The APIC Approach to High Performance Network Interface Design: Protected DMA and Other Techniques†

Zubin D. Dittia
zubin@dworkin.wustl.edu
Tel: +1-(314)-935-4163
Fax: +1-(314)-935-7302

Guru M. Parulkar
guru@cs.wustl.edu
Tel: +1-(314)-935-4621
Fax: +1-(314)-935-7302

Jerome R. Cox, Jr.
jrc@cs.wustl.edu
Tel: +1-(314)-935-7534
Fax: +1-(314)-935-7302

Department of Computer Science
Campus Box 1045 / Bryan 509
Washington University
One Brookings Drive
St. Louis, MO 63130-4899, USA.

All correspondence regarding this submission should be addressed to:
Zubin D. Dittia
zubin@dworkin.wustl.edu

Abstract

We are building a very high performance 1.2 Gb/s ATM network interface chip called the *APIC* (ATM Port Interconnect Controller). In addition to borrowing useful ideas from a number of research and commercial prototypes, the APIC design embraces several innovative features, and integrates all of these pieces into a coherent whole. This paper describes some of the novel ideas that have been incorporated in the APIC design with a view to improving the bandwidth and latency seen by end-applications. Among the techniques described, *Protected DMA* and *Protected I/O* were designed to allow applications to queue data for transmission or reception directly from user-space, effectively bypassing the kernel. This argues for moving the entire protocol stack including the interface device driver into user-space, thereby yielding better latency and throughput performance than kernel-resident implementations. *Pool DMA* when used with *Packet Splitting*, is a technique that can be used to build true zero-copy kernel-resident protocol stack implementations, using a page-remapping technique. *Orchestrated Interrupts* and *Interrupt Demultiplexing* are mechanisms that are used to reduce the frequency of interrupts issued by the APIC; the interrupt service time itself is significantly reduced through the use of a hardware *Notification Queue* which is used to report the occurrence of events in the APIC to software.

Although our ideas have been developed in the context of an ATM network interface, we believe that many of these ideas can be effectively employed in other contexts too (including different types of network interfaces). In particular, we believe that protected DMA and protected I/O could be used by other devices in the system, thereby facilitating the construction of microkernels that can potentially deliver better performance than their monolithic counterparts.

Keywords

ATM, Adapter, Network Interface, I/O, Interrupt, DMA, Protected DMA.

---

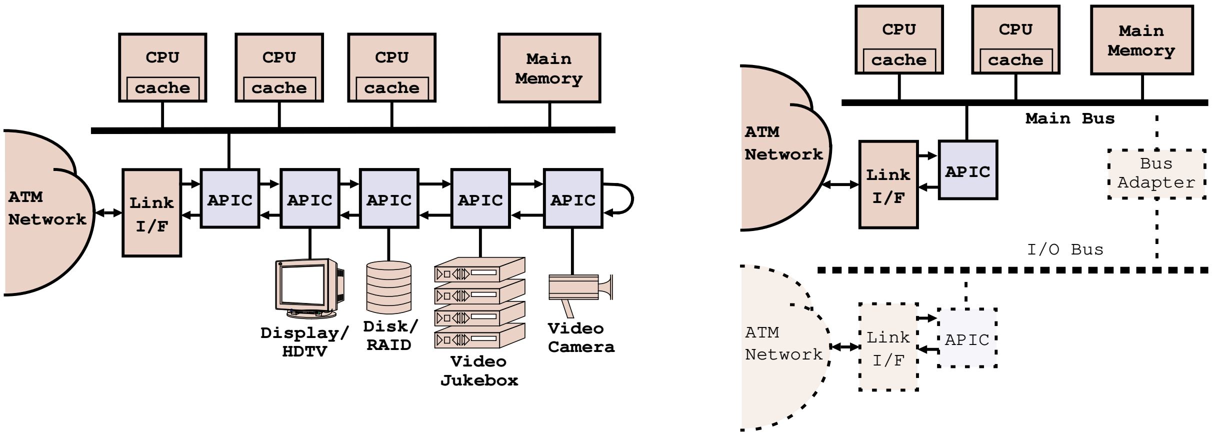
† This work was supported in part by the Advanced Research Projects Agency (ARPA), the National Science Foundation (NSF), Rome Laboratories, and an industrial consortium of Ascom Nexion, Bay Networks, Bell Northern Research, NEC America, NIH, NTT, Samsung, Southwestern Bell, and Tektronix.

1 Introduction

Network interface design and the structuring of protocol implementations in operating systems are two areas of research that have received considerable attention in recent years [1,3,4,5,7,8,9,22,23,28,30]. The sudden interest in these subjects can be attributed to the observed gross disparity between the raw bandwidth available from emerging gigabit networks, and the maximum effective throughput that can be delivered to end applications. The bottlenecks in end-systems that give rise to this disparity are far easier to identify than to rectify. Indeed, it is a well known fact that memory bandwidth limitations, coupled with the overhead of operations such as data copying, software checksumming, servicing of interrupts, and context switching are usually responsible for the observed poor performance. Several of the aforementioned research efforts have identified mechanisms that are at least partially effective in overcoming some of these handicaps. We have attempted to integrate a number of these "proven" good mechanisms, along with some new ones of our own creation, in our attempt to build a state-of-the-art high performance ATM network interface. Our research prototype of this network interface, which is called the APIC (ATM Port Interconnect Controller), is capable of supporting a full duplex link rate of 1.2 Gb/s. Since one of our overriding goals during the course of the project was to minimize cost (of a production version of the interface), the entire interface is being implemented on a single ASIC chip. In addition to being a network interface in the traditional sense, the APIC also has the capability to double as a building block for an I/O Desk-Area Network (DAN) [20,21], or a Very Local Area (ATM) Network (VLAN). This was achieved by building into the chip a small 2×2 ATM switch that would allow multiple APICs to be interconnected to create a small network; each APIC in this resulting network can directly interface either to an I/O device or to a computer system bus. An example of an APIC-based DAN is shown in Figure 1(a). An introduction to the APIC chip and APIC-based DANs, and a brief overview of the internal design of the APIC can be found in [8]. In this paper, we present several innovative mechanisms that have been incorporated in the APIC design with the objective of delivering very high performance (in terms of both throughput and latency) to end-applications running on the host computer.

1.1 A Brief Overview of APIC Features

The APIC has two ATM ports which can be connected either to an optical link interface, or to another APIC. Both ports implement ATM Forum's 16-bit wide UTOPIA (Universal Test and Operations PHY Interface for ATM) standard to connect to receive and transmit links. The prototype APIC interfaces to computer systems and devices over an external memory interface which supports three different bus standards: Sun's MBus, which is a 2.56 Gb/s (peak) 64-bit wide CPU-memory bus, and 32 and 64 bit wide versions of the PCI bus, which is a 1.056 Gb/s peak (for PCI-32, twice that for PCI-64) I/O bus. Thus, as



(a) An APIC-based Desk-Area Network

(b) APIC as a Traditional Network Interface

Figure 1: The APIC Gigabit Network Interface

shown in Figure 1(b), the APIC is capable of interfacing to a host system either over an I/O bus or over a CPU-memory bus. When interfacing to a CPU-memory bus, the APIC issues cache-coherent transactions to ensure cache consistency.

The capability of interfacing directly to a CPU-memory (main) bus is a noteworthy departure from most existing network interface designs, which typically interface to an I/O bus only. Main bus implementations of the network interface enjoy a number of benefits that I/O bus implementations cannot, not the least of which is improved performance. For example, an APIC that interfaces to a CPU-memory bus can physically address the entire range of physical memory in the machine, whereas an I/O bus implementation would, in most typical systems, be forced to use virtual mappings[1] to access physical memory that falls outside of the limits imposed by the I/O bus width. As a case in point, Sun's MBus supports a 36-bit physical address space, but the I/O bus (SBus) supports only 32 bit addresses. Other problems associated with I/O bus interfacing typically arise when the bus adapter that is used to interface the I/O bus to the main bus has been improperly designed. As networking becomes more an integral part of computer systems, we expect that network interfaces will migrate to the main bus on the motherboard in order to achieve maximum performance. This trend is already visible in emerging systems such as Sun's new UltraSparc line of machines, which feature a 100 Mb/s fast Ethernet adapter built into the motherboard.

The APIC implements full **AAL-5 Segmentation and Reassembly** (SAR) functionality on-chip. The APIC does not store-and-forward entire AAL-5 frames; instead, it implements cut-through at cell granularity. For this reason, the APIC can make do with only 256 cells worth of buffering on-chip, and no off-chip buffers. This is in keeping with our objective of very low production costs, since APIC interface

[1] These would have to be setup in advance by the CPU in a special I/O MMU that is built into the bus adapter.

cards do not need any on-board memory and almost no glue logic. In addition to AAL-5 SAR, the APIC also supports transmission and reception of raw ATM cells (**AAL-0**).

