WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

INTEGRATED HARDWARE/SOFTWARE DESIGN OF A HIGH PERFORMANCE NETWORK INTERFACE

by

Zubin Dittia

Prepared under the direction of Professor Gurudatta M. Parulkar

---

A dissertation presented to the Sever Institute of

Washington University in partial fulfillment

of the requirements for the degree of

DOCTOR OF SCIENCE

May 2001

Saint Louis, Missouri

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

ABSTRACT

INTEGRATED HARDWARE/SOFTWARE DESIGN OF A HIGH PERFORMANCE NETWORK INTERFACE

by Zubin Dittia

ADVISOR: Professor Gurudatta M. Parulkar

May, 2001

Saint Louis, Missouri

This thesis describes the design and implementation of a high performance network interface chip called the APIC (ATM Port Interconnect Controller). It also describes architectural enhancements to operating system (OS) software that are necessary to exploit some of the novel features that have been integrated into this chip.

High-performance network interface design has received significant interest from the research community in recent years because traditional design methodologies have not been successful in translating high network bandwidths and low network latencies to improved performance for applications. This can be attributed to several factors: in the past network interfaces have been designed without careful consideration of the operating system software environment in which they get used; main memory bandwidths have not scaled at the same rate as network bandwidths; and network interfaces and protocols have not been designed to support quality of service for applications. These are the problems addressed by this thesis, the objective being to develop new mechanisms which can result in significant improvements in application performance. In addition to incorporating these innovative features, the APIC design borrows proven and useful ideas from a number of commercial and research prototypes.

One of the ways in which the APIC addresses the memory bottleneck alluded to above is to function in a desk-area environment where different memories can be used to spread the load. The idea here is to dedicate one APIC chip and one memory bank to each high-bandwidth device in the system, thereby shedding the load from a host system's main memory. Several such APIC-memory-device combinations can be daisy chained to form a desk-area network with high bandwidth and low latency characteristics.

There are several well-known operating system overheads associated with in-kernel implementations of network interface device drivers. These include context switch latency, system call overhead, and interrupt overhead. It is possible to remove a number of these inefficiencies and allow for increased performance for end applications if the data path of the device driver can be implemented as a library in user-space. While this idea has been proposed in the recent past, the APIC introduces two new mechanisms, Protected DMA and Protected I/O, which together provide for an efficient method for the implementation of user-space drivers.

Another problem which plagues high-speed network adapters is called receive livelock; this term is used to describe the situation in which, under heavy load, an operating system servicing a device might end up spending all its time in the interrupt service routine, and no useful work gets done. The APIC introduces a novel concept called Interrupt Demultiplexing, which taken alone can alleviate the effects of interrupt livelock, but in conjunction with user-space drivers can solve the problem entirely.

Network interfaces, except for ATM interfaces, have traditionally not provided special mechanisms for supporting quality-of-service (QoS) guarantees. Even ATM interfaces have traditionally supported QoS only to a limited extent. By providing pacing support independently for large numbers of connections, the APIC is able to efficiently and reliably support QoS guarantees simultaneously for large numbers of multimedia streams. This can be especially useful in the context of large multimedia-on-demand servers. This feature was made possible through a novel pacer design which uses a hardware d-heap data structure.

The APIC has been successfully implemented in 0.35 micron technology, and is currently in use in several projects both at Washington University and elsewhere, as part of the NSF-sponsored gigabit kits project.

to my parents

Contents

Tables

Figures

Acknowledgements

First and foremost, I would like to thank my advisor Guru Parulkar, to whose help and encouragement I owe everything. This research would not have been possible without him, and I thank him for his patience in dealing with my sometimes difficult work habits, and for always providing the right mix of freedom, encouragement, and perspective that few advisors can provide for their students.

Second, I would like to thank Dr. Jerry Cox, who has been involved with this project from the onset, and who has made significant contributions to the design effort. He was like a second advisor to me in this undertaking. In particular, a large portion of the internal design of the APIC chip is due to him, including many of the details of implementation of the pacing algorithm described here.

I would also like to thank Rex Hill and Will Eatherton, both of whom were responsible for the VHDL coding effort for most of this chip. Although they joined the project late, both made significant contributions to the design as well as the implementation. In particular, I would like to acknowledge Rex for the paragonal memory layout used in the cell store, and for significant innovation in getting the pacing algorithm committed to silicon. I gave Rex many sleepless nights by insisting on a very clean design, and continually adding and removing features. He seldom complained, and always maintained good humor, which made the gruelling task bearable. I also acknowdge Will for his contributions in coding the chips internal clock domain, and the UTOPIA ports. He has left his mark on the chip in the form of an intelligent flow control algorithm which allows the chip to exert flow control signals over optical links.

I owe a lot to John DeHart in this undertaking; he was instrumental in getting the APIC device driver debugged and in a shape in which it could be distributed to Gigabit Kits participants. I admire his abilities and am grateful for his assistance in a very difficult situation, which arose when he had to undertake working with the APIC when all of the chip's developers, myself included, had left the University.

My thanks also go to Dr. Dave Richards, who spent considerable time and effort on implementing and debugging the MBus and PCI prototype cards, and on the UTOPIA port implementation. He also contributed his considerable expertise in tracking down and fixing APIC physical level problems once the chips were back from the foundry. His constant speculations about the schedule of the project were, I am sorry to say, right on mark.

I would also like to thank Margaret Flucke for undertaking the massive layout task for the chip, and Tom Chaney and Fred Rosenberger for spending long hours poring over a printout of the chip's layout to ensure signal integrity and dealing with power and clock distribution issues.

I would like to thank Dr. Jon Turner, who contributed to the project with many very useful comments and suggestions throughout the period of the project. The new and improved pacing algorithm described in Chapter 3 is due to him.

I would like to thank Andy Fingerhut, Anshul Kantawala, and Chuck Cranor for their support and help in software related issues.

I would like to thank everyone at Growth Networks, and in particular Dan Lenoski, for agreeing to let me take time off from work to finish my thesis.

Last, but not the least, I would like to thank all my friends at CCRC and ARL, whose companionship added the necessary ingredients of fun and liveliness without which life has no meaning: Ana, Anshul, Anurag, Apostolos, Brad, Cheenu, Christos, Chuck, Dan, Daphne, Diana, Girish, Gopal, Geppo, Hari, Manamohan, Marcel, Maurizio, Milind, Mini, Nimi, Paula, Penny, Samphel, Sherlia, Shree, and Vykky.

Chapter 1

Introduction

Although gigabit network design has seen major advances in the last decade, the ability of end applications to fully exploit the capacity of these networks has been severely limited. From a throughput standpoint, there is usually a gross disparity between raw network bandwidths and the maximum effective throughput that can be achieved by end-applications. This becomes a very important issue in light of the emergence of several high bandwidth multimedia applications. In terms of latency, the networking subsystem in an end host often adds enough to the end-to-end latency that it often exceeds (or can even be several times) the network propagation and queueing delay. This is especially true in local area networks, where latency is a criticial measure of performance for interactive and distributed computing applications.

The bottlenecks in an end-host that prevent applications from exploiting all or most of the available network capacity manifest themselves in both the hardware and software realms. From a hardware perspective, limitations have traditionally been imposed by the peak system memory and bus bandwidths, and from poor network interface design practices. From a software viewpoint, the overhead of operations such as data copying, checksumming, servicing of interrupts, and context switching are typically responsible for poor performance. Several researchers have recognized the source of these bottlenecks, and identified mechanisms that are at least partially effective in overcoming some of the handicaps. This is evidenced by the considerable litearature that has been published in recent years on network interface design, and on the structuring of protocols in operating systems [2,5,6,7,10,12,14,28,32,39,41].

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 network interface. This document describes the design and implementation of this network interface chip (NIC), which is called the APIC (ATM Port Interconnect Controller).

1.1. Goals

The research presented in this document attempts to answer the following fundamental questions:

- Given current chip technology, is it possible to build an inexpensive network interface that can deliver all or most of the network capacity of a gigabit network to end applications? Can this be done in a way that assures some degree of quality of service (QoS) to these applications?

- Can this same network interface also efficiently support latency sensitive applications that have to coexist with high bandwidth applications?

One of the non-goals of this research is the preservation of the protocol stack software architecture currently used in most commodity operating systems. In other words, we assumed that we had the freedom to change this software architecture in order to best achieve our goal of higher performance.

While this thesis formulates several new mechanisms, not all of them have been validated through experimentation. The reason for this was to limit the scope of the work — design and implementation of a network interface chip is a complex task requiring many man-years of effort, and validation of all of the chip’s features would have required considerable additional investment of time in programming and testing the required software. It is hoped that most of these chip’s features will be exercised by third parties using the chip, both at Washington University and elsewhere.

1.2. Features of the APIC Chip

The prototype APIC design is our attempt at providing an answer to the above questions. Targeted for ATM networks, it is capable of supporting a full duplex link rate of 1.2 Gb/s. Since one

of our goals was to keep (a production version of) the NIC inexpensive, the design is geared towards implementation on a single application-specific integrated circuit (ASIC), with no external memory required for buffering or table handling. Some of the salient features of the APIC design include:

- The ability to act as a building block for System-Area Networks (SAN) and Desk-Area Networks (DAN) [26,27].
- Remote controllability.
- Multipoint and loopback support.
- DMA modes that are designed to reduce the number of data copies to zero (a zero copy architecture).
- Protected DMA and Protected I/O, which are novel techniques that allow buffer management and chip control operations to reside in user-level processes, without compromising OS protection mechanisms.
- Efficient mechanisms designed to reduce interrupt frequency and interrupt service overhead, while retaining the ability to be able to react quickly to latency sensitive events.
- Support for multiple traffic classes for QoS.

The APIC has been coded in VHDL, and passed a detailed co-verification with a C++ simulation of the chip's behavioral model. It has been successfully implemented in 0.35 micron technology, and is currently in use in several research projects at Washington University, as well as at several other universities and research labs that are part of the NSF-sponsored gigabit kits initiative. The software drivers for the chip have been implemented in the NetBSD operating system kernel, and are currently in the process of being ported to Linux.

1.3. Contributions

The primary contributions of this research are:

- Design of a single chip high performance gigabit ATM host network interface with dual ATM ports, that can function both in a standalone environment as well as with other identical chips in a desk-area network environment.

- **Protected DMA and Protected I/O**: A unique meld of hardware and software that enables user-space protocols and applications to efficiently interface directly to the NIC for data movement, without any OS kernel involvement, and without compromising OS security mechanisms.

- **Interrupt Demultiplexing**: A feature that would permit high bandwidth and low latency applications to coexist without the adverse interactions that are common in today's implementations. Additionally, this feature can help reduce the effects of interrupt receive livelock, which is a problem that plagues almost all high performance network interfaces in use today. Taken together with user-space protocol implementations using Protected DMA and Protected I/O, interrupt demultiplexing can entirely eliminate the receive livelock problem.

- **d-Heap Pacing**: In order to support QoS guarantees to individual multimedia streams, the APIC supports paced connections. Compared to traditional network interfaces, the APIC can support independent pacing for very large numbers of connections; this has been made possible through a novel architecture based on hardware implementation of d-heaps.

The thesis formulates the above mechanisms and explains why we believe they will function as described. However, as mentioned earlier, we have chosen not to experimentally validate all of these mechanisms in order to limit the scope of the work; it is hoped that the many users of the chip will invest the time to write the necessary drivers to program those features not exercised by the default driver, and publish results validating many of these mechanisms.

While the APIC targets ATM as a network technology, it is important to note that many of the contributions made in this thesis are applicable in the Internet context too. In particular, it is feasible to implement Ethernet adapters that make use of many of the same techniques, provided the adapter contains a programmable packet classification engine that can classify packets based on Internet port numbers; in that case, an Internet flow would take the place of an ATM connection.

1.4. Outline

The rest of the thesis is organzied as follows: Chapter 2 provides background and motivation for the problem addressed herein. Related research in the area of gigabit network interfaces and desk-area networks is covered in Chapter 3. Chapter 4 lists the main contributions made in this thesis, with reference to the related work presented in Chapter 3. Chapter 5 describes the APIC architecture, while Chapter 6 lays out the chip's internal design. Chapter 7 presents a software architecture and describes implemented pieces of this architecture. Chapter 8 presents results from several experiments performed on a working prototype of the chip. Finally, Chapter 9 concludes with a summary of the contributions and ideas for future research.

Chapter 2

Background and Motivation

Figure 2.1 shows the typical architecture of a modern workstation or server. There are one or more processor modules which interface to the system's main memory and to a primary I/O bus via a processor-memory interconnect. The latter could be implemented as a bus, but in modern machines, it is usually a general switch chipset with internal buffering. A processor module consists of a processor, some cache, and a memory management unit (MMU) which usually contains a translation lookaside buffer (TLB). Most I/O devices interface to the primary I/O bus, either directly or through a secondary I/O bus which in turn is connected to the primary I/O bus using a bus adapter. In the figure, we do not show these secondary I/O buses, because for the most part we are interested in high bandwidth devices, including network interfaces, which interface directly to the primary I/O bus. A good example of a primary I/O bus is the PCI (Peripheral Component Interconnect) bus, which originated as a standard for PCs, but has subsequently gained widespread acceptance in the server and workstation markets too.

There are a few important points to note about this architecture in the context of high performance network interfacing. There are two mechanisms by which devices on the I/O bus can communicate with software running on the host processor(s). In the first technique, the software can use processor instructions to read or write data directly from or to the device. This mechanism, called "Programmed I/O", works by requiring the device to make some of its internal memory and registers available to the host processor; usually, this is achieved by mapping the device into an unused portion of the same address space that is used by the processor to access main memory. In other words, by issuing load and store instructions with these special addresses, the processor can

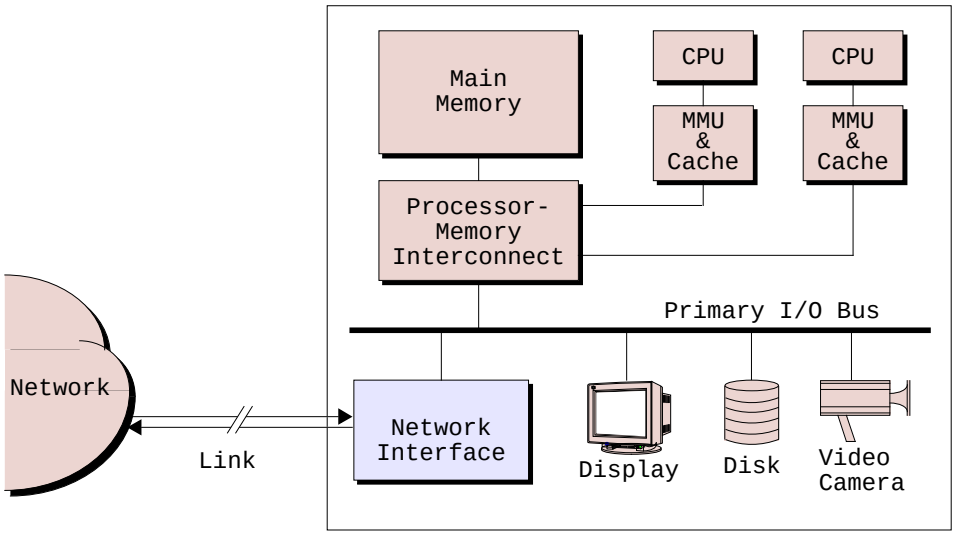

Figure 2.1: Typical Host Architecture

read or write to the registers (or memory) resident on the device. This kind of access method is also commonly referred to as “Memory-mapped I/O”.

The second method used to communicate information between an I/O device and software running on the host processor is called DMA (Direct Memory Access). Here, all communication passes through special “shared” data structures that are allocated in the system’s main memory. What makes these structures shared is the fact that they can be read from or written to by both the processor and the device.

In practice, both methods are used in most devices. Programmed I/O provides a synchronous access interface to the device, while DMA provides an asynchronous interface. Usually, DMA is the preferred technique when large amounts of data are to be transferred to or from memory, because it does not tie up the processor for the duration of the transfer. Programmed I/O is useful when small amounts of data are to be transferred, or when the interaction needs to be synchronous. Usually, control interactions with the device are implemented using programmed I/O.

