1 Introduction

Recent advances in network technology have made it feasible to build high-speed networks using links operating at several 100s of Mbit/second. 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 use, including ATM (Asynchronous Transfer Mode) [18] and Fibre Channel [2].

As network speeds increase, it is important that host interface speeds increase proportionally, so that applications can benefit from the increased network performance. 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 [14, 44, 11], 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.

We have designed a host-network interface architecture optimized to achieve high application-to-application throughput for applications using the socket application programming interface (API) and the internet communication protocols. 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 have been built by Network Systems Corporation (NSC), our competitively selected industrial partner. The first implementation is for the iWarp parallel machine [6] and the other one is for the DEC workstation using the TURBOchannel bus [19]. The interfaces have been used in the Gigabit Nectar testbed at Carnegie Mellon University [38]. The testbed is used to distribute large scientific applications across a variety of computer systems connected by a high-speed network.

The remainder of this report is organized as follows. We first present the hardware and software architecture of the CAB-based interface (Section 2). We then describe the implementation for the workstation (Section 3) and for iWarp (Section 4) host interfaces and discuss their performance. Finally, we discuss some of the applications that used the Gigabit Nectar interfaces in Section 5.

2 The Host-Network Interface Architecture

Many papers have been published that report measurements of the overheads associated with communicating over networks [12, 14, 24, 34, 39, 44]. 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 including per-packet operations (protocol processing, interrupt handling, buffer management), and per-byte operations (copying data, checksumming). As networks get faster, data copying and checksumming will be-

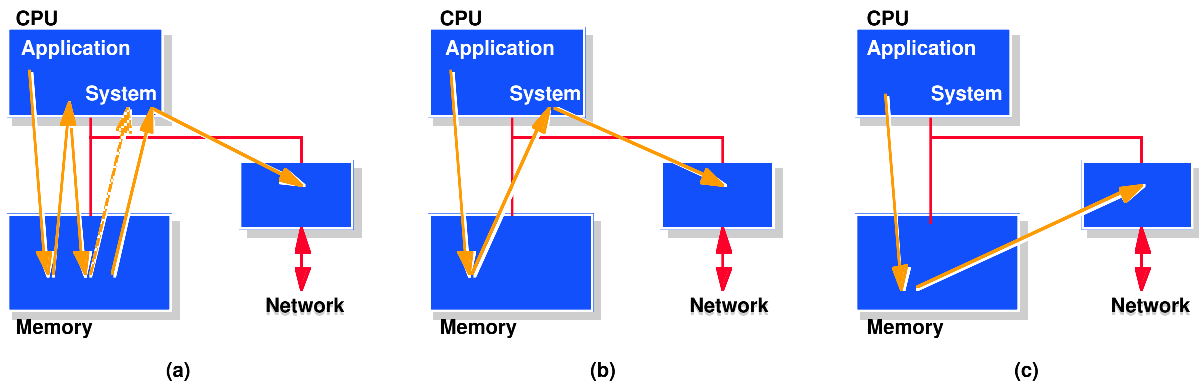

Figure 1: Dataflow in (a) a traditional network interface, (b) a network interface with outboard buffering and (c) a network interface using DMA

come 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.

In the remainder of this section we first discuss how we reduce communication overhead for applications that use BSD sockets and internet protocols, and we then discuss how the CAB architecture supports these optimizations. Using a widely used API (sockets) supports application portability, while using standard communication protocols (TCP and UDP over IP) allows interoperability. Finally, we look at the effect of using different APIs and protocols.

2.1 Optimizing communication

Figure 1(a) shows the dataflow when sending a message using a traditional host interface; receives follow the inverse path. The data is first copied from the user address space to kernel buffers. This copy is needed to implement the *copy semantics* of the socket interface: when the send call returns, the application can safely overwrite the data. The next per-byte operation is the TCP or UDP checksum calculation (dashed line), and finally, the data is copied to the network device. This adds up to a total of five bus transfers for every word sent. 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 1(b). The checksum is calculated while the data is copied. The number of data transfers has been reduced to two. This interface corresponds to the "WITLESS" interface proposed by Van Jacobson [26, 17]. Figure 1(c) shows how the number of data transfers can be further reduced to one by using DMA for the data transfer between host memory and the buffers on the 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.

The per-packet operations include interrupt handling, TCP and IP protocol processing, and buffer management. Protocol processing is often held responsible for limiting communication throughput. Measurements for optimized protocol implementation show that the combined cost of protocol processing on the send and receive side can be as low as 200 instructions [14]. Moreover,

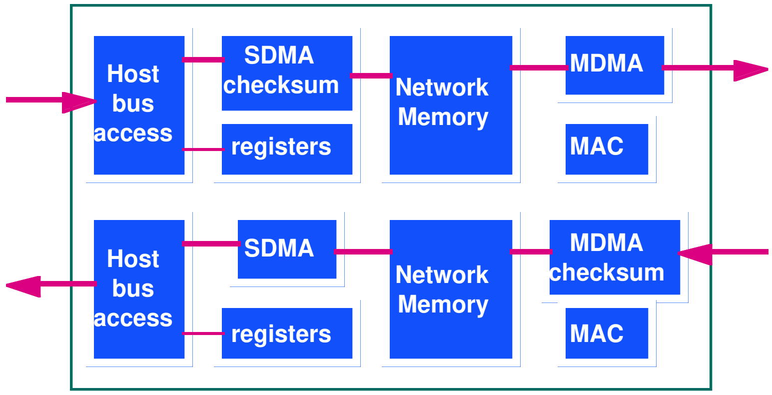

Figure 2: CAB block diagram

moving these tasks outboard can be very complex, so they are performed by the host. Crossing the I/O bus and synchronization (e.g. interrupts) can be quite expensive, so the CAB architecture minimizes the number of host-adapter interactions. For each operation the host requests from the CAB, it can specify whether an interrupt is needed when the operation is finished. This convention allows the host to limit the number of interrupts to one per user write on transmit, and at most one per packet and one per user read on receive.

2.2 Adapter architecture

Figure 2 shows a block diagram of a Communication Acceleration Block (CAB) that supports the optimizations discussed above. It 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 MBytesecond. Data is transferred between host memory and network memory using system DMA (SDMA) and between network memory and the network using media DMA (MDMA). The SDMA engines have a scatter/gather capability so it can collect the packet header and data from different buffers and user data from multiple VM pages that are not adjacent in 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 [21]. The simplest MAC algorithm for a switch-based network is to send packets in FIFO order, but this does not make good use of the network bandwidth because of Head of Line (HOL) blocking. Analysis shows that one can utilize at most 58% of the network bandwidth, assuming random traffic [23, 33]. The CAB uses multiple "logical channels", queues of packets with different destinations, to get around this problem [47].