Among other APIC features, **multipoint and loopback connections**, and **remote controllability** (using control cells which can originate elsewhere in the network) bear mention. Some other features are listed below:

Cell Batching: This refers to the ability to batch cells (from the same connection) that are to be read/written from/to contiguous locations in memory, when executing bus transfers. This is motivated by the fact that larger sized bus transfers are usually more efficient (especially when interfacing to an I/O bus).

Cell Pacing: When transmitting data, the operating system is responsible for scheduling bursts of data to be sent (an example of a burst is a video frame), and the APIC is responsible for pacing out cells belonging to bursts at a rate which is independently specifiable on a per-connection basis. Cell pacing is required to prevent cell buffer overflow in ATM switches, and to adhere to the peak rate specification that was agreed upon at connection setup time.

Low Delay Priority Mechanisms: A connection can be specified to be a “low delay” connection; traffic on such connections always gets priority within the APIC, and can bypass paced traffic.

Best Effort: The APIC allows a transmit connection to be specified as a “best-effort” connection; such connections share all the bandwidth that is left over after accounting for paced and low delay traffic. Future versions of the APIC will also include support for ATM Forum’s rate-based ABR proposal.

Transport CRC Assist: The APIC includes on-chip logic for assisting software in computing a transport level trailer CRC. Note that because the APIC is a cut-through interface, it cannot help with computing checksums (for outbound packets) that reside in transport protocol headers. This means that for legacy transport protocols (TCP) that use header checksums, the checksum will need to be computed in software.

In addition to the above features, the APIC includes DMA mechanisms that are designed to support efficient implementations of zero-copy protocol stacks, and mechanisms designed to reduce the frequency of interrupts and interrupt service overhead. The focus of this paper is on these **DMA and interrupt related mechanisms**. For this reason, it will be sufficient for the rest of the paper to think of the APIC as a traditional network interface (see Figure 1(b)), and to ignore its desk-area networking capabilities.

The prototype APIC is being implemented in 0.5 micron CMOS, with a (conservatively) estimated die size of 1.8 square centimeters, and a package size of a little over 200 pins. The VHDL coding effort for the APIC is currently underway, and we expect to have the first chips back from the foundry in November of 1996. This version of the APIC will support a maximum of 256 transmit and 256 receive connections, which we felt was enough for prototype demonstration purposes. Future versions of the APIC are planned, with support for 1024 transmit and 1024 receive connections.

1.2 Summary of Innovative Claims and Contributions

This paper introduces the following five techniques, in the context of the APIC interface design:

Protected DMA: This is a technique that is used to allow protocol libraries residing in user-space to queue buffers for transmission or reception directly to the APIC, without any kernel intervention, and without compromising the protection mechanisms of the operating system. Although similar functionality has been provided earlier in the Bellcore OSIRIS [9] and in the Cornell U-Net [14] interfaces, their implementations are not amenable to straightforward hardware implementation, and result in higher cost interfaces. Section 3.3 of the paper, which deals with protected DMA, discusses the relative merits of our approach in more detail.

Protected I/O: Protected I/O is a technique that is used to allow protocol libraries residing in user space programmed I/O access to the memory-mapped registers on the APIC, with the specific access permissions to individual registers specifiable by the OS kernel. This is very similar to the protection mechanism used to support *application device channels* in the OSIRIS [9] interface, but our scheme extends that idea to allow the kernel to have a finer granularity of control over access permissions to individual device registers.

Pool DMA with Packet Splitting: This technique enables true zero-copy[2] kernel-resident protocol stack implementations using page remapping. Although the idea of using page remapping to avoid data copying is not new [25], we show how it can be done cleanly and efficiently in the context of ATM networks.

Orchestrated Interrupts: When the CPU queues buffers for transmission or reception to the APIC, it can "mark" some of these buffers; the APIC will issue an interrupt whenever it encounters "marked" buffers. Such (pre-)*orchestrated interrupts* are useful for reducing the frequency of interrupts issued by the APIC.

Interrupt Demultiplexing: By allowing interrupts to be enabled/disabled on a per-connection basis, we can exploit the connection-oriented aspect of ATM networks to significantly reduce the frequency of interrupts raised by the APIC. We call this scheme "*interrupt demultiplexing*."

Notifications: Notifications are messages posted by the APIC to the operating system, recording the occurrence of events such as completion of transmission or reception of a frame, error conditions, etc. By reading notifications off a hardware queue on the APIC, an interrupt service routine can avoid costly scans of per-connection buffer chains to determine what events have occurred. Notifications afford simplifications to the OS software, and reduce interrupt service time.

We would like to point out here that although the above techniques have been developed in the context of an ATM network interface, we believe that many of these ideas can be used to advantage in other

[2] The term "true zero-copy" is used to indicate that data moves directly between application data structures and the network interface without any intermediate copying. A definition of the term can be found in [14].

scenarios too. In particular, we feel that it would be possible for other types of devices such as disks and video frame buffer cards to exploit protected DMA and protected I/O, and we intend to make this a subject for future study.

1.3 Organization of the Paper

The rest of this paper is organized as follows: Section 2 elaborates on the notion of protected I/O. Section 3 describes the three DMA mechanisms supported by the APIC. In Section 4, we explore the ideas of orchestrated interrupts, interrupt demultiplexing, and notifications. Finally, Section 5 makes some concluding remarks. Throughout the paper, comparisons with related work are made where appropriate.

2 Protected I/O

The APIC device driver running on the host CPU communicates with the APIC by two mechanisms: through reading/writing memory-mapped on-chip registers, and through shared data structures in the host's main memory. The APIC's on-chip device registers fall into two categories: global registers, and per-connection registers. The set of global registers contain state that is of global relevance, such as the interrupt status registers, the software reset register, etc. There is one set of per-connection registers for each connection supported by the APIC; these hold state that varies from connection to connection, such as the connection identifier (VPI/VCI), the pacing bandwidth, etc. Protected I/O is designed to give untrusted user-space protocol implementations the ability to directly read or write per-connection registers, in a way that would not compromise the security or protection mechanisms of the operating system. Protected I/O should really have been called "protected programmed I/O," but we chose the former name in the interests of brevity. Protected I/O is really just an extension of the protection mechanism that was used by Druschel et al. to implement application device channels (ADCs) in the OSIRIS interface [9]. The U-Net interface from Cornell [14] also uses the same mechanism as ADCs. Both of these efforts attempt to give an application read/write access to only those memory-mapped I/O registers on the interface that correspond to connections that are "owned" by the application. An application is said to "own" a connection if it was responsible for initiating the connection setup. Figure 2 demonstrates how the ADC protection mechanism works. The per-connection registers on the interface are mapped into its memory-mapped I/O address space in such a way that all registers corresponding to a connection fall into the same physical page frame, and no page frame contains registers for more than one connection. Thus, all device registers for a particular connection can be accessed through physical addresses that fall within the same page frame (or set of page frames, if the machine's page size is not large enough to hold all the registers) in the physically addressed memory-mapped I/O space of the device. When a user process makes a connection setup request to the OS kernel, the kernel modifies the system page table entries such that the page corresponding to the

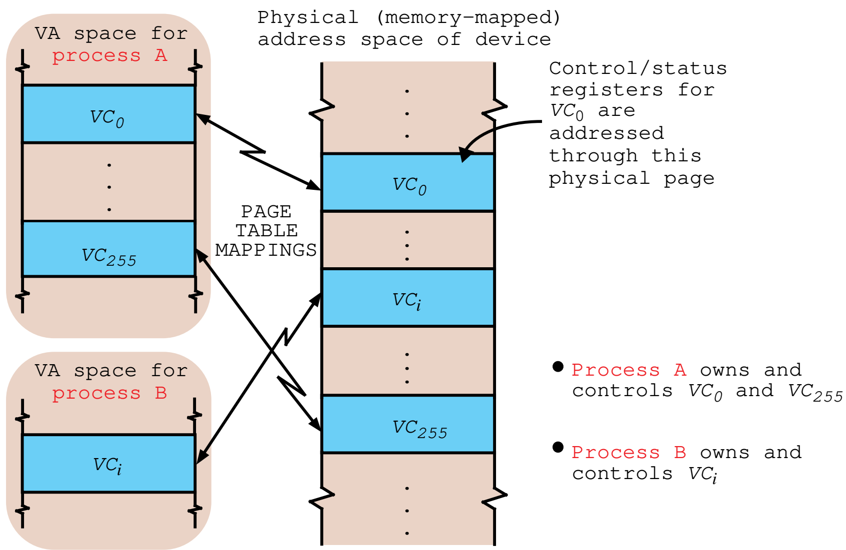


Figure 2: Protection Mechanism Used to Implement Application Device Channels (ADC)

connection is mapped into the process' virtual address space with read/write permissions. The process now becomes the owner of the connection, and it can control the connection by reading or writing the registers on the interface directly, without having to make system calls or otherwise interact with any trusted code. This process cannot, however, control other connections that it does not also own, because the pages corresponding to those connections are not mapped into its virtual address space. To summarize, the system's virtual memory (VM) protection mechanisms are overloaded to provide protected access to per-connection device registers.

