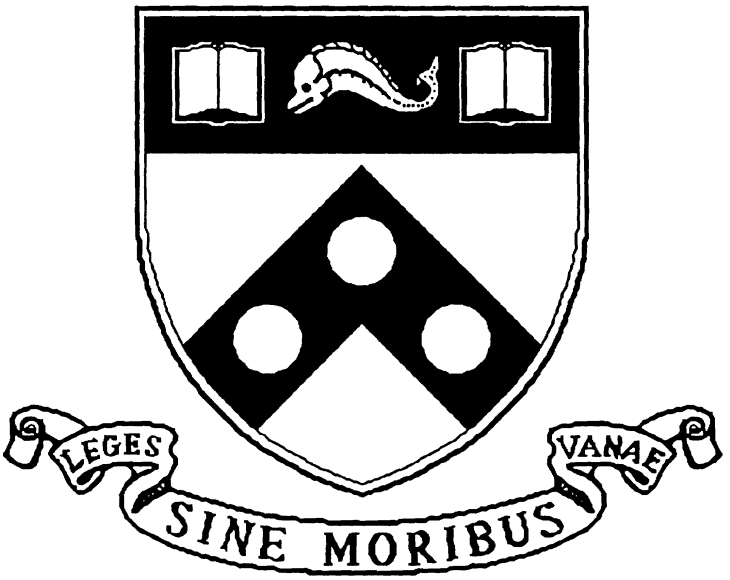
Operating Systems Support for End-to-End Gbps Networking

MS-CIS-93-33
DISTRIBUTED SYSTEMS LAB 30

Jonathan M. Smith
C. Brendan S. Traw



University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department
Philadelphia, PA 19104-6389

March 1993

Operating Systems Support for End-to-End Gbps Networking

Jonathan M. Smith and C. Brendan S. Traw

Distributed Systems Laboratory, University of Pennsylvania
200 South 33rd St., Philadelphia, PA 19104-6389

ABSTRACT

This paper argues that workstation host interfaces and operating systems are a crucial element in achieving end-to-end Gbps bandwidths for applications in distributed environments. We describe several host interface architectures, discuss the interaction between the interface and host operating system, and report on an ATM host interface built at the University of Pennsylvania.

Concurrently designing a host interface and software support allows careful balancing of hardware and software functions. Key ideas include use of buffer management techniques to reduce copying and scheduling data transfers using *clocked interrupts*. Clocked interrupts also aid with bandwidth allocation. Our interface can deliver a *sustained* 130 Mbps bandwidth to applications, roughly OC-3c link speed. Ninety-three percent of the host hardware subsystem throughput is delivered to the application with a small measured impact on other applications processing.

1. Introduction

The past several years have seen a profusion of efforts to design and implement very-high speed networks which deliver this speed "end-to-end". The definition of "very-high" varies, but a good example is the AURORA Gigabit Testbed [5], one of several such testbeds [1]. In AURORA, much of the focus has been on the development of technologies needed to deliver this performance to workstation-class machines rather than supercomputers. It is our belief that these machines will be the majority of endpoints in future Gbps networks.

The difficulty posed by the choice of workstations is the mismatch between the performance of the machines and the bandwidth provided by the network infrastructure such as switches and transmission lines. Specifically, the network bandwidths are within an order of magnitude of the memory bandwidths of most workstations, and the burden on a host's memory architecture must be minimized for maximum performance. As pointed out by Clark and Tennenhouse [4], this forces careful design of protocol processing architectures.

Efficiency can be achieved through many design features, but the main options [25] are: optimizing the processing functions in the protocol architecture, optimizing the operating system support for data transport, and careful placement of hardware for network attachment. In this paper, we will focus on operating system and architectural issues, as we feel that high-performance protocol architecture features such as ordering, errors, duplicates, coordination, and format conversion have been well-covered by others; see for example Feldmeier [14].

Since this paper focuses on operating system issues, we will outline the approaches to host interface hardware and supporting software. We then motivate our selection of a particular set of functions in our implementation of an ATM host interface for the IBM RISC System/6000 workstation [24]. The split of work between the interface and the operating system is discussed, together with several ideas for optimizing end-to-end performance which have been explored and appear to have considerable promise.

1.1. Host Interfaces

The design and implementation of host interfaces has been of interest since the earliest network implementations†. Each succeeding generation has dealt with different types of hosts, networks, protocol architectures and networked applications. Goals have included low cost, high throughput and low delay. Implementations have been optimized towards achieving one or more of these goals in their operational environments. Some of the key implementation decisions have been: (1) the portion of protocol architecture functions performed by the interface; (2) signaling between host and interface; and (3) the placement of the interface in the host computer's architecture. Much of the migration to hardware is intended to obtain an implementation-specific performance advantage - as Watson and Mamrak [25] point out, performance is often due as much to implementation techniques as to careful protocol design. The key question may be the selection of functions to optimize by placement in hardware.

Several interfaces have attempted to accelerate *transport protocol processing.* The VMP Network Adapter Board (NAB) [17] implementation accelerates processing of Cheriton's Versatile Message Transaction Protocol (VMTP). The goals were to reduce the latency required in "request-reply" communications, while delivering high throughput for data-intensive applications. The NAB separated these two classes of traffic to optimize its performance. The NAB included an on-board microcontroller.