2.1. DMA Subsystems

Most modern NICs use DMA as the preferred technique for moving packets to or from main memory. The DMA subsystem is the part of the NIC that is responsible for all DMA related actions.

One of the required features of the DMA subsystem in a NIC is “scatter-gather DMA”. This refers to the ability to be able to handle packets which are fragmented in memory. Such fragmentation occurs because packets are constructed by network protocols so that they usually reside in different regions of the memory. Additionally, a single packet may be broken up into smaller pieces that reside in different memory locations. This happens, for example, if the protocol which constructed the packet used separate buffers for the packet header and packet data.

2.1.1. Why is the DMA Subsystem so important?

The design of the DMA subsystem in a NIC is made complicated by the fact that there are several trade-offs to consider. One of these has to do with choosing between better memory utilization and improved performance. Another, which is more of a software issue but has a major impact on the DMA subsystem, has to do with the selection of an appropriate API (application programming interface) for applications: usually, APIs that are more convenient and easier to use result in worse performance. A poorly designed DMA subsystem which does not take into account these trade-offs and software interactions can result in phenomenally bad overall performance.

One of the most important issues to consider in the design of a DMA subsystem is the number of “data touch” operations. Any time that packet data is read from or written to main memory, it is considered to have been “touched”. A design should try to minimize data touches, because of the large negative impact that they can have on performance. The reason for this is that main memory bandwidth has not kept pace with increases in processor performance, so that any reduction in the number of times memory is accessed for a given piece of data can result in large performance gains. To see this, consider the fact that the main memory bandwidth of a typical PC with a 64-bit memory bus is about 2.2 Gb/s for reads and 1.4 Gb/s for writes (these are numbers from a PIII/450MHz PC). For simplicity of analysis, let us assume that the memory bandwidth is 1.8 Gb/s for both reads and writes. If there were k data touch operations, then each word of data from the network is effectively accessed k times, which means that from the viewpoint of the consumer of the data, the effective maximum throughput that can be achieved is only $(1.8/k)$ Gb/s. For two data touches, this number goes down to 900 Mb/s, for three to 600 Mb/s, and so on. Given the very high raw bandwidths supported by our target network (1.2 Gb/s for the APIC), it is easy to see that for anything more than a single data touch, we may not be able to exploit all of the network’s capacity. And with each

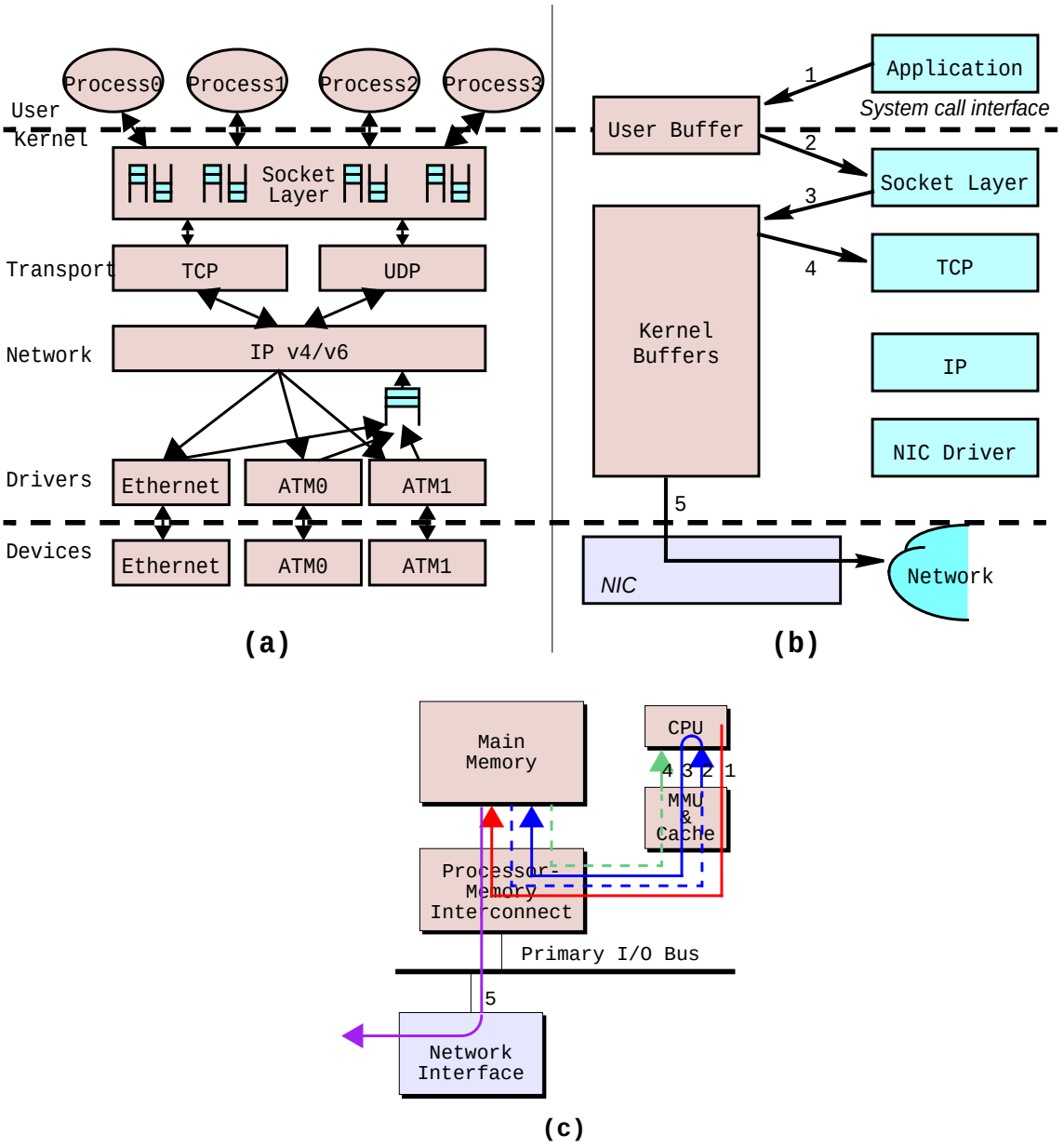

Figure 2.2: Data Touch Overhead in a Typical Protocol Stack

additional data touch, the achievable throughput drops rapidly. Clearly, it is beneficial to keep the number of data touch operations to a bare minimum.

It is not unrealistic to see numbers as high as five data touches; in fact, many modern TCP/IP protocol stacks incur at least that many data touches. To see why this is so, consider Figure 2.2. Part (a) of the figure shows the traditional protocol stack architecture used in most of today's

operating systems (the example shown is for BSD Unix). Part (b) of the same figure shows what happens when an application wants to transmit data. Following the numbered events in the figure, we have:

1. The application, which runs as a process in user space, first generates the data to be sent and writes it to its own private buffer in user-space, following which it makes a system call to the socket layer to transmit the data;

2, 3. The socket layer copies the data from the user buffer into a set of kernel buffers that are used to hold packets.

4. TCP reads the data so that it can compute the checksum which has to be inserted in the packet header.

5. The network interface reads the data from the kernel buffer and transmits it.

Figure 2.2(c) shows what happens in hardware for these five data touch operations. Arrows represent movement of data corresponding to the five data touch operations. Notice that some of the lines are dashed; a dashed line represents the fact that the data movement indicated by the line may not actually occur if the corresponding data is in cache. For example, if the system call to transmit a packet is made soon after the application has generated or otherwise accessed the data, then the read portion of the copy ("2") from user to kernel buffer would with high likelihood be satisfied from cache. Similarly, if TCP decides to transmit the data immediately or shortly after the system call to send it is issued, then the checksum computation step "4" would be satisfied from cache. Thus, in the best case there are three data touches (to memory) for any given piece of data; in the worst case, there are five.

Using our earlier example, with 1.8 Gb/s of memory bandwidth, with five data touches we would be able to achieve a throughput of only (1.8/5) Gb/s, or 360 Mb/s.

Figure 2.2 only showed the data touches for an outgoing packet; a similar but reversed sequence applies on the incoming side.

Data touches are not bad just for throughput performance; they also adversely affect packet latency, because of the extra time the processor spends copying data. Clearly, it is important to be able to minimize data touch operations.

2.1.2. NIC design choices that affect the DMA Subsystem

In this section, we look at some architectural choices available to NIC designers, and how they affect the DMA subsystem.

Cut-through vs. Store-and-forward

Network adapters fall into two main categories: cut-through, and store- and-forward. In a cut-through adapter, it is possible that the transmission of a frame can begin even before the entire frame has been read out of main memory. On the receiving end, a cut-through adapter can store part of a frame in main memory before the entire frame has been received. In a store-and-forward adapter, an entire frame needs to be read from main memory before the adapter will begin transmitting it. Also, a store-and- forward adapter will not write a frame into main memory until it has finished receiving the entire frame. Cut-through adapters have the advantage of lower delay, and of not requiring local memory on the network interface card (NIC) to store frames. They have the disadvantage that they might end up transmitting partial frames if there is an error, and of reporting receipt of partial or corrupted frames (which leaves the job of cleaning up to the software).

One of the possible techniques that can be employed to reduce the number of data touches is to move the transport layer checksum computation function into hardware on the NIC. This is easy to do on a store-and-forward NIC (the checksum can be computed while moving data between the NIC and main memory). But in cut-through adapters, it is not possible to compute and insert a checksum in the header of an outgoing packet, because the header may already have been transmitted by the time we finish computing the checksum. This means that either we would be forced to use a transport protocol with trailer checksumming, or incur the overhead of computing the checksum in software. Since the Internet transport protocols TCP and UDP both use header checksums, we have to incur the overhead of computing the checksum in software for these protocols, at least on the sending side. However, there is a trick which is often quoted (but seldom implemented) that can be used to allow the checksum computation for an outgoing packet to proceed without the overhead of a data touch. Referring to Figure 2.2, if the processor were to compute the checksum while copying data from the user buffer to the kernel buffer (steps 2 and 3), then no extra data touches would be involved. This scheme cannot easily be used on the receiving end, because the outcome of the checksum verification for an incoming packet needs to be known well before the copy from kernel

to user space takes place. So for incoming packets, the only way to avoid a data touch for checksum computation is to implement it on the NIC in hardware. Note that in this case, header checksums are not a problem, because although a cut-through adapter cannot usually verify the correctness of such a frame by itself, it can provide the computed checksum over the frame to the software and leave the job of verification of the checksum (i.e., comparing it to the value in the packet header) to the software.

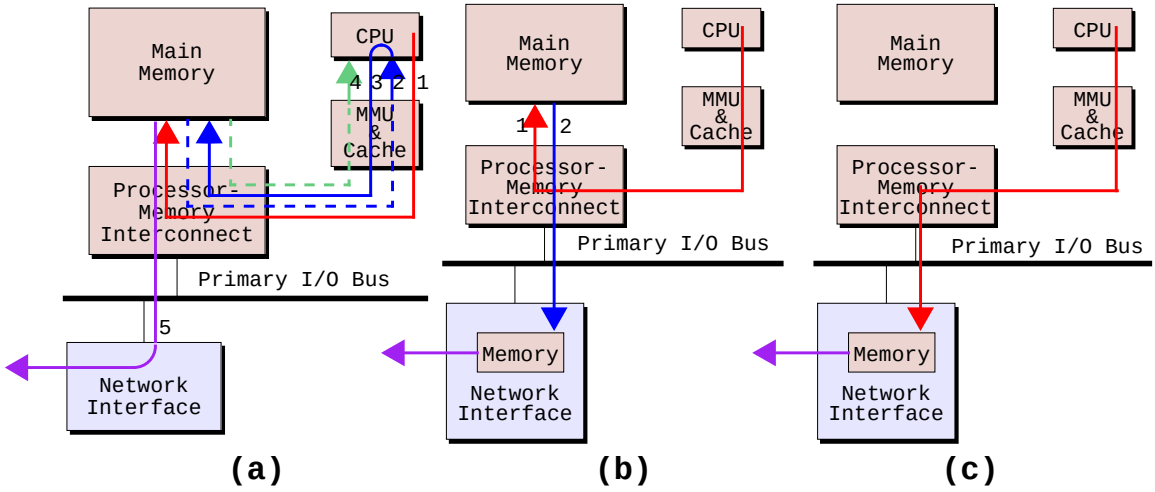

Figure 2.3: Impact of On-board Memory on Data Touches

On-board memory or not?

Another design choice that affects the DMA subsystem is whether or not the NIC has an on-board memory that can be used as a staging buffer for packets. Figure 2.3 shows how the number of data touches can be reduced if a NIC contains on-board memory. Figure 2.3(a) is a repeat of Figure 2.2(c), used for comparisons. In Figure 2.3(b), the kernel buffers (see Figure 2.2(a,b)) are allocated from memory on the NIC, and the kernel moves data from user buffers into these on-board buffers using either programmed I/O or DMA. If programmed I/O is used, the checksum can be computed during the copy loop, thereby resulting in only two data touch operations (as shown in the figure). If DMA is used to move data between main memory and the network interface, then the additional data touch for checksumming can still be avoided if the network interface computes the checksum while performing the DMA operation. Figure 2.3(c) shows a third alternative which involves no main memory accesses at all (zero data touches): the user buffers are allocated from the NIC's on-board memory (we assume on-board checksum support). Although this sounds very

attractive, it is usually not a very practical approach because it would require large amounts of memory on the NIC, and the application would have to know in advance where it should write its data to (i.e., to which NIC's buffer).

What if we didn't have on-board memory? As mentioned earlier, the APIC does not have any on-board memory to keep cost down. What we would like to see happen is for the data to move directly from the application's user-space buffer to the network interface, which can then directly transmit the data. Similarly, on receive, we would like to be able to receive data directly into the user buffer. As we shall see later, the APIC's Protected DMA and Protected I/O features do enable this kind of data movement.

On-board processor or not?

A number of NICs have on-board processors and firmware that is used to perform various NIC related tasks. This has both advantages and drawbacks; often, on-board processors drive up the cost of the NIC, but they provide more flexibility in the sense that more features can be added as necessary without a lot of work. Usually with such NICs, the method by which the driver for the NIC interfaces to the rest of the OS remains the same; however, some researchers have argued for moving portions of the protocol stack onto the NIC. Increasingly, this approach has gained disfavor because it requires very close interaction between the OS on the host and the software running on the NIC's processor.

2.2. Architectural Impact on Latency

So far, we have been focusing on the overhead of data touch operations resulting from data copying and checksumming in the protocol stack. Since data touches reduce the effective available memory bandwidth, they have an adverse effect on throughput. However, because of the extra time involved in copying and checksumming, they also affect end-to-end latency. The relative impact of these operations on latency is quite small however, except in a local-area network (LAN) where the network propagation delay can be of the order of a few tens of microseconds. Several distributed computing applications, such as distributed interactive simulations, Network File Service (NFS), remote procedure call (RPC), etc. could benefit from a very low end-to-end latency. For such latency sensitive applications, it makes sense to not only minimize the impact of data touch

operations, but also operations such as system calls, interrupts, and context switching, all of which have an adverse impact on latency. In modern operating systems, system calls and interrupt overheads can be of the order of a few to several tens of microseconds, which is comparable to the network delay in a LAN. As we shall see later, Protected DMA and Protected I/O achieve the goal of direct movement of data to and from user space without kernel involvement, thereby eliminating system call latency. That leaves the overhead of interrupts. Usually, interrupt overhead cannot be avoided for latency critical applications unless the device is continuously polled. The latter is not a practical alternative, so the cost of fielding an interrupt corresponding to a latency sensitive event, such as packet arrival, cannot be avoided.

