Experiences with a High-Speed Network Adaptor: A Software Perspective[1]

Peter Druschel and Bruce S. Davie and Larry L. Peterson

TR 94-5

Abstract

This paper describes our experiences, from a software perspective, with the OSIRIS network adaptor. It first identifies the problems we encountered while programming OSIRIS and optimizing network performance, and outlines how we either addressed them in the software, or had to modify the hardware. It then describes the opportunities provided by OSIRIS that we were able to exploit in the host operating system (OS); opportunities that suggested techniques for making the OS more effective in delivering network data to application programs. The most novel of these techniques, called *application device channels*, gives application programs running in user space direct access to the adaptor. The paper concludes with the lessons drawn from this work, which we believe will benefit the designers of future network adaptors.

Aug 1994

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

[1] Also appears in the Proceedings of the 1994 ACM SIGCOMM Conference on Communications Architectures, Protocols and Applications

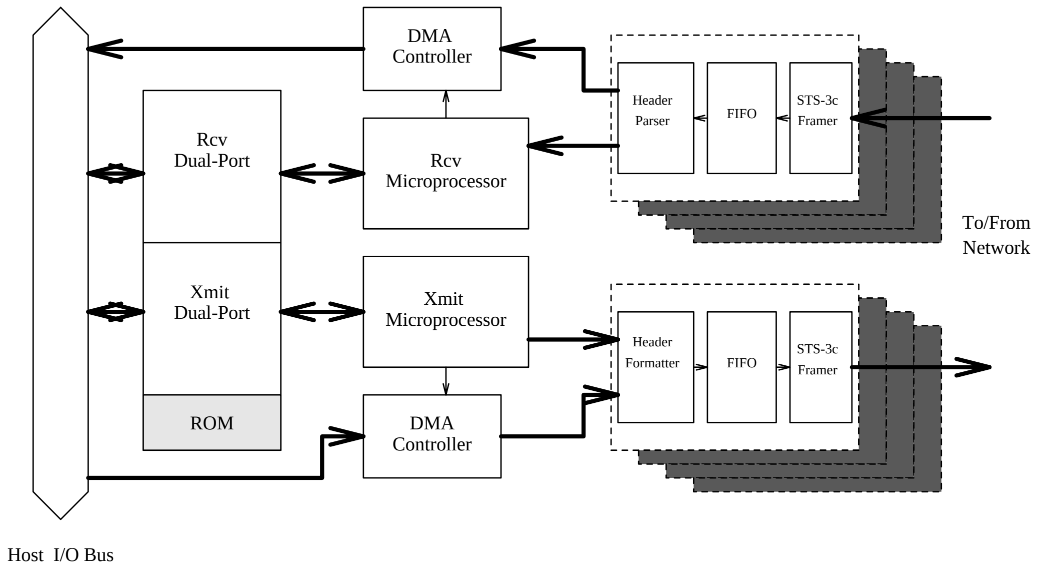

Figure 1: Architectural Overview of OSIRIS

1 Introduction

With the emergence of high-speed network facilities, several research efforts are focusing on the design and implementation of network adaptors [5, 2, 3, 16, 20]. This paper takes the next step in the evolution of adaptors for high-speed networks by reporting our experiences with one particular adaptor—the OSIRIS ATM board built for the AURORA Gigabit Testbed [4, 8]. We consider the network adaptor from a software perspective, identifying the subtle interactions between the adaptor and the operating system software that drives it. Others have looked at this hardware/software interaction as well [18, 6, 16]. In our case, the flexibility built into the OSIRIS board makes this interaction an especially interesting one to study.

The OSIRIS network adaptor was designed specifically to support software experimentation. Therefore, only the most critical, high-speed functions are implemented in hardware, and even these are primarily implemented in programmable logic. The architecture of the interface is depicted in Figure 1. It consists of two mostly independent halves—send and receive—each controlled by an Intel 80960 microprocessor.

The adaptor board attaches to a TURBOchannel option slot provided by DEC workstations. From the host's perspective, the adaptor looks like a 128KB region of memory. A combination of host software and code running in the on-board microprocessors determine the detailed structure of this memory. In general, the memory is used to pass buffer descriptors between the host and the adaptor. Network data is not buffered in the dual-port memory; it is tranferred directly from/to main memory buffers using DMA.

In the transmit direction, the software running on the microprocessor writes commands to a DMA controller and an ATM cell generator. The general paradigm is that the host passes buffer descriptors to the microprocessor through the dual-port RAM, and the microprocessor executes a segmentation algorithm to determine the order in which cells are sent. For example, the host could queue a number of packets and the microprocessor could transmit one cell from each in turn. The microprocessor has the capability to interrupt the host.

In the receive direction, the microprocessor reads from a FIFO the VCI and AAL information that is stripped from cells as they are received. By examining this information, and using other information from the host (such as a list of reassembly buffers), the microprocessor determines the appropriate host memory address at which the payload of each received cell is to be stored. It then issues commands to another DMA controller; typically one command is issued for each ATM cell received. As part of the reassembly algorithm, the microprocessor decides when it is necessary to interrupt the host.

The important point to understand from this brief description is that software running on the two 80960s controls the send/receive functionality of the adaptor, and perhaps just as importantly, this code effectively defines the *software interface* between the host and the adaptor. The other relevant piece of software, of course, is the OS running on the host. In our case, it is the Mach 3.0 operating system [1], retrofitted with a network subsystem based on the *x*-kernel [9, 12]. For the purpose of this paper, there are two relevant things to note about the OS. First, because the *x*-kernel supports arbitrary protocols, our approach is protocol-independent; it is not tailored to TCP/IP. Second, because Mach is a microkernel-based system and the *x*-kernel allows the protocol graph to span multiple protection domains, our approach has to allow for the possibility that network data traverses multiple protection domains; it is not restricted to kernel-resident protocols.

This paper makes two contributions. First, it reports our experiences programming the OSIRIS adaptor to achieve good host-to-host performance. It includes an overview of the problems we encountered, and how we either addressed them in the software, or had to modify the hardware. This discussion is given in Section 2. Second, it describes the opportunities provided by the board that we were able to exploit in the OS; opportunities that facilitated new techniques for making the OS more effective in delivering network data to application programs. The most novel of these techniques, called *application device channels*, gives application programs running in user space direct access to the adaptor. This discussion is given in Section 3. Section 4 then presents the results of a performance study.

Throughout both Sections 2 and 3 we highlight those issues (lessons) that are specific to an ATM adaptor, and those that we believe to be applicable to the design of high-speed network adaptors in general. These lessons are summarized in Section 5.

