Giving Applications Access to Gb/s Networking

Jonathan M. Smith and C. Brendan S. Traw

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

ABSTRACT

Network fabrics with Gigabit per second (Gbps) bandwidths are available today, but these bandwidths are not yet available to applications. The difficulties lie in the hardware and software architecture through which application data travels between the network and host memory. The hardware portion of the architecture is often called a *host interface* and the remainder of the protocol stack is implemented in host software.

In this paper, we outline a variety of approaches to the architecture of such systems, examine several design points, and study one example in detail. The detailed example, an ATM Host Interface and Operating System support built at the University of Pennsylvania, illustrates design tradeoffs for hardware and software, and some of the implications of these tradeoffs on applications performance.

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’’. A good example is the AURORA Gigabit Testbed [6], one of several such testbeds [2]. 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. We believe 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, which as pointed out by Clark and Tennenhouse [5], forces careful design of protocol processing architectures.

Efficiency can be achieved through many design features, but the main options [29] are: optimizing the processing functions in the protocol architecture, optimizing the operating system support for data transport, and careful placement of the hardware required 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 [16].

The remainder of Section 1 outlines approaches to host interface hardware and supporting software. Section 2 describes the design choices made in an implementation of an ATM host interface for the IBM RISC System/6000 workstation [27]. Section 3 analyzes the performance implications of the design choices for applications, and Section 4 concludes the paper.

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

† 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* [23].

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 towards hardware is intended to obtain an implementation-specific performance advantage - as Watson and Mamrak [29] 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.

1.2. Host Interface Attachment

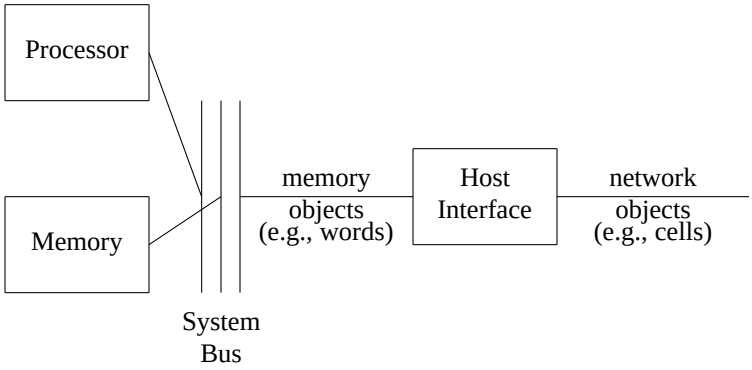

Figure 1: General Host Interface Architecture

Figure 1 illustrates a general architecture for a host interface and will allow us to discuss several options for host attachment. Davie [11, 13] and Ahlgren [15] have also discussed such options. We assess their performance potential for receiving data; sending data is similar.

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. A variant of this scheme would allow the host interface to directly allocate and free host memory, in effect turning management of the host's memory into a peer-peer model rather than a master-slave model. Thus, a given piece of data travels over the System Bus once, from Host Interface to Memory. One potential problem with this approach is that transport protocols may wish to defer data transfer to applications until header processing is complete.
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 rather, traversal is deferred until it is referenced by the processor. This makes the host interface ''look like'' a piece of system memory allocated to the application.
5. The Host Interface can be connected directly to the CPU [6], 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 typically 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 architecturally ‘‘close’’ to the processing unit, so it either must be kept consistent or flushed when new data arrive. Maintaining consistency is considerably easier when the data passes through the processor - Scheme 2 must flush the cache for areas affected by the DMA, and Scheme 4 must flush the cache, either under control of the CPU or the Host Interface. This problem can be avoided by putting the data in non-cached memory, but this may have other negative performance consequences. 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 may also be used to support 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) might present difficulties in I/O operations to and from other devices.

Finally, 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. Scheme 5’s co-processor approach likewise shares processing-unit capacity between computational load and network traffic.

Any evaluation of a host attachment strategy is subject to the specifics of the host, I/O bus, interface technologies and software, as well as the target applications. For these specifics, there are other constraints such as economics, portability, *etc*. Thus, there is no ‘‘best’’ approach - they can only be ranked relative to these constraints.