2.2.1. Impact of interrupts on latency

Interrupts pose another problem for latency sensitive applications, which manifests itself in the presence of other high bandwidth applications with which it may have to coexist. High bandwidth applications typically have a high packet arrival rate, and therefore may get interrupted very often. To reduce the processor overhead of having to service interrupts very frequently, several approaches have been suggested, all of which try to reduce the frequency of interrupts by processing multiple interrupt events with only a single interrupt. In a mixed environment with both latency-critical and high-bandwidth applications, this can have the negative side-effect of significantly increasing the latency seen by delay-sensitive applications. This is because these applications do not get timely notification of packet arrival events, since these events get processed infrequently, and in batches along with large numbers of other events corresponding to other high bandwidth applications. In such mixed environments, it would be beneficial to have some way to allow latency-critical applications to still have their interrupts serviced in a timely manner, without adversely impacting high-bandwidth applications (which are usually *not* latency-sensitive). As we will see later, the APIC's interrupt demultiplexing technique achieves this objective.

2.3. Receive Livelock

In an interrupt driven system, interrupt service takes priority over all other activity. As mentioned earlier, if packets arrive too fast, the system will spend all of its time processing receiver interrupts. It will therefore have no resources left to support delivery of the arriving packets to applications, and no resources to allow the application to consume the received data. The useful

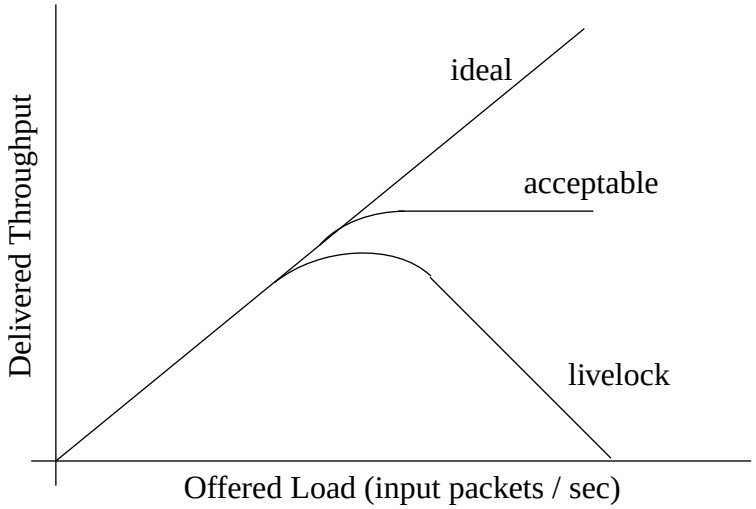

Figure 2.4: Illustration of Receive Livelock

throughput of the system will drop to zero. This condition is referred to as *receive livelock* [38]: a state of the system where no useful progress is being made, because the processor is entirely consumed with processing receiver interrupts.

Figure 2.4 (adapted from [38]) demonstrates the possible behavior of throughput as a function of offered input load. Ideally, no matter what the packet arrival rate, every incoming packet is processed. However, all practical systems have finite capacity, and cannot receive and process packets beyond a maximum rate (determined by the processor speed and the applicatiuon-dependent cost of receiving and processing a packet). Given this practical constraint, we would like the packet packet processing rate to remain pegged at this maximum, even when the rate of arrival of packets is higher than the maximum. However, because of the receive livelock effect, the system throughput may drop off to zero as the packet arrival rate increases, as shown in the figure. Note that this is a direct result of the fact that interrupt service takes priority over packet processing.

Related to the problem of receive livelock is the problem of starvation of transmits under overload. In most systems, packet transmission is done at a lower priority than packet reception, on the assumption that this will cause lower packet loss. However, under heavy receive traffic conditions, the system may spend most of its time servicing interrupts for incoming packets, so that packets

waiting to be transmitted will not get an opportunity for service. This is called transmit starvation [38].

All proposed techniques to alleviate livelock rely on somehow matching the interrupt rate to the rate of consumption of packets, but the problem remains one that dogs all modern adapters. We propose a software technique that can delay the onset of receive livelock, thereby allowing for higher throughput before the onset of livelock. In the context of the APIC, we will also show that by combining Protected DMA and Protected I/O with Interrupt Demultiplexing, we can completely eliminate the problem of receive livelock.

2.4. QoS Support in Network Interfaces

Most conventional network interfaces, barring ATM interfaces, have usually not included any special support for provisioning quality of service (QoS) for different applications. In other words, they have treated all traffic in a best-effort manner. In the case of ATM networks however, the network attempts to provision different grades of service to different applications. It accomplishes this by requiring applications to contract with the network on how much traffic load they would like the network to carry. The application for its part has to ensure that it keeps to its portion of the contract, which usually means some constraints need to be imposed on the rate at which traffic is injected into the network by the application. Such constraints can be implemented in software on the end-stations, but this can provide only a very coarse-grained level of control on the traffic bursts that are fed into the network. By incorporating special QoS support on a network interface, it is possible to offload the packet scheduling work from the processor, and also to achieve much finer grain control over the profile of injected traffic. ATM network interfaces typically provide this level of QoS support in the form of *paced channels*. Each application flow is mapped into a paced channel, and the network interface ensures that all traffic from the channel is injected into the network at a pre-configured *pacing rate*.

In a sense, the pacing rate is the “peak rate” for an application. However, this should not be taken to mean that only constant bit-rate (CBR) applications are supported by this paradigm: Figure 2.5 shows that a variable bit-rate (VBR) application can enqueue bursts of data (for example, video frames) that are to be transmitted on a paced channel with a fixed pacing rate. Each individual burst will be transmitted at the pacing rate, but when the network interface runs out of data

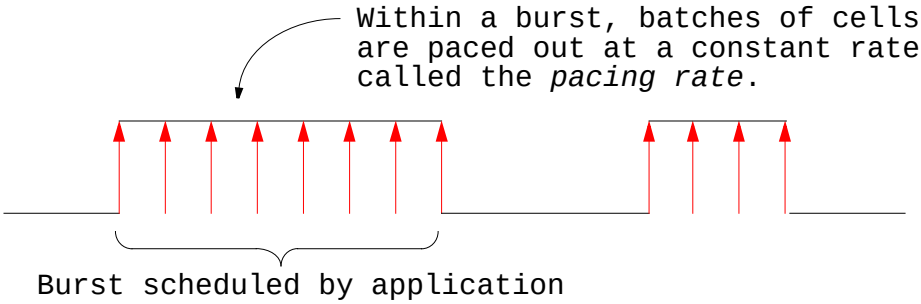

Figure 2.5: Behavior of a paced channel

to transmit at the end of a burst, no more data will be injected into the network until the next burst is enqueued for transmission by the application software. Since there are idle periods between bursts, the resulting traffic profile fits the description of a VBR stream.

How should the pacing rate for a channel be chosen? If the channel is the source of data for a switched VC (SVC) that was setup using some sort of ATM signalling mechanism, then the quality-of-service (QoS) parameters negotiated with the network during the connection setup phase will usually place an upper bound on the maximum rate at which cells on that connection can enter the network. This "peak rate" becomes the pacing rate for the connection, and it is up to the software to ensure that the traffic bursts enqueued on the channel are compliant with other negotiated QoS parameters. For a CBR source, the pacing rate should be set equal to the "constant" bit rate of the source, and the software does not need to do anything more — the APIC will take care of transmitting data from the channel at the correct rate.

What can we do if there is no contract with the network that tells us how to set the pacing rate? This happens with an ATM permanent virtual circuit (PVC), for example. One alternative is to use the network interface in best-effort mode, and rely on higher layer congestion control techniques (eg., TCP congestion control). Another alternative is to vary the pacing rate in response to feedback mechanism from the network. The feedback can be as simple as detection of a packet loss, or it can be more complex — for example, the ATM available bit-rate (ABR) mechanism uses explicit feedback from switches within the network.

Most existing ATM network interfaces do not support more than a small number of paced channels. It is assumed that the software will map multiple application flows into one of the channels.

This becomes difficult to do when there are many applications with varying requirements. The reason that network interfaces have not traditionally supported larger numbers of paced channels is because the problem of hardware scheduling for many different channels with varying pacing rates is non-trivial. The APIC design incorporates a novel mechanism using hardware d-heaps to address this issue. This approach can scale to support many thousands of paced channels, and although a better solution to this problem has subsequently been proposed, the APIC's approach remains a viable, if somewhat expensive, alternative.

2.5. User-space Protocol Implementations

Traditionally, protocol implementations have been OS kernel-resident. In terms of throughput, this model works reasonably well if the frequency of data path operations at the user-kernel boundary is low. This is the case when an entirely kernel-resident implementation of a byte-stream transport protocol such as TCP is being used. This is because send and receive data path operations at the user-kernel boundary can work with large data buffers corresponding to application data units (ADUs), and therefore need not be very frequent. In Unix for example, these operations are implemented using read and write system calls to a socket.

If communication at the user-kernel boundary is in terms of individual packets (also called protocol data units, or PDUs), then the overhead of requiring a system call per packet can result in very poor throughput. There are two important scenarios in which this is important. The first is in the context of datagram applications, for example those using UDP. The second is in the context of user-space library implementations of protocol stacks, which are desirable for several reasons:

- They enable implementation of smaller and more efficient OS kernels (microkernels);
- User-space protocol implementations are easier to program, debug, distribute, upgrade, and maintain;
- Applications can customize protocols depending on their specific requirements;
- The process scheduling policies of the OS kernel trivially cover both application and protocol processing, which makes the task of QoS enforcement within the end-system easier.

Recently, there have been several research efforts [17,18,24,33,40] that have attempted to make user-level library implementations of protocol stacks, in such a way that they can be linked with application programs. Most of these efforts have relied on traditional network interfaces, which necessitate that the device driver be kernel-resident. Thus, even though protocols would be in user-space, transmission and reception of data would require system calls to the kernel. System calls can be expensive; a null system call can take on the order of tens of microseconds in modern 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-senstive LAN applications. But the high cost of system calls can also result in poorer throughput if the protocols have not been implemented carefully. Most researchers who have explored library implementations of protocols have given serious attention to amortizing the overhead of a system call over multiple operations (such as send/receive of a packet) by batching them into a single system call. This places an unnecessary burden on protocol coders, and makes the code less portable and more dependent on the underlying operating system. Furthermore, it exacerbates the aforementioned latency problem.

To solve these problems, system calls and the kernel should be removed from the critical data path. In other words, the device driver for the network interface should be implementable as a library in user-space. While this can be done easily if there is only a single application process, it becomes much more difficult when there are multiple processes running on the host-processor which need to access the network simultaneously. The problem arises because conventional network interfaces are controlled by a device driver that operates in a system-wide trusted context (usually, the kernel). With multiple processes controlling the network interface directly, the issue of maintaining the operating system's protection policies between processes arises. Resolving this issue requires cooperation from the network interface in the form of special support for user-level protocols.

Figure 2.6 illustrates the differences between the traditional kernel-resident control model, and the user-space control model. The figure distinguishes between two kinds of control operations: those on the data path, and those on the control path. Data path operations are executed every time some data is sent or received, while control path operations are relatively rare, and usually need to be executed only once or a few times in a connection's lifetime. This distinction is important, because it means that data path operations are executed very frequently and therefore should be

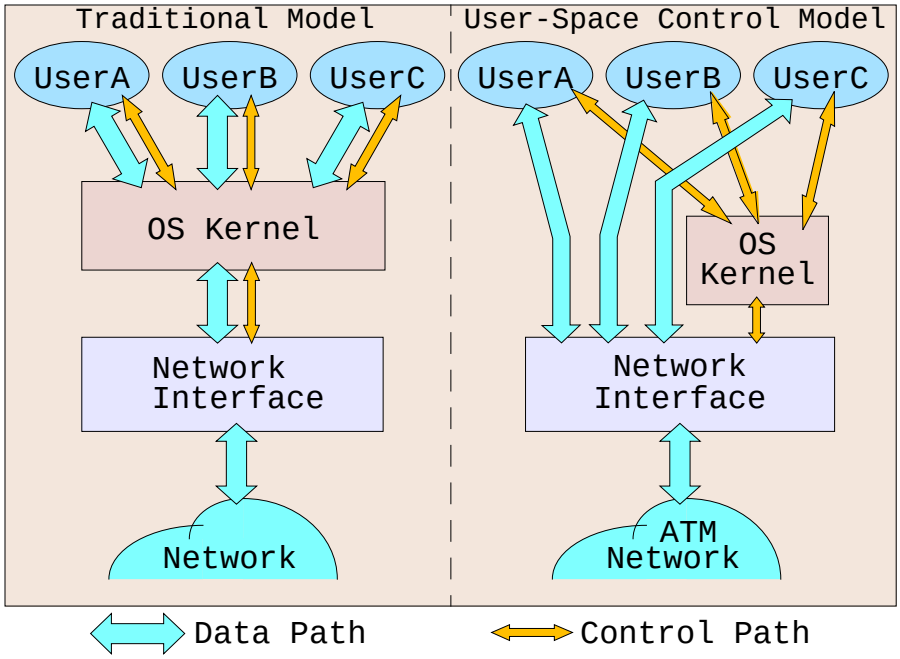

Figure 2.6: Traditional versus User-space Control Model

optimized, whereas control path operations are relatively rare, so there is not much to be gained by optimizing them. Examples of data path operations include: queueing and dequeueing data buffers on queues corresponding to DMA channels, informing the network interface that there are new buffers in these queues, changing the pacing rate of a channel in response to application demands, etc. Examples of control path operations include: setting up DMA channels and connections, adding endpoints to a connection, etc. Note that some operations, such as setting the pacing rate, can be considered to be control path operations if they are executed only once (or rarely) as part of an initialization sequence, and can be considered to be data path operations if repeated executions are necessary every time some data has to be sent or received.

Returning to the figure, we see that in the traditional model, when a user-space process wants to use a network interface device, both control and data path operations need to pass through (and be blessed by) the OS kernel. In the user-space control model, only control path operations need to pass through (and be blessed by) the OS kernel; the frequently occuring data path operations do not need any kernel intervention (and therefore are more efficient).

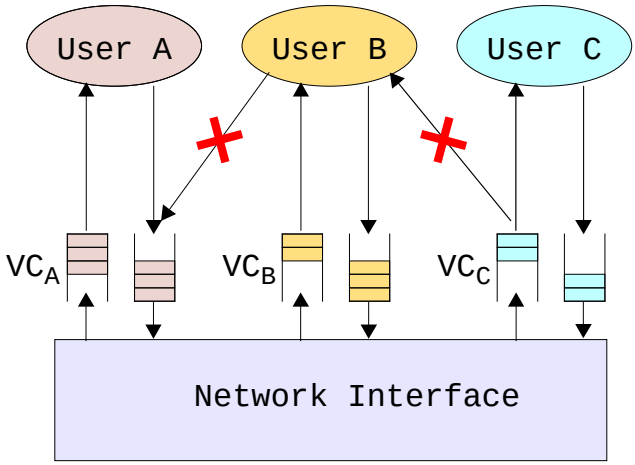

Figure 2.7: Protection concerns in the user-space control model

Since we have removed the kernel from the data path, the network interface has to take over the job of "blessing" all data path operations. Here, "blessing" means ensuring that the protection policies imposed by the kernel are not violated. In the context of an ATM network, these protection policies manifest themselves as sets of operations that are considered to be "legal" for the "owner" of an ATM connection. A user process is said to be the owner of a connection if it holds an OS granted capability for that connection. Usually, the process that is responsible for opening a connection becomes the owner of the connection. Referring to Figure 2.7, it should be illegal for user process B to send data on connection VC_A which is owned by user process A. Similarly, it should be illegal for user process B to be able to receive data arriving on VC_C, which is owned by user process C. As another example, one user process should not be able to change the pacing rate of a connection for which it does not hold a capability (i.e., it is not the owner). In the traditional model, such checks were performed in software by the kernel. With the user-space model, they become the responsibility of the network interface.

Note that in the context of the above discussion, a similar concept can be applied to TCP or other transport level packet flows in an IP network, in the place of ATM connections.

It is important to note that the user-space control model implies a different software structure. In particular, large portions of the network interface device driver can be migrated to user-space. We will henceforth refer to this part of the driver, which runs in an untrusted context and is typically implemented as a library that can be linked with the application, as the user-space driver. The

trusted portion of the driver typically resides in the kernel, and is responsible for all control operations, for dictating protection policies that will apply to user-space drivers, and for fielding device interrupts — we will henceforth refer to this part of the driver as the kernel driver.