2 Basic Functionality

This section describes our experiences programming the OSIRIS board, highlighting the problems it imposed on the software, and how we addressed them. For the most part, this discussion is limited to how we implemented the basic host-to-host functionality, both correctly, and with the highest possible performance; the next section describes how we exploited certain features of the board to implement novel OS techniques that turn this host-to-host performance into equally good user-to-user performance.

2.1 Host/Board Communication

We begin by defining the software interface between the host's device driver and the processors on the OSIRIS board. The host CPU communicates with the on-board processors through shared data structures in the dual-port memory. In addition, each on-board processor can issue an interrupt to alert the host CPU of asynchronous events. The design of the shared data structures and the discipline for using interrupts was guided by the goals of minimizing packet delivery latency and host CPU load, and of achieving host-to-host throughput close to the capacity of the network link. Particular attention was paid to (1) minimizing the number of load and store operations required to communicate with the on-board processors (accesses to the dual port-memory across the TURBOchannel are expensive), (2) avoiding delays due to lock contention while accessing shared data structures in the dual-port memory, and (3) minimizing the number of interrupts, which place a significant load on the host CPU.

2.1.1 Shared Data Structure

As with any shared data structure, measures must be taken to ensure consistency in the presence of concurrent accesses. The dual-port memory itself guarantees atomicity of individual 32bit load and store operations only. Each half of the board provides a test-and-set register that can be used to implement a simple spin-lock. The intended use is to enforce mutually exclusive access to the dual-port memory by mandating that a processor must first acquire the corresponding lock. This approach allows arbitrarily complex shared data structures, but it restricts concurrency between host CPU and on-board processors. As a result, both packet delivery latency and CPU load can suffer due to lock contention.

To avoid this problem, we use simple lock-free data structures that rely only on the atomicity of load and store instructions. The basic data structure used in the dual-port memory is a simple, one-reader-one-writer FIFO queue used to pass buffers between the host and the adaptor. The queue consists of an array of buffer descriptors, a head pointer, and a tail pointer. The head pointer is only modified by the writer and the tail pointer is only modified by the reader. The processors determine the status of the queue by comparing the head and tail pointers, as follows:

$$head = tail \Leftrightarrow queue\ is\ empty$$

$$(head + 1) \bmod size = tail \Leftrightarrow queue\ is\ full$$

The simplicity of these lock-free queues maximizes concurrent access to the dual-port memory, and minimizes the number of load and store operations required to communicate.

A single queue is used for communication between the host CPU and the transmit processor. Each queue element describes a single buffer in main memory by its physical address and length. To queue a buffer for transmission, the host CPU performs the following actions (xmitQueue[head] refers to the buffer descriptor referred to by the head pointer).

- wait until the transmit queue is not full
- de-allocate any previous buffer described by xmitQueue[head]
- queue the buffer using xmitQueue[head]
- increment the head pointer (modulo array size)

The transmit processor continuously performs the following actions.

- wait until the transmit queue is not empty
- read the descriptor at xmitQueue[tail]
- transmit the buffer
- increment the tail pointer (modulo array size)

Two queues are required for communication between the host and the receive processor. The first queue is used to supply buffers to the receive processor for storage of incoming PDUs;[1] the second queue holds filled buffers waiting for processing by the host. Initially, the host fills the free buffer queue. When a PDU arrives, the receive processor removes a buffer from this queue, and stores incoming data into the buffer. When the buffer is filled, or the end of the incoming PDU is encountered, the processor adds the buffer to the receive queue. If the receive queue was previously empty, an interrupt is asserted to notify the host of the transition of the receive queue from the empty state to a non-empty state. The host's interrupt handler schedules a thread that repeatedly performs the following steps until the receive queue is found empty:

- remove a buffer from the receive queue
- add a free buffer to the free queue
- initiate processing of the received data

2.1.2 Interrupts

Handling a host interrupt asserted by the OSIRIS board takes approximately 75μs in Mach on a DecStation 5000/200. For comparison, the service time for a received UDP/IP PDU is 200μs; this number includes protocol processing and driver overhead, but not interrupt handling. Given this high cost, minimizing the number of host interrupts during network communication is important to overall system performance.

In our scheme, the completion of a PDU transmission, which is traditionally signalled to the host using an interrupt, is instead indicated by the advance of the transmit queue's tail pointer. The driver checks for this condition as part of other driver activity—for example, while queuing another PDU—and takes the appropriate action. Interrupts are used only in the relatively infrequent event of a full transmit queue. In this case, the host suspends its transmit activity, and the transmit processor asserts an interrupt as soon as the queue reaches the half empty state.

In the receiving direction, an interrupt is only asserted once for a *burst* of incoming PDUs. More specifically, whenever a buffer is queued before the host has dequeued the previous buffer, no interrupt is asserted. This approach achieves both low packet delivery latency for individually arriving packets, and high throughput for incoming packet trains. Note that in situations where high throughput is required (i.e. when packets arrive closely spaced), the number of interrupts is much lower than the traditional one-per-PDU.

2.2 Physical Buffer Fragmentation

The OSIRIS board relies on direct memory access (DMA) for the actual transfer of network data between main memory and network adaptor. The unit of data exchanged between host driver software and on-board processors is a physical buffer—a set of memory locations with contiguous physical addresses. The descriptors used in the transmit and receive queues contain the physical address and the length of a buffer. The on-board processors initiate DMA transfers based on the physical address of the buffers.

The per-PDU processing cost in the host driver increases with the number of physical buffers used to hold the PDU. Thus, one would like to minimize the number of physical buffers occupied by a single PDU. However, this is made difficult by the fact that the contiguous virtual memory pages used to store a PDU are

[1] For the purpose of this paper, we use the term *protocol data unit* (PDU) to denote a packet processed by a protocol, where the protocol in question is generally given by the context. In this case, the PDU corresponds to the unit of data sent between device drivers.

generally not contiguous in the physical address space. The reason for this lies at the heart of any page-based virtual memory system—the ability to map non-contiguous physical pages to contiguous virtual memory addresses, in order to avoid main memory fragmentation.

