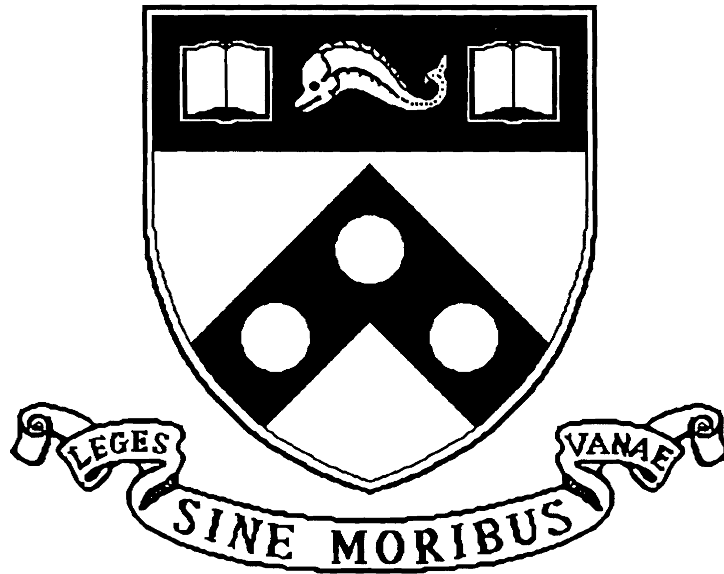
Experimental Evaluation of An ATM Host Interface

MS-CIS-93-18
DISTRIBUTED SYSTEMS LAB 16

Chandler Brendan Stanton Traw
Jonathan M. Smith



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

February 1993

Experimental Evaluation of an ATM Host Interface

C. Brendan S. Traw (traw@aurora.cis.upenn.edu)

Jonathan M. Smith (jms@cis.upenn.edu)

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

ABSTRACT

We have previously reported a design for a host interface board intended to connect workstations to ATM networks, and an implementation that was underway. Since then, we have made some modifications to the hardware architecture (mainly to support the Class 4 ATM Adaptation Layer), nearly completed the hardware implementation, and implemented software support. Our prototype connects an IBM RS/6000 to a SONET-based ATM network carrying data at the OC-3c rate of 155 Mbps.

In this paper, we discuss an experimental evaluation of the interface and supporting software. Our experiments uncovered an unexpected bottleneck in providing high bandwidth to application processes, and we suggest a number of possible improvements to workstation architectures to address this bottleneck.

1. Introduction

Despite rapid advances in workstation processor and memory subsystem performance, the next generation of high speed (Gbps), wide area networks [11] threatens to exceed the data management capabilities of the hosts. To assist these hosts, specialized host interfaces are being developed at Penn [12], Bellcore [7], Carnegie-Mellon/Fore Systems [4], and elsewhere.

The host interface work at Penn has been centered on developing a high-performance host interface for workstation hosts in the AURORA Gigabit Testbed environment [3]. We have chosen to focus on workstations since we believe that they will be the predominant processor class connected to such networks.

1.1. Goals and Design Philosophy

One important outcome of the work is a high-performance host interface for IBM RS/6000 [1] workstations in the AURORA testbed, but our research goals are somewhat more ambitious and far-reaching. In particular, we wanted:

(1) A hardware/software architecture which is flexible and allows experimentation with portions of the protocol stack.

(2) A focus on architectural solutions to achieve good cost/performance, so our results scale across technology choices.

(3) Low absolute cost, so that large-scale replication is possible.

We believe that the resulting host interface will meet these goals. The design philosophy for our architecture is based on providing a ''common denominator'' set of services in dedicated hardware. All per cell activities such as CRC creation and verification, segmentation, and reassembly are performed in high density programmable logic. The host is responsible for all higher level activities. We anticipate this combination will meet our goals and provide an excellent balance between performance and flexibility.

Other possibilities include a minimal hardware approach [4] or the use of powerful offboard processing engines [7]. The minimal hardware approach, characterized by the assignment of almost all tasks to the workstation host including adaptation layer processing, 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. However, such an approach takes advantage of aggressive workstation technology improvements.

Powerful offboard engines are attractive from a parallel processing point of view, since they migrate processing and data movement control away from the host CPU. In addition to this performance, significant flexibility is gained from the reprogrammability of the host interface behavior. However, this approach is more costly, and extremely careful programming is required to achieve tight performance goals, especially when portions of multiple protocol stacks must be supported.