The Nectar Communications Accelerator Board (CAB) [22] includes a microcontroller with a complete multithreaded operating system. The host-CAB interaction is via messages sent over a VME bus, synchronized using a mailbox scheme. The programmability can be used by applications to customize protocol processing. Cooper, *et al.* [7], report that TCP/IP and a number of Nectar-specific protocols have been implemented on the CAB.

It remains unclear whether the entire transport protocol processing function needs to migrate to the interface; Clark, *et al.* [3] argue that in the case of TCP/IP the actual protocol processing is of low cost and requires very few instructions on a per-packet basis, and thus could be left in the host with minimal impact.

Less protocol processing is performed by two ATM host interfaces built at Bellcore and Penn. Bellcore's [10] ATM Host Interface implementation attaches to the TURBOChannel bus of the DECstation 5000 workstation. The interface operates on cells, and communicates protocol data units (PDUs) to and from the host. The design relies on two Intel 80960 RISC microcontrollers to perform the protocol processing and flow control at a rate of 622 Mbps. Like the CAB, it is designed to be programmable, although the programming is mainly designed to explore Segmentation-and-Reassembly (SAR) algorithms. At this time, Bellcore's interface provides the highest burst performance reported for an ATM host interface [11].

Off-board processors can migrate many processing and data movement tasks away from the host CPU. However, it is not clear that flexibility requires a general purpose processor, as opposed to a solution using, e.g., programmable logic devices. At Penn, we have developed a scalable host interface architecture for providing segmentation and reassembly functions in dedicated logic [24]. All per cell processing including error detection is performed by the hardware. An initial implementation attaches to the IBM RISC System/6000 workstation through its Micro Channel I/O bus.

A second implementation of our segmentation and reassembly architecture is currently in progress to further explore the design space. The result will be an ATM Link Adapter for use with the HP 9000/700 series workstations equipped with Afterburner [18] cards. The Afterburner/ATM Link Adapter combination will provide the same basic functionality as the initial implementation with two exceptions. First, for additional ATM Adaptation Layer 5 (AAL5) support, the CRC32 will be generated and checked as reassembled AAL5 PDUs are moved from the ATM Link Adapter to the Afterburner. Second, the Afterburner will provide support for computing IP checksums in hardware as well as a per-PDU processor interrupt.

Fore Systems, Inc. [6], and Cambridge University/Olivetti Research [15], have each explored an approach which puts minimal functionality in interface hardware. This approach assigns almost all tasks to the workstation host including ATM adaptation layer processing.

Minimalist approaches can take advantage of aggressive workstation technology improvement, which might outstrip that of host interface components. However, such an approach has two potential failings. First, RISC workstations are optimized for data processing, not data movement, and hence the host must devote significant

† Detailed information on some of these interfaces and supporting software is available in a Special Issue of the *IEEE Journal on Selected Areas in Communications* [21].

resources to manage high-rate data movement. Second, the operating system overhead of such an approach can be substantial without hardware assistance for object aggregation and event management.

Table I summarizes some high-level design features for some example host interfaces.

Feature	NAB	CAB	Bellcore	Penn/HP	Penn	Cambridge	Fore
Event Flag	I	Mbox	I	I	CI	I	I
Event	PDU	PDU	PDU	PDU	PDU	Cell or PDU	Cell
Processor?	Y	Y	Y	N	N	N	N

Table I: **Signaling, Units, and Intelligence**
Legend: I - Interrupt, CI - Clocked Interrupt

Table II further lists features of some ATM network host interfaces.

Feature	Bellcore	Penn/HP	Penn	Cambridge	Fore
Connection MUX	HW	SW	SW	SW	SW
Connection DeMUX	HW	HW	HW	SW	SW
HW Err. Detect Unit	Header	Cell/PDU	Cell	Header/PDU	Cell
Loss Detect	HW	SW	SW	SW	SW

Table II: **ATM-oriented features**

1.2. Host Interface Attachment

From the point of performance, an application can do no better than have a dedicated host interface available to it. The host interface would communicate via memory shared with the application, so that copying was minimized both on the processor and on the I/O bus. The operating system might involve itself in, e.g., scheduling, but it would have a limited role so that the application peak bandwidth would approach the limit of network or memory bandwidth. **Figure 1** illustrates a general architecture for a host interface and will allow us to discuss several options for host attachment.

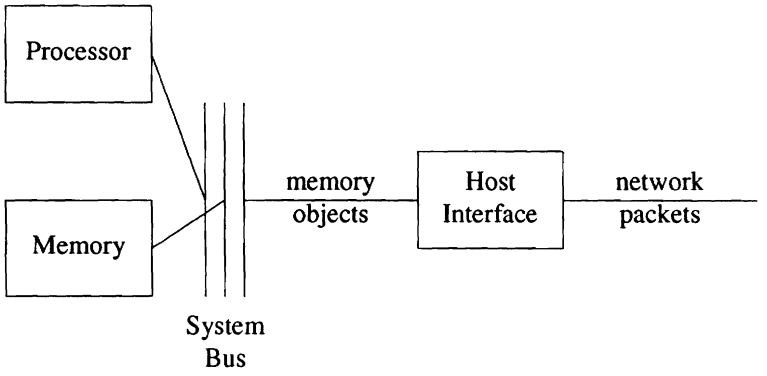


Figure 1: General Host Interface Architecture

We will enumerate a set of options using this figure and attempt to assess their performance potential. This enumeration loosely follows B. Ahlgren's [13].