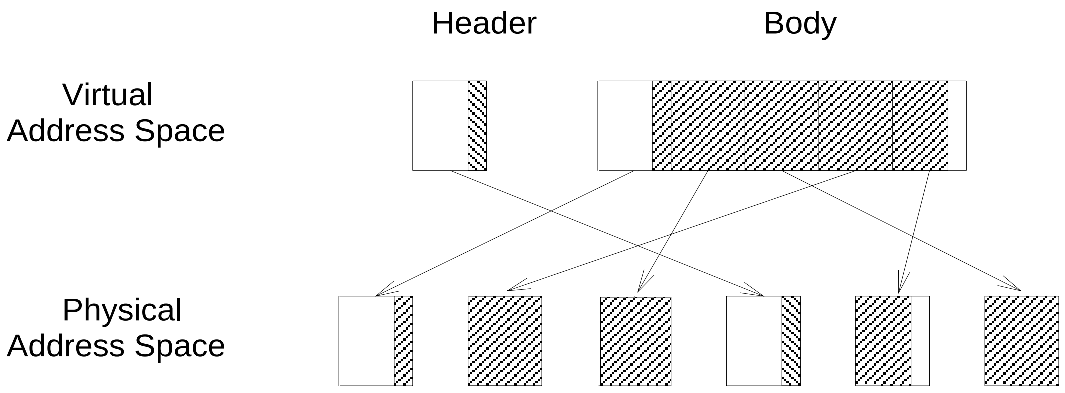

Figure 2: PDU Buffer Fragmentation

Figure 2 depicts a PDU passed to the OSIRIS driver for transmission. The PDU consists of two parts—a header portion, which contains protocol headers, and the data portion. The header portion usually contributes one physical buffer. The data portion is typically not aligned with page boundaries, and may thus occupy $\lceil (message\ data\ size \Leftrightarrow 1)/page\ size \rceil + 1$ pages. When the physical pages occupied by the data portion are not contiguous, each page contributes a physical buffer. In practice, a PDU with a data portion of length n pages usually occupies $n + 2$ physical buffers.

Message fragmentation at the protocol level can aggravate this proliferation of physical buffers. The problem is that unless the fragment boundaries in the original message coincide with page boundaries, each fragment may generate excess physical buffers in the driver. As an example of an extreme case, assume that a contiguous 16KB application message is transmitted using UDP/IP with a maximal transfer unit (MTU) of 4KB,[2] which is also the system's page size. The inclusion of the IP header reduces the data space available in each fragment to slightly less than 4KB. Consequently, the data portions of most fragments are not page-aligned, and occupy two physical pages. In addition, the IP header attached to each fragment occupies a separate page. As a result, the transmission of a single, 16KB application message can result in the processing of up to 14 physical buffers in the driver. This compounding effect of IP fragmentation and buffer fragmentation can be avoided by ensuring page alignment of application messages, and by choosing an MTU size that is a multiple of the page size, plus the IP header size. This ensures that fragment boundaries align with page boundaries.

A similar problem exists on the receive side. Recall that the host driver allocates receive buffers, and queues these buffers for use by the receive processor. Most operating systems do not support the dynamic allocation of physically contiguous pages. In this case, the size of the receive buffers is restricted to the system's memory page size, since it represents the largest unit of physically contiguous memory that the driver can allocate. This limit on the size of receive buffers causes the fragmentation of all incoming network packets larger than the page size.

The proliferation of physical buffers is a potential source of performance loss in the OSIRIS driver. A general solution to this problem would require the use of physically contiguous memory for the storage of

[2]Keep in mind that the OSIRIS driver, not the hardware, defines the MTU. We are just using 4KB as an example.

network data. In traditional operating systems, where network data is copied between application memory and kernel buffers, this can be achieved by statically allocating contiguous physical pages to the fixed set of kernel buffers. Unfortunately, this approach does not readily generalize to a copy-free data path [9], since applications generally cannot be allowed to hold buffers from a statically allocated pool. We are currently experimenting with OS support for dynamic allocation of contiguous physical pages on a best-effort basis.

Several modern workstation, such as the IBM RISC System/6000 and DEC 3000 AXP Systems provide support for virtual address DMA through the use of a hardware virtual-to-physical translation buffer (scatter/gather map). Host driver software must set up the map to contain appropriate mappings for all the fragments of a buffer before a DMA transfer. When data is transferred directly from and to application buffers, it may be necessary to update the map for each individual message. As a result, physical buffer fragmentation is a potential performance concern even when virtual DMA is available.

2.3 Cache Coherence

The cache subsystem of the host we were originally using—the DECstation 5000/200—does not guarantee a coherent view of memory contents after a DMA transfer into main memory. That is, CPU reads from cached main memory locations that were overwritten by a DMA transfer may return stale data. To avoid this problem, the operating system normally executes explicit instructions to invalidate any cached contents of memory locations that were just overwritten by a DMA transfer. Unfortunately, partial invalidations of the data cache take approximately one CPU cycle per memory word (32bits), plus the cost of subsequent cache misses caused by the invalidation of unrelated cached data.[3] This cost has a significant impact on the attainable host-to-host throughput, as quantified in Section 4 (Figure 3).

The key idea for avoiding this cost is to take a lazy approach to cache invalidation, and to rely on network transmission error handling mechanisms for detecting errors caused by stale cache data. When a data error is detected at some stage during the processing of a received message, the corresponding cache locations are invalidated, and the message is re-evaluated before it is considered in error. The feasibility of this approach depends on the following conditions.

1. The underlying network is not reliable, and therefore mechanisms for detecting or tolerating transmission errors are already in place.
2. The rate of errors introduced by stale cache data is low enough for the lazy approach to be effective.
3. Revealing stale data does not pose a security problem.

While the first condition is true for most networks, the second condition deserves some careful consideration. The OSIRIS driver employs a free buffer queue and a receive queue with a length of 64 buffers each, and a buffer size of 16KB. This implies that once a receive buffer is allocated and queued on the free buffer queue, normally 63 other buffers are processed by the host until that buffers re-appears at the top of the received buffer queue. In order to become stale, a cached data word from a particular buffer has to remain in the cache while 63 other receive buffers are being processed. During this time, the CPU typically reads the portion of the input buffers occupied by received data, as well as other data relating to protocol processing, application processing and other activities unrelated to the reception of data. These accesses are likely to evict all previously cached data from the DECstation's 64KB data cache.

[3] The DECstation also supports a fast instruction that swaps the data and instruction cache, which amounts to an invalidation of the entire cache contents. However, the high cost of subsequent cache misses makes this not an attractive solution.

Experimentally, we have seen no evidence of stale data at all while running our test applications. This suggests that the error rate should be low enough for this optimization to be very effective. It should be noted that lazy cache invalidation is not likely to scale to machines with much larger caches. Fortunately, hardware designers have recognized the high cost of software cache invalidation, and tend to provide support for cache coherence on these machines. For example, the DEC 3000 AXP workstation data cache is updated during DMA transfers into main memory.