		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	programmed IO
PIO_C	(with checksum)
DMA	direct memory access
DMA_C	(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

2.3 Host view

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

2.4 Host interface taxonomy

BSD sockets and internet protocols are only one of many ways in which applications can communicate over networks. [46] presents a taxonomy of host interfaces as a function of three parameters:

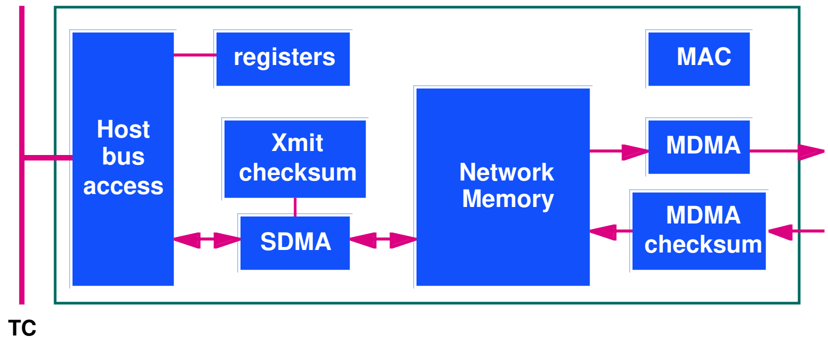

Figure 3: Workstations CAB adapter architecture

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 adapter. 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 rows and columns in Table 1 clearly show some of the tradeoff in the area of host interface design. First, in many scenarios (rows in the table), it is possible to improve performance by making the adapter hardware more complex. Second, for many adapter architectures (columns in the table), applications that use an API with copy semantics will see less efficient communication than applications that use an API based on shared buffers, which is harder to use. The CAB architecture (last column) is the most complex architecture (highest degree of hardware support), but it is also the only architecture that supports all API/protocol combinations with a single DMA transfer across the memory bus (Figure 1c).

3 The workstation interface

The CAB architecture was implemented using off-the-shelf components by NSC for DEC workstations with a TURBOchannel IO bus [19]. The workstation CAB uses a single memory for both incoming and outgoing packets and a single SDMA engine (Figure 3). Network memory is im-

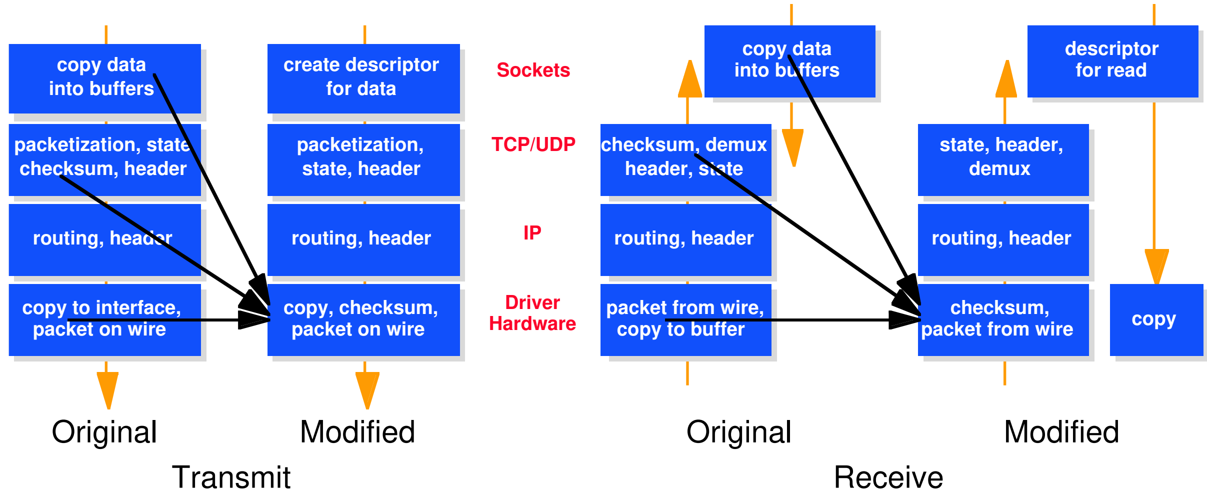

Figure 4: Software architecture

plemented using DRAM, and the DMA engines are controlled using a 29K microprocessor. The workstation CAB sits in a separate box that connects to a TURBOchannel paddle card. While the CAB hardware is designed for bandwidths up to 300 Mbit/second, the microcode currently limits throughput to about 200 Mbit/second (Section 3.3).

3.1 A single-copy protocol stack for BSD

To make efficient use of the CAB, data should be transferred directly from user space to CAB memory and vice-versa, and this model is different from that found in Berkeley Unix operating systems, where data is channeled through the system's network buffer pool [32]. 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. Formating 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 4 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.

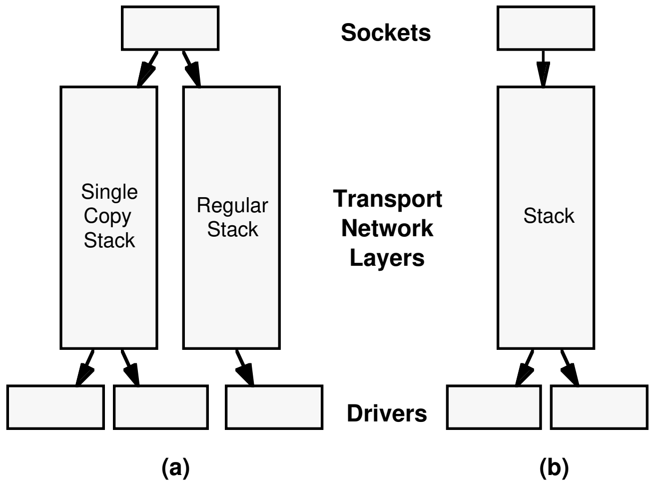

Figure 5: Single versus multiple stacks

3.2 Implementation in DEC OSF/1

This software architecture was implemented in a Net2 BSD protocol stack, as it exists in DEC OSF/1 v2.0 [27]. The single copy path can be added as a separate stack, or it can be merged with the traditional multi-copy stack (Figure 5). Adding the single-copy path as a second, separate stack (Figure 5a), is more complicated since it raises the issue of what stack to use for each read/write and each incoming packet, and it also complicates operations that involve both stacks, such as IP routing. For this reason, we added a single-copy path to the existing protocol stack (Figure 5b).