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 between a network and the computing system’s memory. The software builds an application communications model over this abstraction. 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 the 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 supporting continuous media. We believe that approaches optimized towards a particular traffic type, such as low-latency transactional traffic [24], 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 user’s applications are typically executable programs, or groups of such programs cooperating on a task. 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. 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. The device-dependent portion is implemented as a ‘‘device driver.’’ This is not strictly necessary as demonstrated in Mach 3.0, which moves both protocols and much of the device driver code out of the kernel. 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, or bus mastered transfers†), what is the partitioning of functionality between the host software and the host interface hardware? For example, use of DMA or bus-master transfers 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, performance for some applications might be gained by customizing stacks [5] using a new programmer interface. Multiple protocol stacks could be supported, at some cost in effort; this allows both older applications and new applications with greater bandwidth requirements to coexist. Methods such as the *x*-Kernel [18, 21] 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 [19]. An example would be using an interrupt to signal grosser events, such as half-full queues in the interface.

1.4. Communicating State Changes between Host and Interface

One of the key issues in the design of operating system features which support interactions with external events (such as arriving data) is the state exchange 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 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

---

† With programmed I/O, the CPU controls the transfer; with DMA a third party controls the transfer, and with bus mastered operation the peripheral controls the transfer [9].

implementation environment, but an illustrative example is given by the UNIX [25] *callout* table design, used for operating system management of pools of teletypewriter lines. Clocked interrupts represent a dynamic midpoint between polling and data-driven interrupts; appropriate clocking rates can make the scheme resemble either of the other two.

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 $\lambda \cdot k \cdot (C+\alpha)$. 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, for clocked interrupts delivered at a rate β per second, the capacity limit is $1 > \beta \cdot C + \lambda \cdot k \cdot \alpha$. 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 performance, 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.

Box 1: Clocked Interrupts

1.5. Example Host Interface Architectures

Table I summarizes some high-level design features for several host interface architectures presented in the literature.

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
Bus Arch.?	VME	VME	TURBOChannel	SGC	MCA	TURBOChannel	SBus

Table I: **Signaling, Units, Intelligence and Attachment**

Legend:
PDU - Protocol Data Unit, an object size dictated by the protocol
I - Interrupt
CI - Clocked Interrupt
** - Processor can signal arbitrary events, e.g., Cell, PDU, timeout, etc.*

Several interfaces have attempted to accelerate *transport protocol processing.* The VMP Network Adapter Board (NAB) [19] 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) [24] 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.* [8], 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.* [4] 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.

Table II lists additional features of some ATM network host interfaces. Connection multiplexing and demultiplexing refers to combining data from different connections into a cell stream when sending, and splitting connections from the cell stream when receiving.

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

Table II: **ATM-oriented features**

Legend:
HW - Hardware
HSW - Host Software
OSW - Outboard Software

Less protocol processing is performed by two ATM host interfaces developed at Bellcore and Penn. Bellcore's [12] 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. Like the CAB, this host interface relies on programmable processors, in this case, two Intel 80960 RISC microcontrollers, to perform protocol processing and flow control. In this case, the programmability is targeted primarily at exploring various segmentation-and-reassembly (SAR) algorithms. At this time [13], Bellcore's interface provides the highest burst performance reported for an ATM host interface.

Off-board processors can migrate many processing and data movement tasks away from the host CPU. However, it is not clear that a general purpose processor is the most efficient mechanism to support these services. At Penn, we have developed a scalable host interface architecture for providing segmentation and reassembly functions in dedicated logic [27]. The architecture can support link speeds from 160 Mbps to over a gigabit per second [26]. All per cell processing including ATM header and ATM Adaptation Layer (AAL) 3/4 processing as well as demultiplexing are performed by the hardware. An initial implementation which can support link speeds of up to 160 Mbps 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 [28] for use with the Hewlett-Packard HP9000/700 series workstations equipped with Afterburner [20] cards. The Afterburner/ATM Link Adapter combination will provide the same basic functionality as the initial implementation with two exceptions. First, the AAL5 will be supported, and the CRC32 will be generated and checked as AAL5 PDUs are moved between the ATM Link Adapter and 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. [7], and Cambridge University/Olivetti Research [17], 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, e.g., computing checksums, checking segment numbers, *etc.*

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 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.