The third condition is satisfied whenever reliable protocols are used that detect data errors before the data is passed to an unprivileged application. However, with unreliable protocols, an application could access stale data from a previous use of the receive buffer, potentially violating the operating system's security policy. This problem can be solved by ensuring the reuse of receive buffers on the same data stream. In this way, stale data read by an application is guaranteed to originate from an earlier message received by that application, thus eliminating security problems. The reuse of receive buffers on the same data stream has other advantages, as described in Section 3.1.

2.4 Page Wiring

Whenever the address of a buffer is passed to the OSIRIS on-board processors for use in DMA transfers, the corresponding pages must be *wired*. Wiring, also referred to as *pinning*, refers to the marking of a page as being non-eligible for replacement by the operating system's paging daemon. Since changing the wiring status of a page occurs in the driver's critical path, the performance of this operation is of concern.

Our initial use of the Mach kernel's standard service for page wiring resulted in surprisingly high overhead. One problem is that Mach's implementation of page wiring provides stronger guarantees than are actually needed for DMA transfers. In particular, it prevents not only replacement of the page itself, but also of any pages containing page table entries that might be needed during an address translation for that page. We now use low-level functionality provided by the Mach kernel to prevent replacement of pages with acceptable performance.

2.5 DMA Length

The length of DMA transactions has a significant effect on performance. As mentioned above, DMA usually takes place one ATM cell at a time. This provides maximum flexibility in the transmit direction (e.g. to interleave several outgoing PDUs) and avoids the need for a reassembly buffer in the receive direction. However, the initial decision to fix all DMA transactions at exactly one cell payload (44 bytes, because of AAL overhead) had some undesirable performance impact, for two reasons: (1) the DMA overhead for the TURBOchannel is high enough to make transfers as short as 44 bytes rather inefficient, and (2) fixed-length DMA, especially when the length is not a power of two, causes a range of problems at the edges of buffers. We now discuss each of these problems, in turn.

2.5.1 DMA Overhead

As reported previously [8], the maximum data transfer speed that can be sustained with 44 byte transfers over the TURBOchannel on a DECstation 5000/200 is 367 Mbps in the transmit direction and 463 Mbps in the receive direction. These figures, which have been measured for brief periods on the actual hardware, can be derived simply by considering the minimal overhead for DMA transactions in each direction—8 cycles for writes, 13 cycles for reads. Thus, for example, the maximum throughput for transmission is $11/(11+13)\times 800 = 367$ Mbps.

Clearly, it would be advantageous to increase the length of DMA transfers. In the transmit direction, the only penalty for increasing DMA length is an increase in the granularity of multiplexing. We argued previously that fine-grained multiplexing is advantageous for latency and switch performance reasons [7]. However, when the adaptor is used in a mode where the goal is to maximize throughput to a single application, neither of these reasons is relevant. It is therefore reasonable, and straightforward, to modify the DMA controller so that it can perform DMA transactions longer than one ATM cell. Note that if we allowed transfers of 88 bytes at a time, the maximum rate that data could be moved across the bus would be $22/(22+13)\times800 = 503$ Mbps. This is close to the 516 Mbps data bandwidth available in a 622 Mbps SONET/ATM link when 44 byte cell payloads are used.

In the receive direction, the primary advantage in doing single-cell DMAs is that it removes the need for a reassembly buffer on the adaptor; cells can be placed directly in host memory as they arrive. Not only does this reduce the hardware complexity of the interface, but it also reduces the likelihood that inadequate reassembly space is available.

In some circumstances, however, it is possible to preserve the advantages of not having a reassembly buffer on the adaptor while performing DMAs longer than one cell. The quantity that we really wish to optimize is the user-to-user throughput for a single application. In this case, as long as cells arrive in order, most successively received cells will contain data that is to be stored in contiguous regions of host memory, the only exception being at the end of a buffer. Since there is a small amount of FIFO buffering of cells on the adaptor, the microprocessor can look at two cell headers before deciding what to do with their associated payloads. If the header information suggests that the two payloads should be stored contiguously, then a single, 88-byte DMA can be initiated.

We have implemented this change to the DMA controller logic; the maximum throughput of the hardware is now $22/(22+8)\times800 = 587$ Mbps—more than the payload of an OC-12 channel. Note that the biggest gain is achieved just by going to double-cell DMAs, since we have already driven the overhead down from 42% to 26%. With any further increase in DMA length the returns diminish. The measured performance of doing 88-byte DMAs is reported in Section 4.

2.5.2 DMA Length Variation

So far we have considered DMA transactions that are multiples of the ATM cell payload. The decision to restrict DMA length was made to simplify the DMA controller design, since the logic for this component is by far the most complex part of OSIRIS. It initially seemed reasonable to assume that data could be passed between the host and the adaptor in contiguous buffers of arbitrary size, and that only the last cell of a buffer would need to be partially filled. However, there are several drawbacks to this approach.

The crux of the problem is that, for reasons of efficiency, the host should not simply pass contiguous buffers to the adaptor, but it should pass complete PDUs. Since PDUs are generally composed of a number of discontiguous buffers, and the size of the buffers is rarely a multiple of the ATM payload size, it becomes necessary to send partially filled cells in the middle of PDUs. Not only is this inelegant, but it also makes interoperating with other systems impossible and adds sufficient complexity to the microprocessor reassembly code that it becomes difficult to meet the tight instruction budget. The consequences for the reassembly code complexity are even worse when partially filled cells are received out-of-order, as discussed in Section 2.6.

Another problem arises when PDU sizes are multiples of the page size, as is the case for network file system (NFS) traffic. In the transmit direction, the last cell of a page contains a few bytes of the next physical page. This is almost certainly data that does not belong to the sending application, so this may be considered a security risk. In the receive direction, the only legitimate option is to stop filling the page when the next cell would cause the page-boundary to be crossed, and start on a new buffer. However, this is likely to break many higher-layer services that expect to see full pages (e.g. NFS).

The ideal solution would be to implement a DMA controller that could handle arbitrary length DMA transactions. The main drawback to this approach in our case was the hardware complexity, which may have exhausted the resources in the available programmable logic. The problem could also be dealt with by some amount of copying by the host, but this would adversely affect performance. Fortunately, a solution that avoids copying but which is simpler to implement than arbitrary-length DMA was found to be acceptable. It turns out that, in the x-kernel/Mach environment, it is straightforward to arrange for all the buffers of a PDU (except the last) to be aligned in such a way that they end at page boundaries.