1. The Host Interface is capable of Direct-Memory Access (DMA), which means that it can communicate with the system memory directly, without processor intervention. The typical system (e.g., UNIX) uses the Host Interface DMA capability to copy the data from the network into a buffer managed by the operating system. The data is then copied by the CPU from the system buffer to a buffer owned by the application, which also resides in the system memory. Thus, a given piece of data travels over the System Bus three times: Host

Interface to Memory, Memory to CPU, and CPU to Memory.

2. The Host Interface is capable of DMA, and the operating system is able to arrange for data to be transferred directly to the application address space. Thus, a given piece of data travels over the System Bus once, from Host Interface to Memory.
3. The Host Interface has a processor-addressable memory area, which the operating system manages. When data arrives in the Host Interface's buffers, the operating system copies this data into the user address space. In this model, the data must travel the bus twice, once from Host Interface to CPU, and then from the CPU to Memory.
4. The Host Interface has a processor-addressable memory area, where the application buffers are located. This means that data never traverses the system memory bus, or does not until it is referenced by the processor. This is equivalent to connecting the host interface directly to the system memory.
5. The Host Interface can be connected directly to the CPU [5], as in augmenting the processing unit with a co-processor. As in Scheme 4, there is no memory bus traversal, and further, the connection is to a system component which operates at speeds higher than memory bandwidths.

Each of these schemes is affected by a number of other considerations.

First, most modern architectures include a *cache,* which decreases the access latency of frequently used data, but must be kept in a state consistent with system memory. The cache is typically co-located with the processing unit, so it either must be kept consistent or flushed when new data arrive. Maintaining is considerably easier when the data passes through the processor - Scheme 2 must flush cache for areas affected by the DMA, and Scheme 4 must flush the cache, either under control of the CPU or the Host Interface. Schemes 1 and 3 should be able to obtain up-to-date cache copies when the data is copied into the user address space.

Second, host interfaces will also be used for applications which require specialized peripherals, such as video conferencing. Thus it is important to keep a good balance between I/O and memory accessibility. The DMA based schemes do this, but the memory-on-interface schemes (Schemes 3 and 4) provide an unorthodox I/O model, and thus would present difficulties in I/O operations to and from other devices. The coprocessor approach would involve the processing unit in all operations connecting the network to other devices.

Third, Schemes 1 and 3 involve the processor in copying data across address-space boundaries. Thus, the processor must reduce its processing capacity by the amount of time spent copying data - this would be a considerable fraction of its instruction-processing capacity at high bandwidths. The co-processor approach likewise shares processing-unit capacity between processing load and network traffic.

Finally, I/O channel architectures provide a number of practical attractions, among which are access to other peripherals, structuring, concurrency control, and features such as virtual address translation by channel controller hardware. In addition, connection to a bus can aid portability across CPU and system architectures, and access to vendor's I/O bus specifications is considerably less restricted than to system memory bus specifications.

1.3. Interaction with Software

Operating system software plays a key role in the achievement of high end-to-end networking performance. The abstraction provided by the host interface is that of a device which can transfer data units between a network and system memory. The software must build upon this abstraction to satisfy application requirements. A significant constraint on such software is its embedding in the framework of an operating system which satisfies other (possibly conflicting) requirements. Particular application needs include transfer of data into application-private address spaces, connection management, high throughput, low latency, and the ability to support a variety of traffic types. Traffic types include traditional bursty data communications traffic (such as transaction-style traffic), bulk data transfer, and the sustained bandwidth requirements of applications using continuous media. We believe that approaches optimized towards particular traffic types, such as low-latency transactional traffic [22], will suffer if the traffic mix varies considerably.

The software operating on the host is usually partitioned functionally into a series of layers defined by protection boundaries. Typically, each software layer contains several protocol layers. The applications are typically executable programs, or groups of such programs cooperating on a task, which a user might invoke. Applications which require network access obtain it via abstract service primitives such as *read(), write(),* and *sendto().* These service primitives provide access to an implementation of some layers of the network protocol, as in the UNIX system's

access to TCP/IP through the socket abstraction. The protocol is often designed to mask the behavior of the network and the hardware connecting the computer to the network, and its implementation can usually be split into device-independent and device-dependent portions.

Significant portions of protocol implementations may be embedded in the operating system of the host, where the service primitives are system entry points, and the device-dependent portion is implemented as a "device driver." Such device drivers have a rigidly specified programmer interface, mainly so that the device-independent portions of system software can form a reasonable abstraction of their behavior. Placement of the protocol functions within the operating system is dictated by two factors, *policies* and *performance*. The key policies which an operating system can enforce through its scheduling are *fairness* (e.g., in multiplexing packet streams) and the prevention of starvation. High performance may require the ability to control timing and task scheduling, the ability to manipulate virtual memory directly, the ability to fully control peripheral devices, and the ability to communicate efficiently (e.g., with a shared address space). All of these requirements can be met by embedding the protocol functions in the host operating system. In practice, the main freedoms for the host interface software designer lie in the design of the device driver, since it forms the boundary between the host's device independent software and the functions performed by the device.