2. The Penn ATM Interface

To illustrate some of the detailed design choices and their consequences, Section 2 examines an implementation in considerable detail. The performance implications are studied later, in Section 3.

2.1. A Programmer's View of the Hardware

The Host Interface is implemented as a pair of cards, one card for transmission and the other for reception. The transmission card segments variable-sized Protocol Data Units (PDUs) into fixed-size ATM cell bodies, generates the ATM header and the AAL, and transmits the cell. The programmer must specify information for the header, the location and size of the PDU to segment, check the status of the interface, and start the transfer.

The Reassembler card receives multiplexed streams of ATM cells, which it demultiplexes using fields in the ATM cell header and AAL into a number of queues, one queue per virtual circuit or PDU. The queue numbers are used as names by the host, which absorbs data by presenting a queue identifier, a location to place the data, and a cell count, after which it initiates a transfer.

Thus, the programmer has a relatively simple model, consisting of some queue management facilities, status and synchronization information, and a number of control operations. All data movement operations are carried out by the hardware on the variable-sized PDUs stored in system memory. This support for 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.* Calls are costly due to their generality, as well as crossing of an application/kernel address space protection boundary. Pu, *et al.* [22] 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 [25] 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. Various data structures in the Reassembler are initialized, and a sequence of set-up operations for the software such as initializing the AIX device switch table are performed. The set-up operations also 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. **Figure 2** illustrates a simple program to access the interface.