Thus, the DMA controller does not need to perform arbitrary length DMA, as long as it can avoid doing DMA across page boundaries. We implemented the following modification to the DMA logic: if the address handed to the DMA controller by the microprocessor is within 44 bytes of a page boundary, the DMA will stop when it reaches the boundary. The DMA controller then waits for another address from the microprocessor, which it uses to DMA enough bytes to fill the remainder of the ATM cell. Typically, this second address will be the start address of the next buffer in the PDU.

It is noteworthy that the cause of the problem here was a mismatch of abstractions between hardware and software. The hardware designer's abstraction was that the host would pass contiguous buffers to the adaptor. For satisfactory software performance, however, a better abstraction was to pass a PDU consisting of a chain of discontiguous buffers.

The clear lesson here is the importance of being able to design adaptor hardware in concert with the host software that will drive it. The original scheme of single-length DMA might have been workable if all the host software were designed to fit that model. However, it is clearly unreasonable to design an entire operating system to fit in with the quirks of a network adaptor. The combination of programmable logic and software control in the OSIRIS adaptor enabled it to be modified to suit the requirements of the host software.

2.6 Cell Misordering

One of the features of the OSIRIS interface is that it uses *striping* to achieve a network speed of 622 Mbps. By this we mean that four 155 Mbps channels are grouped together and treated as a single logical channel, with data striped at the cell level. Striping is a well-established technique that enables an end user to achieve a network bandwidth in excess of that which can readily be supported in the network itself.

The principal drawback of striping in an ATM network is that it has the potential to introduce misordering, which is explicitly prohibited in the ATM standard.[4] There are three main causes of misordering: (1) different delays experienced by each physical link because of different physical path lengths; (2) different delays introduced into the physical links by multiplexing equipment in the network; and (3) different queuing delays experienced by cells on different links as they pass through distinct ports on the switches in the network.

The first cause can be eliminated by multiplexing all physical links onto a single fiber, as is done in AURORA. It is also possible to eliminate the third cause by adding some complexity to the switch; the switch must coordinate the different ports to keep all queue lengths equal. However, adding this complexity has the undesirable effect of negating the advantage of striping—to provide higher bandwidth to those (presumably few) users who need it, without forcing an upgrade to the network. The second cause came as a surprise, and it was not within our power to eliminate it. For these reasons, we decided to live with the misordering.

[4] Note that the standards do not address striping.

The misordering introduced by these factors is not arbitrary; cells transmitted on a given physical link will arrive in order relative to each other, but may be delayed relative to cells sent on other links. In our case, with four links, the first and fifth cells of a PDU will travel on the same physical link, and the fifth will always arrive after the first. However, the second, third and fourth cells may arrive ahead of the first. We refer to this limited class of misordering as skew, and we identified two strategies for coping with it.

The first strategy involves putting a sequence number in the AAL header of each cell. Since the OSIRIS design allows each cell to be individually placed at a specific location in host memory, the only change is to the 80960 code to handle out-of-order arrivals. The sequence number is used to determine the host memory address at which each cell is to be stored. This approach has several drawbacks, however. First, if skew is introduced by different queuing delays in the switches, it is essentially unbounded and thus we can never guarantee that the sequence number space is large enough. Second, the possibility that the first cell of a PDU will not be the first one received adds significant complexity to the reassembly code.

The second approach takes advantage of the fact that this is not arbitrary misordering. Since cells on a given physical link arrive in order, we can view the reassembly of a PDU as four concurrent reassemblies, where the four "packets" happen to be interleaved with each other in memory. In this case, we can use AAL5-style reassembly on each of the four packets and when all four packets are complete, as indicated by the framing bit in the AAL5 header, we can declare the reassembly of the PDU to be complete. There is a small problem if a PDU is less than 4 cells long, since we would not receive four framing bits in this case. We could deal with this using one additional framing bit in the ATM header to indicate the very last cell of a PDU. The only real drawback of this approach, aside from its impact on standards, is that it was difficult to implement in the small instruction budget available in the 80960. Since the goal of the design was to permit experimentation with algorithms that would ultimately be implemented in hardware, we feel this is not a fatal flaw.

Whatever means are used to deal with skew, it does have a serious disadvantage. As discussed in Section 2.5.1, the performance of the interface can be significantly enhanced by combining successively received cell payloads on the board and transferring the combined data to the host as a single, longer DMA. Once skew is introduced, the probability that two successive cells will be received in order is greatly reduced.

2.7 DMA versus PIO

One of the most lively debates in network adaptor design is over the relative merits of DMA and programmed I/O (PIO) for data movement between the host and the adaptor. Both the literature on the subject (e.g. [16, 2, 6]) and our own experience have led us to the conclusion that the preferable technique is highly machine-dependent. In the case of the DEC workstations we used, the low throughput achievable using PIO across the TURBOchannel ensures that, with well designed software (i.e. no unnecessary copies) DMA is preferable.

We argue that the best way to compare DMA performance versus PIO is to determine how fast an application program can access the data in each case. For example, when data is DMAed into memory on a DECstation 5000/200, it will not be in the cache; an additional read of the main memory is necessary when the application accesses the data. On the DECstation, reading data into the cache causes a dramatic decrease in throughput from the pure DMA results, but the throughput remains above that which can be achieved by PIO simply because of the high performance penalty for word-sized reads across the TURBOchannel. On DEC's Alpha-based machines, a greatly improved memory system with a crossbar switch that connects TURBOchannel, main memory and cache allows cache/memory transactions to occur concurrently with DMA transfers on the TURBOchannel. In addition, DMA writes to main memory update the second level cache. On these machines, applications are able to access the data at the rate of and concurrent with its DMA transfer into main memory (see Section 4).

In the PIO case, with carefully designed software, data can be read from the adaptor and written directly to the application's buffer in main memory, leaving the data in the cache [13, 6]. If the application reads the data soon after the PIO transfer, the data may still be in the cache. According to one study, the PIO transfer from adaptor to application buffer must be delayed until the application is scheduled for execution, in order to ensure sufficient proximity of data accesses for the data to remain cached under realistic system load conditions [15]. Loading data into the cache too early is not only ineffective, but can actually decrease overall system performance by evicting live data from the cache. Unfortunately, delaying the transfer of data from adaptor to main memory until the receiving application is scheduled for execution requires a substantial amount of buffer space in the adaptor. With DMA, instead of using dedicated memory resources on the adapter, incoming data can be buffered in main memory. Using main memory to buffer network data has the advantage that a single pool of memory resources is dynamically shared among applications, operating system, and network subsystem.