Since we last reported on this work [12], we have been carrying our design philosophy through to a realization. Here, we update our discussion of the architecture, detail the host software, and present performance results.

2. Hardware

This Host Interface is comprised of two logical sections, each of which occupies a standard sized Micro Channel Architecture [5] (MCA) board in the RS/6000. These two logical sections are the Segmenter and the Reassembler. The architecture is briefly reviewed in the following two sections. The primary change from our earlier discussion [12] is the inclusion of logic to process the Class 4 ATM Adaptation Layer (AAL) [9]. This specialized AAL hardware does not preclude the use of other adaptation layers, as they can be implemented in host software.

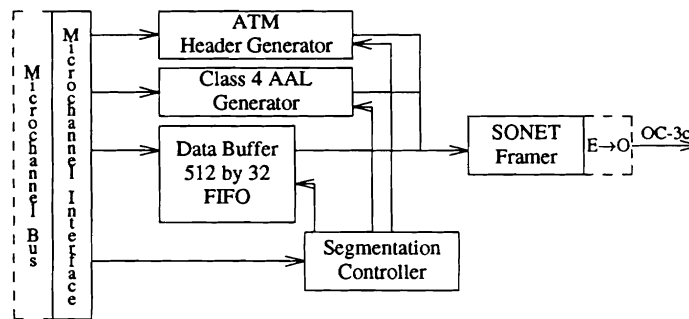


Figure 1: Segmenter

2.1. The Segmenter

A block diagram of the Segmenter is presented in Figure 1. When data is to be transmitted into the network, the virtual circuit identifier (VCI) to be used is loaded into the header generator. A multiplexing identifier (MID) is also loaded into the AAL generator if the data is to be transmitted via the Class 4 AAL. The host then sets up a streaming mode [6] (an optimized bus transfer mode for contiguous data) transfer to move the data which is to be transmitted from a pinned buffer in host memory to the FIFO buffer on the Segmenter. The location and size of the host's buffer are specified during stream set-up. While this transfer is being made, the Segmenter produces the header check CRC and formats the control information into the appropriate ATM and AAL header formats. As soon as sufficient data has been placed into the FIFO buffer, the segmentation controller removes the data for the first cell from the FIFO buffer and adds an ATM header, AAL header and AAL trailer. If the cell is carrying Class 4 data, the payload CRC is calculated as the data is moved to the SONET framer [10] and placed in the appropriate field at the end of the cell. This process is repeated until the FIFO buffer is drained.

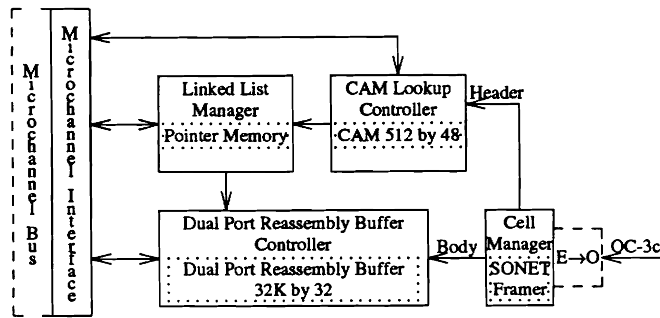


Figure 2: Reassembler

2.2. The Reassembler

The Reassembler is illustrated in Figure 2. It is composed of four major subsections which operate concurrently. The cell manager, CAM lookup controller, and the linked list manager (LLM) exploit this concurrency to form an ATM cell-processing "pipeline".

The cell manager verifies the integrity of the header and payload (if the cell is carrying Class 4 data) of cells received from the network by the SONET framer. It extracts the VCI from the ATM header and the MID and length from the AAL header. We currently ignore the 4-bit sequence number in the Class 4 AAL as we believe it is insufficient to provide a reliable means for cell loss detection. The body of the cell is placed in a FIFO buffer for later movement into the dual port reassembly buffer (DPRB).

The CAM lookup controller manages two CAMs which provide lookup support for a total of 256 simultaneous virtual connections and the reassembly of 256 datagrams. The host is able to flush undesired virtual circuits and datagrams from the reassembler through the CAM lookup controller.