With a single stack implementation, data can flow through the protocol stack in three different formats: data in kernel buffers, data in user space, and data in outboard buffers. 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). The new mbuf types include new data structures holding information on the checksum location, the task that issued the read or write (used for notification), and location of data in the user's address space or the outboard memory.

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. On transmit, the

checksum routines includes information about the location and offset of the checksum in the packet descriptor so that the CAB can calculate the checksum and insert it in the TCP/IP header. On receive, the checksum routine uses the checksum that was calculated by the CAB hardware. descriptor

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.

The CAB uses DMA to achieve good efficiency on the data transfer between the network memory and host memory. DMA devices typically transfer data between kernel buffers and the device, but with the single-copy stack, data is transferred directly between user space and the device. This means that user buffers have to be pinned and remapped before they can be accessed by the CAB DMA engines. As a result of this overhead, it is more efficient to use the original stack for small reads and writes, since it DMAs to and from kernel buffers, which is more efficient, and also allows tcp to coalesce data.

Besides user-level applications communicating through the CAB, we have to consider two more important scenarios:

- 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 6 shows the different paths through the protocol stack. The single-copy path described described in this section is shown in black. Given the nature of the changes to the protocol stack, in-kernel applications communicating through existing interfaces are not affected (thick grey arrow in Figure 6) since they use "regular" mbufs, which are still supported.

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 6) as a result of the new mbuf types. 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. Interoperability of the different code segments is maintained by doing transformations on the data representation at the module boundaries. For example, outboard data (M_WCAB mbuf) is DMAed into kernel buffers (regular mbufs) before it is passed into a file server.

3.3 Performance

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

3.3.1 Design of experiments

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

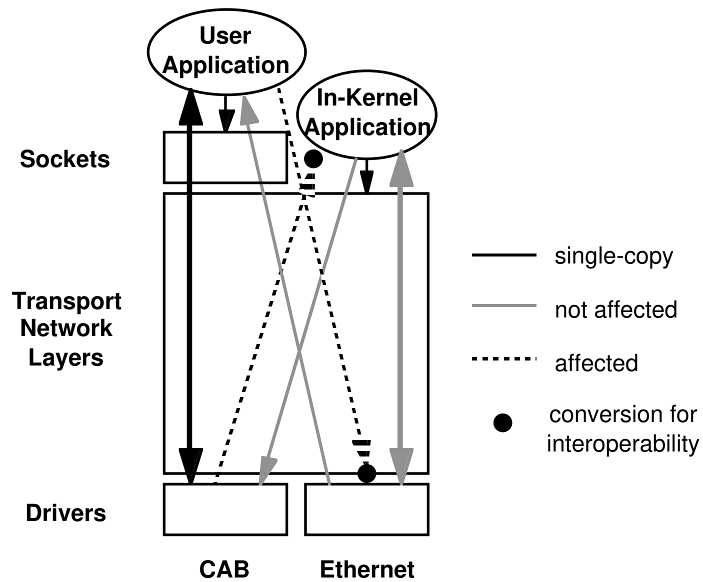

Figure 6: Paths through the protocol stack

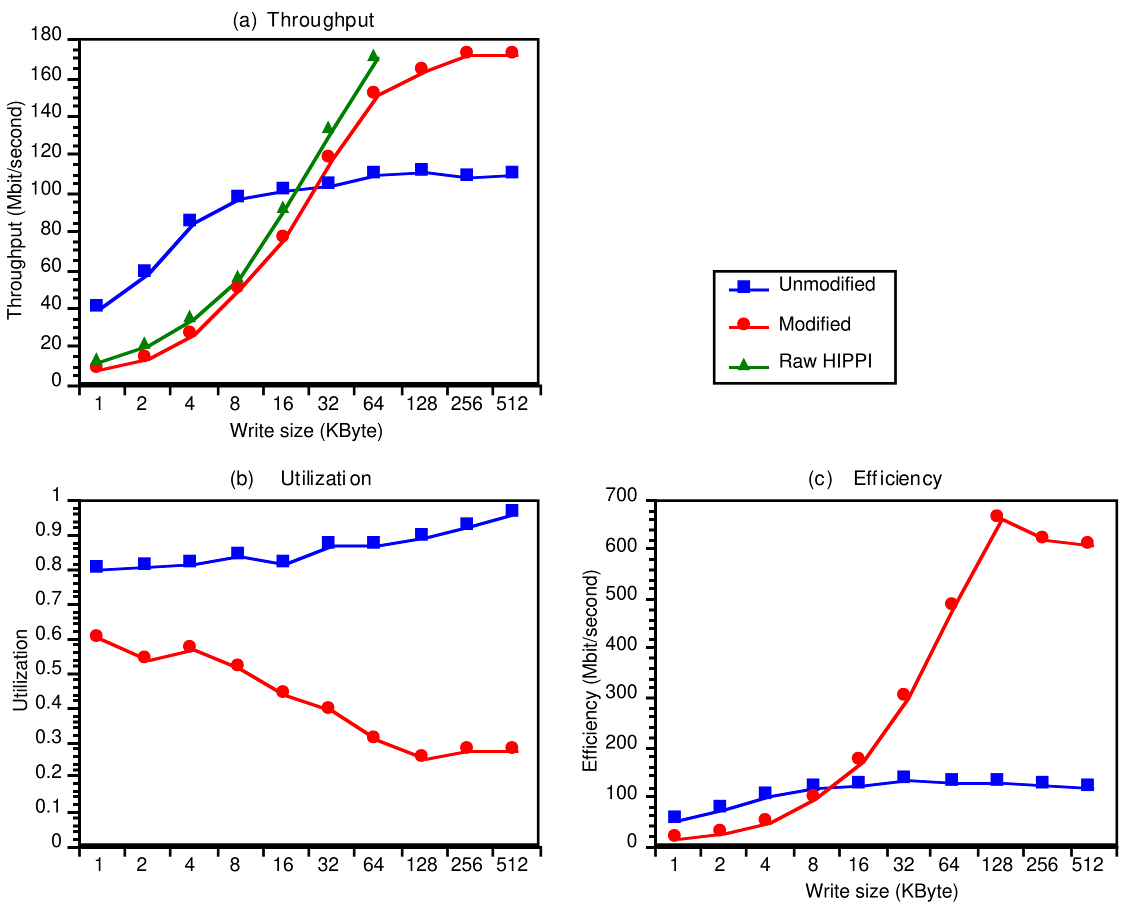

Figure 7: Throughput, utilization and efficiency as a function of read/write size

window scaling [7]. The network device used is the CAB [47] and the Maximum Transmission

Unit (MTU) is 64 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 and user-level applications communicating through the CAB.

