," *Computer Communication Review*, vol. 21, pp. 99–110, January 1991.

6 S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. S. Tseng, J. Sutton, J. Urbanski, and J. Webb, "iWarp: An Integrated Solution to High-Speed Parallel Computing," in *Proceedings of Supercomputing '88*, (Orlando, Florida), pp. 330–339, IEEE Computer Society and ACM SIGARCH, November 1988.

7 DEC, "TURBOchannel Overview," April 1990.

8 Report, "Gigabit Network Testbeds," *IEEE Computer*, vol. 23, pp. 77–80, September 1990.

9 J. Lumley, "A High-Throughput Network Interface to a RISC Workstation," in *Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, IEEE, February 1992.

10 H. Kung, P. Steenkiste, M. Gubitoso, and M. Khaira, "Parallelizing a New Class of Large Applications over High-Speed Networks," in *Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 167–177, ACM, April 1991.

11 L. Tucker and G. Robertson, "Architecture and Applications of the Connection Machine," *IEEE Computer*, vol. 21, pp. 26–38, August 1988.

12 L.-F. Cabrera, E. Hunter, M. J. Karels, and D. A. Mosher, "User-Process Communication Performance in Networks of Computers," *IEEE Transactions on Software Engineering*, vol. 14, pp. 38–53, January 1988.

13 N. C. Hutchinson and L. L. Peterson, "Implementing Protocols in the x-kernel," Tech. Rep. 89-1, University of Arizona, January 1989.

14 S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren, "Amoeba: A Distributed Operating System for the 1990s," *IEEE Computer*, vol. 23, pp. 44–53, May 1990.

15 M. Schroeder and M. Burrows, "Performance of Firefly RPC," *ACM Trans. Computer Systems*, vol. 8, pp. 1–17, February 1990.

16 P. Steenkiste, "Analyzing Communication Latency using the Nectar Communication Processor," in *Proceedings of the SIGCOMM '92 Symposium on Communications Architectures and Protocols*, (Baltimore), pp. 199–209, ACM, August 1992.

17 V. Jacobson, "Efficient Protocol Implementation." ACM '90 SIGCOMM tutorial, September 1990.

18 E. Cooper, P. Steenkiste, R. Sansom, and B. Zill, "Protocol Implementation on the Nectar Communication Processor," in *Proceedings of the SIGCOMM '90 Symposium on Communications Architectures and Protocols*, (Philadelphia), pp. 135–143, ACM, September 1990.

19 M. G. Hluchyj and M. Karol, "Queueing in High-Performance Packet Switching," *IEEE Journal on Selected Areas in Communication*, vol. 6, pp. 1587–1597, December 1988.

20 S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, "The Design and Implementation of the 4.3BSD UNIX Operating System." Addison-Wesley, 1989.

21 R. Braden, D. Borman, and C. Partridge, "Computing the Internet Checksum," *Computer Communication Review*, vol. 19, pp. 86–94, April 1989.

22 H. Kanakia and D. R. Cheriton, "The VMP Network Adaptor Board (NAB): High-Performance Network Communication for Multiprocessors," in *Proceedings of the SIGCOMM '88 Symposium on Communications Architectures and Protocols*, pp. 175–187, ACM, August 1988.

23 G. Chesson, "Protocol Engine Design," in *Proceedings of the Summer 1987 USENIX Conference*, pp. 209–215, June 1987.

24 B. Beach, "UltraNet: An Architecture for Gigabit Networking," in *Proceedings of the 15th Conference on Local Area Networks*, pp. 232–248, IEEE, September 1990.

25 E. A. Arnould, F. J. Bitz, E. C. Cooper, H. T. Kung, R. D. Sansom, and P. A. Steenkiste, "The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers," in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 205–216, ACM, April 1989.

26 J. Mahdavi, G. L. Huntoon, and M. B. Mathis, "Deployment of a HIPPI-based Distributed Supercomputing Environment at the Pittsburgh Supercomputing Center," in *International Parallel Processing Symposium*, (Los Angeles), IEEE, April 1992.

27 R. J. Vetter, D. H. C. Du, and A. E. Klietz, "Network Supercomputing," *IEEE Network Magazine*, vol. 6, pp. 38–44, May 1992.