The ADC scheme does not allow the kernel fine grain control over which individual per-connection registers the application is given access to, or selectively control the type of access (no access, read-only, or read-write) to these registers. This is important because some administrators may be willing to grant more privileges to their users than others, and some processes may similarly be given more control by the operating system than others. For example: Should a user process be allowed to change the pacing rate for a connection that it owns? Should the operating system permit a user process to dictate under what conditions the interface will raise an interrupt, when a connection-specific event occurs? There is no way currently to use the ADC protection mechanism to provide this type of fine grain access control, short of mapping each device register into a separate page, and setting up page table entries and protections appropriately. This scheme is not acceptable because if there are many registers per connection, the number of page mappings in each process' context becomes large, resulting in inefficiencies from potentially slower page table lookups in the MMU, and thrashing in the TLB. Protected I/O provides the solution to this problem by extending the ADC protection idea to include the concept of an *access mask register*. The access mask register is a register that augments the set of per-connection device registers, so that each connection

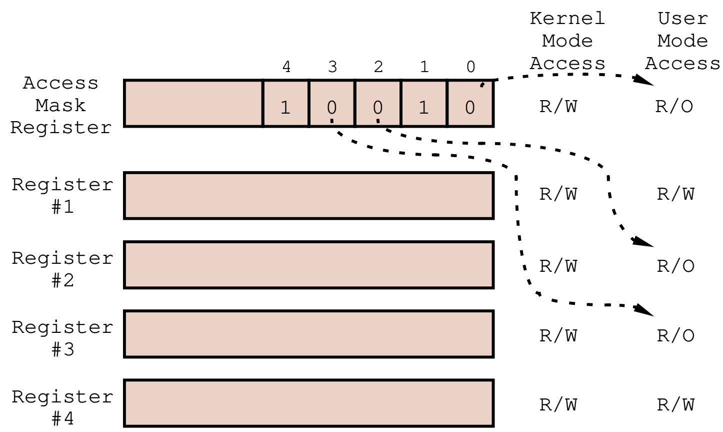


Figure 3: Illustration of Fine Grain Access Control using Protected I/O

has an associated access mask register. In addition, all per-connection registers (including the access mask register) are mapped twice, and in different page frames, in the physically addressed memory-mapped I/O space of the device. Note that this means that each hardware per-connection register can be accessed (read or written) using two different physical addresses, each of which resides in a separate page frame. We term these two different methods of access as kernel-mode and user-mode accesses. The kernel ensures that the virtual memory page tables are setup in such a way that the kernel can use kernel mode accesses for all registers, but user processes are forced to use user-mode accesses. Furthermore, the ADC protection mechanism is used to ensure that only a process that owns a connection can access the corresponding per-connection registers (using user-mode accesses). The device (APIC) differentiates read or write accesses to a per-connection register based upon the access type. Only kernel-mode accesses are guaranteed to be successful. Thus, the kernel has full read/write access to all device registers, including the access mask registers for all connections. As shown in Figure 3, the access mask register contains one bit for every per-connection register that is associated with the corresponding connection; for each bit that is set, the device allows user-mode write access to the corresponding register. Otherwise, only read access is granted. The kernel is responsible for setting up the access mask register to grant appropriate permissions to the user process that owns the corresponding connection. Clearly, this scheme can be extended to allow the option of providing no access (in addition to R/W and R/O) to registers by having two bits per register instead of just one, in the access mask register; however, this added functionality has not been included in the current version of the prototype APIC, because there were no per-connection registers that needed to be completely hidden from applications.

3 APIC DMA Mechanisms

The APIC uses DMA for all movement of transmit/receive data from/to main memory. DMA is also used by the APIC to read/write control data structures stored in main memory called buffer descriptors,

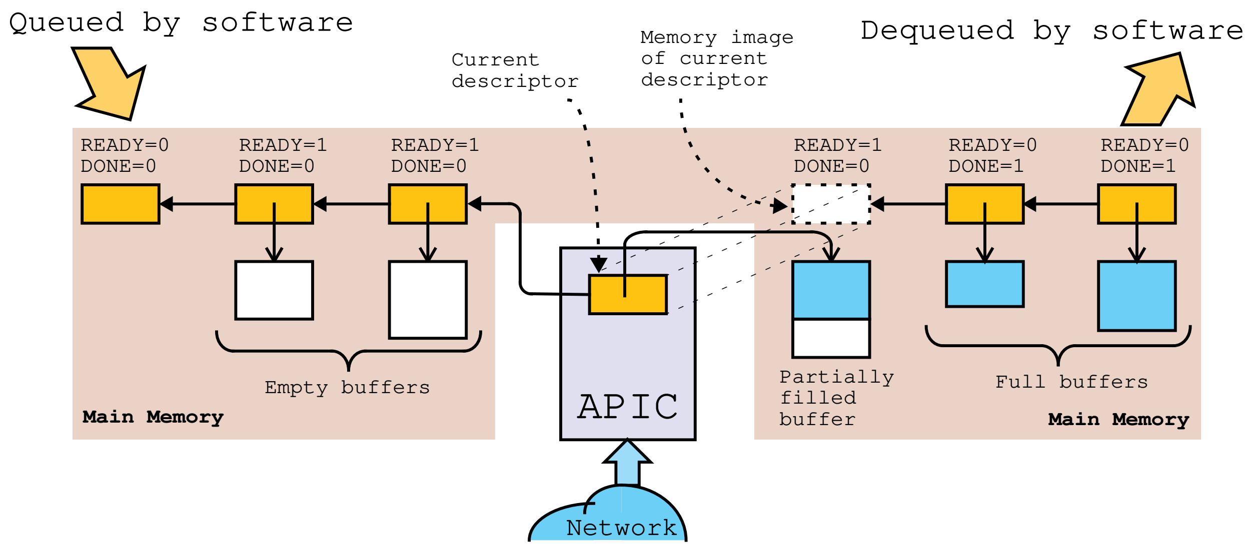


Figure 4: Example to demonstrate how data is received using buffer descriptors

which are needed to support scatter/gather DMA. All buffer descriptors are 16 bytes long, and contain information about a data buffer in physical memory, such as the physical address of the buffer, its length, and the type of data in the buffer (AAL-0 or AAL-5). Buffer descriptors are used by the host CPU to queue buffers containing data to be transmitted, or to queue empty buffers that will be used to hold received data. Figure 4 demonstrates the use of buffer descriptors in receiving data. For transmission, the scenario is almost identical, except that the buffers on the right in the figure would contain data that has already been transmitted (or is in the process of being transmitted), while the buffers on the left hold data that has yet to be transmitted. To allow buffer descriptors to be queued, each descriptor contains the address of the next descriptor in a singly linked list (chain) of descriptors. One descriptor in a chain is always designated as the “current” descriptor; this is the descriptor that the APIC is currently using. When the APIC is done with the current descriptor (i.e., it has finished transmitting all the data in the corresponding buffer, or it has filled the buffer with received data), it writes the descriptor back to main memory, and fetches the next descriptor in the chain. The current descriptor (which is on-chip) is visible to software through APIC device registers. These registers are also used by the software to setup the first descriptor in the chain, at the time of connection initialization. Each descriptor contains two bits, called the `READY` and `DONE` bits, that are used to synchronize APIC and CPU accesses of the descriptors. The `READY` bit is set by the CPU to inform the APIC that the corresponding buffer is ready to be transmitted from, or received into. The `DONE` bit is set by the APIC when it has finished sending data from (or receiving data into) the corresponding buffer. The CPU can “catch up” with the APIC by chasing the next descriptor links until it comes to a descriptor that has its `DONE` flag set to zero (this is the current descriptor). In addition to the fields already mentioned, buffer descriptors contain several control and status flags. These include an `INTERRUPT` flag