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

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:

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

3.3.2 Experimental results

Figures 7(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 7(a) includes the throughput for raw HIPPI reads and writes. The raw HIPPI throughput test generates well-formed packets that can be handled efficiently by the microcode, so the raw HIPPI results represent the highest throughput one can expect for a given packet size. 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 3.2) 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 modified stack has a higher throughput: the peak throughput is 172 Mbit/second for the modified stack versus 110 Mbit/second for the unmodified stack. The utilization measurements shows that the modified stack uses fewer CPU cycles to provide the higher throughput. The unmodified stack uses over 90% of the CPU, i.e. data copying and checksumming by the CPU limits throughput, while the modified uses only 25% of the CPU, i.e. even at 170 Mbit/second it still leaves many CPU cycles for use by applications.

To better evaluate the overhead we define the communication *efficiency* as how many Mbit/second of communication can be supported if the full CPU were utilized for communication, i.e. the ratio of the throughput and efficiency graphs in Figure 7(a) and (b). Figure 7(c) shows the efficiency for

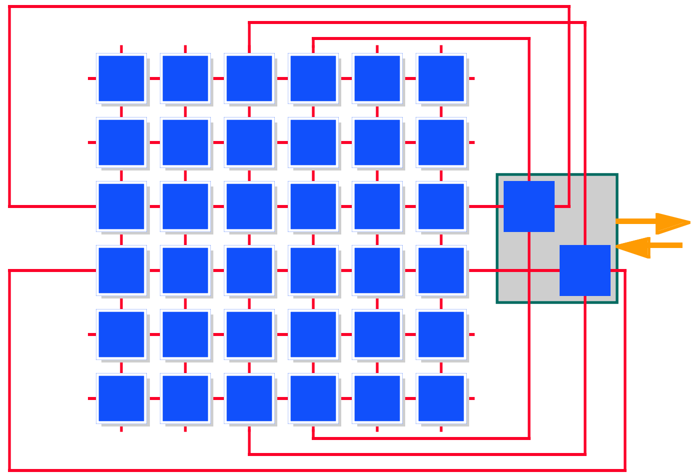

Figure 8: Connection of HIPPI network interface to iWarp distributed-memory system

the both implementations of the stack. We see that the single-copy stack is more than five times more efficient than the unmodified stack for large writes, but less efficient for small writes. The cross over point is between 8 and 16 KByte, indicating that the single-copy stack will pay off 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 adapter.

4 The iWarp interface

The most powerful supercomputers today are distributed-memory systems that link a large number of workstation-class processors together using a high performance interconnect. Their success can be attributed to the fact that they are inherently scalable and provide relatively inexpensive computing cycles compared with traditional supercomputers. However, while distributed-memory systems are effective compute engines, network IO has proven to be a problem. The reason is that network IO is typically supported through an IO node with the same computational power of one of the compute nodes, i.e. a workstation-class node has to support network IO for a supercomputer. The IO architecture used for the HIPPI interface for the iWarp system [6] relies on a careful distribution of the network functions, e.g. protocol processing, data formating and connection management, between the distributed-memory system and the network interface to achieve high-bandwidth network IO.

4.1 IO architecture

Distributed-memory systems communicate over a networks through a *network interface* node connected both to the external network (e.g. HIPPI) and the internal interconnect of the system (Figure 8). The role of the network interface is to forward data between the internal and external network. The sequential network interface tends to become a bottleneck.

In the iWarp HIPPI interface [45], these problems are addressed by mapping each task onto the subsystem that is the most appropriate for it (network interface or distributed memory system):

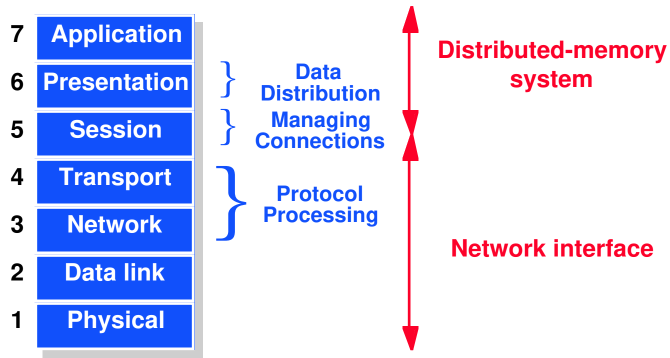

Figure 9: Mapping of protocol stack

1. Communication protocol processing (transport and network layers) generally does not parallelize well. We map this task to the network interface, and use the CAB architecture to provide hardware support for time-critical tasks (Section 4.2).

2. Managing connections between the distributed-memory system and the outside world through the network interface translates into a problem of allocating resources in the system (e.g. link bandwidth, buffer space, ..). This task is mapped onto the streams software (Section 4.3).

3. Data sent or received over the network is typically distributed over the private memories of the nodes. This means that the communication software has to perform scatter or gather operations as part of the IO operation [20]. In our IO architecture we map the task of combining/distributing the data, onto the distributed-memory system (Section 4.4).

Note that these functions correspond to the transport, network, session and presentation layers of the OSI network model (Figure 9).

Our implementation shows that this architecture is very effective. We have measured sustained throughputs of 55 MBytes/second for simple applications (e.g. displaying images on a HIPPI framebuffer), and 40 MBytes/second for complex applications that send data to the PSC Cray C90.

4.2 iWarp CAB

Since protocol processing does not parallelize well, this task is performed by the network interface. Figure 10 shows the architecture of the iWarp-HIPPI network interface, or HIPPI Interface Board (HIB). It consists of two iWarp processor and a CAB. The iWarp processors link the network interface into the iWarp torus and perform protocol processing, while the CAB provides support for critical protocol processing operation: data transfer, checksumming and buffering (Section 2). The operation of the network interface is similar to that of a sequential system, except that the data source and sink is the distributed-memory system, and not the and not the memory of the iWarp processors on the interface.