As we noted above, the user-space control model helps in terms of both throughput and latency. Furthermore, it enables efficient implementation of application-customizable user-space protocols. The user-space control model is a relatively new paradigm, and as we shall see, is supported by the APIC using two new mechanisms: Protected I/O and Protected DMA.

Chapter 3

Related Work

Several research groups have attempted to design and implement high-speed host-network interfaces over the past few years. The APIC design builds upon the success of these efforts by adopting a number of useful features and improving or improvising them for our target environment.

3.1. Related work in network interface design

One of the earliest efforts in high-speed network interface design was the network adapter board (NAB) [32], which was especially designed to support the VMTP transport protocol [4]. As shown

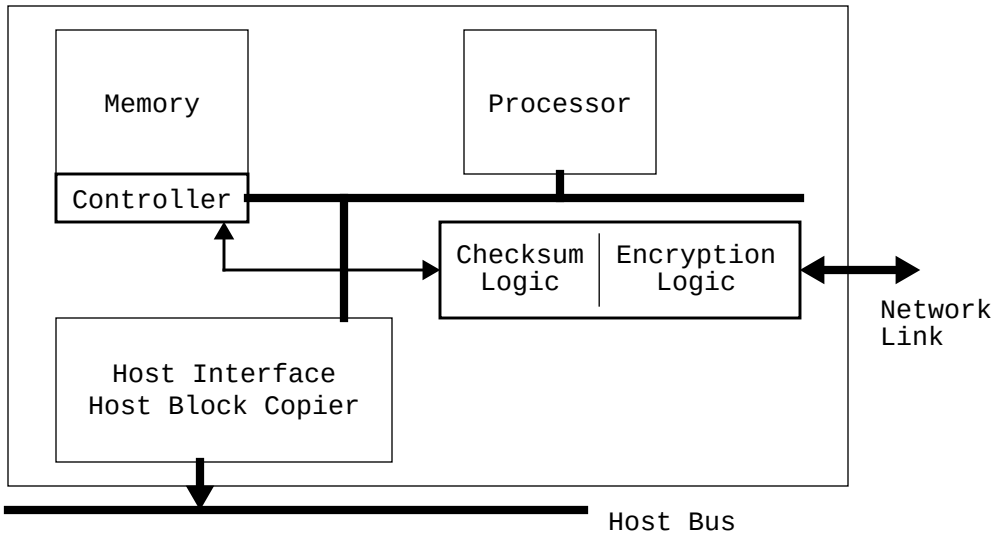

Figure 3.1: Network Adapter Board (NAB) Architecture

in Figure 3.1, the board includes a microprocessor and memory subsystem. The buffer memory, which is implemented using a VRAM, is used as a staging area for the transmission and reception of packets. The serial access port of the VRAM is used for transfers between the host and the NAB, and between the NAB and the network. The random access port is used for the on-board processor to manipulate data in the memory. This processor is responsible for a lot of the "common case" packet processing, which include a firewall function and the error-free transmission of packets, while the host processor is responsible for the "rare case" processing, including acknowledgement and retransmission of packets. The main contribution of the NAB architecture was the idea of smartly partitioning protocol processing between the host CPU and the board, a concept which has seen continued use in most modern NIC designs, including the APIC.

The Nectar communications accelarator board (CAB) [1] is a host-network interface that connects through a 10 MByte/s VME interface to the host system. It too has an on-board processor, but the difference is that the CAB processor is responsible for all of the transport protocol processing. By mapping the CAB's on-board memory into the address space of the application running on the host processor, it is possible to achieve a zero-copy architecture, as was demonstrated in Figure 2.3(c). However, this style of NIC design has lost favor because of the large amounts of memory that would be required on the NIC, and because it requires close interaction between software running on the host and that on the NIC's on-board processor. Additionally, Clark et al. [5] have argued that in the case of TCP/IP the actual protocol processing is of low cost and requires very few instructions on a per-packet basis, and thus could be left on the host with minimal impact.

Washington University's Axon project [39] represents an attempt at designing a high performance host communications architecture for high-bandwidth distributed applications. This architecture allows processes to share their virtual address spaces; when a process attempts to access a segment/page that is not in its main memory, it can be retrieved from a local disk or from a remote machine. The NIC design allows network data to be copied directly into the application's address space without any store-and-forward hop, and argues that all per-packet data path protocol processing (including the transport protocol) should be implemented in hardware. As mentioned above, it has since become apparent that higher level protocol processing is best left to host software for reasons of flexibility and portability, and the impact on performance would be minimal. The Axon architecture was never implemented, but a simulation of the architecture showed promising results.

ATM network interfaces bearing mention include the one from Fore Systems and Cambridge University/Olivetti Research [25], which puts minimal functionality in interface hardware. This approach assigns almost all tasks to the host processor, including segmentation and reassembly processing. This approach suffers from two drawbacks: first, modern host processor architectures are optimized for data processing, not data movement, and so the host would have to devote significant resources to manage the high-rate data movement. Second, operating system overhead of this approach can be substantial without hardware assistance for object aggregation and event management.

Another ATM adapter is Bellcore's OSIRIS Decstation 5000 interface [10], which connects the host's TURBOchannel I/O bus to a 622 Mb/s OC-12 ATM link. This interface places all data movement and per-cell operations in custom hardware, while control path functions including segmentation and reassembly (SAR) are implemented using two on-board processors. While this approach does simplify the NIC design, it is often more cost-effective to implement SAR functionality in custom hardware, and implement many higher level control operations directly on the host processor.

The University of Pennsylvania ATM network interface [41] was designed for the IBM RS/6000 workstation, and was used to connect it's Microchannel bus to a 155 Mb/s OC-3 ATM link. It featured hardware implementation of the ATM segmentation and reassembly pipeline, using off-the-shelf programmable logic devices and memory chips. The protocol stack is partitioned so that all per-cell functions are carried out using dedicated hardware, while host software is responsible for higher-level protocol processing, and for controlling packer movement to/from memory. One aspect of this interface that bears mention is that it can optionally use the Microchannel bus' I/O channel controller in order to allow the network interface to have contiguous access to scattered pages in physical memory. This requires the I/O channel controller to have a memory-management unit (I/O-MMU), which needs to be setup by the host processor prior to data transfer. Use of this I/O-MMU provides the network interface or other devices the ability to directly stream data into or out of the address space of user-space processes. Since most modern PC and workstation architectures do not feature I/O channel controllers with software programmable MMUs for virtual address space access by devices, this design is not generally applicable unless such a MMU is implemented on the NIC itself.

Van Jacobson's WITLESS [31] interface design first introduced the concept of a single-copy interface. The idea was to have a shared memory (usually resident on the NIC card) which allows random access by both the NIC and the host processor without affecting each other's performance. When a program sends data, the networking code copies the data immediately into buffers in the shared memory. The various protocol handling routines work on the data in the shared memory, including prefixing of header information, etc. The network interface can then transmit the packet in a single operation. On the incoming side, the interface places received packets in buffers in the shared memory before informing the network code of their arrival. The data remain in these buffers until a program asks to receive them, at which point they are copied into the program's buffer. Note that in both cases, since the processor is doing the copying, it can also compute the checksum over the data in the copy loop, thereby avoiding an additional data touch operation for checksumming. However, on the incoming side, the computed checksum is usually required by network protocols before they can copy data into the application's address space, so that a WITLESS NIC would have to provide on-board checksumming to avoid the overhead of an additional data touch.

There is a subtle point which bears emphasis here; note that the WITLESS architecture is called *single-copy* because the host processor is responsible for copying data between the application buffer and the NIC-resident shared memory. If the NIC were doing the copying (using DMA), the architecture would have been termed *zero-copy*, as exemplified by the University of Pennsylvania interface design. Note that in both cases, the number of data touch operations is identical, the difference is in whether host-processor cycles are used to do the copying or not. There is some confusion as to what to call an architecture where the application buffers themselves reside on the NIC, in which case there are no copies made at all, by either the host processor or the NIC, and the number of data touch operations to items in the system's main memory is zero. Some argue that this approach should be termed zero-copy, but since this style of architecture is almost never used (because it ties the application design very closely to the system architecture), we will ignore its nomenclature for the purposes of this thesis.

The WITLESS single-copy architecture has the advantage over zero-copy architectures in that it can achieve the same number of data touches without requiring any changes to applications using the socket API; it has the disadvantage that processor cycles are spent copying the data, and it requires large memories resident on the NIC.

There are two implementations of the WITLESS architecture in the literature. The HP Medusa [2] interface was targeted at connecting a HP PA-RISC Apollo series 700 workstation to an FDDI network. The second generation of this design, called the Afterburner [9], was designed to connect the same machine to a variety of network links (including HIPPI and ATM) at upto 1 Gbps.

3.2. Network interfaces supporting user-space control

Chapter 2 introduced the concept of a user-space control model for network interfaces to support user-space protocols. With this model in mind, the APIC includes support for two special mechanisms called Protected DMA and Protected I/O. In parallel with the APIC effort, there were two other research efforts that attempted to achieve the same objective, although they did it using mechanisms different from those used by the APIC.

The first of these, proposed by Druschel et al. [14], was a software extension applied to the on-board processor on the Bellcore OSIRIS interface described earlier. Equivalent to a variant of the APIC's Protected I/O feature, this scheme introduces the concept of an application device channel (ADC). An ADC allows an application running in user-space read/write access to only those memory-mapped I/O registers on the interface that correspond to connections that are owned by the application. The memory-mapped I/O registers on the device can correspond either to device control registers, or to physical memory resident on the device. In the latter case, the protocols running in user space can move data directly to and from buffers resident on the device using programmed I/O with an ADC.

Figure 3.2 demonstrates how the ADC mechanism works. The per-connection registers on the network interface are mapped into its memory-mapped 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 (as a control path operation), the kernel modifies the system page table entries such that the page corresponding to the connection is mapped into the process' virtual address space with

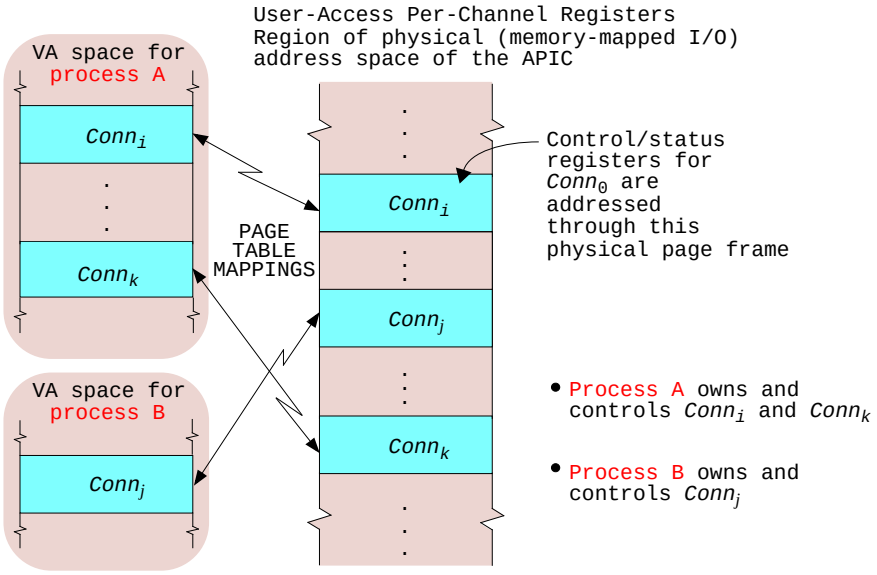

Figure 3.2: Providing protected access to registers using VM overloading

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. The process cannot, however, control other connections that it doesn't 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.

In the case of the OSIRIS interface, the ADC concept is used to allow user processes to read and write to buffer descriptors resident in the on-board memory. Each page in that memory is bound to an ATM connection, and contains a queue of transmit or receive buffer descriptors. In addition, the operating system kernel has to provide the on-board processor with a list of main memory pages which an application using the ADC is allowed to access. In the outbound direction, the user process uses an ADC to enqueue a buffer descriptor containing a pointer to the data to be transmitted. The OSIRIS' on-board processor checks to see if the pointer address falls within one of the allowed physical pages for the connection before commencing DMA to read and send the data from the main memory buffer. If it is determined that the address is not valid for the ADC on which it was queued, the on-board processor asserts an interrupt, and the operating system in turn raises an access

violation fault in the offending application process. The addresses of receive buffers enqueued by an application are similarly checked in the inbound direction.

The U-Net interface work from Cornell [19] implements a mechanism that is functionally identical to ADCs.. The U-Net interface was implemented as a modification to the firmware on a Fore Systems SBA-200 ATM adapter, which features an on-board i960 processor. Because the target host is a SparcStation-20/10, the adapter resides on the SBus. Like most I/O buses, the SBus has fewer address lines than the main memory bus, which limits the region of memory that can be accessed by I/O devices. This necessitates data transfer to and from specially designated communication segments in the main memory. As with ADCs, U-Net communication segments have to be bound to an ADC channel (referred to as a "endpoint" in U-Net papers). This requires pre-programming the network interface with a list of all the pages corresponding to a segment. Again, it is the responsibility of the on-board processor to verify that an application is only sending from or receiving into segments that have been bound to an endpoint that it owns. This involves matching each queued address against the list of valid pages for an endpoint.

U-Net specifies a software architecture for use with a U-Net enabled network interface. This architecture enables use of Active Messages [20], which allow receipt of a message to result in upcalls to an application routine that is specified in the message; the Active Messages enables efficient overlapping of communication and computation in multiprocessors. U-Net style ADCs allow the Active Messages concept to be extended to multiprocessors constructed from networks of workstations running in a more distributed environment (e.g. over a LAN).

More recently, an industry consortium comprising Compaq, Intel, and Microsoft announced the development of a *Virtual Interface (VI) Architecture* specification [43], which is targeted at standardizing the mechanism by which network interfaces provide direct user-space access to applications over system-area networks (SANs). The VI architecture requires applications to register address ranges corresponding to virtually contiguous wired memory regions with the VI network interface. This information is used by the adapter to build page tables corresponding to legal pages in use by each application, similar to the ADC concept. Because it is targeted for use in a SAN environment however, the VI architecture specifies the communication protocols which include setup to establish an end-to-end connection identifier that is used to tag packets (similar to an ATM VC). It also supports unconventional data transfer operations such a remote direct memory access

(RDMA), in which the sender of a message is allowed to specify the destination buffer for a transfer. Also supported is reliable delivery of messages through a low-level transport protocol.

3.3. Desk-Area Networks

As mentioned earlier, the APIC architecture allows multiple APICs to be interconnected to one another and to I/O devices in order to form a desk-area network (DAN). A DAN is an architecture in which a packet switch serves as a workstation interconnect. Work on desk-area networks was pioneered at the University of Cambridge [26], and was also pursued independently at MIT by Tennenhouse et al [27] as part of the VuNet project. While the Cambridge design aims to use a single such interconnect for all components including CPU, memory, and devices, the VuNet system is more like a system-area network in that it targets interconnection of multiple workstations, storage, and I/O devices to one-another.

Both of these efforts are aimed at using a general switch based interconnect within a host computer. The Cambridge DAN was built using a Fairisle ATM switch and a home-grown multiprocessor operating system that runs on each of the nodes connected to the switch. Each device in this architecture has an associated port controller featuring a processor-memory subsystem; the port controller is responsible for communicating with other port controllers and to the host operating system. The VuNet system relies on a switch which is a dumb crossbar; it is assumed that the devices connected to the switch will contain functionality to enable them to command the switch to route cells to the appropriate output ports. This will usually involve a ATM virtual circuit lookup within the network interface connecting devices to the switch.