3 New OS Mechanisms

This section introduces two novel OS mechanisms facilitated by the OSIRIS board—*fast buffers* (fbufs) and *application device channels* (ADCs)—that are designed to improve user-to-user throughput and latency, respectively. Whereas the previous section focuses on how we achieved good host-to-host performance, the mechanisms discussed in this section address the problem of delivering equally strong performance to application programs.

3.1 Fast Buffers

One of the key problems faced by the operating system, especially a microkernel-based system in which device drivers, network protocols, and application software might all reside in different protection domains, is how to move data across domain boundaries without sacrificing the bandwidth delivered by the network. The fbuf mechanism is designed to address this problem—it is a high-bandwidth cross-domain buffer transfer and management facility.

The fbuf mechanism itself is simple to understand. It combines two well-known techniques for transferring data across protection domains: page remapping and shared memory. It is equally correct to view fbufs as using shared memory (where page remapping is used to dynamically change the set of pages shared among a set of domains), or using page remapping (where pages that have been mapped into a set of domains are cached for use by future transfers). Since fbufs are described in detail elsewhere [10], this section concentrates on the OSIRIS features that we were able to exploit.

The effectiveness of fbufs depends on the ability of the adaptor to make an early demultiplexing decision. That is, the "data path" through the system that the incoming packet is going to traverse must be determined by the adaptor so that it can be stored in an appropriate buffer; one that is mapped into the right set of domains. We say that an fbuf that is already mapped into a particular set of domains is *cached*. Being able to use a cached fbuf, as opposed to an uncached fbuf that is not mapped into any domains, can mean an order of magnitude difference in how fast the data can be transferred across a domain boundary.

In the case of the OSIRIS adaptor, the device driver employs the following strategy. It maintains queues of preallocated cached fbufs for the 16 most recently used data paths, plus a single queue of preallocated uncached fbufs. The adaptor performs reassembly of incoming packets by storing the ATM cell payloads into a buffer in main memory using DMA. When the adaptor needs a new reassembly buffer, it checks to see if there is a preallocated fbuf for the virtual circuit identifier (VCI) of the incoming packet. If not, it uses a buffer from the queue of uncached fbufs.

One of the interesting aspects of this scheme is how we use VCIs. The x-kernel provides a mechanism for establishing a path through the protocol graph, where a path is given by the sequence of sessions that will process incoming and outgoing messages on behalf of a particular application-level connection. Each path is is then bound to an unused VCI by the device driver. This means that we treat VCIs as a fairly abundant resource; each of the potentially hundreds of paths (connections) on a given host is bound to a VCI for the duration of the path (connection). This approach is not compatible with a regime that treats VCIs as a scarce resource, and in particular, a resource that the network charges for.

Early demultiplexing has advantages beyond that of enabling efficient delivery of data to applications. It is also the basis for the appropriate processing of prioritized network traffic under high receiver load [11]. The threads that de-queue buffers from the various receive queues may be assigned priorities corresponding to the traffic priorities of the network stream they handle. During phases of receiver overload, lower-priority receive queues will become full before higher priority ones, allowing the adaptor board to drop the lower priority packets before they have consumed any processing resources on the host.

3.2 Application Device Channels

Fbufs take advantage of the OSIRIS demultiplexing capability to avoid costs associated with the transfer of data across protection domain boundaries on the end host. These costs would otherwise limit the attainable application-to-application throughput. Application device channels (ADCs) take the on-board demultiplexing approach a significant step further. An ADC gives an application program restricted but direct access to the OSIRIS network adaptor, bypassing the operating system kernel. This approach removes protection domain boundaries from both the control and data path to the network adaptor, resulting in minimal application-to-application message latencies.

ADCs are implemented as follows. The transmit dual-port memory is divided into sixteen 4KB pages, each of which contains a separate transmit queue. The receive dual-port memory is similarly partitioned so that each page contains a distinct free buffer queue and receive queue. One transmit queue, and one pair of free/receive queues are used by the operating system in the usual way. The remaining pages are grouped in pairs of one transmit and one receive page.

When an application opens a network connection, the operating system may decide to map one pair of pages into the application's address space to form an application device channel. Linked with the application is an ADC channel driver, which performs essentially the same functions as the in-kernel OSIRIS device driver. Also linked with the application is a replicated implementation of the network protocol stack. The technology of application-linked network protocols has been demonstrated elsewhere in the literature [19, 14], and is also supported by the x-kernel.

The operating system assigns a set of VCIs, a priority, and a list of physical pages to the ADC. The receive processor queues incoming PDUs on the receive queue of an ADC if the VCI of the PDU is in the set of VCIs assigned to that ADC. The priority is used by the transmit processor to determine the order of transmissions from the various ADCs' transmit queues. The list of physical pages is used to maintain proper memory access protection; it determines which pages the application can legally use as receive and transmit buffers. When an application queues a buffer with an unauthorized address, the on-board processor asserts an interrupt, and the operating system in turn raises an access violation exception in the offending application process. Host interrupts are always fielded by the kernel's interrupt handler. If the interrupt indicates the transition of an ADC's receive queue from the empty to a non-empty state, the interrupt handler directly signals a thread in the ADC channel driver, as described in Section 2.1.2.

At first glance, ADCs may appear similar to the mapped device drivers used in Mach [17] and other microkernel-based systems. In these systems, the user-level UNIX server is granted direct access to, and control of, the network device. However, application device channels are different from mapped device drivers in two important ways. First, the OS kernel remains in control of the device in the case of ADCs; only certain kinds of access are granted to the application domain. Second, the device can be fairly shared among and directly accessed by a number of untrusted applications; the device is not mapped into a single domain, as is the case with mapped device drivers. That is, the device is shared by multiple end-user domains, rather than a single network server domain.

The way in which ADCs allow applications direct access to the network adaptor is analogous to the way an application is allowed direct access to the CPU and to main memory. The operating system restricts the use of certain CPU instructions, and permits access to only a subset of main memory in order to remain in control of the machine's resources. The OS kernel itself is normally only involved in scheduling resources, as well as initialization and finalization of program execution. In many distributed applications, such as multimedia, network I/O is a frequent and common component of program execution. ADCs recognize this and allow the operating system kernel to be bypassed in the common case of network data delivery. The OS need only be involved in connection establishment and termination.