Other distributed-memory systems also rely on the network interface to perform protocol processing. While we run a relatively standard TCP/IP implementation on the network interface, others use outboard protocol processing (e.g. [48]), or a customized protocol implementation relying heavily on hardware support (e.g. [43]).

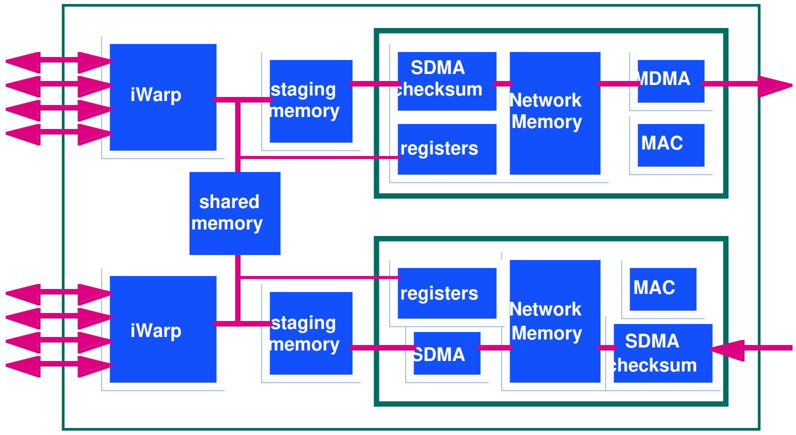

Figure 10: iWarp HIPPI interface architecture

The architecture uses two iWarp processors instead of one because of data bandwidth requirements. The critical resource in the architecture is the memory bus of the iWarp processors since all data that is sent/received has to flow over it, and the bus is also used for program and local data accesses. Using two iWarp processors instead of one doubles the bandwidth available for these operations from 160KByte/sec to 320KByte/sec.

The main role of the staging memories in the architecture is to efficiently gather data coming from the different iWarp buses (transmit) or to scatter data (receive). They are implemented as dual-port RAMs with a bandwidth of 160MByte/second on the iWarp side and 100MByte/second on the network side. Since the DMA engines on the iWarp chip interleave the data on the iWarp memory bus in small blocks (8 bytes), it is necessary to use static RAM to achieve high throughput. As a result, the staging memories are small: 128KByte for each direction.

The implementation of the TCP/IP protocol stack for iWarp differs from a traditional workstation implementation in a number of ways. The most obvious difference is that processing is distributed over two processors, so the transmit and receive components of TCP/IP have to be separated. A shared memory (Figure 10) allows the two components to keep a consistent protocol state. Second, the protocol stack has to be modified to make use of the outboard storage and checksum calculation. These changes are similar to those discussed in Section 3.1, except that iWarp runs a light-weight runtime (instead of Unix) and interoperability with existing devices and applications is not a concern.

The iWarp CAB can send data at 75 MByte/sec for 64 KByte packets. Measurements for raw HIPPI and UDP over IP give the same throughput results, i.e. the UDP/IP implementation is very efficient. The main difference between the two cases is that the idle time on the iWarp cell is lower in the UDP case (40%-56%) than in the raw HIPPI case (74%-79%). The current bottleneck in the system is the microcode on the CAB: it limits us to sending about 3000 packets per second.

4.3 The streams package

The transfer of data between the application on the system and the network is a two phase process. In a first phase (transmit), data is transferred from the system to the interface cell, and in the second phase, the data is sent over the network. The data transfer over the network is controlled by the communication protocols, as described above. Managing the first phase is mainly a resource management problem: both the distributed-memory system and the network interface have limited

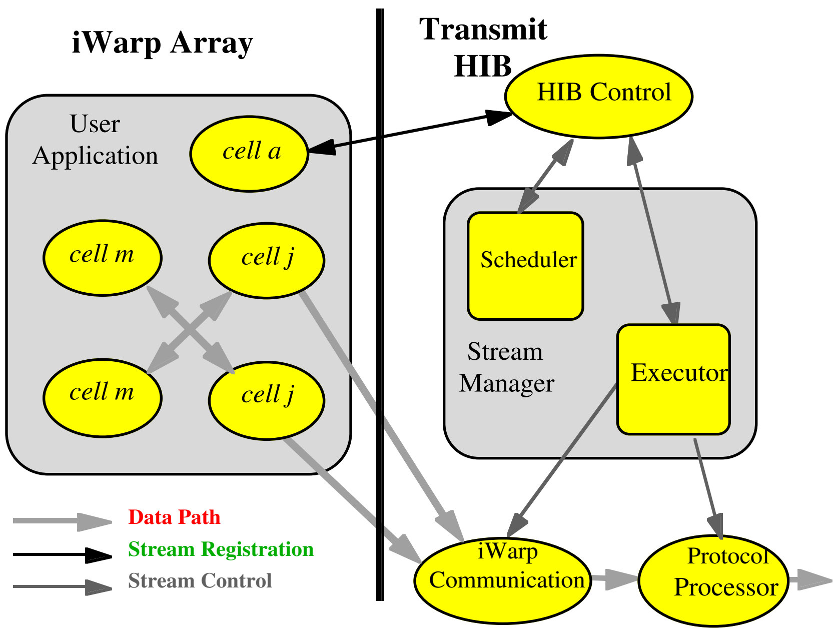

Figure 11: Streams architecture

resources (memory, buses, ..), and how they are allocated to support I/O will have a significant impact on performance. On sequential systems, this task is traditionally performed by the operating system. However, applications on distributed-memory systems can have very different IO requirements. They can differ with respect to the distribution of the data, the use of communication resources inside the distributed-memory system, and choice of communication library. For this reason, a single solution built into the operating system or supported by a general library will not be able to support IO efficiently for a wide range of applications.

For iWarp, communication between the distributed memory system and the HIB is supported using the *streams package*. It distinguishes between tasks that need to be performed on the distributed-memory system and tasks that need to be performed on the network interface [22]:

- The Streams Manager on the network interface is responsible for efficient data movement from the interconnect of the distributed-memory to the network for multiple *streams*. A stream is a connection from the application on the distributed-memory system to another system connected to the external HIPPI network. The streams manager is also responsible for invoking the appropriate communication protocols, which format and prepend headers to each packet.

- The application on the distributed-memory system is responsible for distributing (or collecting) the data to (or from) the network interface node using existing communication libraries. This architecture does not imply that each application has to provide the code to transfer data to and from the network interface. Rather, libraries can be built for common data distributions (Section 4.4).