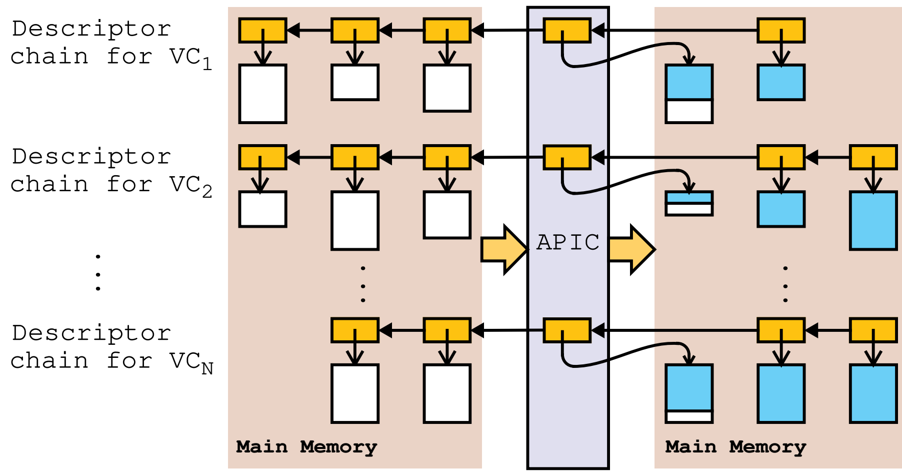


Figure 5: Illustration of Simple DMA

which can be set by the CPU to instruct the APIC to issue an interrupt when it first reads in the descriptor (see Section 4.1 for details on how this flag is used), an `END-OF-FRAME` flag, which is used to mark the last buffer of an AAL-5 frame, and a `CRC-OK` flag which is set by the APIC to inform the processor whether AAL-5 CRC verification was successful.

It should be noted that a single AAL-5 frame can span multiple data buffers. The APIC will always transmit all the data in a transmit buffer. However, the APIC does not allow data from more than one AAL-5 frame in a single receive buffer, so receive buffers may not be completely filled by the APIC (this happens if the portion of the received frame in the buffer is less than the size of the buffer).

The APIC supports three different varieties of scatter/gather DMA: Simple DMA, Pool DMA, and Protected DMA. The choice of which to use depends on whether protocols are implemented in the kernel (or equivalently, in a trusted user level server process) or in an untrusted user level library, and on whether data is being transmitted or received. We now go on to describe all three DMA mechanisms.

3.1 Simple DMA

With simple DMA, each connection supported by the APIC is associated with a separate and dedicated chain of descriptors, as shown in Figure 5. Since the prototype APIC supports a maximum of 256 transmit and 256 receive connections, there can potentially be as many as 512 active connections that have 512 different descriptor chains associated with them. Clearly, the APIC needs to have on-chip storage for 512 *current* descriptors.

Simple DMA can be used by kernel-resident (or trusted user-level server) protocol stack implementations only. This is because if untrusted user-level programs were given write access to descriptors, they could queue arbitrary buffers from main memory for transmission or reception (including those not other-

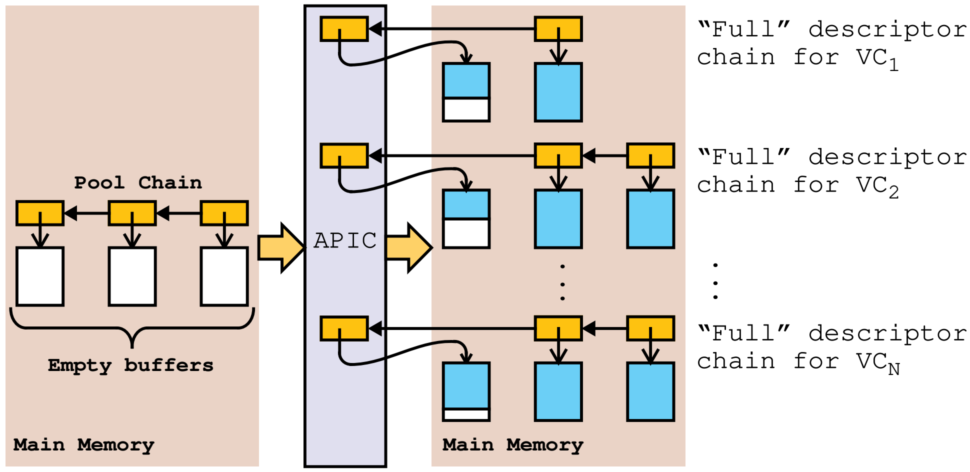


Figure 6: Illustration of Pool DMA

wise accessible to the process), thereby compromising the security and defeating the protection mechanisms of the operating system. Protected DMA, which is described in Section 3.3, is designed to be usable by untrusted user-level protocol stack implementations.

Simple DMA can be used for both transmit and receive connections. However, simple DMA can be very inefficient in its use of memory resources if it is used for receiving data. This is because simple DMA requires free (or empty) buffers to be dedicated to each open receive connection. Because free buffers queued on one connection cannot be used by another, and because there is no way to anticipate when data will arrive on a connection, the amount of free buffer space required can be very large. Pool DMA is designed to overcome this limitation of simple DMA.

3.2 Pool DMA

Pool DMA allows a set of receive connections to all share the same pool of free buffers (see Figure 6). The APIC fetches a descriptor from a global "pool chain" whenever a connection needs a new empty buffer to place arriving data in. Before it writes the old "full" descriptor back, however, the APIC fills in the next descriptor pointer field in that descriptor with the address of the new descriptor. This results in the buffer descriptors (and buffers) getting demultiplexed into separate chains of "full" buffers, one per connection. Note that pool DMA is intended for use by receive connections only; transmit connections cannot use pool DMA.

The prototype APIC supports four global pool chains, and it is possible to specify which of these chains to use for each connection. This can be useful, because we may know in advance that some connections always carry very small packets (DNS, for example), so we can use a pool of small buffers for all connections of that type. Other connections may draw from pools with larger sized buffers. Since the APIC

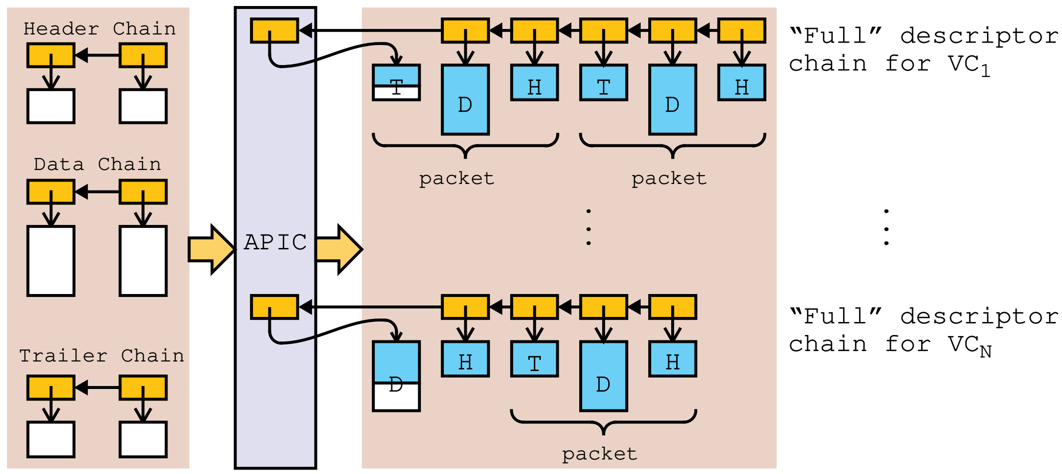


Figure 7: Pool DMA with Packet Splitting

does not allow multiple packets in a single buffer, this scheme results in further reductions in memory resource usage for buffers.

3.2.1 Pool DMA with Packet Splitting

The APIC allows a received packet (AAL-5 frame) to be split into a header portion, a data portion, and the AAL-5 trailer, with each of these portions written to buffers drawn from different pool chains. This is illustrated in Figure 7. The location of the header-data split is independently specifiable for each connection as an offset from the start of the packet; this "header length" is fixed for the lifetime of the connection. The location of the data-trailer split is determined by the APIC using the length field in the AAL-5 trailer.