A reference resulting from the CAM lookup operation is passed to the LLM. The LLM, as its name suggests, establishes and maintains a linked list data structure for each of the virtual circuit and datagrams that is being received. Data received from the network is placed at the end of the appropriate list while data destined for host memory is read from the front of the list.

The LLM allocates space in the DPRB for data coming into the reassembler from the network and passes the location to the DPRB controller. The cell body which was placed into the FIFO by the cell manager is removed and written into the appropriate reassembly buffer.

The host is able to read data from a particular virtual circuit or datagram by specifying a list reference to the LLM which determines the location in the DPRB where data is stored. The location is passed to the DPRB controller which removes the data from the buffer for transfer into host memory over the MCA bus.

3. Software

The current host interface support software consists of AIX character-special device drivers. Several have been implemented so that we could test various hypotheses about the effects of data copying on achieving high performance. We used AIX's capability to support dynamically-loadable device drivers; this allowed us to make progress despite the unavailability of kernel source code. The next section describes startup processing common to all drivers; later sections are devoted to particular driver functions and alternative implementation strategies.

3.1. Driver Set-up and access

The driver can be configured into the system at boot time if the device is detected, or later under program control. The host interface presents a unique device identifier when probed, and this identifier is used to gather descriptive information (including driver routines) from a system object database. Configuration includes allocating addresses for use by the device; the device uses these addresses for its control registers and to support streaming mode transfers.

The interface is initialized when the device special file */dev/host{n}* is first opened (*n* is a small integer, 0 on our test system). Initialization consists of probing the device at a distinguished address which causes it to be reset, as well as performing various set-up operations for the device driver software. The operations currently include pinning the driver software's pages into real memory and, for the driver discussed in Section 3.2.1, allocating two 64K-byte contiguous buffers which are also pinned. After initialization, the device and driver are ready for operation; routines for all appropriate AIX calls (e.g., *read(), write(), ioctl(), etc.*) are provided. The code fragment in the Appendix illustrates how a programmer would access the device for writing; this particular fragment is taken from the measurement apparatus we used for the data of Section 4.3.

3.2. Segmenter Software

When the *write()* call is invoked on the device, data is copied from the user address space into one of the 64K buffers. When a hardware-provided status flag on the segmenter indicates the device is inactive, a streaming mode transfer is set up by initializing a number of translation control words (TCWs) [6] in both the RS/6000 and in the Micro Channel's I/O Channel Controller (IOCC). The TCWs allow both the device and the CPU to have apparently contiguous access to scattered pages of real memory. This is illustrated in Figure 3.

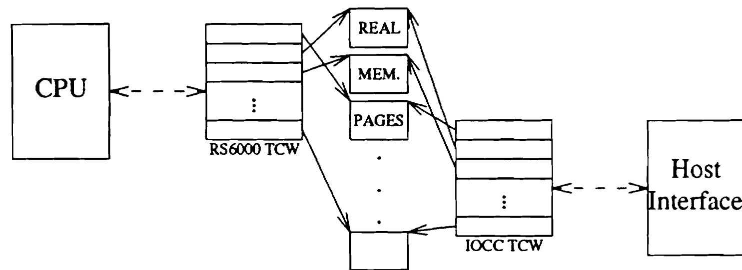


Figure 3: Illustration of TCW usage

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

3.2.1. In-kernel buffering

At this point, the driver marks the other buffer inactive and returns control to the user process. This combination of a hardware-provided state flag and double-buffering permits overlapped operation of the host interface and the host processing unit.

While this architecture supports overlapped operation, the copying between user and kernel address spaces is a major impediment to high-performance operation. The provision for TCWs in the IOCC allows large contiguous transfers directly to and from the address space of an AIX user process. We have another prototype device driver which supports such transfers.

3.2.2. User-process buffering

Overlapped operation from user address spaces is somewhat trickier than from 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 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 is easily accomplished by tagging the active buffers TCW 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 we intend to try next.

3.3. Reassembler Software

The reassembler software is considerably more complex than the segmenter software, since its activation is controlled by external events such as arriving cells. We have avoided the use of interrupts in our interface system [12] due to the software overhead, since with rapid arrival of small data objects (such as ATM cells), the interrupt service time can exceed the data service time. This remains true for considerably larger aggregations of cells. Without interrupts, however, the host is obligated to poll the interface. For the segmenter, we poll for completion of a streaming transfer using a status register value indicating that the card is idle; only performance is affected if we are delayed in observing a transition. On the receiver, however, the consequence of a missed state transition may be lost data and severe state inconsistencies between the host and the interface. Thus, the design of the reassembler software requires support for real-time operations (such as clock-driven polling) and must perform well to keep up with arriving traffic. Much of the additional software complexity of the receiver is support for polled operation.