The components interact through a control interface and a data interface:

- The data interface transfers data between the the network interface and the application on the distributed memory system. The main parameters controlling data transfers are the data format (quantity and ordering if striping is used), and address information.

- The control interface allows the application to instruct the network interface on how it should perform communication. Typical operations include opening or closing a connection, issuing an IO operation, or inquiring about status.

The streams package provides the flexibility needed to support application-specific I/O and a wide range of applications have used the package to communicate over HIPPI [22]. Examples include MRI medical image reconstruction, chemical flow sheeting, and real time video display.

If the distributed-memory systems is programmed using a programming tool, such as a parallelizing compiler, that tool is in an ideal position to manage the IO, e.g. select the right data movement libraries and insert calls to stream manager. The tool already manages the resources in the distributed-memory system based on an understanding of the characteristics of the application, and can optimize IO operations internally (between the array's nodes) and externally (to the network node interface)

4.4 Data distribution

To efficiently utilize the large number of processors in a distributed-memory computer, applications typically use data parallelism. Data is partitioned into equal-sized blocks, which are distributed across the processors, and each processor operates on the data that is assigned to it. Both the type and granularity of data partitionings varies widely between application. As a result, I/O operations include an extensive scatter/gather operation that is application specific. The scatter/gather operation also has to deal with striping the data stream over multiple links, if the link speed over the internal interconnect is lower than that of the external network.

Distributed-memory machines can deal with the data distribution in a number of ways. In a first approach, array cells send data to the interface independently and the data is sorted out and grouped by the interface. This approach is simple to implement, but even if the overhead for sending and receiving blocks to or from the internal interconnect is small, the interface will become a bottleneck for fine-grain data partitionings [20].

In the approach we selected [8], the processor array is responsible for the scatter/gather operation. On transmit, it constructs large messages and presents them to the interface in an efficient way, for example striped across multiple links. On receive, it distributes the data across the processors. This approach is attractive for two reasons. First, the interface cell only has to deal with large blocks of data, independent from the data partitioning inside the array. This minimizes the cost on the network interface of exchanging data with the system. Second, distributed-memory machines typically support high-bandwidth inter-node communication, and since reshuffling parallelizes very well, many links can be used at the same time. As a result, the processor array can reshuffle data efficiently. A similar approach has been proposed for disk I/O [5].

Figure 12 (left) shows how quickly we can reshuffle data; the results are for the reshuffling of a block-cyclic distribution with a certain block size (x axis) to a distribution with 8KByte blocks, which is the optimal size for communication with the HIB. Even for very small block sizes, reshuffling can be performed at rates that far exceed HIPPI network rate (100 MByte/second).

In a second experiment, data is transferred to the HIB, starting with the data mapped on the system using block-cyclic distributions with different block sizes. Figure 12 (right) shows the

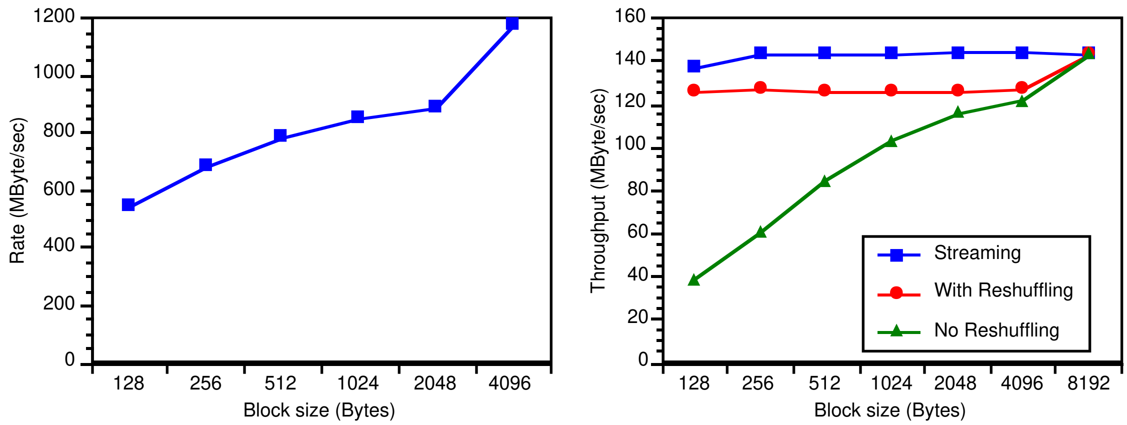

Figure 12: Performance of the Reshuffling Algorithm (left) and Throughput to HIB (right)

throughput both with and without reshuffling. We observe that when reshuffling is used, we can easily match the HIPPI bandwidth. For comparison, the curve labeled "streaming" shows the maximum throughput. It was obtained by having each cell send the data blocks directly to the HIB, striped across 4 links but ignoring the order of the data.

5 Gigabit Nectar testbed

The Gigabit Nectar iWarp and workstation interfaces were deployed in the Gigabit Nectar testbed (Figure 13). The testbed connects computer system on the CMU campus and at the Pittsburgh Supercomputer Center (PSC) by a HIPPI network. The systems include an Intel iWarp and Paragon system and 24 DEC Alpha 3000/400 workstations on the CMU campus, and a Cray C90 and Thinking Machines CM2 at PSC. The network consists of two local area HIPPI networks, one on the CMU campus and one at PSC, connected by a serial HIPPI link and by an experimental ATM-SONET link [25, 13]. The ATM-SONET link (HAS for HIPPI-ATM-SONET) was developed jointly with Bellcore. The testbed is being used for a variety of applications.

5.1 Heterogeneous supercomputer applications

A first class of applications that makes use of the testbed consists of applications that are distributed across a small number of supercomputers. Examples include a medical resonance imaging application [37] that uses the iWarp, C90 and Paragon systems, and a chemical flow sheeting application [15] that was distributed over the iWarp, C90 and CM2 systems. Applications that use special purpose devices have similar characteristics. An example is a video capture application that capture a stream of images, processes them on iWarp and displays them on a HIPPI framebuffer [49].

The use of heterogeneous systems makes it possible for each task in the application to run on the system for which it is best suited. This is for example the case for the chemical flow sheeting application which consists of a massively-parallel memory-intensive task (CM2), a highly parallel task (iWarp), and a scalar task (C90).