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

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

if (write(fd, buf, count ) == -1)
   perror("write failure");
```

Code fragment to 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 Virtual Circuit Identifiers (VCIs) and Multiplexing Identifiers (MIDs) for use by the hardware when formatting ATM cells. The VCI and MID are currently specified to the driver on a per-file descriptor basis for efficiency, since for a stream of PDUs they would typically not change. The VCI and MID 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 [29] and confirmed in other implementations [3]. 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.

When the *write()* call is invoked on the device, the virtual addresses of user data are available to the driver through a `uio` structure element. The `uio` structure element is used to mark the pages in the application virtual address space as *pinned*, and to obtain a ''cross-memory descriptor'' which allows the application data pages to be addressed by a device on the Micro Channel bus using real addresses, as illustrated in **Figure 2**. For the *read()* call, the pages are pinned as well. Use of pinned storage may present difficulties for system memory allocation if many channels with large PDU sizes are active concurrently.

When appropriate hardware-provided status flags indicate the device is inactive, a bus-master transfer is set up. The software prepares for bus-mastered transfers by initializing a number of translation control words (TCWs) [9] 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 a lookup table 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 2**.

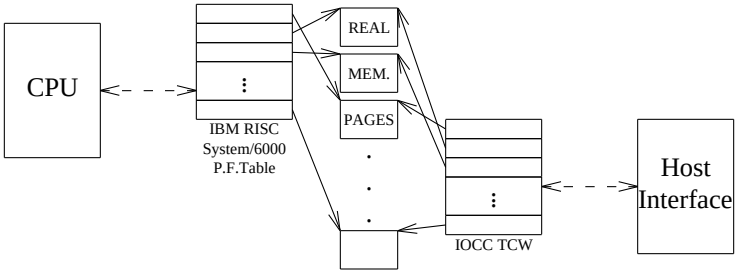

Figure 2: Illustration of TCW and PFT usage

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. The major tradeoff here is that the pages used must be pinned into real memory while these operations are taking place, thus removing them from the pool normally available to applications.

2.4. Timer Implementation

We have chosen clocked interrupts as the signaling mechanism for our host interface because of the operating environment. In particular, as pointed out in [14] 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,

† In the IBM RISC System/6000 Model 580, a newer implementation called the XIO is used. It provides considerably higher performance.

but *scheduling* is the operating system mechanism most suited to allocating bandwidth, as it is a form of time-division.

A periodic timer interrupt is generated using the AIX timer services [10]. 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 [10] before there is significant negative performance impact from timer processing. At a timer frequency of 60 Hz, less than 2% of the processor cycles are used for clocked interrupt processing on a RISC System/6000 Model 580, as measured by the `sar` command [1]. In one sixtieth of a second, about 6000 cells can arrive on an OC-3c at full rate, and the Reassembler buffer can accommodate over 7000 Cells [28]. 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 [19]. We are currently studying the problem of setting the timer interval. As discussed above, the timer interval should be a function of traffic characteristics, and at this time, real traffic considerations for high-speed cell networks 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. The basic controls for the measurement program include a repetition count, a buffer (PDU) size, and a bit pattern with which to populate the buffer.

A script which varied the PDU size and number of repetitions to achieve a constant total of bytes was written. The parameters used ranged from a PDU 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 complete set of tests is plotted in **Figure 3**.

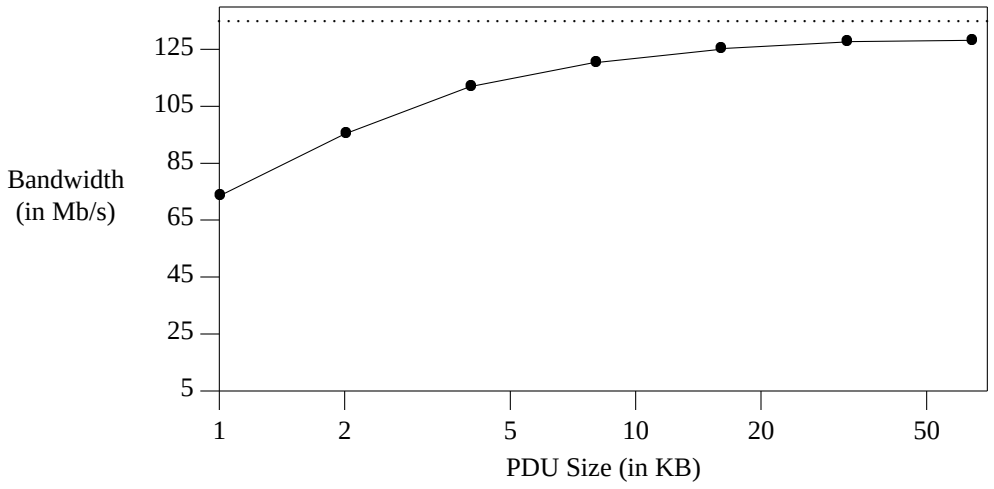

Figure 3: Measured performance, Mb/s vs. PDU size (RISC/System 6000 Model 580)

On the IBM RISC System/6000 Model 320, the bottleneck is the I/O Channel Controller (IOCC) which limits the performance to about 130 Mbps, while on the XIO-equipped IBM RISC/System 6000 Model 580, the physical layer data rate is the bottleneck at 135 Mbps. We have measured sustained I/O bus to memory transfer rates of 270 Mbps on the XIO architecture on the IBM RISC System/6000 Model 580.

Given the observed measurements summarized above, the software is not the system bottleneck. We can also read data at these rates, thus there is no read vs. write asymmetry in performance. The throughput seen by an application using a protocol suite such as TCP/IP may be less than shown here due to additional complexity. However, these measurements reflect an upper bound on the throughput achievable with *any* implementation which copies data to and from host memory.

3.1. 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 traffic.