As described before, the current support software is implemented as an AIX device driver. The reassembler software operates using mainly three AIX system calls, *open(), ioctl(),* and *read().* Before initialization (for example, by loading at system boot time or later), the device driver is inactive, but since there are no interrupts from the device, this does not affect system integrity. At initialization, a number of data structures are created and processes dependent on these data structures are begun. The three main data structures are the VC table, the DG table, and the POLL table.

3.3.1. VC table

The hardware supports 256 Virtual Circuit Identifiers (VCIs); a 256-entry table is used to track activity on each virtual circuit. Each array element is of type *vc_t*:

```
typedef struct {
    int vc_status;       /* status flags for this VC                */
    struct xmem vc_x;    /* pinned pages, d_master()ed area         */
    caddr_t vc_buf;      /* parameters for buffer mapped to...      */
    int vc_len;          /* ...this virtual circuit                 */
    vci_t vc_next;       /* identifier of next active VC            */
    vci_t vc_prev;       /* identifier of previous active VC        */
    long vc_poll_rate;   /* polls per second                        */
    long vc_poll_time;   /* clock time for next  poll               */
    dg_list_t vc_dgs;    /* list of datagrams on this VC            */
} vc_t;
```

The `vc_x` entry is a "cross-memory descriptor" used by AIX to control transfers between processor virtual address spaces (e.g., user space) and the Micro Channel's virtual address space. The `vc_buf` pointer and the `vc_len` byte count are used to prevent overwrites of user data, unpin pages when a particular VC is closed, and to remove the cross-memory mapping. The polling strategy uses the `vc_next` and `vc_prev` entries to maintain an active list; the `vc_poll_rate` and `vc_poll_time` entries also exist to support polling. The `vc_dgs` entry points to any datagrams which may have arrived on this virtual circuit.

3.3.2. DG table

Connectionless data transmission is also supported by the AAL [9], multiple datagrams may arrive over a single VC. The reassembler board assembles the datagram and when the datagram is complete, a transfer can be initiated into a processor memory area. We currently transfer data from the board into a 64Kbyte buffer allocated from the kernel's pinned heap. After the datagram has been transferred from the interface, its length is available from a device register; this length is used to copy the data into buffer areas provided by the user process. The DG table has 256 entries, each of which is quite similar to the VC table entry illustrated above, and is of type `dg_t`. The polling parameters are deleted, there are no cross pointers to other tables, and no pointers to support doubly-linked lists. The `vc_dgs` entry of the VC table is supported with `dg_list_t`, which is the head of a singly-linked list of DG table entries.

3.3.3. POLL table

The POLL table is constructed to support polling operations on the VCs; it implements a linked list of pointers to VC table entries. The linked list (which bears considerable resemblance to the data structures used by many UNIX TTY drivers) is sorted on `vc_poll_times`, so that the head of the list immediately yields the time to next poll; subtracting the current system clock time from this value yields the time with which a fine-granularity alarm timer is set. Insertion into the list is potentially expensive, since insertion into the ordered list takes linear time. However, the lookup required for polling and deletion of the processed table entry are constant-time operations.

4. Performance Measurements

In this section, we focus on performance in a ''bottom-up'' fashion. First, we analyze the Segmentation and Reassembly hardware. We then analyze the transmit data path performance when this hardware is connected to the IBM RS/6000 Model 320's implementation of the MCA. Finally, we study the performance of the entire hardware/software transmission architecture using the AIX device drivers discussed in Section 3.

4.1. Segmentation and Reassembly Hardware

As of January 1992, the Segmenter has been prototyped and tested except for the the AAL generator; the Reassembler prototype is nearly complete and has been partially tested.

Important performance measurements of the Segmentation and Reassembly hardware are as follows:

- Header generation in the Segmenter requires 5 clock cycles (250 ns).
- In the Reassembler, the Cell Manager is able to verify cell integrity (both header and class 4 payload via CRC check) and extract necessary control fields in one cell time (2.6 μs at the OC-3c rate).
- The worst case per cell operation on the CAM Lookup Controller requires 11 clock cycles (550 ns).
- The longest per cell operation performed by the LLM requires 12 clock cycles (600 ns). NOTE: This result is based on device level simulations.