The packet splitting feature of the APIC can be used to implement a true zero-copy protocol stack using page-remapping techniques, under the following assumptions: the network and transport protocol headers are of fixed size for each connection, and the amount of data in each packet is an exact multiple of the receiving host's page size[3]. The first of these assumptions is not as restrictive as it may seem: it is sufficient if the total (network + transport) header size is fixed over extended bursts of packets. Indeed, an examination of TCP/IP packet traces has shown us that in almost all cases, the combined TCP/IP header size remains fixed for the lifetime of a connection (excluding the connection setup and teardown phases). The second assumption, however, is not true of most existing transport protocols. It is required to get zero-copy behavior on the receiving end if we want to efficiently handle network errors and loss — to see this, note that if a packet is lost, there is a hole in the receive sequence space which must be filled by one or more received packets without a copy; this can be accomplished only if the retransmitted data fills the page

[3] If there are multiple receiving hosts, and if we assume that pages are always a power of 2 in size, then the data portion of each packet would have to be a multiple of the largest of the page sizes of all receivers.

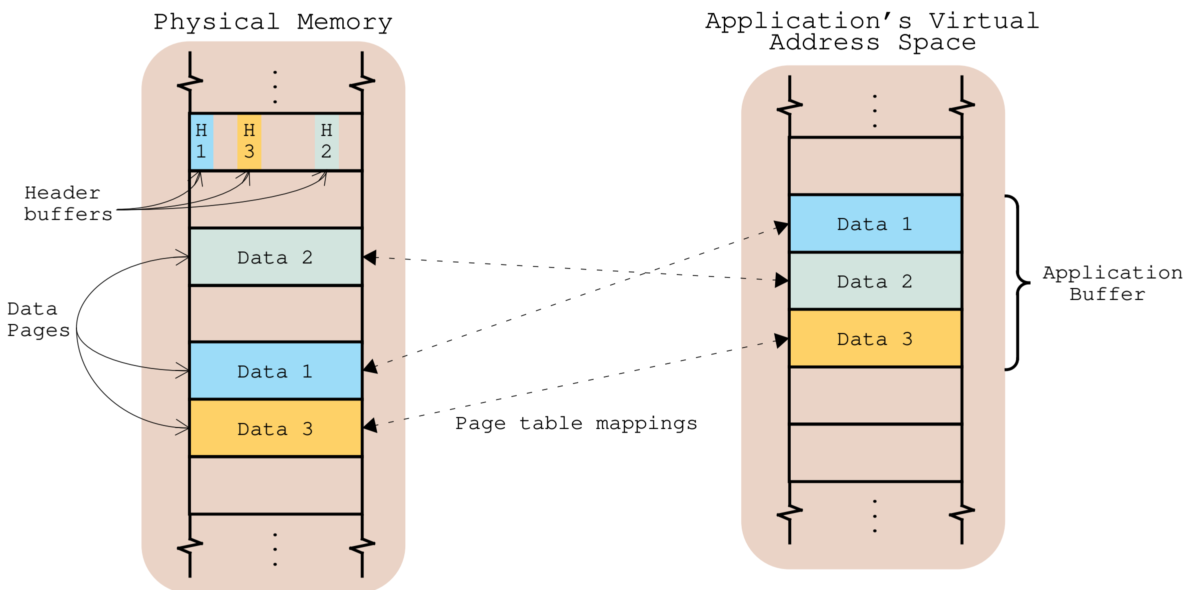


Figure 8: Zero-Copy Using Packet Splitting and Page Remapping

frames in the "hole" completely. This second requirement however, can be relaxed if we are willing to tolerate performance degradation resulting from copying data when there has been loss in the network. We are currently working on developing a high performance ATM transport protocol that satisfies both of the above assumptions.

The page-remapping strategy for implementing the receive side of a zero-copy protocol stack is illustrated by the example in Figure 8. The APIC splits received packets into headers and data (trailers, which are handled in the same way as headers, have intentionally been omitted to simplify the example). The split is possible because the APIC knows the fixed header length in advance. The header buffers are all drawn from a pool of buffers that are mapped into the kernel's address space. Buffers that will hold the data portion of packets are all drawn from a pool of free physical pages. Since each packet holds one or more pages worth of data, no page will be left partially filled. When packets are received, the operating system examines the header of the packet to determine where the corresponding data page(s) should reside in the application's address space; it then modifies the page table entries appropriately. The end-result is that data from consecutively sequenced packets appears contiguous in the application's address space. The (sometimes high) cost of modifying page table entries can be minimized by processing packets in batches, thereby avoiding the need to flush the TLB for every packet; this is known as lazy updating of page tables. The transmit side of the zero-copy protocol stack is considerably simpler: data can be transmitted directly from an application buffer using simple DMA; no page-remapping is needed.

Note that in a zero-copy protocol stack implementation, a BSD socket type of API cannot be presented to applications. A suitable API is provided by fbufs [10], container shipping [26], or UCMIO [6]. These APIs avoid the problem of conflicts between the transport protocol, which wants to keep data in its retransmission queue unchanged until acknowledged by the receiver, and the application process, which may want to overwrite data in these buffers before it has been acknowledged. The fbuf and container shipping schemes, for example, solve this problem by taking write permissions for transmit buffers away from the sending process, thereby forcing it to write new data into a different buffer.

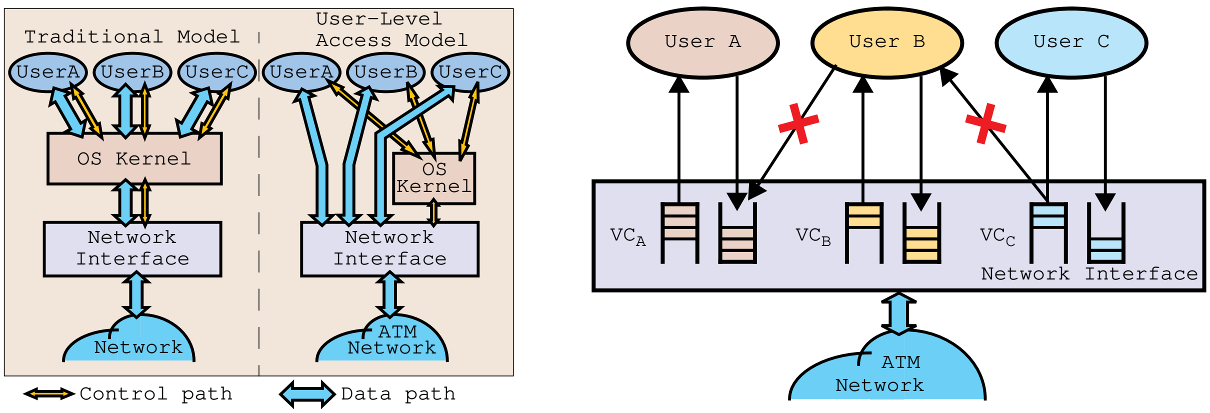
Finally, note that using page-remapping to eliminate data copying is by no means a new idea [25]. The Axon [28] network interface was among the first to introduce the use of page-remapping as a method for avoiding data copying.

3.3 Protected DMA

3.3.1 Motivation and Related Work

There have been several research efforts [12,13,19,24,29] that have attempted to make user-level library implementations of protocol stacks that can be linked with the application. Most of these efforts have left the network interface device driver inside the kernel, thereby requiring the use of system calls for transmitting and receiving data. System calls are expensive — our experiments show that a null system call takes between 10 and 50 microseconds in state-of-the-art workstations running derivatives of the Unix operating system. In and of itself, this is bad from the viewpoint of minimizing end-to-end latency for latency-critical LAN applications (many distributed computing applications, RPC, and NFS fall in this category). But the high cost of system calls can also result in poorer throughput if the protocols have not been implemented carefully. Most of the researchers who have explored library implementations of protocols have given serious attention to amortizing the overhead of a system call over multiple operations by batching them into a single system call. We believe that this places an unnecessary burden on protocol coders, and makes the code less portable and more dependent on the underlying operating system. Furthermore, this technique does not solve the higher end-to-end latency problem discussed earlier.

To solve the afore-mentioned problems, system calls should be eliminated from the critical data path. As shown in Figure 9(a), the data path should be between the user-space and the network interface. The control path operations (for example, connection setup and teardown), can still pass through the kernel without incurring a significant performance penalty. With this objective in mind, Druschel et al. [9] proposed using ADCs (described earlier in Section 2) to queue buffer descriptors directly on the network interface. The ADC protection mechanism would ensure that one user could not queue/dequeue descriptors to/from another user's descriptor queue (see Figure 9(b)). The network interface was given the task of



(a) Eliminating kernel from data path **(b) Using ADCs to queue/dequeue descriptors**

Figure 9: Providing User-Level Access to the Network Interface

ensuring that buffer descriptors queued by an application pointed to valid buffers to which the application had appropriate read or write access. To make these checks feasible, the network interface relied on ownership and access information for buffers that was provided to it in advance (at connection setup time), by the operating system kernel. The checks themselves were performed by the two on-board *i*-960 processors that were part of the OSIRIS network interface. The U-Net interface from Cornell [14] also relied on these same ideas to achieve a direct data path between user-space and the network adapter, but this work was done by modifying the firmware for the on-board *i*-960 processor on Fore-System's SBA-200 ATM adapter.