The software architect is presented with the following choices as to detailed implementation strategy:

1. Based on the capabilities of the interface (e.g., its provision for programmed I/O, DMA, and streaming), what is the partitioning of functionality between the host software and the host interface hardware? For example, use of DMA or streaming removes the need for a copying loop in the device driver to process programmed I/O, but may require a variety of locks and scheduling mechanisms to support the concurrent activities of copying and processing. Poor partitioning of functions can force the host software to implement a complex protocol for communicating with the interface, and thereby reduce performance.
2. Should existing protocol implementations be supported? On the one hand, many applications are immediately available when an existing implementation is supported, e.g., TCP/IP or XNS. On the other, significant performance (and hopefully new applications) can be gained by ignoring existing stacks in favor of stacks optimized to the new Gbps networks [4] and interface hardware, using a new programmer interface. Or, both stacks could be supported, at a significant cost in effort; this allows both older applications and new applications with greater bandwidth requirements to coexist. Methods such as the *x*-Kernel [16, 19] may provide a method for customized stacks to be built on top of operating system support such as we describe in this paper.
3. How are services provided to applications? One key example is the support for paced data delivery, used for multimedia applications. As the host interface software is a component in timely end-to-end delivery, it must support real-time data delivery. This implies provision for process control, timers, etc. in the driver software.
4. How do design choices affect the remainder of the system? The host interface software may be assigned a high priority, causing delays or losses elsewhere in the system. Use of polling for real-time service may affect other interrupt service latencies. The correct choices for tradeoffs here are entirely a function of the workstation user's desire for, and use of, network services. While any tradeoffs should not preclude interaction with other components of the system, e.g., storage devices or frame buffers, increasing demand for network services may bias decisions towards delivering network subsystem performance.

Given the cost of interrupts and their effect on processor performance, strategies which reduce the number of interrupts per data transfer can be employed [17]. An example would be using an interrupt only as an event indicator. The transfer of bursts of ATM cells may arise as a consequence of the mismatch between larger application data units and the ATM payload of 48 bytes would be accomplished in a scheduled manner, e.g., using polling.

1.4. Interrupts - Clocked versus Data-Driven

One of the key issues in the design of operating system features which support interactions with external events (such as arriving data) is the signaling protocol. There are three common approaches used:

1. Pure "busy-waiting", where the external event can be detected by a change in, e.g., an addressable status register. The processor continuously examines the stateword until the change occurs, and then resumes processing with the newly-arrived data. "Busy-waiting" is rarely if ever used in multitasking systems, since it effectively precludes use of the processor until the event arrives. It is more commonly used by dedicated controllers. "Busy-waiting" can be used with priorities to enforce some degree of isolation among activities on the processor.

2. *Interrupts* are an artifact of the desire to timeshare processors among activities. The basic idea is that the event arrival (most likely detected by a low-level busy-waiting scheme in the external device) causes the processor to be *interrupted,* that is, to cease its current flow of control and to begin a new flow of control dictated by data arrival. Typically, this involves transferring the data to a processor storage location where the data can be processed later, using a normal flow of control. When interrupt service is complete, the processor resumes the interrupted flow of control. The two difficulties with interrupts are their asynchronous arrival and their cost. The asynchronous arrival forces concurrency control techniques to be employed, and the interrupt service time improves much more slowly than microprocessor speeds.
3. *Clocked* interrupts try to achieve a somewhat different balance of goals. A periodic software timer is used to interrupt the flow of control of the processor as with any other interrupt. Interrupt *service* then consists of examining changed statewords, as in the "busy-waiting" scheme. The tradeoffs here are closely tied to the implementation environment, but an illustrative example is given by the UNIX [23] *callout* table design, used for operating system management of pools of teletypewriter lines.

We have chosen clocked interrupts as the signaling mechanism for our host interface because of the operating environment. In particular, as pointed out in [12] multiplexing is a key issue, and in an end-to-end architecture, the end-points are processes. While the host interface demultiplexes traffic into per-virtual circuit queues, these queues must be transferred to the appropriate applications processes. In addition, Quality of Service guarantees, especially allocated bandwidths, must be supported. Our view is that like other system managed resources, bandwidth sharing can be split between *policy* and *mechanism.* The policy is largely a function of higher layers in a protocol hierarchy, but *scheduling* is the operating system mechanism most suited to allocating bandwidth, as it is a form of time-division.

Using clocked interrupts is an engineering decision based on factors such as costs and traffic characteristics. A simple calculation shows the tradeoff. Consider a system with an interrupt service overhead of C seconds, and k active channels, each with events arriving at an average rate of λ events per second. Independent of interrupt service, each event costs α seconds to service, e.g., to transfer the data from the device. The offered traffic is $\lambda \cdot k$, and in a system based on an interrupt-per-event, the total overhead will be $\lambda \cdot \alpha \cdot k \cdot C$. Since the maximum number of events serviced per second will be $1 / C+\alpha$, the relationship between parameters is that $1<\lambda \cdot k \cdot (C+\alpha)$. Assuming that C and α are for the most part fixed, we can increase the number of active channels and reduce the arrival rate on each, or we can increase the arrival rate and decrease the number of active channels.

However, assuming clocked interrupts delivered at a rate β per second, we get the relationship $1<\beta \cdot C+\lambda \cdot k \cdot \alpha$. Since α is very small for small units such as characters, and C is very large, it makes sense to use clocked interrupts, especially when a reasonable value of β can be employed. In the case of modern workstations, C is about 10^{-3} second. Note that as the traffic level rises, more work is done on each clock "tick", so that the data transfer rate $\lambda \cdot k \cdot \alpha$ asymptotically bounds the system rather than the interrupt service rate. To be fair, one should note that traditional interrupt service schemes can be improved, e.g., by aggregating traffic into larger packets (this reduces λ significantly, while typically causing a slight increase in α), or by using an interrupt on one channel to prompt scanning of other channels.

Given the wide variety of traffic proposed for ATM networks, and our desire to accommodate such traffic, we chose to explore the use of clocked interrupts.