Since multiple Cell Managers can be used in parallel to interface to higher bandwidth trunks composed of multiple OC-3c connections, it is useful to make a rough calculation of the performance of the Reassembler's ATM cell processing pipeline.

Assuming that the Reassembler is not required to service any host requests, the limiting component in the pipeline is the LLM. Since the worst case per cell operation requires 600 ns, and there are 424 bits per cell, the pipeline bandwidth is about 700 Mbps. In actual operation, this bandwidth would be reduced by up to 50% since the host must also utilize the LLM to drain cells from the reassembly buffer.

4.2. MCA Bus

We have thoroughly studied the performance of data transfers between the host's main memory and the host interface across the MCA Bus. An RS/6000 Model 320 was used for these measurements. We have chosen 32 bit bus-mastered streaming transfers to be our primary mechanism for providing bulk data movement across the bus. Streaming is a specialized transfer mode where the time required to initiate a transfer can be amortized over numerous data transfer cycles, potentially resulting in nearly twice the bus bandwidth. In addition, being a bus master allows concurrent access to memory by the device and the host CPU. This concurrency is significant, as we had achieved about 100 Mbps using programmed I/O, but the CPU was totally dedicated to this task and hence unavailable for application execution.

Using 32 bit streaming transfers, we have found that the bus itself is capable of sustained data transfers of slightly less than 320 Mbps, its peak rate for 32 bit transfers. These data rates have been observed card-to-card between peripheral cards on the MCA bus.

Unfortunately, when transferring data between the host's main memory and the host interface, significantly lower performance is observed. The difficulty lies in the I/O Channel Controller (IOCC). The IOCC is the connection between the MCA bus and the interface processor/memory bus of the RS/6000. To minimize the latency of the host's main memory, the IOCC assigns a set of buffers to each arbitration level used on the bus. Each set of buffers is composed of sixteen 32 bit words. Thus, when a write bus cycle is initiated, the data to be transferred must be loaded into the IOCC buffer from main memory before it can be transferred across the bus to the destination. While

these buffers are being loaded, the bus is in a suspended state awaiting the completion of the loading.

We have characterized the bus behavior using a logic analysis mainframe with 10 ns resolution. The measured period of bus suspension ranges between 2 and 3 μs. Once the IOCC buffer has been loaded, only 1.6 μs (or 100 ns per word) are required to transfer the 16 words across the bus. Thus, not more than about 44% of maximum bus bandwidth can be utilized for transfers involving the host's main memory. This duty cycle would suggest that the maximum obtainable bandwidth of the RS/6000 Model 320's MCA bus is slightly less than 142 Mbps when a peripheral is bus master.

We believe that a small increase in IOCC complexity, e.g., enough parallel operation to permit double-buffering when streaming transfers are enabled, would allow us to maximize bus performance. We expect that later models of the RS/6000 will contain an improved IOCC to allow higher bus performance.

4.3. Software and System Performance

4.3.1. Experimental Setup and possible sources of error

A short AIX program to gather timing measurements was written, of the basic form given in Appendix I. 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 patterns could be generated on the bus for display on the logic analyzer. The defaults used are 1, 65536 (bytes), and a pattern of bytes derived from a counter. A modified version of this program which recopies the pattern into the buffer before each *write()* system call was also used in the tests (this program gives rise to the solid lines marked ''with copy'' in Figures 4-6); the primary version of the program does not do this, as our focus was the performance of our hardware/software architecture, not RS/6000 data movement performance. However, the additional copy may make the measurements more relevant for protocol stacks built above our architecture.

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. While this may have been too short a test, we verified the measured values by rerunning the 64KB cases with a repetition count of 32K, and this case (2GB) matched the shorter case to 3 significant digits. All measurements are repeatable to 3 significant digits of accuracy; at this point, there is ''noise'' due to such factors as background activities on the processor and AIX timing granularity.

These measurements do 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 of Clark and Tennenhouse's Application Layer Framing and Integrated Layer Processing [2]. 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.

4.3.2. Measurements

Shown in Figure 4 is the performance of the hardware/software combination for the device driver implementation (described above in Section 3.2.1), where the AIX kernel copies the user buffer data into a kernel buffer and then initiates a streaming transfer using the kernel copy of the buffer as a source.