Another important benefit of connecting heterogeneous systems with a high speed network is that it simplifies code reuse. Many applications, e.g. interdisciplinary applications, are constructed

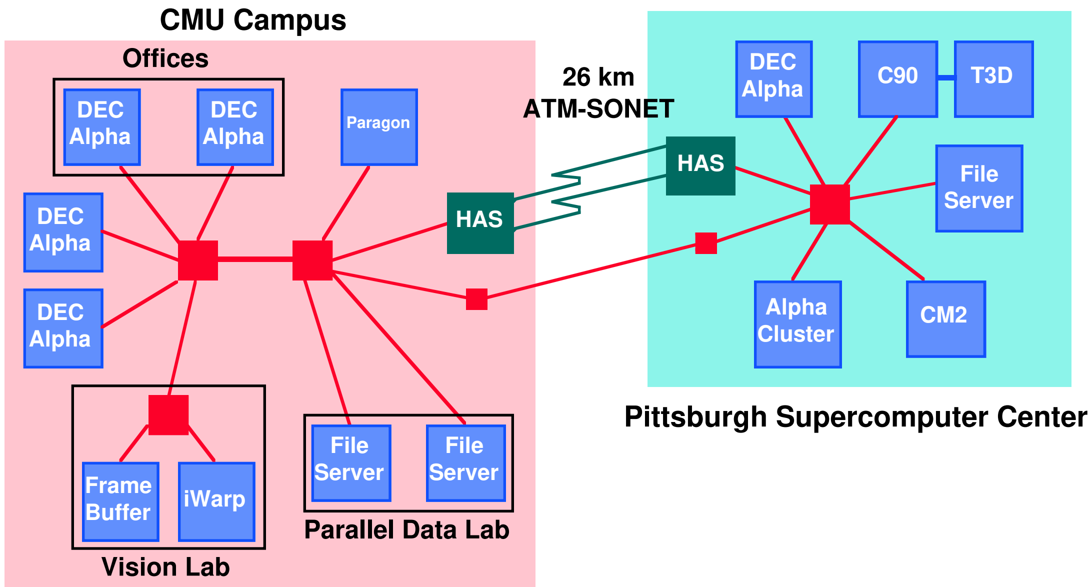

Figure 13: The Gigabit Nectar testbed

by combining existing programs. In many cases, these programs have been developed for different parallel systems, and porting all programs to the same system so they can be integrated would be a significant effort. A heterogeneous multicomputer makes it possible to avoid the porting step: each component executes on the platform for which it was developed and the components in the combined application use the network to exchange data. Since data sets are usually very large, the network must have a high bandwidth.

5.2 Workstation cluster applications

Workstation clusters (or "Network Of Workstations") are an attractive platform for many applications, and the ARPA-funded Nectar project [4, 30], the predecessor of Gigabit Nectar, explored issues in the areas of high-speed communication in switched-based LANs [44, 16] and support for application distribution [31]. This work is being continued using the Gigabit Nectar cluster consisting of 24 Alpha workstations distributed across labs and offices. The applications that use the cluster include the NSF Grand Challenge applications on Environmental Modeling [29] and on Ground Motion Modeling.

The focus of the Gigabit Nectar cluster, however, has been on supporting research on programming tools for distributed computing. Implementing applications on a network-based multicomputer is a non-trivial effort, and programming tools that simplify that task are needed. In the context of the Nectar and Gigabit Nectar systems, we have demonstrated tools in three critical areas: monitoring tools that help the programmer understand the behavior of their application [10, 9], support for data sharing across the network [35, 36], and load balancing tools that help in distributing work to make efficient use of the cycles on the nodes [51, 50, 41, 42, 40].

Recent research has focused on providing programming abstractions at a higher level than message passing. The ARPA funded Fx parallelizing FORTRAN compiler [30] supports both data and task parallelism, and, as a result, it can be used for a large number of application domains, including embedded signal processing applications. Other efforts include the Dome distributed object library [3], which provides runtime support for load balancing and fault tolerance, and the DCABB environment [28] which supports distributed branch and bound algorithms. The Scotch parallel file system is also connected to Gigabit Nectar.

6 Conclusion

The Gigabit Nectar project demonstrated that by optimizing per-byte operations, it is possible to communicate over networks efficiently using standard communication protocols (internet protocols) and APIs (BSD sockets). For workstations we use outboard buffering and checksumming and a modified BSD protocol stack to achieve single copy communication, i.e. data is touched only once on its path from the application space to the network. For the distributed memory systems such as iWarp, we rely on the distributed memory system to create large contiguous blocks of data that can be handled efficiently by the network interface. Protocol processing is performed on the network interface using hardware support for per-byte operations, similar to that on the workstation interface.

The workstation and iWarp interfaces were deployed in the Gigabit Nectar testbed and were used both in heterogeneous supercomputer and workstation cluster applications. One interesting result was that the presence of a high-speed network allowed us to quickly build several complex applications by combining application components that had been developed independently for different systems. Our workstation cluster results also show that a higher speed network allows us to run applications more effectively, i.e. to use more nodes and get better speedups.

References

[1] ANSI. High-performance parallel interface - mechanical, electrical and signalling protocol specification (HIPPI-PH). ANSI X3.183-1991, 1991.

[2] ANSI. Fibre channel physical and signaling interface (fc-ph). Working draft proposed American National Standard for Information Systems, 1992.

[3] Jose Arabe, Adam Beguelin, Bruce Lowekamp, Eric Seligman, Mike Starkey, and Peter Stephan. Dome: Parallel programming in a heterogeneous multi-user environment. Technical Report CMU-CS-95-137, Computer Science Department, Carnegie Mellon University, April 1995.

[4] Emmanuel Arnould, Francois Bitz, Eric Cooper, H. T. Kung, Robert Sansom, and Peter 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*, pages 205–216, Boston, April 1989. ACM/IEEE.

[5] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings of Supercomputing '93*, pages 452–461, Oregon, November 1993. ACM/IEEE.

[6] Shekhar Borkar, Robert Cohn, George Cox, Sha Gleason, Thomas Gross, H. T. Kung, Monica Lam, Brian Moore, Craig Peterson, John Pieper, Linda Rankin, P. S. Tseng, Jim Sutton, John Urbanski, and Jon Webb. iwarp: An integrated solution to high-speed parallel computing. In *Proceedings of the 1988 International Conference on Supercomputing*, pages 330–339, Orlando, Florida, November 1988. IEEE Computer Society and ACM SIGARCH.