4 Performance

This section reports on several experiments designed to evaluate the network performance achieved with the OSIRIS board, and the impact of various optimizations described in earlier sections. All presented results refer to message exchanges between test programs linked into the kernel. For user-to-user performance using application device channels (ADCs), the measured results were within the error margins of those obtained in the kernel-to-kernel case on an otherwise unloaded system. This is significant, since it implies that there is no penalty for crossing the protection domain boundary between OS kernel and unprivileged user processes. The effectiveness of fbufs, independent of ADCs, is reported elsewhere [10].

Machine	Protocol	Message size (bytes)			
DEC model		1	1024	2048	4096
5000/200	ATM	353	417	486	778
	UDP/IP	598	659	725	1011
3000/600	ATM	154	215	283	449
	UDP/IP	316	376	446	619

Table 1: Round-Trip Latencies (μs)

Throughout this section, we report results obtained on two generations of workstations: the DECStation 5000/200 (25Mhz MIPS R3000), and the DEC 3000/600 (175MHz Alpha). Table 1 shows the round-trip latencies achieved between a pair of workstations connected by a pair of OSIRIS boards linked back-to-back. The rows labeled "ATM" refer to the round-trip latency of PDUs exchanged between test programs configured directly on top of the OSIRIS device driver. In the "UDP/IP" case, round-trip latency was measured between two test programs configured on top of the UDP/IP protocol stack[5]. IP was configured to use an MTU of 16KB, and UDP checksumming was turned off. The measured latency numbers for 1 byte messages are comparable to—and in fact, a bit better than—those obtained when using the machines' Ethernet adaptors under otherwise identical conditions. This is a reassuring result, since it demonstrates that the greater complexity of the OSIRIS adaptor did not degrade the latency of short messages.

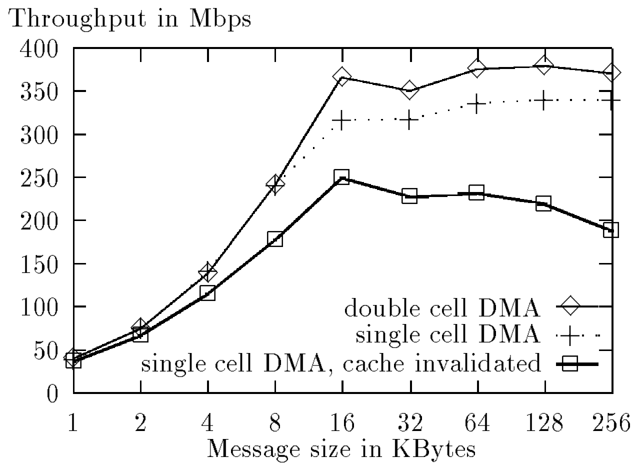

Figure 3: DEC 5000/200 UDP/IP/OSIRIS Receive Side Throughput

The next set of measurements was designed to evaluate the network performance of the receiving host in isolation. For this purpose, the receiver processor of the OSIRIS board was programmed to generate fictitious PDUs as fast as the receiving host could absorb them. Figure 3 shows the measured data throughput achieved on a DEC 5000/200 with the UDP/IP protocol stack, where the IP MTU was set to 16 KB. The graphs depict results measured with DMA transfer sizes of one and two cell payloads, and with cache invalidation in the OSIRIS driver.

We make the following observations. First, the maximal throughput achieved is 379 Mbps with double cell DMA, 340 Mbps with single cell DMA, and 250 Mbps with single cell DMA when the data cache is pessimistically invalidated after each DMA transfer. The last number shows the significant impact of cache invalidations on throughput.

In the DECStation 5000/200, all memory transactions occupy the TURBOchannel and no part of a DMA transaction can overlap with the CPU accessing main memory. Thus, memory writes and cache fills that result from CPU activity reduce DMA performance. Conversely, DMA traffic increases the average memory access latency experienced by the CPU. The combined effect of DMA overhead and main memory contention result in a maximum throughput rate of 340 Mbps in the receive direction. Note that in this experiment, network data is never accessed by the CPU. In the case where the data is read by the CPU (e.g., to compute the UDP checksum), the maximal throughput decreases to 80 Mbps, due to the limited memory bandwidth on this machine.

Figure 4 shows the corresponding results obtained using DEC 3000/600 workstations. This machine has a greatly improved memory system. A buffered crossbar allows DMA transactions and cache fills/cache write-backs to proceed concurrently, and hardware ensures cache coherence with respect to DMA. The

[5]Our otherwise standard implementations of IP and UDP were modified to support message sizes large than 64KB.

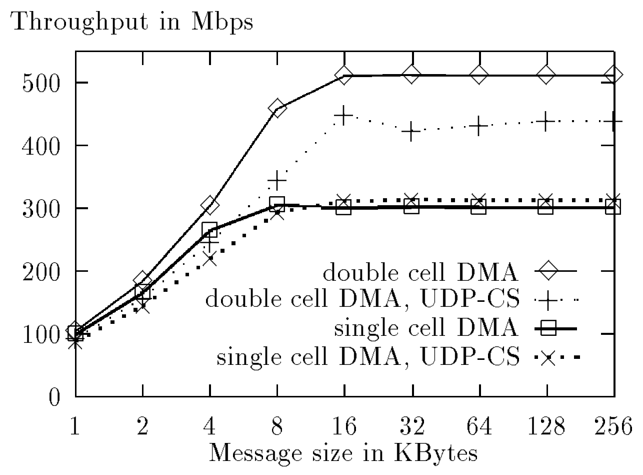

Figure 4: DEC 3000/600 UDP/IP/OSIRIS Receive Side Throughput

experiment was run with single and double DMA transfers, and with UDP checksumming turned on and off. With double cell length DMA, the throughput now approaches the full link bandwidth of 516 Mbps for message sizes of 16KB and larger. With DMA checksumming turned on, the throughput decreases slightly to 438 Mbps. This is an important result; it implies that the network data can be read and checksummed at close to 90% of the network link speed. Also, the throughput for small messages has improved greatly, thanks to the reduced per-packet software latencies on this faster machine.