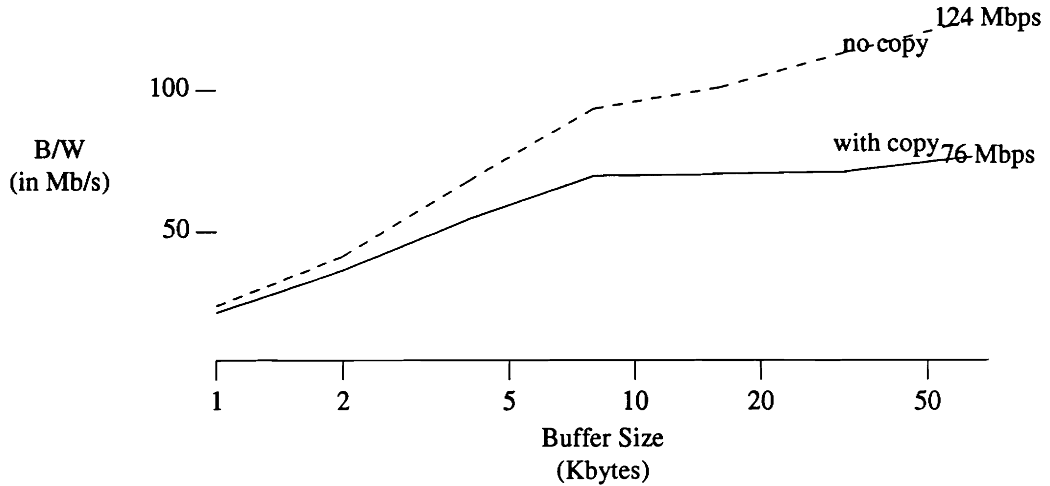


Figure 4: Performance of `test_wr`, streaming from kernel buffers

Figure 5 shows the performance of a driver (please refer to Section 3.2.2) which copies the data directly from the user address space using the RS/6000's facilities for virtual address translation. The device driver also includes some minor optimizations discovered after the first driver was written.

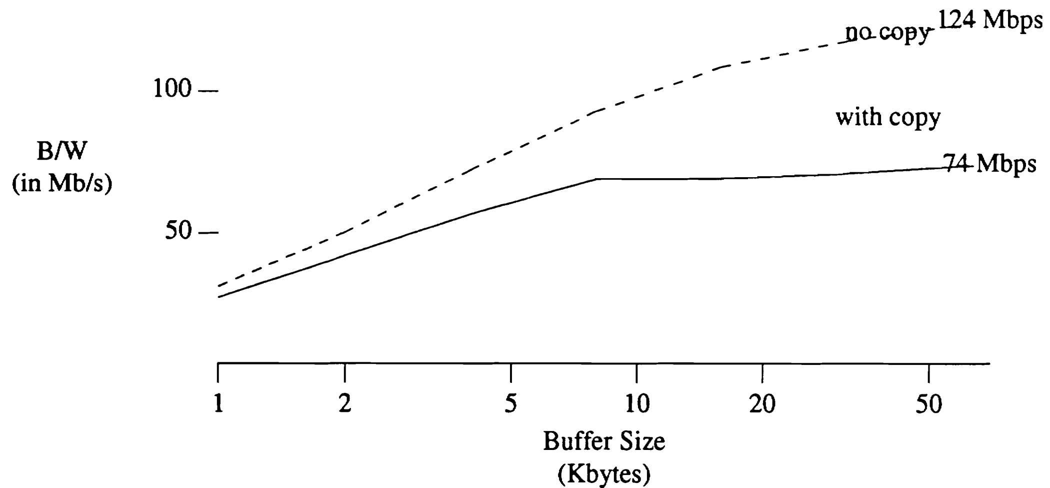


Figure 5: Performance of `test_wr`, streaming from user buffers

After the detailed performance analysis of the hardware showing the IOCC bottleneck (discussed below), we modified the user-buffer driver so that we could measure driver overhead versus other factors such as memory copying and host memory access performance. This was done by deactivating about 5 lines of code in the driver which initiate the streaming transfer, and another 5 which poll the host interface status register for completion of the transfer. These results thus correspond to the case of an "infinitely fast" host interface card connected to a current generation RS/6000 through an infinitely fast MCA bus.