We performed a series of experiments to quantify the effect of the host interface and supporting software on overall system performance. Our goal was estimating the remaining applications processing capacity when the networking subsystem was being exercised. We had two measures: the effect on the availability of CPU cycles, and the effect on main memory bandwidth. We discuss each of these in the following two subsections.

3.1.1. CPU Impact

The measurement apparatus was an IBM RISC System/6000 running AIX 3.2. We used the `sar` command [1] to observe the fractions of processing unit capacity devoted to user applications, system processing, waiting for I/O, and the ‘‘idle’’ loop. The command was run using a monitoring interval of 1 second and the data logged to a file for analysis. The overhead of operating the measurement apparatus and the other background activity on the system was about five percent, and we subtracted this overhead from the measurements in order to focus on the effects of the networking subsystem. The factors of interest were the workload, the use of the networking subsystem, and the impact on both networking and applications performance. Workload was generated by concurrently running five processes which repeatedly attempted to factor the prime *11,111,111,111,123*. The processor is called Loaded when this workload is being processed, and is otherwise Unloaded. A total of 4.8 million bytes were transferred using three different PDU sizes, as shown in *Table III*.

PDU Size	48 bytes	4,800 bytes	48,000 bytes
Transfer Time, Unloaded CPU	0.50 sec.	0.34 sec.	0.30 sec.
Transfer Time, Loaded CPU	5.4 sec.	0.75 sec.	0.46 sec.
CPU % Remaining for Application	50%	65%	70%

Table III: **CPU Availability versus networking**

3.1.2. Memory Impact

Memory impact was studied on the same RISC System/6000 Model 580 using a program we designed which measures the performance of the main memory subsystem. It does this by striding across an area of memory larger than the system's cache, using cache-line sized steps. The stepping serves to minimize the number of instructions required for the measurement, while the choice of the size of the memory area serves to ensure that caching is not effective.

The program reads a system clock, and then repeatedly strides through the memory for a fixed number of repetitions. After the memory strides are done, the system clock is re-read, and the memory bandwidth is computed. This computed memory bandwidth is then output by the program. On the Model 580, the baseline memory bandwidth was measured to be about 2.5 Gbps by this program. In comparison, on an IBM RISC System/6000 Model 320, the observed bandwidth was about 1Gbps; on a Model 530H, the observed bandwidth was about 1.8 Gbps.

Experiments were done by repeatedly executing the bandwidth measurement program and writing its results to a log file for later analysis. The networking subsystem was exercised, and its effects on the memory bandwidth analyzed from the data in the log file. Both 48 byte and 48,000 byte PDU sizes were used to transfer data. In both cases, the total impact on measured memory bandwidth (including the execution of the networking test application program, the device driver, and the transfer of data between the application and host interface) was less than 50%. A more careful analysis of the measurements shows that timesharing the processor between the networking application and the bandwidth measurement program reduces the observed bandwidth considerably, since the bandwidth utilized when the program is not executing is zero. It is still able to obtain over 1.5Gbps of main memory bandwidth. The implication is that there is a large fraction of memory bandwidth available for applications processing.

These CPU usage and memory bandwidth measurements suggest that less extremal applications may easily operate concurrently with the networking subsystem. For example, informal benchmarking done while the networking subsystem was operating on a lightly-loaded IBM RISC System/6000 Model 320 showed little or no observable system performance degradation. A simplistic 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 wall-clock time.

3.2. Analysis of Results

With reference to **Figure 3**, for small PDU sizes, software is the limiting factor to system performance. Smaller PDU sizes force the application to make frequent system calls, which force the AIX system to context-switch frequently. Larger PDU 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 PDU size. The performance is increases greatly as PDU size is increased from 1KB to 2KB, but the increase from 32KB to 64KB gives only a slight gain.

3.3. Double-Buffered Shared Memory Interface