3.4. Reducing Interrupt Overhead

Interrupt overhead can play an important role in determining the performance of a high bandwidth network interface. There have been various suggestions aimed at reducing this overhead. For example, in the context of the OSIRIS interface, Druschel et al. [14] suggest disabling transmit interrupts altogether, and checking for the completion of transmission as a 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 driver; this has the desirable effect of the interrupt being issued only once per burst of incoming packets. Traw et al [41] suggest disallowing interface interrupts

altogether, and instead polling the interface on periodic hardware clock interrupts. These proposals do not address the adverse impact such schemes have on latency-sensitive applications, and there has been very little in the literature on supporting such mixed environments.

3.5. Receive Livelock Elimination

The receive livelock problem mentioned in the previous chapter is described in [38]. There have since been efforts to address this problem. Jeffrey Mogul and K.K.Ramakrishnan describe a software-centric method [34] which works to avoid livelock by requiring the operating system kernel to carefully schedule network interrupts the same way it schedules process execution. This approach requires that the system check to see if interrupt processing is taking more than its fair share of resources, and if so, disabling interrupts temporarily. The operating system can infer impending livelock because it is discarding packets due to queue overflow, or because higher layer protocol processing or user code is making no progress, or by measuring the fraction of CPU cycles used for packet processing. Interrupts can be re-enabled when internal buffer space becomes available, or upon expiration of a timer.

A scheme to avoid receive livelock by careful design of a network interface is described in [30]. This approach requires the network adapter to be equipped with enough intelligence to be able to detect host input load levels and use this information to dynamically modulate the rate at which it interrupts the host for packet input.

Another solution to livelock is *lazy receiver processing (LRP)* [13]. In the LRP paradigm, the network interface is tightly coupled to higher layers of the protocol stack, and it demultiplexes packets to their destination socket queue. Protocol processing is performed at the priority of the receiving application, rather than at interrupt priority. The assumption here is that the network interface has an on-board processor, the firmware of which is specially designed to interface with the host processor's protocol stack in order to demultiplex packets directly to their destination sockets.

3.6. QoS support for network interfaces

There have been several research papers that describe mechanisms to implement QoS in network switches and routers [44,11]. However, there have been relatively few schemes designed to work in the context of a network interface. One reason is that network interfaces for non-ATM

networks typically provide best-effort service only for all packets, and do not attempt to shape the outgoing traffic streat in any way. With ATM network interfaces too, the QoS support has been fairly limited in most commercial adapters. For example, the Efficient Networks ATM interface supports a small number of paced channels for which a pacing rate can be specified, and all connections should be mapped into one of these channels.

A pacing scheme based on *Active VCI rings* is described in [35]. In this scheme, VCI numbers are entered into slots in a circular array. Each slot in the array corresponds to a transmit opportunity on the wire; a free running pointer advances through the array once per opportunity. If the current array slot contains a valid VCI, a cell is transmitted on that connection, otherwise the transmission opportunity is lost. In either case, the pointer is advanced to the next slot in the circular array. The size of the array limits the minimum rate that can be specified. The software is assigned the task of setting up the elements in the ring to match the desired rates of the different connections. Unfortunately, the contents of the ring have to be modified every time a connection becomes idle or active.

More recently, following recognition of some of the weaknesses in the APIC's pacing scheme, Jonathan Turner proposed a novel pacing algorithm based on timing wheels [42]. Turner's scheme allows spacing between two consecutive cells on a connection to vary from the ideal (determined based on the connection's pacing rate) by a few percent, but corrects for this variation over time in order to prevent drift. Through the use of this approximation, Turner achieves scalability in the pacing algorithm without excessive logic cost.

As shown in Figure 3.3, Turner's scheme works by maintaining multiple timing wheels, each of which has a nominal cell rate associated with it. The pointer in each wheel advances at the wheel's nominal rate; if the rate for wheel i is one cell every b cell times, then the pointer would advance by one every b cell times. When this happens, the list of cells waiting in the current slot is moved to the output list, which contains cells ready to be transmitted on the link. The output list is serviced at the rate of one cell per cell time. To schedule a channel to transmit a cell in the next i cell times, we insert the cell i positions ahead of the pointer in the wheel with the finest granularity (i.e., the highest nominal rate). If i is so large that the we would wrap in this wheel, then we successively try wheels with coarser granularities, until an appropriate wheel has been found. Because of the coarser granularity of some wheels, we may not be able to schedule the cell exactly i cell times

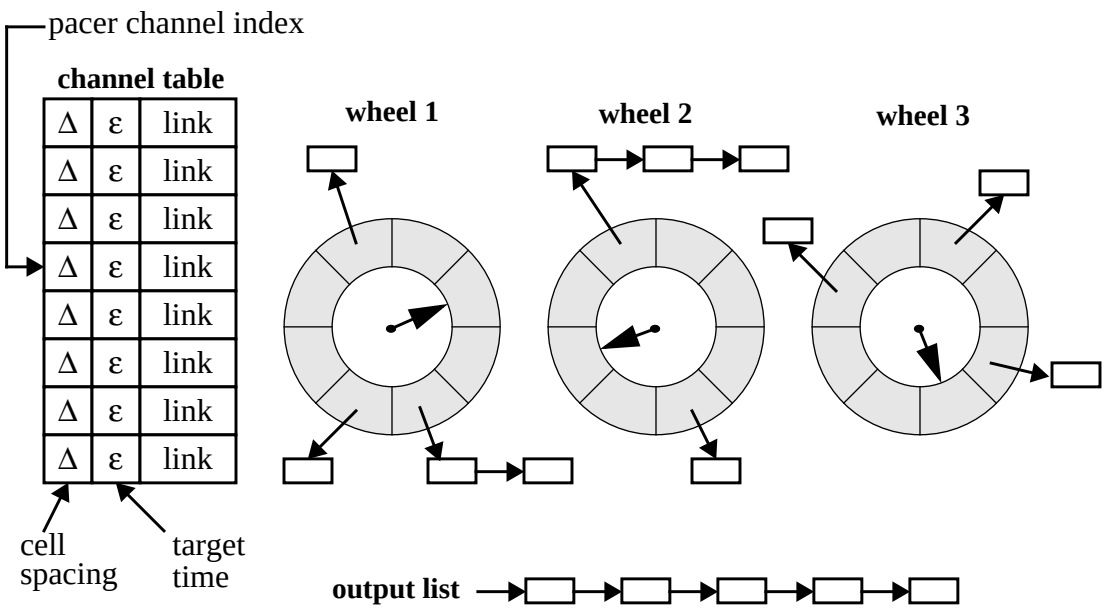

Figure 3.3: Turner's Pacing Algorithm

in the future; in that case, we keep track of the difference in a channel table, and use it to make up by adjusting the spacing between future cells correspondingly.

Chapter 4

Contributions

In this chapter, we first list the different problem areas related to high-speed adapter design that are addressed by the research presented in this thesis. We will then list our contributions, show how they address the problem areas identified, and in light of the related work presented in Chapter 3, demonstrate an advancement in the state-of-the-art.

4.1. Problem Statement

As outlined in Chapter 2, there are a few different problem areas that can be identified with respect to high speed network interface design:

- Memory bandwidth and I/O bus limitations constrain the number of high bandwidth multimedia devices that can be used within a host.
- Typical implementations of current protocol stacks involve many data touches. This results in poor throughput into and out of memory for bandwidth-intensive applications, as well as higher latencies for latency-sensitive applications.
- To enable efficient user-space protocol implementations, network interfaces need to be able to support the user-space control model effectively.
- There is no easy way for bandwidth intensive applications to co-exist with latency sensitive applications given current network interface design techniques and operating system structures.

- The receive livelock problem continues to remain a major obstacle to achieving high performance in modern network interfaces.

- For QoS support, network interfaces need to be able to perform pacing over large numbers of connections with different and independent pacing rates for each connection.

4.2. Overview of Solutions

The APIC directly addresses many of these problems, while simultaneously enabling construction of low cost interfaces that have no on-board processor or memory.

The number of high-bandwidth devices in a system is constrained by the total memory bandwidth available to these devices. The APIC allows more memories to be used, in order to increase the total effective memory bandwidth. It does this by permitting selected devices to have dedicated staging memories, and by allowing construction of a packet-switched desk-area network comprising these devices. A unique daisy-chained desk-area network architecture is described, and the concept of remote control of devices is introduced to effectively support this architecture. Unlike the University of Cambridge or VuNet desk-area networks, the APIC does not require a processor dedicated to each device. It also does not require a general purpose switch to form the heart of the desk-area network; the APIC's daisy-chained architecture allows for graceful cost scaling in proportion to the number of connected devices.

The APIC allows for a zero-copy architecture. Zero-copy is enabled through two different mechanisms. For monolithic kernel-resident protocols, a DMA technique referred to as *Pool DMA* can be combined with *packet-splitting* to allow data transfers between an application and the APIC to occur with no intervening copies. For user-space protocols, the APIC supports zero-copy with the user-space control model using the *Protected DMA* and *Protected I/O* mechanisms. The user-space control model removes kernel intervention from the data path, and therefore permits the APIC to support very low latencies for applications that demand it. All earlier attempts at providing zero-copy behavior have required at least one of the following:

- have required protocol processing to be implemented on the network interface card (e.g. CAB, Axon), thereby requiring complex interaction with the host operating system

- have required the system to provide an I/O MMU (e.g. UPenn interface)
- have required an on-board MMU on the NIC (e.g. OSIRIS, U-Net).

The APIC does not suffer from any of these drawbacks. In addition, all but the UPenn interface have required on-board processors to implement the zero-copy functionality, while the APIC design is able to make do without one.

The APIC has one of the first implementations of the user-space control model for network interfaces. While the ADC and U-Net work was done simultaneously, the APIC's approach has several key advantages. The Protected I/O mechanism is very similar to the ADC and U-Net approaches, but it allows for much finer grained protection policies to be enforced; in particular, the operating system can enforce protection down to individual registers on the device. This is not possible with the ADC or U-Net approaches. Additionally, Protected DMA is a novel mechanism that supports DMA directly to and from user buffers without the need for on-board or system-supplied I/O MMUs. Since most systems do not sport an I/O MMU, and since provisioning one on a network interface can increase the cost of the interface, there is a significant advantage to the APIC's approach.

Existing network interface and operating system design techniques do not gracefully support simultaneous execution of bandwidth-intensive and latency-sensitive applications. To give an analogy here, operating system schedulers have for a long time provided the means for the processor resource to be effectively shared by CPU-intensive applications and delay-sensitive interactive applications. However, the same is not true for the network resource that is managed by a network interface; no known solutions exist in the literature to the problem of allowing high-bandwidth applications to coexist with delay-sensitive ones. The problem, as mentioned earlier, is that bandwidth-intensive connections perform better with a low frequency of interrupt events, which imply that interrupts need to be widely spaced apart in time, or that the interface needs to be polled. Latency-sensitive applications, on the other hand, would prefer immediate delivery of event notifications through interrupts. These are contradicting goals if the interrupt policy used for both types of connections is the same. The APIC solves this problem through the introduction of a new technique called *interrupt demultiplexing*, which allows the interrupt policy to be separately specified for the two types of applications.

The interrupt demultiplexing technique also enables the APIC to address the problem of receive livelock. Taken by itself, this technique can delay the onset of livelock by allowing more useful work to be done on each packet on a per-interrupt basis. Taken together with the user-space control model as implemented by Protected DMA and Protected I/O, the APIC can completely eliminate receive livelock. This is achieved by allowing both protocol processing and driver processing to occur at the same priority as the application. Unlike with Lazy Receiver Processing (see Chapter 3), the APIC's approach does not require the network interface to run special software that is aware of and closely interacts with the host operating system. This is a significant advantage, given the number of different operating systems and the number of time operating systems need to be upgraded. Also, unlike the other approaches to the receive livelock problem described in Chapter 3, the strategy used by the APIC does not require complex feedback-based techniques that rely on modulating the interrupt frequency in response to changes in load, making the APIC approach easier to implement and more portable to different operating environments.

The APIC also effectively supports quality of service by allowing the specification of pacing rates independently for large numbers of connections. This is achieved through the innovative *d*-heap pacing algorithm that will be described in Chapter 5. The Active VCI rings scheme outlined in Chapter 3 suffers from the following drawbacks: it requires very large amounts of memory to handle fine grained specification of rates; enabling and disabling connections are intensive operations that can take a long time to complete, which is significant because a connection needs to be disabled whenever it runs out of data, and re-enabled whenever new data is available to be transmitted; and it requires complex algorithms to add or drop connections. The APIC approach suffers from none of these disadvantages.

Turner's pacing scheme, also described in Chapter 3, is possibly a better way to do pacing than the APIC's approach, but it was developed afterwards, and in response to the deficiencies recognized with respect to the APIC scheme. The APIC's scheme does have the advantage that it is ideal to a very fine granularity, which could become important if the receiving device has very small buffers or expects an even rate of traffic reception. However, except in these cases, Turner's scheme appears to be the preferred state-of-the-art solution to the pacing problem.

In conclusion, Table 4.1 presents a comparison of the APIC vis-a-vis some of the other network interface designs presented in Chapter 3.

Table 4.1: Comparison with Other Network Interfaces

	On-board processor?	On-board memory?	Location of Protocol Processing	Location of SAR processing	Number of data copies	Support for user-space control?	QoS support?
NAB	X	X	Host	N/A	>1		
CAB	X	X	Firmware	N/A	0		
Axon	X	X	Firmware	N/A	0		X
Fore Systems, Olivetti			Host	Host	>1		
OSIRIS	X	X	Host	Firmware	0 (requires on-board MMU)	X	
UPenn		X	Host	Hardware	0 (requires system to have I/O MMU)		
WITLESS, Medusa, After-burner		X	Host	N/A	1		
U-Net	X	X	Host	Firmware	0 (requires on-board MMU)	X	
APIC			Host	Hardware	0	X	X

Chapter 5

Architecture Overview

This chapter provides an overview of the APIC architecture, including many of its features. Several of the novel mechanisms alluded to in previous chapters are described here, along with indications as to how they might be used. Note however that many details as well as some of the more arcane features of the chip have been omitted from this document.

5.1. APIC as a Network Interface Device

The primary and most important function of the APIC chip is that of an ATM host-network interface device. In other words, its main job is to connect a computer system (PC, workstation, or server) to an ATM switch port (see Figure 5.1). When functioning in this capacity, the APIC presents two main interfaces to the outside world: one is a bidirectional I/O bus interface to the attached computer system, and the other is an ATM port interface that is used to send/receive ATM cells to/from an ATM switch. Since the APIC is an electronic device, both of the above interfaces are electronic. However, the connection to the switch is usually over optical fiber, so in the normal case there needs to be a special transceiver device that connects the APIC's ATM port to an optical fiber (transmit/receive) pair. Thus, a traditional-style network interface card (NIC) built around the APIC would hold, in addition to an APIC chip, the transceiver device and some support logic. Such a board would plug into an available I/O bus slot in the computer system, and present a socket in the backplane for the optical fiber pair that will connect it to the switch.

In Figure 5.1, the NIC transmits data by doing a direct memory access (DMA) to read data from specified buffers in the system's main memory, and injecting an ATM cell stream built using that

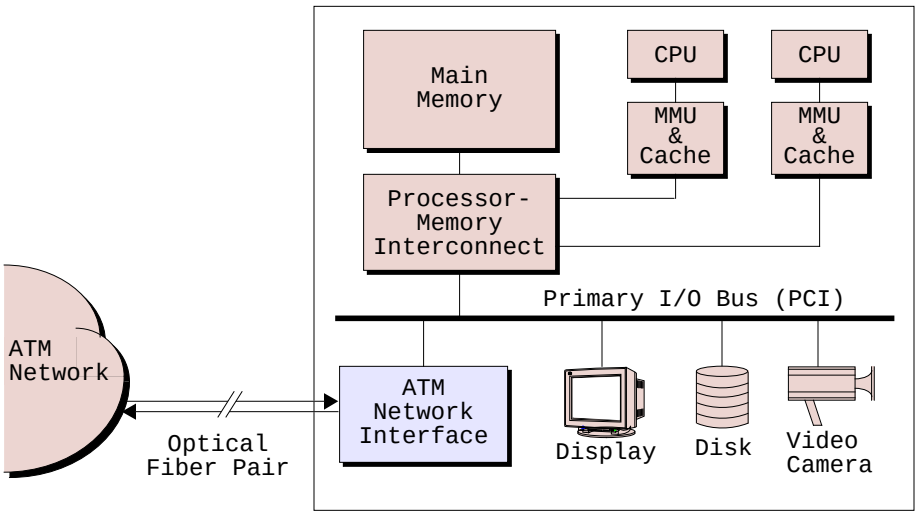

Figure 5.1: Location of an ATM NIC in a Computer System

data into the ATM network. Similarly, when the NIC receives data in cells on its ATM port, it writes the data into buffers in the main memory, again using DMA. Thus, from the NIC's perspective, the main memory is the producer and consumer of ATM data. If a device (for example, the camera) has data to send to the ATM network, the processor will first have to arrange to get the data from the camera into the main memory from where the NIC can read and transmit it. Similarly, if a video stream arriving on the ATM input port of the NIC has to be sent to the display, it will have to go through the system's main memory. As described earlier, this can be a problem if there are many high bandwidth devices in the system, because of the limited main memory bandwidth. This leads us to the second main function of the APIC, which is to serve as a building block for a Desk-Area Network (DAN) that would allow data streams to bypass the main memory and provide a direct path from the network to various devices in the system.