There are three main problems with the OSIRIS/ADC and U-Net approaches that make them unsuitable for use with APIC-like network interfaces:

1. All buffer descriptors have to reside on the network interface board. For high bandwidth network interfaces, the total amount of buffer descriptor space needed can be quite large (a megabyte or more). In any case, the amount of space needed would be much more than can fit on a single chip. This means that the network interface would need to have a significant amount of external on-board memory, thus increasing the cost of the interface substantially (especially if the memory is implemented as SRAM).
2. The network interface needs to store information about all valid communication buffers that are in use by each application, and their access permissions, in order to ensure that applications cannot queue illegal buffers to the interface. This means even more memory on the network interface board, and consequently, even higher cost.
3. A fast on-board processor is needed to perform the buffer validity tests. This involves checking the buffer that is queued against all valid buffers that can be queued by an application. A buffer indexing or a hashing scheme can be used to minimize this overhead, but in any case, it is not amenable to easy

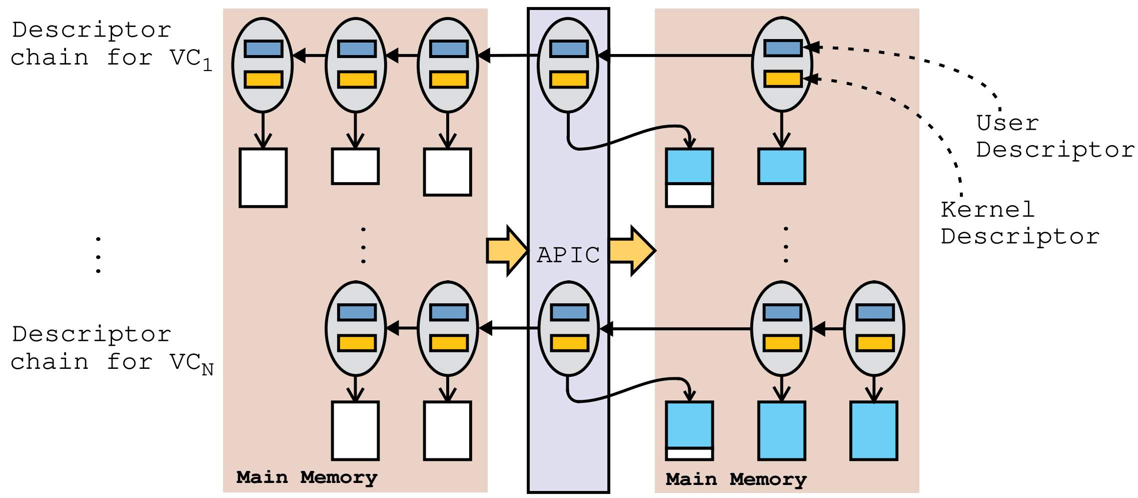


Figure 10: Illustration of Protected DMA

hardware implementation. We would like to point out here that the details of the types of checks performed, or the mechanism used, has not been elaborated upon in ADC or U-Net publications [9,11,14].

Our approach to solving the problem of providing untrusted user-space protocols the ability to queue or dequeue data directly to/from the interface is called *Protected DMA*. This approach does not suffer from any of the three drawbacks mentioned above, and it is usable by both APIC and OSIRIS/U-Net types of interfaces.

3.3.2 Description of Protected DMA

In protected DMA, all buffer descriptors reside in main memory. Protected DMA is in many ways similar to simple DMA; however, each buffer is associated with two descriptors instead of one (see Figure 10). We will call these *kernel descriptors* and *user descriptors*. Before the APIC uses a buffer, it always reads both descriptors for the buffer. The user descriptor contains values supplied by a user level program, so these values cannot be trusted. However, the kernel sets up descriptors in such a way that the kernel descriptor can be used to check the validity of the values supplied in the user descriptor. Thus, before the APIC begins using a buffer, it uses the contents of the kernel descriptor to validate, or *notarize*, the values supplied by the application in the user descriptor. The contents of the kernel descriptor are guaranteed by the operating system to be trustworthy; the kernel ensures this by using the VM protection mechanism of the system to disallow user-space access to kernel descriptors.

The association between communication buffers and their descriptors is done by the kernel at the behest of the user, through the use of system calls. When the user process makes such a request, the kernel first wires down the pages corresponding to the buffer in main memory, picks a pair of descriptors for the buffer, initializes the contents of the kernel descriptor, and returns the user descriptor to the user. If the

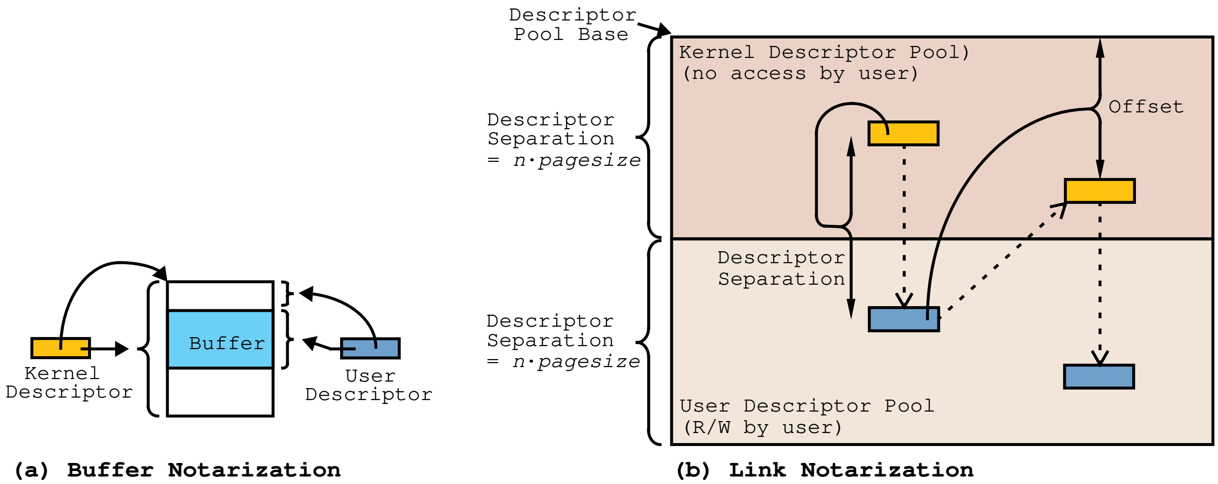


Figure 11: Notarization for Protected DMA

buffer spans multiple non-contiguous physical pages, then multiple pairs of descriptors, one pair per page, will be associated with the buffer. Kernel descriptors always reside in a page that is not mapped into the user's virtual address space, and user descriptors always reside in a page to which the user has full read/write access. This ensures that user processes can only modify user descriptors, but not kernel descriptors.

3.3.2.1 Buffer Notarization Kernel descriptors are initialized by the kernel to contain the actual physical address of the buffer they reference, and its length (see Figure 11(a)). User descriptors are interpreted by the APIC to contain an offset into the buffer, and a length. Thus, the application is allowed to queue subsets of communication buffers. The APIC certifies the validity of a buffer by ensuring that the sum of the offset and length specified in the user descriptor is no greater than the length specified in the kernel descriptor. We call this checking process *buffer notarization*. If a buffer fails notarization, the corresponding connection is suspended and the APIC issues an exception interrupt to the processor; the kernel can then take whatever action it deems fit. If a buffer passes notarization, the APIC creates a new *resultant descriptor* which contains the length field from the user descriptor, and an address field which is computed as the sum of the address field from the kernel descriptor and the offset field from the user descriptor. This resultant descriptor is then used just like a normal (simple DMA) descriptor would. When the APIC is done transmitting or filling the buffer, it writes back the resultant descriptor to the location of the user descriptor. This allows the user process to be able to examine the status of the buffer/connection. The kernel descriptor is only read and never written, since the kernel is not responsible for keeping track of the state of the connection. Note also that the APIC needs on-chip storage for only one (resultant) descriptor per connection.