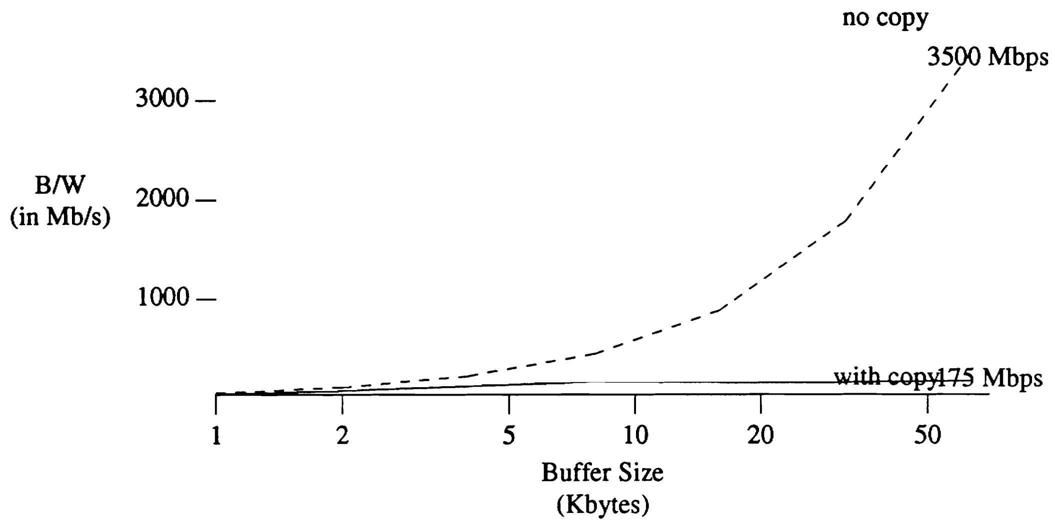


Figure 6: Performance of test_wr, infinitely fast interface & bus

4.3.3. Discussion of Results

The script was run on a lightly-loaded IBM RS/6000 Model 320. Benchmarking done by another process showed little or no system performance degradation, even when competing for I/O resources (e.g., a several megabyte FTP copying data from a remote IBM PC/RT connected through an Ethernet).

It's clear from each of the three graphs that for small block sizes, software is the limiting factor to system performance. Smaller block sizes force the AIX system to context-switch frequently. Larger block sizes reduce software overhead and the hardware performance limits discussed in Section 4.2 become the limiting factor. This can be seen by examining 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.

For many sources of traffic, the 64KB blocks, and hence the performance figures, may be unrealistic. We are studying device driver strategies which can give us good performance with smaller block sizes, perhaps by optimizing the device driver strategy for stream-startup.

5. Conclusions and a Look to the Future

The hardware and software we designed and implemented performed remarkably well, and simulations on the cell manipulation logic on the board show that it can operate well into the range of 700 Mbps or beyond. Our approach of pursuing architectural solutions, such as concurrent operation (as in the parallelism in the header processing pipeline), allows us to take advantage of improvements in technology which would allow higher clock speeds. The software experiments positing an "infinitely fast" device show that the software design scales well to higher-performance platforms. We were somewhat frustrated in our performance goals by the implementation of the Micro Channel Architecture on the IBM RS/6000 Model 320. While the clock rates of the current MCA could support higher speeds (up to 320Mbps, multiplying data width by the clock rate), the I/O Channel Controller limits performance to about 140 Mbps. We were surprised to discover this bottleneck, as we expected software or the MCA bus itself to be the limiting factor. It is hard to blame the designers, as networking at this speed was probably not a consideration in bringing the machine to fruition.

Our research plans for the immediate future (for this interface) are threefold. First, we will interconnect it to the ATM host interface designed by Davie [7] of Bellcore, and to the Sunshine switch [8] in the context of the AURORA collaboration. This will help to iron out any misinterpretations of standards or unwarranted assumptions. Second, our colleagues at IBM Research have implemented an ORBIT [3] card for the RS/6000's MCA; our use of the RS/6000 suggests that internetworking PTM and ATM using the RS/6000 as a bridge would be a very interesting engineering experiment. Since both cards are Micro Channel cards, the IOCC need not be involved in data

transfer, and a 64-bit wide streaming data mode can be used. Third, and finally, we hope to replicate a small number of cards so that our collaborators can deploy them at their sites concurrently with the facilities deployment during Summer 1992. This will allow testing of the interface as a component in the AURORA high-speed WAN and give us an opportunity to perform protocol processing experiments such as implementations of congestion control strategies.