5.2. APIC-based DANs

In order to support desk-area networking, the APIC incorporates a second ATM port which can be used to daisy chain multiple APICs together as shown in Figure 5.2. Note that this is a deviation from conventional ATM network adapters, which usually have only a single ATM port. In the daisy chained DAN shown in the figure, there is one primary APIC chip which interfaces to the system's I/O bus, and serves as the primary network interface for the system. All the remaining APICs are each connected to an I/O device and some memory. These "I/O modules" may optionally also have

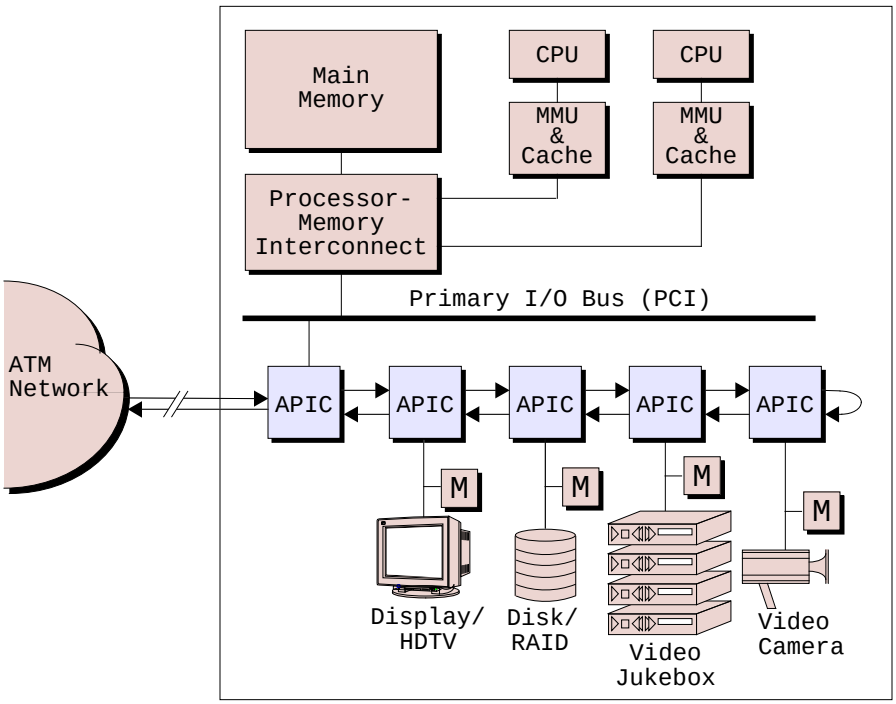

Figure 5.2: An APIC Interconnect as a Desk Area Network

a local control processor, but more typically they are controlled directly by the system's main processor(s). With this DAN architecture, data can move directly between the network and I/O devices, without having to pass through the system's main memory. Thus, for example, the APIC could take a video data stream originating at the camera and transmit it to the ATM network as an ATM cell stream. Note that cells from this stream may transit (unchanged) through other APIC chips enroute to the network. As another example, an ATM cell stream containing video data that is received from the network can be sent directly to a display device without having to go through the system's main memory. The DAN can also be used to directly communicate between different devices within the same system. As an example, a video stream from the disk could be sent to the display device using the APIC interconnect.

☞ The connection between the ATM ports of two APICs can be either electronic (PCB traces or a ribbon cable) or optical fiber. In the latter case, optical-to-electronic transceiver devices will need to be used. Distances of up to 10 feet have been achieved using ribbon cable; for longer distances, optical fiber is needed.

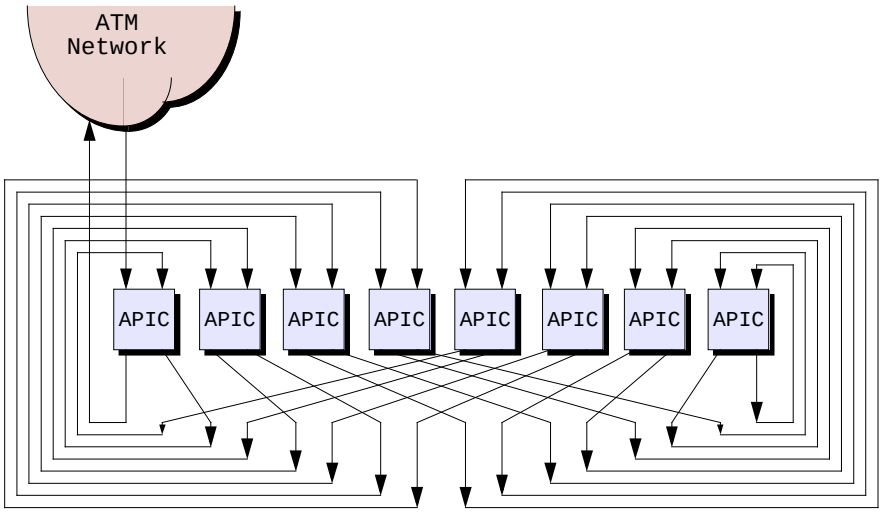

Figure 5.3: Perfect Shuffle Topology

A daisy chain is not the only interconnect topology that is possible to build using APICs. Figure 5.3 shows an interconnect with a perfect shuffle topology, which has the advantage of lower delay for cells: the number of hops from any APIC to any other APIC is $O(\lg n)$, as opposed to $O(n)$ for the daisy chain, where n is the total number of APICs in the interconnect. (Note that to simplify the figure, we have chosen not to show the individual devices that would be connected to each APIC over its bus port). It is also possible to build other topologies such as a toroidal mesh or a manahattan street network using APIC chips.

Although we have described the use of the two ATM ports of the APIC only in the context of desk area networks, other uses can be envisioned as well. For example, APICs can be used:

- to build local area networks (LANs);
- for processor interconnects in parallel (super)computers;
- to build ATM switch port cards that could be used to selectively process packets entering or leaving a switch port, or for traffic scheduling and/or policing;
- as a network tap used to monitor traffic on an ATM link and collect statistics.

- to build servers and clients with enhanced reliability resulting from dual paths between servers and clients.

☞ Washington University is currently working on a few projects that utilize the APIC in one or more of the application scenarios mentioned above. One of the projects uses the APIC to construct switch port cards (SPCs). Another project uses the APIC as a network monitoring tool. A third project uses the APIC to attach an IP processor to each port of an ATM switch, thereby enabling the construction of very high performance and scalable gigabit IP routers.

5.3. Ports and Connections

5.3.1. The ATM Ports

The prototype APIC chip supports a 1.2 Gb/s maximum link rate on each of its two input and two output ATM ports. These ports comply with the ATM Forum's UTOPIA (Universal Test and Operations PHY Interface for ATM) standard, which specifies the (electronic) interface between an ATM-layer device (the APIC) and a PHY-layer device (the optical/electronic transceiver). Each UTOPIA port has a 16 bit data path which is used for speeds of 622 Mb/s (OC-12) and 1.2 Gb/s. Operation at 155 Mb/s (OC-3) is also possible using only 8 bits of the 16 bit data path.

5.3.2. The Bus Port

The APIC's I/O bus interface is compliant with the industry standard PCI local bus. Both PCI-32 (32 data lines) and PCI-64 (64 data lines) versions are supported. Only the 33 MHz version of the PCI bus is supported, giving us a bus peak rate of 1.05 Gb/s for PCI-32 and 2.11 Gb/s for PCI-64. The APIC is theoritically capable of sourcing and sinking data at the maximum possible rate achievable on the PCI bus.

☞ The maximum achievable data rate is lower than the peak bus rate because of address and turn-around cycles between transactions, during which no data can be transferred on the bus. Larger transactions on the bus are more efficient, because there are fewer wasted cycles relative to the number of useful cycles. The obtainable data rate is also dependent on other factors such as the maximum memory bandwidth (which is sometimes as low as 500 Mb/s), the current load on the bus and memory subsystems, and the architecture and performance of the bridge chips between the PCI bus and memory.

5.3.3. Virtual Connections (VCs)

The APIC supports a total of 256 ATM virtual connections (VCs) each for transmit and receive. Each supported VC can be configured to use any VPI in the range from 0 to 255, and any VCI in the range from 0 to 65535. In other words, the entire VPI/VCI range is supported. The only constraint is that no two open receive VCs can have the same low-order 8 bits in their VCIs. All state associated with a VC is stored on-chip.

☞ The number of supported VCs may sound like very few compared to the numbers being quoted by many commercial vendors. We feel that 256 is more than enough for workstations and PCs. For demanding server applications, more VCs may be required. Our choice in this matter was guided by the number we felt could fit comfortably on the chip, because we wanted to avoid having any additional off-chip memory to hold VC state. At one time, we did consider (and indeed, spent considerable effort in) trying to maintain the VC state in the host's memory, and implementing a type of caching mechanism to bring only required VC state into the chip in an on-demand fashion. However, the hardware and timing issues involved with such a design proved very difficult to tackle. Future versions of the chip that use a smaller feature size could easily hold state for 1024 or more VCs each for transmit and receive, which is enough even for all but the most demanding server applications.

5.4. Basic Operation

The default behavior of the APIC (after reset) is to forward without modification any cells arriving at one of the two input ATM ports to the other ATM port for output. We will henceforth refer to this path through the chip as the *transit path*. There are two other paths that cells can take through the chip: a *receive path* and a *transmit path*. These are described below.

In order to achieve anything other than default transit forwarding through the device, it is necessary for the controlling processor to configure VCs on the chip (we will describe the method used to configure the chip later).

The APIC will pass all cells received on an input ATM port that belong to an open receive (Rx) VC to the host. This is done by writing the data from the cell into buffers in memory accessed over the APIC's bus port. This is the *receive path*.

Note that the memory that is accessed over the bus port is either the host's main memory or the local memory for a device; we will henceforth refer to this as an APIC's *external memory.*

On the outgoing side, a transmit (Tx) VC can be configured to send data on a specified output port. Data for a Tx VC is read from buffers in external memory by the APIC and used to generate a stream of ATM cells. These are forwarded to the specified output port, where they are interleaved with transit cells before being transmitted This is the *transmit path*.

5.4.1. Segmentation and Reassembly

The data that is read (written) from (to) external memory is in the form of ATM Adaptation Layer (AAL) frames. The process of breaking an AAL frame into ATM cells which can then be transmitted is called segmentation, and the process of taking one or more received cells and reconstructing an AAL frame from those cells is called reassembly. ATM network interface devices are often referred to as SAR (segmentation and reassembly) devices. The APIC supports two AAL types: AAL-0 and AAL-5.

5.4.2. Packets and Frames

Throughout this document, we use the term *frame* to refer to either an AAL-0 frame or an AAL-5 frame, and the term *packet* to refer to the higher level unit of data that is encapsulated within a frame. On the transmit side, a packet is usually passed to the APIC driver code by either a network layer protocol (eg., IP), or a native ATM transport layer protocol. The driver is responsible for encapsulating the packet in a frame, and passing the frame to the APIC for segmentation and transmission. On the receiving end, the APIC passes reassembled frames to the driver, which is responsible for extracting the packet from the frame and delivering it to the appropriate higher layer protocol. It is very important to note that the APIC operates only on frames, and is for the most part oblivious to the existence of packets.

5.4.3. Cut-through Behavior

In Chapter 2, we described the difference between a cut-through adapter and a store-and-forward adapter. The APIC is designed as a cut-through adapter. When the APIC has to transmit a frame, it reads portions of the frame called *batches* into its on-chip memory and transmits those as soon as the link becomes available. Batches are composed of one or more cells worth of data; they

are described further in Section 5.5.5. On the receiving end, the APIC will attempt to write batches of cells belonging to a frame to external memory as soon as the bus becomes available; note that it does not wait to receive all cells belonging to the frame before doing this.

☞ An APIC NIC can possibly also be used as a store-and-forward adapter by adding some local memory on the NIC board into which the processor places frames to be transmitted, and which the APIC can use to store received frames. However, there are several hardware level issues related to adding such local memory on the board that have not been addressed by the APIC design team, since we felt that the value of doing this was questionable (it increases the cost of the boards without much added benefit).

One of the primary motivations for designing the APIC as a cut-through adapter is to avoid the need for any on-board memory on the NIC (and the need to have extra pins on the chip to access such a memory). This significantly reduces the cost of a NIC, and achieves better sharing of the system's main memory resource. It also eliminates the problem of deciding the size of on-board memory: no one size works for all applications. But more importantly, it means that an APIC NIC has no scarce resources on the board; as we will see later, this allows for better structuring of protocol stack implementations in an operating system.

Of course, our choice does mean that we have to live with the drawbacks associated with having a cut-through design (see Chapter 2). One problem has to do with the fact that a cut-through adapter can end up transmitting a partial frame if there is an error, and it could report receipt of partial or corrupted frames to software. This problem is not a serious one, since such events are rare, and it is easy to recover from them in software. Of more concern is the problem of (transport protocol) header checksums. Because of pre-existing standards (TCP), it is not possible to mandate that the checksum field should reside in the packet trailer. Of course, we could just leave the checksumming all to software, but this can have an adverse impact on TCP performance which we would like to avoid. The APIC provides for a TCP checksum assist on the receiving end, as described in Section 5.9.1. However, as we mentioned earlier, the problem is more with the sending side, and here the APIC offers no hardware level solution.

☞ We will see later that even this is not such a serious problem, because current implementations of TCP/IP require a copy of data from user space to the kernel on the sending side; as mentioned

in Chapter 2, by rolling the checksum computation into this copy loop, most of the cost of checksum computation can be eliminated. Although there is a similar copy (from kernel to user space) on the receiving end, it is much more difficult to roll that checksum computation into the copy loop because of recovery (rollback) problems if the checksum turns out to be incorrect (Note: the receive side hardware-level checksum assist makes this a non-issue in the APIC context).

5.4.4. Channels and Connections (VCs)

Throughout the rest of this chapter, we make a strong distinction between a *channel* and a *connection* (or *VC*). A *connection* is an ATM virtual circuit (VC); we will use these latter two terms interchangeably to mean the same thing. A connection is therefore an ATM layer entity: each connection is associated with a fixed VPI and VCI. A *channel* corresponds to a DMA data stream. Each channel is associated with a single FIFO queue in external memory that serves as a source or sink for data that is read or written by the APIC. Although in many cases there is a one-to-one mapping between connections and channels (i.e., they refer to the same data stream), this need not be the case. For example, in some cases a single channel can source cells that are transmitted on multiple connections. The reverse is also possible, though not very useful: multiple channels can source cells all belonging to the same connection. Similar one-to-many and many-to-one relationships between connections and channels are possible in certain special cases in the receive direction too.

5.5. Summary of Features

The APIC design has been targeted to permit efficient operation in many different software environments, and to support a wide spectrum of applications. Since one of our primary goals in this undertaking was to make the chip flexible enough to serve as a research platform for high performance protocol implementations, not all of the features described may be useful for a given flavor of application. In this section, we give a high level overview of some of the important features supported by the chip.