3.3.2.2 Link Notarization Although buffer notarization is sufficient to ensure that a user process never queues an illegal buffer, it cannot guarantee that the user will not queue an illegal descriptor. To get around this, the APIC has to subject the next descriptor link in user descriptors to another check which we call *link notarization*. Figure 11(b) shows how link notarization works. Every user process is allocated a dedicated set of $2 \cdot n$ contiguous physical memory pages that will be used to hold user and kernel descriptors for that process — this can be likened to the file descriptor table for a process. The value of n is fixed, and is configured into the operating system kernel at compile time. The first n pages contain only kernel descriptors, and will be referred to as the kernel descriptor pool. Likewise, the last n pages contain only user descriptors, and will be referred to as the user descriptor pool. Pairs of user and kernel descriptors are always allocated such that both descriptors are located at the same offset relative to the start of their respective descriptor pools. All the pages in the user descriptor pool are mapped with read/write permission into the process' virtual address space, while the pages in the kernel descriptor pool are accessible to the kernel only. The APIC per-connection state includes a pointer to the descriptor pool base of the process which owns the connection. This allows user processes to be able to use offsets within the descriptor pool to reference descriptors, rather than physical memory addresses. It also makes the job of link notarization easy for the APIC. As shown by the dotted arrows in the figure, the APIC always reads the kernel descriptor for a buffer before reading the corresponding user descriptor. Kernel descriptors always contain the relative offset of the corresponding user descriptor (this *descriptor separation* is always $n \cdot pagesize$); the APIC uses this to determine the address of the user descriptor. User descriptors specify the link to the next descriptor as an offset into the descriptor page pool. In order to notarize this link, the APIC has to ensure that the next kernel (user) descriptor falls within the corresponding kernel (user) descriptor pool. It does this by making sure that the offset specified in the user descriptor is no larger than the descriptor separation specified in the corresponding kernel descriptor. If the link fails notarization, the APIC suspends the corresponding connection and issues an exception interrupt to the processor. If the link passes notarization, the APIC initializes the next descriptor link of the resultant descriptor to hold the address of the next kernel descriptor, which is computed as the sum of the descriptor pool base and the offset in the user descriptor.

Thus, with two simple inequality checks, the APIC ensures that no illegal buffers or buffer descriptors can be queued by a user process. Once a process has queued a buffer, it can inform the APIC that this has been done by issuing an *attention* command to the APIC (using protected I/O), thus completely removing the kernel and system calls from the critical data path. Protected DMA can therefore be used by untrusted user-space library protocol implementations, with the confidence that the security and protection mechanisms of the operating system will not be compromised.

We would like to point out here that although we have made all control path operations on the APIC pass through the kernel, there is no reason that they could not instead be made to pass through a trusted user-level server directly to the APIC. If this is done, then we have succeeded in moving almost the entire protocol stack including the APIC device driver, into user space (some kernel code would still be required to field interrupts from the APIC, but this code can be very small because all it needs to do is wake up relevant processes). This technique is particularly attractive for microkernel implementations, and has the potential to outperform monolithic kernel implementations that keep the entire device driver in the kernel.

4 APIC Interrupt Mechanisms

Interrupts are issued by the APIC to report the occurrence of asynchronous events, such as completion of transmission or reception of data, or an error condition. It is important that interrupts not be issued very frequently, because interrupt service overhead can be substantial. Furthermore, interrupts have a bad effect on cache performance. Our experiments with a prototype MBus test board have shown that the minimum interrupt service latency (the time from the instant when the interrupt was issued to the time when the first instruction of the device interrupt service routine is executed) for a Sun Sparcstation-10 running SunOS 4.*x* is 14 microseconds. This implies that the time to service even a null interrupt can be greater than the interarrival time between successive packets! Several researchers have recognized the importance of reducing the frequency of interrupts [9,17,27,30] to a host interface. All proposed solutions rely on batching multiple events so they can be handled with a single interrupt. Druschel et al. [9] suggest disabling transmit interrupts altogether, and checking for the completion of transmission as part of other driver activity. On receive, they interrupt only when new data arrives and there is no old data that has not already been dequeued by the OS; this has the desirable effect of the interrupt being asserted only once per burst of incoming packets. Traw et al. [30] suggest disallowing interface interrupts altogether, and instead polling the interface on periodic hardware clock interrupts. This scheme is shown to perform well for high bandwidth applications, but is unsuitable for latency-critical LAN applications because the time between successive clock interrupts is typically of the order of milliseconds. The APIC reduces interrupt frequency through two approaches: *orchestrated interrupts*, and *interrupt demultiplexing*. Furthermore, the APIC includes a *notification* mechanism that reduces the time to service interrupts.

4.1 Orchestrated Interrupts

As mentioned earlier, buffer descriptors used by the APIC contain an `INTERRUPT` bit flag. If this flag is set in a descriptor that has been queued on a transmit or receive chain, the APIC will interrupt the processor when it first reads the descriptor. This variety of interrupts are termed orchestrated, because they are issued in response to a fabricated and pre-orchestrated event (viz., the reading of the descriptor). This feature is very useful when transmitting data, because the processor can avoid getting interrupted for every packet, and only get interrupted when the last packet in a large burst has been transmitted. This is achieved by only setting the descriptor interrupt bit in the last descriptor corresponding to the last packet of the burst. Orchestrated interrupts are also used to notify the processor when a (transmit or receive) descriptor chain underflow is imminent. This is done by setting the interrupt bit in a descriptor that is located close to the end of a chain of descriptors; when the APIC reaches this descriptor, it will issue an interrupt and in response, the kernel can take steps to replenish buffers in the chain. Note that orchestrated interrupts can be used to advantage in both connection-oriented and traditional network interfaces.

4.2 Interrupt Demultiplexing

Interrupt demultiplexing is a strategy employed by the APIC to reduce the frequency of interrupts that are used to report packet arrivals (note that orchestrated interrupts cannot help here). With interrupt demultiplexing, traffic patterns and protocol and application processing requirements for individual applications directly affect the frequency of interrupts that are generated. For example, latency-critical applications may cause an interrupt to be issued for every packet that is received, while high bandwidth multimedia applications which are willing to tolerate the higher latency (that is a consequence of attempts to reduce the interrupt frequency) may trigger an interrupt per burst of packets. The connection-oriented aspect of ATM is exploited to differentiate (demultiplex) between applications with different traffic patterns and protocol and application processing requirements—hence the name *interrupt demultiplexing*.

In the APIC, interrupt demultiplexing is implemented by including a one-bit flag in the per-connection state which is used to tell whether interrupts are enabled for the connection. When an event (e.g., packet arrival) occurs on a particular connection, an interrupt is generated only if interrupts were enabled for that connection, and immediately following this the APIC automatically disables interrupts for that connection. Subsequent packet arrivals on that connection will not trigger an interrupt (although events on other connections might) until after the host CPU has specifically re-enabled interrupts by writing to an APIC per-connection command register. The CPU always re-enables interrupts for a connection after it has finished processing all outstanding packets that have arrived on that connection. Processing of packets belonging to latency-critical applications is always given higher priority than other types of packets. In fact, the higher the priority at which packets are processed, the lower the latency that such packets will experience. Clearly, a single interrupt would suffice for processing an entire burst of rapidly arriving packets. If a latency-critical packet arrives in the midst of a burst of packets belonging to a high bandwidth connection, the APIC would issue an interrupt which would result in the low latency packet getting immediate service, while service of the high bandwidth burst is postponed.

The amount of processing that will be done on packets before interrupts are re-enabled, and the scheduling policy that is used for software interrupts or threads, are both critical in determining how well interrupt demultiplexing performs. Some of the possibilities are outlined below:

- A kernel software interrupt routine can do all protocol processing and deposit received packets in a (socket) buffer for the application to pick up later, and then re-enable interrupts before returning.
- An application thread may be delegated the responsibility of re-enabling interrupts after it has finished processing any received packets. Protected I/O would provide secure access to the APIC register that is used to re-enable interrupts.
- Received packets may be delivered to an application using upcalls [5]. Interrupts can be re-enabled when the upcall returns. The upcall can be responsible for simple protocol processing, or it could be used to incorporate the contents of the packet into an on-going computation (à la Active Messages [15]).
- The scheduling policy used could be a simple Unix-like priority scheme, or a periodic real-time scheme such as that used with Real-Time Upcalls (RTUs) [18,19]. In the latter case, it is possible to entirely disable interrupts for a connection, and the application can periodically (in an RTU) check the receive queue for packet arrivals. This scheme has all the benefits of clocked interrupts [30], and has the advantage that low latency applications can also be supported efficiently.

Clearly, interrupt demultiplexing allows for a great deal of flexibility in the way in which protocol stacks are structured. A future paper will report on the relative benefits and trade-offs involved with implementing interrupt demultiplexing in a variety of environments.