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

[8] Claudson Bornstein and Peter Steenkiste. Data reshuffling in support of fast I/O for distributed-memory machines. In *Proceedings of the Third International Symposium on High-Performance Distributed Computing*, pages 227–235, San Fransisco, August 1994. IEEE.

[9] Bernd Bruegge. A portable platform for distributed event environments. In *PDDEB91*, volume 26, pages 184–193, Santa Cruz, California, December 1991. ACM.

[10] Bernd Bruegge and Peter Steenkiste. Supporting the development of network programs. In *Proceedings of the Eleventh International Conference on Distributed Computing Systems*, pages 641–648. IEEE, May 1991.

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

[12] Luis-Felipe Cabrera, Edward Hunter, Michael J. Karels, and David A. Mosher. User-process communication performance in networks of computers. *IEEE Transactions on Software Engineering*, 14(1):38–53, January 1988.

[13] N. K. Cheung and H. T. Kung. Gigabit/sec wide area computer networks: Potential applications and technology challenges. In *OFC'91*, San Diego, February 1991. invited paper.

[14] 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.

[15] Robert Clay and Peter Steenkiste. Distributing a chemical process optimization application over a gigabit network. In *Proceedings of Supercomputing '95*, page To appear. ACM/IEEE, December 1995.

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

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

[18] M. de Prycker. *Asynchronous Transfer Mode*. Ellis Harwood, 1991.

[19] DEC. Turbochannel overview, April 1990.

[20] Thomas Gross and Peter Steenkiste. Architecture implications of high-speed I/O for distributed-memory computers. In *Proceedings of the 8th ACM International Conference on Supercomputing*, pages 176–185, Manchester, England, July 1994. ACM.

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

[22] Michael Hemy and Peter Steenkiste. Gigabit IO for distributed-memory systems: Architecture and applications. In *Proceedings of Supercomputing '95*, page To appear. ACM/IEEE, December 1995.

[23] 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.

[24] Norman C. Hutchinson and Larry L. Peterson. Implementing protocols in the x-kernel. Technical Report 89-1, University of Arizona, January 1989.

[25] M. Z. Iqbal, M. Stern, J. Young, H. Izadpanah, R. Standley, and J. L. Gimlett. A 2.5 gb/s sonet datalink with sts-12c inputs and hippi interface for gigabit computer networks. In *GLOBECOM '92 Conference Record*, pages 1196–1200, Orlando, FL, December 1992. IEEE.

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

[27] Peter Steenkiste Karl Kleinpaste and Brian Zill. Software support for outboard buffering and checksumming. In *Proceedings of the SIGCOMM '95 Symposium on Communications Architectures and Protocols*, page To Appear, Boston, August 1995. ACM.

[28] G. Kudva and J. F. Pekny. DCABB: A distributed control architecture for branch and bound calculations. *Computers and Chemical Engineering*, 1994. In Press.

[29] Naresh Kumar, Armistead Russell, Edward Segall, and Peter Steenkiste. Parallel and distributed application of an urban and regional multiscale model. Submitted for Publication, 1994.

[30] H.T. Kung, Robert Sansom, Steven Schlick, Peter Steenkiste, Francois J. Bitz Matthieu Arnould, Fred Christianson, Eric C. Cooper, Onat Menzilcioglu, Denise Ombres, and Brian Zill. Network-based multicomputers: An emerging parallel architecture. In *Proceedings of Supercomputing '91*, pages 664–673, Albequerque, November 1991. IEEE.

[31] H.T. Kung, Peter Steenkiste, Marco Gubitoso, and Manpreet Khaira. Parallelizing a new class of large applications over high-speed networks. In *Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 167–177. ACM, April 1991.

[32] 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.

[33] Qingming Ma and Peter Steenkiste. Performance of circuit switched lans under different traffic conditions. In *Proceedings of the 19th Conference on Local Computer Networks*, pages 266–276. IEEE, October 1994.

[34] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba: A distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, May 1990.

[35] Hiroshi Nishikawa and Peter Steenkiste. Aroma: Language support for distributed objects. In *International Parallel Processing Symposium*, pages 686–690, Los Angeles, April 1992. IEEE.

[36] Hiroshi Nishikawa and Peter Steenkiste. A general architecture for load balancing in a distributed-memory environment. In *Proceedings of the Thirteenth International Conference on Distributed Computing Systems*, pages 47–54, Pittsburgh, May 1993. IEEE.

[37] Doug C. Noll, Jon A. Webb, and Tom E. Warfel. Parallel fourier inversion by the scan-line method. *IEEE Transactions on Medical Imaging*, in press, 1995.

[38] Report. Gigabit network testbeds. *IEEE Computer*, 23(9):77–80, September 1990.

[39] Michael Schroeder and Michael Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[40] Bruce Siegell. *Automatic Generation of Parallel Programs with Dynamic Load Balancing for a Network of Workstations*. PhD thesis, Department of Computer and Electrical Enginerring, Carnegie Mellon University, 1995. Also appeared as technical report CMU-CS-95-168.

[41] Bruce Siegell and Peter Steenkiste. Automatic generation of parallel programs with dynamic load balancing. In *Proceedings of the Third International Symposium on High-Performance Distributed Computing*, pages 166–175, San Fransisco, August 1994. IEEE.

[42] Bruce Siegell and Peter Steenkiste. Controlling application grain size on a network of workstations. In *Proceedings of Supercomputing '95*, page To appear. ACM/IEEE, December 1995.

[43] Raj K. Singh, Stephen G. Tell, Shaun J. Bharrat, David Becker, and Vernon L. Chi. A programmable HIPPI interface for a graphics supercomputer. In *Proceedings of Supercomputing '93*, pages 124–132, Oregon, November 1993. ACM/IEEE.

[44] 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.

[45] 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.

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

[47] 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.

[48] Richard Thompson. Los alamos multiple crossbar network crossbar interfaces. In *Workshop on the Architecture and Implementation of High Performance Communication Subsystems*. IEEE, February 1992.

[49] Jon Webb. Latency and bandwidth considerations in parallel robotics image processing. In *Proceedings of Supercomputing '93*, pages 230–239, Oregon, November 1993. ACM/IEEE.

[50] I.-C. Wu and H.T. Kung. Communication complexity for parallel divide-and-conquer. In *1991 Symposium on Foundations of Computer Science*, pages 151–162, San Juan, October 1991.

[51] I-Chen Wu. *Multilist scheduling: A New Parallel Programming Model*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1993. Also appeared as technical report CMU-CS-93-184.