2. The Penn ATM Interface

2.1. A Programmer's View of the Hardware

The Host Interface is implemented as a pair of cards which share a physical layer transceiver. The Segmenter card breaks variable-sized Protocol Data Units (PDUs) into fixed-size ATM cell bodies, prepends headers, and transmits the cell. The Reassembler card receives multiplexed streams of ATM cells, which it demultiplexes using the ATM cell header into a number of queues, one queue per virtual circuit. The queue numbers are used as names by the host, which absorbs data by presenting a queue identifier and initiating a transfer. Each card behaves as a bus master, which provides a DMA-like capability to transfer data using the bus bandwidth-maximizing streaming transfer mode.

The card pair provides the illusion of a fast network with variable-sized data units using an ATM network and segmentation and reassembly logic. The interface performs adaptation layer processing, which provides a higher-

level interface to the ATM network for applications. Since adaptation layer processing typically involves integrity checks such as CRC checksums, the checks are done in hardware. The illusion of variable-sized PDUs is used to significant advantage by the host software support, discussed next.

2.2. Software Implementation Overview

UNIX and its derivatives are the development platform for almost all host software research, because they are the dominant operating systems on workstations. These operating systems unfortunately impose a number of additional constraints on the designer, in particular, the high cost of *system calls* due to their generality and the crossing of an application/kernel address space protection boundary. Pu, *et al.* [20] report that over 1000 instructions are executed by a *read()* call before any data are actually read. UNIX also embeds a number of policy decisions about scheduling, which as indicated above, is event-driven and designed to support interactive computing for large numbers of users. While several UNIX derivatives have been modified to support "real-time" behavior, these are non-standard, making solutions dependent on them non-portable. A number of other evolutions in UNIX, however, appear promising for high performance implementations and efficient application-kernel communication, such as shared memory, memory-mapped files, and provision for concurrency control primitives such as semaphores.

The current host interface support software consists of an AIX character-special [23] device driver. The software enables the host interface hardware to copy data directly from the application address space.

The interface is initialized when the device special file */dev/host{n}* is first opened. Initialization consists of probing the device at a distinguished address which causes it to be reset, build data structures in the Reassembler, as well as performing various set-up operations for the software. The operations currently include pinning the software's pages into real memory by removing them as candidates for page replacement. After initialization, the device and software are ready for operation; routines for all appropriate AIX calls (e.g., *read(), write(), ioctl(), etc.*) are provided. The *read()* and *write()* calls perform data transfer operations, while *ioctl()* is used for control operations such as specifying Virtual Circuit Identifiers to be associated with a particular channel. The code fragment shown below in **Figure 2** illustrates how a programmer would access the device for writing; this particular fragment is taken from the measurement software we used for performance evaluation.

```
if ((fd = open("/dev/host_s0", O_WRONLY)) == -1){
   perror("Couldn't open");
   exit(-1);
}

set_header( fd, vci, mid ); /* calls ioctl() */

for(i=0; i<repeats; i++){
   if (write(fd, buf, count ) == -1)
      perror("write failure");
}
```

Figure 2: Code fragment to access and exercise Segmenter

The software is accessed mainly through the *ioctl(), read(),* and *write()* system entry points. *Ioctl()* is employed for such control tasks as specifying VCIs and MIDs for use in formatting ATM cells; the VCI and MID are specified to the driver on a per-file descriptor basis. They are used, e.g., to specify header data to the segmenter card so that it can format a series of ATM cells for transmission. *Ioctl()* is used for any behavioral customization of the software, such as bandwidth allocations, maximum delays, and pacing strategies. Data transfer is done with *read()* and *write(),* providing a clean separation between transfer and control interfaces.

2.3. Reduced Copying

As we have discussed above, it is desirable to reduce copying. The advantage of reducing copying has been observed by others, e.g., Watson and Mamrak [25] and confirmed in other implementations [2]. The key issue is coordinating the hardware and software in such a way that the copying cost can be reduced; we have done this by copying data directly to and from user address spaces, as discussed in Section 1.2, above.

When the *write()* call is invoked on the device, user data is available to the driver through a `uio` structure element. If the data is to be put into kernel buffers, it is copied from the user address space into one of the 64K

buffers. If data is to be copied from the user process address space, the `uio` structure element is used to mark the application pages as pinned, and to obtain a "cross-memory descriptor" which allows the user data to be addressed by a device on the Micro Channel bus. When a hardware-provided status flag on the Segmenter indicates the device is inactive, a streaming mode transfer is set up. The software prepares for streaming by initializing a number of translation control words (TCWs) [8] in the Micro Channel's I/O Channel Controller (IOCC). In addition, page mappings are adjusted for pages in the host memory; the RISC System/6000 uses an Inverted Page Table also referred to as the Page Frame Table (PFT). The TCWs and Page Frame Table entries allow both the device and the CPU to have apparently contiguous access to scattered pages of real memory. The pointer tables are illustrated in **Figure 3**.

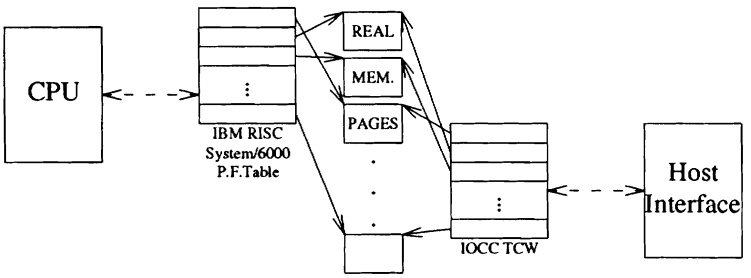


Figure 3: Illustration of TCW and PFT usage

After the TCWs and other state are set up, the device is presented with the data size and buffer address, which initiates the transfer.