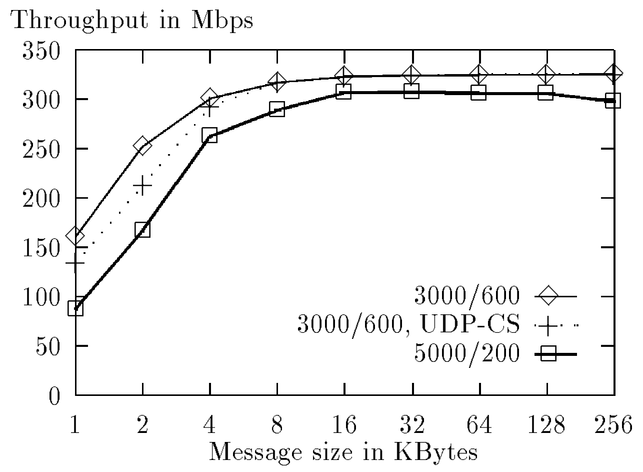

Figure 5: UDP/IP/OSIRIS Transmit Side Throughput

The final set of measurements evaluates the network performance on the transmit side. The results for both the DEC 5000/200 and the 3000/600 are shown in Figure 5. The maximal throughput achieved on the transmit side is currently 325 Mbps. This number is limited entirely by TurboChannel contention due to the high overhead of single ATM cell payload sized DMA transfers. A hardware change to allow longer DMA transfers in this direction is underway, but was not completed at the time of this writing.

With double cell DMA transfers on the transmit side, the host-to-host throughput attained is expected to fall between the graphs for single cell DMA and that for double cell DMA on the receive side (Figure 4). The exact result depends on the rate of double cell DMA transfers on the receiving host, as detailed in Section 2.6.

5 Conclusions

Based on the experience of writing software for the OSIRIS network adaptor, we draw three broad conclusions. First, the flexibility built into the adaptor was critical to its success as an experimental apparatus. This flexibility was primarily embodied in the fact that both the adaptor's algorithms, and the interface it presents to the host, are defined by software; programmable logic provides additional flexibility. This provided several distinct benefits.

- It allowed us to work around unexpected problems. For example, in the case of the network introducing skew that we were powerless to remove, we were able to re-program the segmentation/reassembly code running on the board's microprocessors.
- It helped us to avoid forcing the abstractions of the hardware designer onto the software architect. The major example of this was the problem of fixed-length DMA.
- It allowed us to tune the host/adaptor interface, thereby making it easier to write efficient operating system software. Simple examples of how we optimized this interface include minimizing locking contention between the host and the board, and reducing receive interrupts to less than one-per-PDU. More complex examples include fast buffers and application device channels.

While speed is often sacrificed for flexibility, it is noteworthy that we were still able to reassemble ATM cells in the common case and in the absence of misordering at approximately OC-12 speeds in software. Given that production adaptors will probably use custom hardware for reassembly, which will be faster but less flexible, we feel this is strong evidence that the cost of reassembly is not excessive.

Second, there were several difficult (and non-obvious) problems that the operating system had to address, all of which are essentially independent of the OSIRIS board. Examples include dealing with buffer fragmentation, page wiring, and cache coherence.

Finally, given that the OSIRIS adaptor was designed to provided maximal flexibility, it contains many more features than one would include in a production board. Based on our experience, we have found the following two features to be important, and would recommend that they be considered in future board designs.

- The ability to make an early demultiplexing decision; treating VCIs as an abundant resource that represents end-to-end connections is a reasonable way to do this on an ATM network. This is used by both the fbuf and ADC mechanisms.
- The ability to support multiple transmit and receive queues, and map each of them directly into user-level protection domains. It was this feature that facilitated the ADC mechanism.

Acknowledgements

Special thanks to David Mosberger for porting the x-kernel to the DEC Alphas. We also thank Jody Davie for suggesting the name OSIRIS, in honor of the Egyptian god reported to be the first victim of segmentation and reassembly. According to legend, he was cut into pieces by his brother Seth and reasembled by his wife, Isis.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for Unix development. In *Proceedings of the USENIX Summer '86 Conference*, July 1986.

[2] D. Banks and M. Prudence. A high-performance network architecture for a PA-RISC workstation. *IEEE Journal on Selected Areas in Communications*, 11(2):191–202, February 1993.

[3] G. Blair, *et al*. A network interface unit to support continuous media. *IEEE Journal on Selected Areas in Communications*, 11(2):264–275, February 1993.

[4] D. Clark, *et al*. The AURORA gigabit testbed. *Computer Networks and ISDN Systems*, 25:599–621, 1992.

[5] Eric Cooper, *et al*. Host interface design for ATM LANs. In *Proc. 16th Conf. on Local Computer Networks*, Minneapolis, MN, October 1991.

[6] C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley. Afterburner. *IEEE Network*, 7(4):36–43, July 1993.

[7] B. S. Davie. A host-network interface architecture for ATM. In *Proc. ACM SIGCOMM '91*, Zurich, September 1991.

[8] B. S. Davie. The architecture and implementation of a high-speed host interface. *IEEE Journal on Selected Areas in Communications*, 11(2):228–239, February 1993.

[9] P. Druschel, M. B. Abbott, M. Pagels, and L. L. Peterson. Network subsystem design. *IEEE Network (Special Issue on End-System Support for High Speed Networks)*, 7(4):8–17, July 1993.

[10] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, Dec. 1993.

[11] D. C. Feldmeier. Multiplexing issues in communication system design. In *Proc. ACM SIGCOMM '90*, pages 209–219, Philadelphia, PA, Spetember 1990.

[12] N. C. Hutchinson and L. L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan. 1991.

[13] V. Jacobson. Efficient protocol implementation. *ACM SIGCOMM '90 tutorial*, Sept. 1990.

[14] C. Maeda and B. Bershad. Protocol service decomposition for high-performance networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, Dec. 1993.

[15] M. Pagels, P. Druschel, and L. L. Peterson. Cache and TLB effectiveness in the processing of network data. Technical Report 93-4, Department of Computer Science, University of Arizona, Mar. 1993.

[16] K. K. Ramakrishnan. Performance considerations in designing network interfaces. *IEEE Journal on Selected Areas in Communications*, 11(2):203–219, February 1993.

[17] F. Reynolds and J. Heller. Kernel support for network protocol servers. In *Proceedings of the USENIX Mach Symposium*, pages 149–162, Monterey, Calif., Nov. 1991.

[18] J. M. Smith and C. B. S. Traw. Giving applications access to Gb/s networking. *IEEE Network*, 7(4):44–52, July 1993.

[19] C. Thekkath, T. Nguyen, E. Moy, and E. Lazowska. Implementing network protocols at user level. In *Proceedings of the SIGCOMM '93 Symposium*, Sept. 1993.

[20] C. B. S. Traw and J. M. Smith. Hardware/software organization of a high-performance atm host interface. *IEEE Journal on Selected Areas in Communications*, 11(2):240–253, February 1993.