5.5.1. Multipoint and Loopback

At an abstract level, the APIC can be thought of as a switch with three inputs and three outputs: two input/output pairs corresponding to the two ATM ports, and the third corresponding to the bus

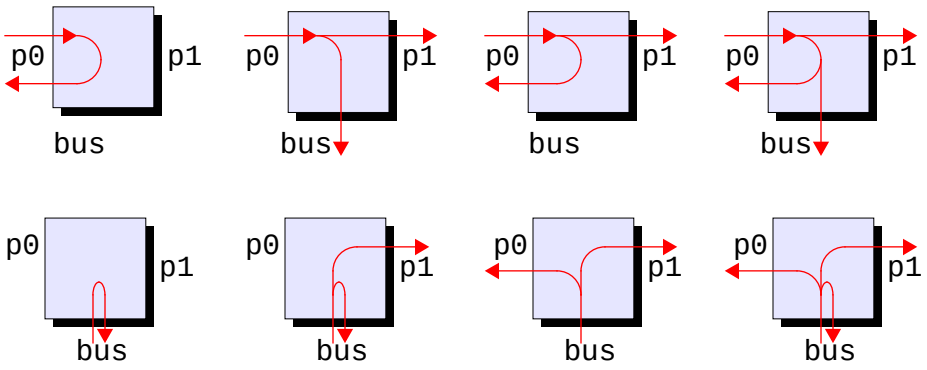

Figure 5.4: Instances of Multipoint and Loopback Connections

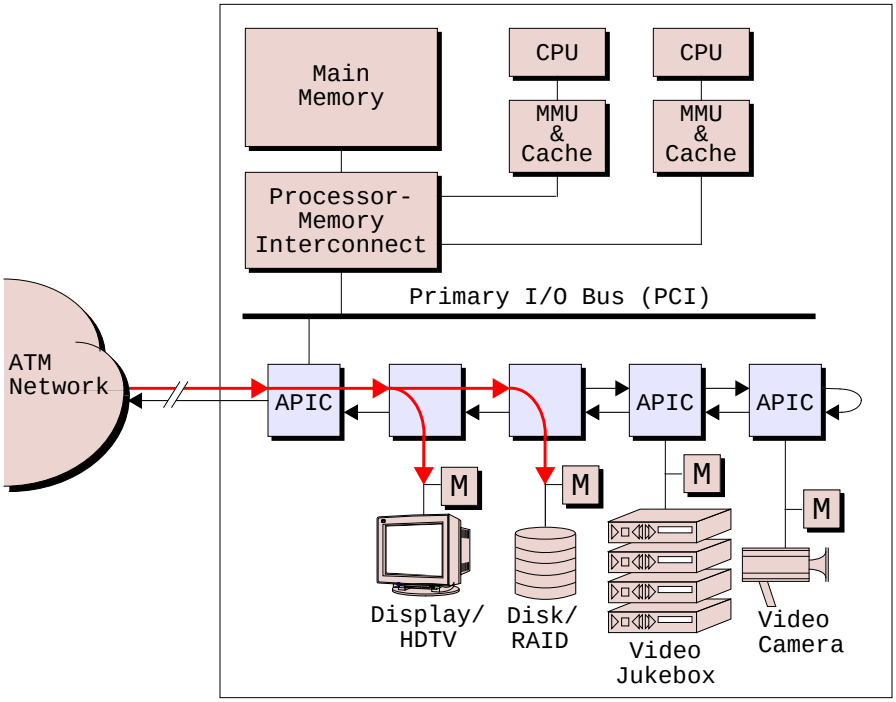

Figure 5.5: An Example Multipoint Application

port. Data originating at any of the three possible inputs can be switched to any subset of the three outputs. Some of the possibilities are shown in Figure 5.4. The APIC allows the subset of outputs to be specified separately for each receive VC (data input is from one of the two ATM ports) and each transmit channel (data input is from the bus port). Notice that point-to-point, multipoint, and loopback connections are all enabled by this mechanism. The loopback feature is useful for testing the operation of a port, while the multipoint feature can be used to implement broadcast in a daisy

chain, or to support multipoint applications (for example, directing a video stream from the network both to a display as well as to a disk for local storage — see Figure 5.5).

5.5.2. AAL-0

As mentioned earlier, the APIC supports two types of ATM adaptation layers: AAL-0 and AAL-5. AAL-0 is also called the null AAL, and its main function is to allow the host to individually send and receive completely specified ATM cells. AAL-0 is intended to be used for the following purposes:

- To communicate with a device that operates on raw ATM cells and does not understand any AALs.

☞ Washington University's Multimedia Explorer (MMX) is a device that falls into this category.

- When it is necessary to send special types of control cells into the network without having to first establish a connection, or where special formatting of a cell and its header are necessary.

☞ For example, Washington University's Gigabit ATM Switch is controlled using specially formatted control cells that can be sent using AAL-0.

- For software emulation of AALs not supported by the APIC.

There is no standard AAL-0 frame format; an implementation is free to choose its own format. The APIC uses a 56 byte format, which consists of a special 4 byte word called an *internal header*, followed by the first 4 bytes of the ATM cell header and 48 bytes of cell payload. The internal header contains (among other information) the ATM port that the cell should be transmitted on (or was received on). When transmitting an AAL-0 frame, the APIC constructs the corresponding cell by stripping the internal header and adding the ATM header error check (HEC) byte between the cell header and the payload. On receive, the APIC removes the HEC byte from the incoming cell, prepends an internal header, and writes the resulting AAL-0 frame to external memory. This process is illustrated in Figure 5.6.

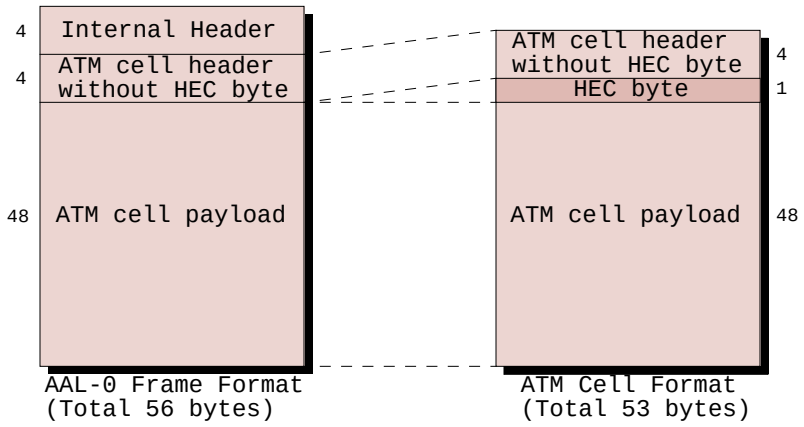

Figure 5.6: AAL-0 Frames and SAR

5.5.3. AAL-5

Although several standard AALs have been defined (AAL-1, AAL-2, AAL-3/4, AAL-5) for different classes of ATM traffic, AAL-5 has (arguably) become the defacto AAL of choice for almost all applications, because of its simplicity. For this reason, AAL-5 is the only adaptation layer (other than AAL-0) implemented by the APIC in hardware. If support for some other AAL is needed, it can be implemented in software (albeit at a loss in performance) using the AAL-0 feature of the APIC.

5.5.4. Traffic Types

There are several places both within the APIC and in external memory where data (cells or frames) can be enqueued on one of a set of two or more queues, only one of which can be serviced at a time. In most such cases, the policy used to service the queues is based on the *traffic type* of the data contained in the queue. Note that each queue is assumed to hold data of a single traffic type. However, it is possible for multiple queues in a set to hold data of the same traffic type.

Consider for example the set of transmit DMA channels that are ready (i.e., they have data queued for transmission in external memory). These channels need to be serviced by the APIC according to some pre-defined service policy. Servicing a channel in this case corresponds to reading enough data from the corresponding queue in external memory to make a batch of cells, and queueing those cells within the chip for transmission on the appropriate output port. The decision

of when and from which channel to read the next batch of data is made based on the traffic types associated with the various channels.

The APIC defines three traffic types for transmit data: *low delay, paced,* and *best-effort,* and two traffic types for receive data: *low delay* and *normal delay.*

We would like to point out a subtlety here, the full import of which will become apparent only after we have finished discussing the internal design of the chip. It should be clear by now that traffic types are associated with certain data streams. Earlier, we pointed out the difference between channels and connections, and noted that both have an associated sequential data stream (see Section 5.4.4.). So the question arises: is a traffic type associated with a channel or a connection? In the APIC context, the traffic type is a property of a channel for transmit data, and a connection for receive data. Because of this peculiarity, we talk of low delay, paced, and best-effort *channels* for transmit, and low delay and normal delay *connections* for receive.

Receive Traffic Types

On receive, the traffic type (low delay or normal delay) is used only for internal queuing in the APIC. At places where such queueing occurs, low delay traffic is always given priority over normal delay traffic. Since there is very little buffering on the chip itself (a little more than 256 cells), the traffic type for receive does not play as significant a role as for transmit.

Transmit Traffic Types

On transmit, one important APIC configuration parameter that affects the discussion of the different traffic types is the *maximum sourcing rate*. This parameter determines the maximum rate at which an APIC will source data. Of course, the configured value should take into account physical limitations such as the link rate and the maximum bandwidth achievable on the bus.

Low Delay

Low delay is the highest priority traffic type for transmit channels. The APIC will always read and transmit data from low delay channels in preference to paced or best-effort channels. If there are multiple active low delay channels, they will be serviced in round robin order. Low delay traffic is transmitted at the maximum sourcing rate of the APIC.

Paced

Next to low delay traffic, paced traffic has the highest priority. As mentioned in Chapter 2, data from a paced channel is transmitted at a peak rate that has been configured for that channel; this *pacing rate* is independently specifiable for all transmit channels.

Best Effort

The lowest priority transmit traffic type is best-effort. A best-effort channel uses up all the remaining bandwidth that is leftover after accounting for low delay and paced channels. If there are multiple best-effort channels, they are serviced in round-robin order, so they will end up sharing the leftover bandwidth equally. Note that the leftover bandwidth is what is remaining given the maximum sourcing rate of the APIC. Note that if there are no active low delay or paced channels, then a best-effort channel behaves almost exactly like a low delay channel. The best-effort traffic class is work-conserving.

☞ The fact that the APIC shares available bandwidth equally between all active best-effort channels can be exploited as a fair queueing (FQ) mechanism. Since software does not need to be involved, this is a very low cost solution to the problem of FQ. The ideal FQ discipline is bit-by-bit round robin (also called generalized processor sharing, or GPS). Using best-effort channels to implement fair queueing gives us a nearly optimal fair queueing discipline, which we call "cell-by-cell round-robin" (CCRR). Future versions of the APIC could implement a weighted version of CCRR that can be used as an implementation of weighted fair queueing (WFQ).

5.5.5. Batching

The APIC has to issue transactions on the PCI bus every time it wants to write or read a sequential block of data to or from external memory. Each transaction has some overhead associated with it.

☞ Each transaction requires one address cycle. For reads, there is additionally a turnaround cycle between the address cycle and the first data cycle, because the direction of movement of data on the bus changes. There is usually a dead tick between every pair of transactions. And for reads, there may be several wait cycles during which data is being fetched from the memory.

Additionally, modern memory architectures (SDRAM, Rambus, Page-mode DRAM) tend to favor large sequential accesses. For these reasons, larger transactions are usually more efficient than smaller ones and result in higher performance. The payload from a single ATM cell (48 bytes) is too small to be an effective transaction size. For this reason, the APIC attempts to move batches of cells to and from external memory in a single transaction.

☞ Batching is also useful if the APIC is interfacing to the serial port of a VRAM. The size of the serial access memory (SAM) of a VRAM can be as large as 512 bytes. If a transfer cycle is required for very small partial fills of the SAM, then there will be no throughput advantage gained by using the serial port.

For transmit channels, the APIC uses a configurable *batch size* parameter to determine the maximum number of cells that can comprise a single transaction. Batch sizes of 1 to 8 cells are supported. For receive channels, there is no batch size parameter; the APIC attempts to create as large a batch as possible from cells received on that channel.

5.5.6. Remote Control

When operating in a traditional NIC environment, an APIC device driver running on a host processor controls the APIC by using memory-mapped I/O to read or write the APIC's on-chip registers. What about the case when an APIC is connected to an I/O device in a DAN environment? If we could have a special control processor associated with each device that can issue loads and stores to the APIC's register space over the PCI bus, and has access to the shared data structures in the APIC's external memory, then there is no problem. However, in most cases dedicating a control processor to each device in the DAN will be prohibitively expensive. What we would like is for the APIC as well as its connected device to be controlled directly by the host's main processor. The APIC accomplishes this kind of "remote control" by defining three special types of ATM cells: *control cells*, *response cells*, and *interrupt cells*. In a DAN environment, the controlling processor will usually be the host processor, but the remote control feature of the APIC is completely general: we assume that control cells can originate anywhere, including from remote hosts in the ATM network. This means that the remote control feature would permit isolated "ATM devices" that are distributed over a wide geographical area (eg., surveillance cameras) to all be controlled from a centralized control computer.

Control cells are sent to remote APICs on dedicated control VCs. They define the control operation that has to be performed at the remote APIC. How does a control cell target a specific APIC in a daisy-chain of APICs? We use a pin-configured 16-bit address called the *APIC ID* to identify the target APIC for the control operation. The control operation itself is encoded as a read or write operation to an internal or external address (which is specified in the control cell). *Internal accesses* are used to access the APIC's on-chip registers. If an internal access is specified, the APIC internal register corresponding to the specified address is read or written. Note that the APIC presents the same register address space for both local (memory-mapped) control and remote (control cell) control. *External accesses* are used to access shared data structures in the remote APIC's external memory, and to control any devices that reside on the remote APIC's local PCI bus. If an external access is specified, the remote APIC will become bus master on its local PCI bus, and issue a read or write transaction to the specified address. This mechanism can be used to program the device connected to a remote APIC: send a control cell to that APIC commanding it to read or write to a specified control register in the device's memory-mapped I/O address space.

Response cells serve as an acknowledgement for a control cell, and also return a success/failure indication and the result from the control operation (if it was a successful read operation) to the controlling processor. Unlike control cells, response cells are not sent on a special control VC; the cell header used in a response cell is completely specified within the corresponding control cell. This leaves the choice of the VC for response cells entirely up to the controlling processor.

A CRC field in control and response cells allows for error detection. To ensure reliable control in the face of cell loss and/or corruption, the APIC implements an alternating bit protocol. This is a stop-and-wait protocol, which means that only one control cell operation can be outstanding at any time, and the next control cell cannot be sent until the first has been acknowledged. The controlling processor can retransmit a control cell if no response is received in some timeout period. The alternating bit protocol's one bit sequence number is used to guarantee at-most once semantics, i.e., the operation specified in the control cell will never be performed more than once, and the correct response from the operation will always be returned.

Interrupt cells are used to report asynchronous events (interrupts) that occur at a remote APIC to the controlling processor. There is no special interrupt cell format. When the remote APIC raises its interrupt line in response to some event, it can optionally also resume a specially configured

transmit channel called the *interrupt channel*. This channel should be setup by the host as a paced channel with a very low pacing rate, and primed with some cells that can have any format the controlling processor wishes (they can even contain no data at all). When the interrupt channel is resumed, it causes cells (which we call interrupt cells) to begin being transmitted from that channel. The VC corresponding to the interrupt channel is expected to be setup so that it terminates at the APIC that is directly connected to the controlling processor. This will mean that the interrupt cells will be received by that APIC and result in a real interrupt being issued to the control processor. The latter can then examine the interrupt cell that has been received to decide which remote APIC generated it, and take appropriate action. This includes acknowledging the interrupt and suspending the interrupt channel so that no more interrupt cells are transmitted. Since we assume that the pacing rate for an interrupt channel is setup so that it is very low, in most cases only a single interrupt cell will be transmitted in response to which the control processor will suspend the interrupt channel. However, if for some reason the first interrupt cell is lost, the remote APIC will transmit another. These cells will continue being sent at the pacing rate of the interrupt channel, until the processor reacts by issuing control cells to acknowledge the interrupt and suspend the interrupt channel. This guarantees that a remote interrupt will never be lost.