4.3 Notifications

In an APIC interrupt service routine, it is important to be able to find out which connections have had some activity on them, so that only the corresponding application threads or software interrupts are scheduled. In order to avoid having to scan the descriptor queues for all open connections to test for the occurrence of events on a connection, the APIC has an on-chip queue, called the *notification queue*, which contains a list of "active" connection IDs. The CPU can read entries off the queue; once an entry has been read, it is removed from the queue. Only those connections that have had some activity since the time the queue was last emptied by the processor, are considered to be "active." Each notification on the queue contains, in addition to the connection ID, a notification code which encodes the current status of the connection. Notifications can be used either in an interrupt service routine, or in a clock interrupt to poll the APIC for activity quickly and efficiently.

5 Conclusions

In this paper, we have presented several techniques for high performance network interface design. These techniques enable the APIC to efficiently support both high throughput and latency-critical applications. Note that this is achieved without imposing a specific structure on protocol implementations or the operating system. Instead, the programming interface presented by the APIC to the software includes several mutually exclusive mechanisms that can be mixed and matched in different ways by a systems programmer in order to extract the maximum performance from the interface. Furthermore, the connection-oriented nature of the underlying network allows different compositions of these mechanisms to be used for different applications, thereby allowing for customizing of protocol implementations. For example, one application may choose to be interrupted by the interface on every packet arrival, while another may choose to be interrupted only once per burst of packets, and yet another may disable interrupts altogether and instead decide to poll the interface. Customization is further facilitated by the fact that the APIC includes support for user-space driver implementations.

In addition to interfacing to standard workstations, displays, and video cameras, the APIC is also going to be used to implement a high performance video storage server [2], and a video wall. The APIC driver software and VHDL programming work is currently in progress. We have also successfully built and tested MBus prototype cards to test the bus interface. We expect the first set of APIC chips to be back from fabrication in late 1996. A future paper will report on the measured performance of prototype APIC-based systems in our gigabit local-ATM testbed.

While all the mechanisms described in this paper were presented in the context of the APIC network interface, we are convinced that some of these mechanisms can be used to advantage in other types of I/O devices too. This conviction is based on the fact that I/O typically involves multiple sequential streams of data, with individual streams associated with different applications. This property is reminiscent of ATM connections. It should therefore be possible to use techniques like protected DMA, protected I/O, and interrupt demultiplexing for other types of I/O devices. In particular, protected DMA and protected I/O can be used to off-load the task of controlling hardware in the system to applications running in user space. This facilitates the construction of new kernel architectures such as the MIT Exokernel [16]. By exposing the hardware to applications (as opposed to hiding it under an abstraction), significant performance gains are possible. Our future research plans include exploring the feasibility of using protected DMA and protected I/O mechanisms for other types of devices, and in particular for mass storage devices like disks and RAIDs.

References

[1] Banks, D., and Prudence, M., "A High Performance Network Architecture for a PA-RISC Workstation," *IEEE JSAC*, Vol. 11 No. 2, Feb. 1993.

[2] Buddhikot, M.M., Parulkar, G.M., and Cox, J.R., "Design of a Large Scale Multimedia Server," *Journal of Computer Networks and ISDN Systems*, Dec. 1994.

[3] Clark, D.D., Jacobsen, V., Romkey, J., Salwen, H., "An Analysis of TCP Processing Overhead," *IEEE Communications Magazine*, Vol. 27, No. 6, 1989.

[4] Clark, D.D., and Tennenhouse, D.L., "Architectural Considerations for a New Generation of Protocols," *Proc. ACM SIGCOMM 90*, Aug. 1990.

[5] Clark, D.D., "The Structuring of Systems Using Upcalls," *Proc. 6th Symposium on Operating System Principles (SOSP)*, 1985.

[6] Cranor, C.D., and Parulkar, G.M., "Design of Universal Continuous Media I/O," *Proc. 5th Intl. Workshop on Network and Operating System Support for Digital Audio and Video*, Apr. 1995.

[7] Davie, B.S., "The Architecture and Implementation of a High-Speed Host Interface," *IEEE JSAC*, Vol. 11, No. 2, Feb. 1993.

[8] Dittia, Z.D., Cox, J.R., and Parulkar, G.M., "Design of the APIC: A High Performance ATM Host-Network Interface Chip," *Proc. IEEE INFOCOM 95*, April 1995.

[9] Druschel, P., Peterson, L., and Davie, B.S. "Experiences with a High-Speed Network Adaptor: A Software Perspective," *Proc. ACM SIGCOMM 94*, Sep. 1994.

[10] Druschel, P., and Peterson, L., "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility," *Proc. 14th Symposium on Operating System Principles (SOSP)*, Dec. 1993.

[11] Druschel, P., "Operating System Support for High-Speed Networking," *University of Arizona Ph.D. Dissertation CS-94-24*, Aug. 1994.

[12] Edwards, A., Watson, G., Lumley, J., Banks, D., Calamvokis, C., and Dalton, C., "User-Space Protocols Deliver High Performance to Applications on a Low-Cost Gb/s LAN," *Proc. ACM SIGCOMM 94*, Sep. 1994.

[13] Edwards, A., and Muir, S., "Experiences Implementing a High Performance TCP in User-Space," *Proc. ACM SIGCOMM 95*, Aug. 1995.

[14] Eicken, T. von, Basu, A., Buch, V., Vogels, W., "U-Net: A User-Level Network Interface for Parallel and Distributed Computing," *Proc. 15th ACM Symposium on Operating System Principles*, Dec. 1995.

[15] Eicken, T. von, Culler, D.E., Goldstein, S.C., and Schauser, K.E., "Active Messages: A Mechanism for Integrated Communication and Computation," *Proc. 19th ISCA*, May 1992.

[16] Engler, D.R., Kaashoek, M.F., and O'Toole, J., "Exokernel: An Operating System Architecture for Application-Level Resource Management," *Proc. 15th ACM Symposium on Operating System Principles*, Dec. 1995.

[17] Forin, A., Golub, D., and Bershad, B., "An I/O System for Mach 3.0," *Proc. USENIX Mach Symposium*, Nov. 1991.

[18] Gopalakrishnan, R., and Parulkar, G.M., "Bringing Real-time Scheduling Theory and Practice Closer for Multimedia Computing," *Proc. ACM SIGMETRICS 96*.

[19] Gopalakrishnan, R., and Parulkar, G.M., "Application Level Protocol Implementations to provide QoS Guarantees at Endsystems," *Proc. 9th Annual IEEE Workshop on Computer Communications*, Oct. 1994.

[20] Hayter, M.D., and McAuley, D.R., "The Desk Area Network," *Operating Systems Review*, Vol. 25, No. 4, Oct. 1991.

[21] Houh, H.H., Adam, J.F., Ismert, M., Lindblad, C.J., and Tennenhouse, D.L., "The VuNet Desk Area Network: Architecture, Implementation, and Experience," *IEEE JSAC* Vol. 13, No. 4, May 1995.

[22] Hutchinson, N.C., and Peterson, L.L., "The *x*-Kernel: An architecture for implementing network protocols," *IEEE Trans. Software Engineering*, Vol. 17, No. 1, Jan. 1991.

[23] Kanakia, H., and Cheriton, D.R., "The VMP Network Adapter Board (NAB): High Performance Network Communication for Multiprocessors," *Proc. ACM SIGCOMM 88*, Aug. 1988.

[24] Maeda, C., and Bershad, B. "Protocol Service Decomposition for High-Performance Networking," *Proc. 14th ACM Symposium on Operating System Principles*, Dec. 1993.

[25] Partridge, C., "Gigabit Networking," *Addison Wesley*, 1994.

[26] Pasquale, J., Anderson, E., and Muller, P.K., "Container Shipping: Operating System Support for I/O-Intensive Applications," *IEEE Computer*, Vol. 27, No. 3, Mar. 1994.

[27] Ramakrishnan, K.K. "Performance Considerations in Designing Network Interfaces," *IEEE JSAC* Vol. 11, No. 2, Feb. 1993.

[28] Sterbenz, J., and Parulkar, G.M., "Axon: Host-Network Interface Architecture for Gigabit Communication," *Protocols for High Speed Networking*, Elsevier (North Holland), 1991.

[29] Thekkath, C., Nguyen, T., Moy, E., and Lazowska, E. "Implementing Network Protocols at User Level," *Proc. ACM SIGCOMM '93*, Sep. 1993.

[30] Traw, C.B.S., and Smith, J.M., "Hardware/Software Organization of a High Performance ATM Host Interface," *IEEE JSAC* Vol. 11, No. 2, Feb. 1993.