The provision for TCWs in the IOCC allows large contiguous transfers directly to and from the address space of an AIX user process. The IOCC's translation table removes the burden of copying data across the protection boundary from the software, imposing it on the hardware portion of the interface architecture.

Overlapped operation (e.g., double-buffering) to and from user address spaces is somewhat more complex than for transfers to and from PDU copies kept in kernel buffers, due to the risks inherent in concurrent access to shared state by the device and the process. Two obvious approaches are: (1) blocking (i.e., ceasing execution of) the process until streaming is complete, and (2) trusting the process to not access the data (e.g., the process could do its own double-buffering). The first approach prevents a single process from using the hardware's capability for overlapped operation. This seems unwise (although it is what we do currently), since most applications use the CPU to transform data which travels to and from the network. The second approach assumes either intelligence or benevolence. However, as we have seen in practice, the inevitable crashes due to inconsistent data in the kernel punish other users for a transgression. A third approach is to force the process to block (cease execution) when it accesses a "busy" buffer. In this way, "well-behaved" processes can achieve maximum overlap, while AIX is protected from the indiscretions of "poorly-behaved" processes. This can be accomplished by tagging the active buffer's PFT entries with "fault-on-write"; the process is then blocked until the streaming transfer is complete and the page fault can be resolved. This combines the good features and removes the complications of the other two schemes, and is the approach currently being explored.

2.4. Timer Implementation

A periodic timer interrupt is generated using the AIX timer services [9]. The timer interrupt service routine examines the control tables in order to decide which actions are to be taken next. All operations are of short duration (e.g., examining the CAMs on the host interface card) so that several can be performed during the interrupt service routine. In addition, the status of the device and its internal tables are determined, in order to drain active VCs and receive reassembled CS-PDUs. Logical timers in the tables which have expired are updated and reset when service is performed.

AIX on the IBM RISC System/6000 Models 520 and 320 can support timer frequencies of 1000 Hz [9] before there is significant negative performance impact from timer processing. At a timer frequency of 60 Hz, at least 90% (worst case, ~98% average) of the processor capacity should remain available to applications. In one sixtieth of a second, about 6000 cells can arrive on an OC-3c at full rate, and the Reassembler buffer can accommodate about 7500 Cells. While less-frequent polling improves throughput and host performance, it has some potentially negative

consequences for latency; for example a 60 Hertz timer would give a worst-case latency of over 16.7 milliseconds before data reached an application, far slower than desired for many LAN applications [17]. We are currently studying the problem of setting the timer interval, but as discussed above, the timer interval should be a function of traffic. At this point in time, real traffic considerations are not well-understood.

A key feature of using timers is provision for *bandwidth allocation* by limiting per-connection data transfers. This turns out to be trivial with a clocked interrupt system, as PDUs can be delivered at a multiple of the base clock rate, or the number of PDUs per clock interval can be controlled. Since *read()* and *write()* serve to synchronize the process with data motion, a simple bandwidth allocation scheme is enabled. We control these allocations using parameters passed via *ioctl()*.

3. Performance

A key test of the various architectural hypotheses presented is their experimental evaluation; since many of these claims are related to performance, our experiments are focused on timing and throughput measurements, and analyses of these measurements. Since application performance is the final validation, any experiments should be as close to true end-to-end experiments as possible. In our case, data should pass from a user process (the application), through the software and hardware subsystems, to the network.

A simple program to gather timing measurements was written, of the basic form shown above in **Figure 2**. While the option-handling is not shown for the sake of brevity, the basic options include a repetition count, a buffer size, and a bit pattern with which to populate the buffer. This latter option was included so that recognizable data patterns would be produced on the logic analyzer used to monitor the experiments. The defaults used are 1, 65536 (bytes), and a counting pattern.

A script which varied the buffer size and number of repetitions to achieve a constant total of bytes was written. The parameters used ranged from a buffer size of 1KB and repetition count of 8K to a size of 64KB and a count of 128, yielding a total byte count of 8MB. The performance for buffer sizes of 1KB, 4KB, 16KB and 64KB were 31Mbps, 72Mbps, 108Mbps and 125Mbps, respectively. The complete set of tests is plotted in **Figure 4**.