We are now exploring strategies which can give us better performance for smaller PDU 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. This should have a particularly strong effect on the performance of the software for small PDU sizes, as system call overhead was more significant for small PDUs, as illustrated in **Figure**

4.

In this model, the O.S. Kernel and Application share address space, as illustrated in **Figure 4**. This implementation removes the per-PDU system call by using shared status flags for kernel/user communication and synchronization. The use of buffer pairs allows the CPU and bus master card operate concurrently, and thus derive performance improvements from ‘‘overlapped’’ transfers.

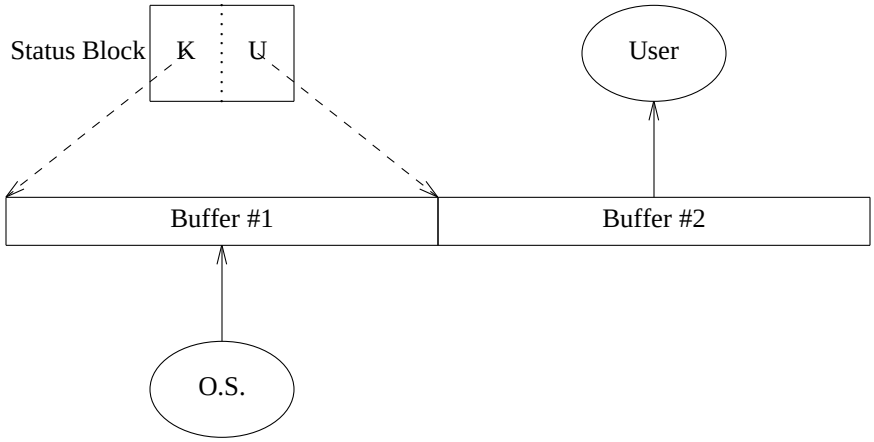

Figure 4: Shared Memory Buffer Model (Reading)

We have implemented this scheme for the reassembler device driver, and the measurements for an IBM RISC System/6000 Model 580 used in a loopback mode are promising; the reassembler shared-memory model software easily keeps up with the maximum performance of the segmenter card and its system-call interface model software.

4. Conclusions

Operating Systems employed in high-speed networks must reduce copying to deliver maximal throughput to applications, and they must deliver this throughput *while preserving the capability of the host to perform applications processing.* We believe that they must also provide efficient support for delay-sensitive traffic, as many proposed high-bandwidth applications incorporate such 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.

Finally, the use of a shared memory programming paradigm reduces the impact of system call overhead, and allows overlapped operation of the processor and the host interface’s data transfer functions.

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 were 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 more complex protocol [24] than might otherwise be needed.

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 continuous-media traffic, and interrupts are expensive, these ideas appear useful.

5. Notes and Acknowledgments

Our collaboration with Bruce Davie has had a strong intellectual impact on this work, and his comments greatly improved an earlier version of this article. Dave Farber, Dave Sincoskie, Dave Tennenhouse, Marc Kaplan and Dave Clark have all provided insights and constructive criticism. David Feldmeier's and K. K. Ramakrishnan's suggestions improved our presentation.

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] ''SAR (System Activity Report) Command,'' in *IBM AIX Version 3 for RISC System/6000 - Commands Reference, Volume III* (March 1990), pp. 1969-1972.

[2] *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`

[3] D. Banks and M. Prudence, ''A High-Performance Network Architecture for a PA-RISC Workstation,'' *IEEE Journal on Selected Areas in Communications (Special Issue on High Speed Computer/Network Interfaces)* **11**(2), pp. 191-202 (February 1993).

[4] 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).

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

[6] 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).

[7] 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.

[8] 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.

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

[10] 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

[11] Bruce S. Davie, ''Host Interface Design for Experimental, Very High Speed Networks,'' in *Proc. SPRING Compcon*, San Francisco, CA (February 1990), pp. 102-106.

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

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

pp. 228-239 (February 1993).

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

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

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

[17] 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).

[18] 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).

[19] 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.

[20] 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).

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

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

[23] 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), pp. 169-172 (February 1993).

[24] 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.

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

[26] C. Brendan S. Traw, ‘‘Host Interfacing at a Gigabit,’’ Technical Report MS-CIS-93-43, CIS Department, University of Pennsylvania (April 21st, 1993).

[27] 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), pp. 240-253 (February 1993).

[28] J. T. van der Veen, C. Brendan S. Traw, Jonathan M. Smith, and H. L. Pasch, ‘‘Performance Modelling of a High Performance ATM Link Adapter,’’ in *Proceedings, Second International Conference on Computer Communications and Networks*, San Diego, CA (June 28-30, 1993).

[29] 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).