The longer-term research questions raised by these experiments are centered around workstation architectures. The RS/6000, unlike many current-generation workstations, has adequate memory bandwidth to support high-speed networking. The fact that it is not accessible through the Micro Channel suggests that perhaps direct-to-memory operations are necessary, with a host interface connected directly to the system memory bus. Our collaborator David Tennenhouse of MIT goes further, in suggesting that the network interface connect directly to the processing unit, in the style of a coprocessor! But I/O channel architectures such as the Micro Channel provide a number of attractions, among which are structuring, concurrency control, and features such as virtual address translation with the IOCC. In addition, connection to a bus which is less closely coupled to the CPU can aid portability. For example, we debugged the host interface on an IBM PS/2 Model 50.

It is unclear how the networking community will resolve its ferocious need for bandwidth, but there seems little question that workstation vendors must provide higher performance access to computational resources and to memory. This performance must be available to attached devices and networks, whether through I/O channels or novel attachment schemes.

6. Notes and Acknowledgments

We would like to thank Bruce Davie for his detailed and constructive criticisms of this work and its presentation in this paper. Dave Farber planted the seed which started the research, by suggesting the implementation of an ATM to Ethernet bridge. Steve Heimlich helped us in the initial phases of the device driver implementation and Fred Strietelmeier helped us understand the IOCC.

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 and DARPA provide funds to the University participants in AURORA. Bellcore is providing support to MIT and Penn through the DAWN project. IBM has supported this effort by providing RS/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.

RS/6000, AIX, PC/RT, PS/2 and Micro Channel are trademarks of IBM. Ethernet is a trademark of Xerox. UNIX is a trademark of UNIX Systems Laboratories.

7. References

[1] H. B. Bakoglu, G. F. Grohoski, and R. K. Montoye, "The IBM RISC System/6000 processor: Hardware overview," *IBM Journal of Research and Development* **34**(1), pp. 12-22 (January, 1990).

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

[3] David D. Clark, Bruce S. Davie, David J. Farber, Inder S. Gopal, Bharath K. Kadaba, W. David Sincoskie, Jonathan M. Smith, and David L. Tennenhouse, "The AURORA Gigabit Testbed," *Computer Networks and ISDN Systems*, (to appear) (1991).

[4] 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.

[5] IBM Corporation, *IBM RISC System/6000 POWERstation and POWERserver: Hardware Technical Reference, Micro Channel Architecture*, IBM Order Number SA23-2647-00, 1990.

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

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

[8] J. Giacopelli, J. Hickey, W. Marcus, W. D. Sincoskie, and M. Littlewood, "Sunshine: A High-Performance Self-Routing Broadband Packet Switch Architecture," *IEEE Journal on Selected Areas in Communications* **9**(8), pp. 1289-1298 (October, 1991).

[9] CCITT Recommendation I.363, *B-ISDN ATM Adaptation Layer (AAL) Specification*, 1990.

[10] Thomas J. Robe and Kenneth A. Walsh, "A SONET STS-3c User-Network Interface IC," in *Proceedings, Custom Integrated Circuits Conference*, San Diego, CA (May, 1991).

[11] Computer Staff, "Gigabit Network Testbeds," *IEEE Computer* **23**(9), pp. 77-80 (September, 1990).

[12] C. Brendan S. Traw and Jonathan M. Smith, "A High-Performance Host Interface for ATM Networks," in *Proceedings, SIGCOMM 1991*, Zurich, SWITZERLAND (September 4-6, 1991), pp. 317-325.

8. Appendix: Experimental Apparatus

```
/*
 *  testwr.c - main block (no declarations or set-up shown)
 */

if ((fd = open("/dev/host0", O_WRONLY)) == -1){
   perror("Couldn't open dd");
   exit(-1);
}

gettimeofday( &tv1, &tz );

for(i=0; i<repeats; i++){
/*
 * copies are added here, with memcpy( buf, SOMETHING );
 */

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

gettimeofday( &tv2, &tz );

clock = elapsed( tv2, tv1 );

printf( "elapsed time: %d microseconds\n", clock );
```