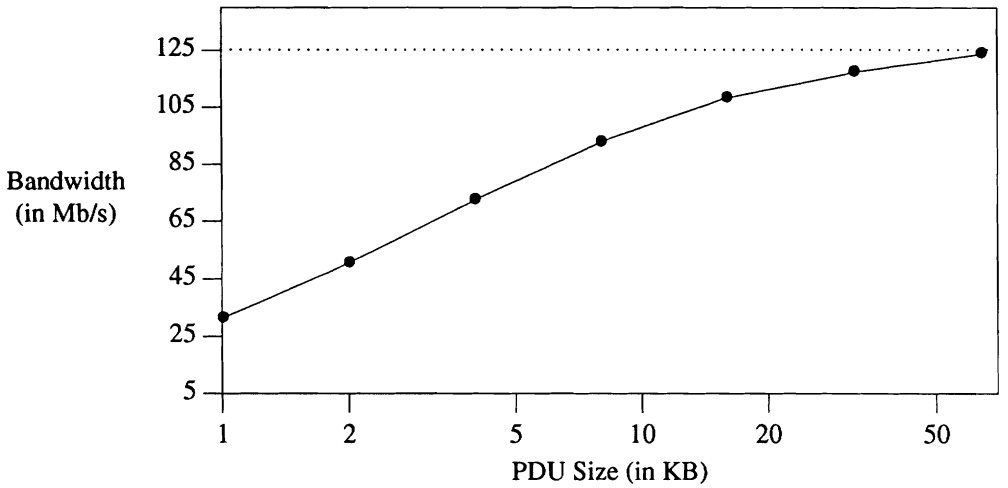


Figure 4: Measured performance, Mb/s vs. PDU size

The maximum measured performance we have observed is 130Mbps, which is about 93% of the limit of the bottleneck in the hardware subsystem, the I/O Channel Controller.

3.1. Discussion

Given the observed measurements summarized above, the software is not the system bottleneck. We have also been able to read data at these rates, thus there is no read vs. write asymmetry in performance.

These measurements may not reflect the throughput that would be seen by an application using a protocol suite such as TCP/IP, although they may reflect an upper bound on the throughput achievable with an

implementation.

The tests do not represent end-to-end throughput measurements between processors across the network, but rather rates sustainable by the host when delivering data to the network. Thus, they set an upper bound on the protocol performance for protocol architectures using our operating system work as a base.

3.2. Effect on unrelated applications

Since the "end-to-end" Gbps goal includes application processing, any solution must preserve the ability of the workstation to run applications while interacting with the network fabric. In addition, whatever solution is chosen must support classes of applications likely to exercise the system's capabilities. The operating system must ensure that the local portion of the application is able to gain sufficient processor resources to send or absorb its traffic.

Informal benchmarking done from another terminal while these scripts were run on a lightly-loaded IBM RISC System/6000 Model 320 showed little or no observable system performance degradation. A trivial test which competes for I/O and processing resources, a multimegabyte FTP copying data from a remote IBM PC/RT connected through an Ethernet, required about 5% more time. We are now performing formal studies using multifactor performance indices such as SPECmarks to estimate the impact of various tunable parameters in our implementation.

3.3. Analysis of Results

For small block sizes, software is the limiting factor to system performance. Smaller block sizes force the application to make frequent system calls, which force the AIX system to context-switch frequently. Larger block sizes reduce the per-byte software overhead, since the system calls are amortized over a larger data transfer. As this overhead becomes (relatively) smaller, the data transfer rate dominates the performance, and since the software does not participate in actual transfer to and from the device, the hardware performance limits bound the throughput. This can be seen by studying the relative performance gain for each doubling in block size. The performance is almost doubled as block size is increased from 1KB to 2KB, but the increase from 32KB to 64KB gives only a 10% gain.

We are now exploring strategies which can give us better performance for smaller block sizes. One such idea is the use of an area of shared memory to allow the kernel and applications to communicate without system calls, thus eliminating their performance impact.

4. Conclusions

Operating Systems employed in high-speed networks must reduce copying and provide support for isochronous traffic.

We have shown here one way to reduce copying by enabling data transfers directly to and from buffers located in application-process address spaces. The method has been demonstrated experimentally and shown to deliver high throughputs. Operating System support must also include scheduling, which allows bandwidth-allocated traffic streams to be delivered. The implementation we described provides resource scheduling for network users, and considerably reduces interrupt overhead.

Clocked interrupts have been tested over a range of values from 1Hz to 500Hz, and high throughput is delivered to applications. Setting the base rate is an interesting (and unsolved) optimization problem which trades higher throughput at low clock rates against lower delays at high clock rates. Until we have a real mix of applications traffic, it will be hard to intelligently set the value.

One important (and often overlooked) observation we would like to make about our interface is that it was remarkably easy to program. This was not an accident; the hardware and software developed together. The result of a simple programming model, though, is simple software. The simplicity of the software allows it to run efficiently, and eases later optimizations. One difficulty we have observed in practice with implementations employing on-board protocol processing is that communication with the interface requires a protocol [22]; we believe that this is undesirable.

All strategies are functions of their environment and the economics of various tradeoffs within that environment. When memory-bandwidth is constrained relative to network bandwidth, applications have requirements for

high-bandwidth isochronous traffic, and interrupts are expensive, these ideas appear useful.

5. Notes and Acknowledgments

Our collaboration with Bruce Davie has had a strong impact on this work. Dave Farber, Dave Sincoskie, Dave Tennenhouse, Marc Kaplan and Dave Clark have all provided insights and constructive criticism. David Feldmeier's suggestions made this a much better paper.

AURORA is a joint research effort undertaken by Bell Atlantic, Bellcore, IBM Research, MIT, MCI, NYNEX, and Penn. AURORA is sponsored as part of the NSF/DARPA Sponsored Gigabit Testbed Initiative through the Corporation for National Research Initiatives. NSF (Cooperative Agreement Number NCR-8919038) and DARPA provide funds to the University participants in AURORA. Bellcore is providing support through the DAWN project. IBM has supported this effort by providing RISC System/6000 workstations, and this work was partially supported by an IBM Faculty Development Award. The Hewlett-Packard Company has supported this effort through donations of laboratory test equipment and workstations.

RISC System/6000, AIX, PC/RT, PS/2 and Micro Channel are trademarks of IBM. Ethernet is a trademark of Xerox. TURBOChannel and DECstation are trademarks of Digital Equipment Corporation. HP9000 is a trademark of Hewlett-Packard. UNIX is a trademark of UNIX Systems Laboratories.

6. References

[1] *Gigabit Testbed Initiative Summary,* Corporation for National Research Initiatives, 1895 Preston White Drive, Suite 100, Reston, VA 22091 USA (January 1992). `info@nri.reston.va.us`

[2] D. Banks and M. Prudence, "A High-Performance Network Architecture for PA-RISC Workstations," *IEEE Journal on Selected Areas in Communications (Special Issue on High Speed Computer/Network Interfaces)* **11**(2) (February 1993).

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

[4] D. D. Clark and D. L. Tennenhouse, "Architectural considerations for a new generation of protocols," in *Proc. ACM SIGCOMM '90*, Philadelphia, PA (September 1990).

[5] D. D. Clark, B. S. Davie, D. J. Farber, I. S. Gopal, B. K. Kadaba, W. D. Sincoskie, J. M. Smith, and D. L. Tennenhouse, "An Overview of the AURORA Gigabit Testbed," in *Proceedings, INFOCOM 1992*, Florence, ITALY (1992).

[6] Eric Cooper, Onat Menzilcioglu, Robert Sansom, and Francois Bitz, "Host Interface Design for ATM LANs," in *Proceedings, 16th Conference on Local Computer Networks*, Minneapolis, MN (October 14-17, 1991), pp. 247-258.

[7] Eric C. Cooper, Peter A. Steenkiste, Robert D. Sansom, and Brian D. Zill, "Protocol Implementation on the Nectar Communication Processor," in *Proceedings, SIGCOMM '90*, Philadelphia, PA (September 24-27, 1990), pp. 135-144.

[8] IBM Corporation, *IBM RISC System/6000 POWERstation and POWERserver: Hardware Technical Reference, General Information Manual*, IBM Order Number SA23-2643-00, 1990.

[9] IBM Corporation, "AIX Version 3.1 RISC System/6000 as a Real-Time System," Document Number GG24-3633-0, Austin, TX (March 1991). International Technical Support Center

[10] Bruce S. Davie, "A Host-Network Interface Architecture for ATM," in *Proceedings, SIGCOMM 1991*, Zurich, SWITZERLAND (September 4-6, 1991), pp. 307-315.

[11] Bruce S. Davie, "The Architecture and Implementation of a High Performance Host Interface," *IEEE Journal on Selected Areas in Communications (Special Issue on High Speed Computer/Network Interfaces)* **11**(2) (February 1993).

[12] D. C. Feldmeier, "Multiplexing Issues in Communication System Design," in *Proc. ACM SIGCOMM 90*, Philadelphia, PA (September 1990), pp. 209-219.

[13] D. C. Feldmeier, *High Performance Protocol Meeting - Descriptions of Slides*, August 20-21st, 1992.

[14] D. C. Feldmeier, "A Framework of Architectural Concepts for High-Speed Communication Systems," *IEEE Journal on Selected Areas in Communications* **11**(4) (May 1993).

[15] David J. Greaves, Derek McAuley, and Leslie J. French, "Protocol and interface for ATM LANs," in *Proceedings, 5th IEEE Workshop on Metropolitan Area Networks*, Taormina, Italy (May 1992).

[16] N. C. Hutchinson and L. L. Peterson, "The *x*-Kernel: An architecture for implementing network protocols," *IEEE Transactions on Software Engineering* **17**(1), pp. 64-76 (January 1991).

[17] Hemant Kanakia and David R. Cheriton, "The VMP Network Adapter Board (NAB): High Performance Network Communication for Multiprocessors," in *Proceedings, ACM SIGCOMM '88* (August 16-19 1988), pp. 175-187.

[18] John Lumley, "A High-Throughput Network Interface to a RISC Workstation," in *Proceedings, IEEE Workshop on the Architecture and Implementation of High-Performance Communications Subsystems (HPCS '92)*, Tucson, AZ (February 17-19, 1992).

[19] S. O'Malley and L. L. Peterson, "A Dynamic Network Architecture," *ACM Transactions on Computer Systems* **10**(2) (May 1992).

[20] Calton Pu, Henry Massalin, John Ioannidis, and Perry Metzger, "The Synthesis System," *Computing Systems* **1**(1) (1988).

[21] Jonathan M. Smith, Eric C. Cooper, Bruce S. Davie, Ian M. Leslie, Yoram Ofek, and Richard W. Watson, "Guest Editorial," *IEEE Journal on Selected Areas in Communications (Special Issue on High Speed Computer/Network Interfaces)* **11**(2) (February 1993).

[22] Peter A. Steenkiste, "Analyzing Communication Latency Using the Nectar Communication Processor," in *Proceedings, SIGCOMM '92 Conference*, Baltimore, MD (August 17-20, 1992), pp. 199-209.

[23] K.L. Thompson, "UNIX Implementation," *The Bell System Technical Journal* **57**(6, Part 2), pp. 1931-1946 (July-August 1978).

[24] C. Brendan S. Traw and Jonathan M. Smith, "Hardware/Software Organization of a High-Performance ATM Host Interface," *IEEE Journal on Selected Areas in Communications (Special Issue on High Speed Computer/Network Interfaces)* **11**(2) (February 1993).

[25] Richard W. Watson and Sandy A. Mamrak, "Gaining Efficiency in Transport Services by Appropriate Design and Implementation Choices," *ACM Transactions on Computer Systems* **5**(2), pp. 97-120 (May 1987).