5.6. User-Space Control

In Chapter 2, we introduced the concept of user-space control of a network interface. We will now see how the APIC supports the user-space control model. In particular, two novel techniques are presented which enable this feature in the APIC: Protected I/O and Protected DMA. The former is used to enable user-space drivers to perform “protected” memory-mapped I/O accesses to on-chip registers. The latter allows these same drivers to have “protected” access to the shared data structures in main memory (i.e., the DMA channel queues). In either case, the degree of protection that is enforced by the APIC depends on policies defined by the kernel driver; the APIC only provides the mechanisms needed to enforce these policies. Protected I/O and Protected DMA are covered in the sections that follow.

5.6.1. Protected I/O

As mentioned above, *Protected I/O* is the mechanism that enables portions of an APIC’s memory-mapped I/O space to be made accessible to untrusted user-space driver code. The decision of

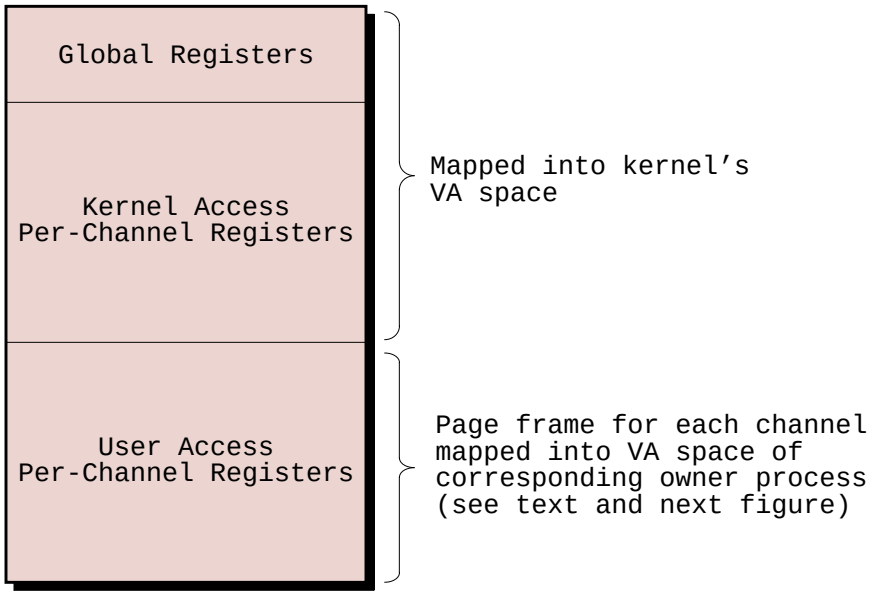

Figure 5.7: Memory-Mapped I/O Address Space of the APIC

which portions of the register space should (or should not) be accessible to a particular user-space process is made by trusted software (the kernel driver). It is important to remember that protected I/O is only a mechanism provided by the APIC; the policy is in the hands of software (the kernel driver) running on the host processor. The Protected I/O scheme is in many respects similar to the ADC work presented in Chapter 3.

The APIC's memory-mapped registers fall into two categories: *global registers*, and *per-channel registers*. The set of global registers contain state that is of global relevance, such as the interrupt status register, the software reset register, etc. There is one set of per-channel registers for each channel supported by the APIC; these hold state that varies from channel to channel, such as the channel pacing rate, the corresponding connection's VPI and VCI, etc.

☞ Although we mentioned earlier that there need not be a one-to-one mapping between channels and connections, the Protected I/O feature only makes sense when we consider channels to be bound to connections. Note that this applies only to channels that are controlled using protected I/O.

Figure 5.7 shows that the APIC's memory-mapped I/O address space is divided into three regions. One region is used to access all the global registers. The other two regions are used to

access the per-channel registers. Each physical per-channel register is mapped into both of these regions, and therefore can be accessed by loads/stores to either of two addresses. The mapping of addresses to registers is such that:

1. Each of the three regions in the APIC's address space occupies an integral number of physical page frames.

☞ A physical page frame is the address space corresponding to a single page in the machine's physical address space: it is the granularity at which the memory-management unit (MMU) enforces virtual memory (VM) page protections. The page frame size varies in different machines, but it is always a power of 2; a typical value is 4 KB.

2. In the *user-access per-channel registers* region (see figure), all the registers corresponding to a channel fall into the same physical page frame, and no single page frame contains registers for more than one channel.

The physical page frames containing global registers and *kernel-access per-channel registers* are assumed to be mapped by the kernel driver into the kernel's virtual address (VA) space. This gives the kernel driver unrestricted access to all of the APIC's physical registers. Additionally, whenever a user process is granted the ownership (capability) for a connection, the kernel driver will map into that process' VA space the page frame for the corresponding channel from the user-access per-channel registers region. This was illustrated in Figure 3.2, which is repeated here as Figure 5.8 for convenience. Notice that only the owner process of a connection will be able to control it by modifying the corresponding registers. The user-space driver accesses these registers using only virtual memory loads and stores—there are no system calls involved. A process cannot access the registers for connections/channels that it does not own, because the kernel driver would not have mapped the corresponding page frames into its VA space. What we have done here is to overload the system's virtual memory (VM) protection mechanisms to provide protected access to per-channel device registers.

Since it is possible for the kernel to map in pages from user-access region of the APIC's address space, why do we need a separate kernel-access region? The problem is that because each connection/channel occupies one page frame in the user access region, the kernel would have to map in as many pages as the total number of connections (or at least, those that are open). On most systems,

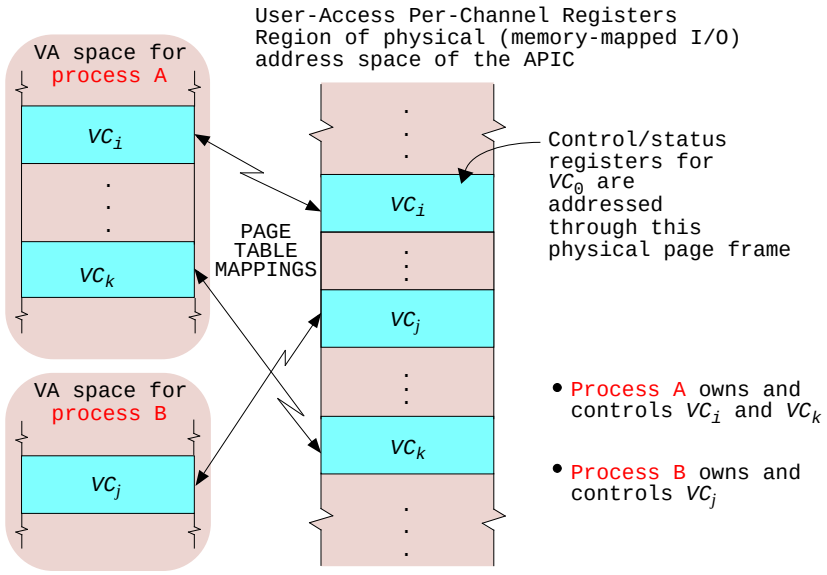

Figure 5.8: Providing Protected Access to Registers using VM Overloading

the kernel page table context cannot support large numbers of pages, making it impossible to map the entire user-access region into the kernel's address space. We can exploit the fact that most of this region is sparse; the actual number of registers per connection/channel is very small, so we can concatenate the registers for all the connections/channels into a single contiguous region of the physical address space, which can be easily mapped into the kernel's address space. This forms the kernel-access region of the address space. Note that the kernel-access region is dense, while the user-access region is sparse to allow registers for different connections/channels to fall into different physical page frames.

The scheme described above does not give the kernel driver fine grain control over which individual per-channel registers are accessible to the owner process of a channel. For example, should a process be allowed to change the pacing rate for a channel that it owns? Clearly, this is a policy decision that should be left to the APIC driver programmer, rather than being hardwired in the chip.

☞ One way to achieve this kind of fine grain access control would be to map each device register into a separate page frame in the APIC's address space; then the kernel driver can selectively map these pages into process' address spaces based on its register access policies. This approach is not feasible when there are many registers per channel, because 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 translation lookaside buffer (TLB).

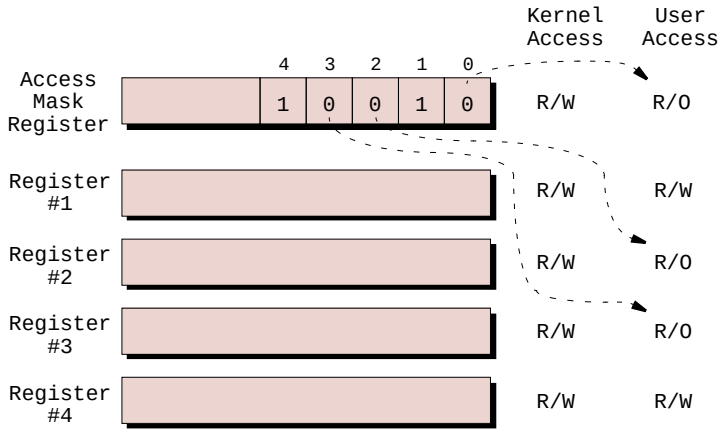

Figure 5.9: Fine Grain Access Control Using Protected I/O

Protected I/O enables fine grain access control to per-channel registers by augmenting each set of per-channel registers with an *Access Mask Register (AMR)*. As shown in Figure 5.9, each bit in an AMR controls access to one other register in the corresponding set of per-channel registers. If a bit in the AMR is set, then the APIC will allow write accesses to the corresponding per-channel register from the user-access region of its address space to succeed; if the bit is cleared, only read accesses are permitted. Any accesses made to registers using addresses in the kernel-access region always succeed: this provides the means by which the kernel driver can program the AMR itself. Since user processes can access per-channel registers only through the user-access region of the APIC address space, their access is limited to those registers that have their corresponding bits set in the channel's AMR.

☞ The scheme described above can be extended to include the option of providing no access (in addition to read-write and read-only) to registers by having two bits in the AMR per register instead of just one. However, this added functionality is not included in the current APIC prototype, because there were no per-channel registers that (we felt) needed to be completely hidden from applications.

Note that the fine-grained access protection supported by Protected I/O is not available with the ADC or U-Net approaches described in Chapter 3.

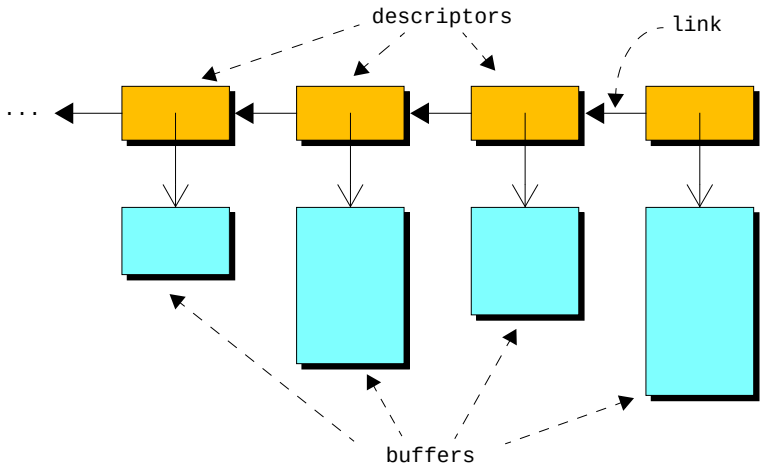

Figure 5.10: A Descriptor Chain

5.7. DMA Modes

The APIC uses DMA for all movement of transmit/receive data from/to external memory. It also uses DMA to read/write control data structures called *buffer descriptors* that are stored in external memory, and which are used to implement the FIFO queues of buffers needed for scatter/gather DMA operations. All buffer descriptors are 16 bytes in size, and contain information for a single data buffer. This information includes the physical address of the buffer in external memory, its length, and some flags. In addition, each buffer descriptor also contains a pointer to another buffer descriptor, which is used to create linked lists of buffer descriptors. Figure 5.10 shows a linked list of buffer descriptors, with each descriptor pointing to a different buffer. We will henceforth refer to such linked lists as *descriptor chains*, and to the linked list pointer in each descriptor as a *link*.

Descriptor chains are built by the driver software, and are used by the APIC as FIFO queues that serve as sources and sinks for data. Figure 5.11 illustrates how a descriptor chain is used by the APIC to transmit data. For each supported DMA channel, the APIC always has one on-chip working descriptor called the *current descriptor*. The current descriptor is visible to the driver software through APIC device registers, and it is initialized by the driver to be the first descriptor in a chain of descriptors for that channel. Whenever the driver has new buffers containing data that is to be transmitted, it sets up a descriptor for each such buffer and appends these descriptors to the tail end of the descriptor chain. The APIC transmits data from the buffer pointed to by the current descriptor (i.e., the *current buffer*) until there is no more data in that buffer. It then writes back the

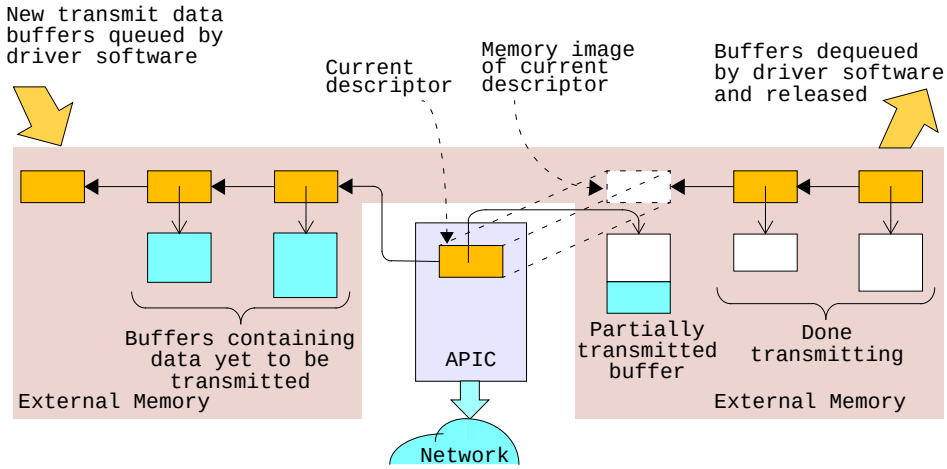

Figure 5.11: Transmitting Data Using a Descriptor Chain

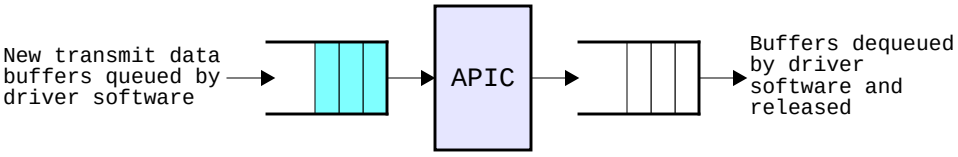

Figure 5.12: FIFO Queue Model for a Transmit Descriptor Chain

current descriptor to external memory, follows the link pointer to read the next descriptor in the chain, and then starts reading and transmitting data from the buffer pointed to by this new current descriptor. The driver can dequeue buffers that contain data which has already been transmitted from the head of the descriptor chain, and free or recycle them as needed. In Figure 5.11, the portion of the descriptor chain shown to the left of the APIC can be thought of as a FIFO queue that is the source of buffers containing data to be transmitted, while the part of the descriptor chain to the right of the APIC can be thought of as a FIFO queue that is the sink for buffers that have been used up (i.e., the data has already been transmitted). This is illustrated in Figure 5.12.

☞ Note that descriptor chains are simply an implementation technique for achieving the desired FIFO queue behavior. Other FIFO queue implementation techniques could also have been used. One popular alternative that is used in many NIC designs is to have an array of descriptors, with head and tail indices that wrap past the end of the array (i.e., circular arrays). One problem with this approach is that the size of the array needs to be fixed at some conservative number. A linked list, on the other hand, can be of variable length and can grow on demand.

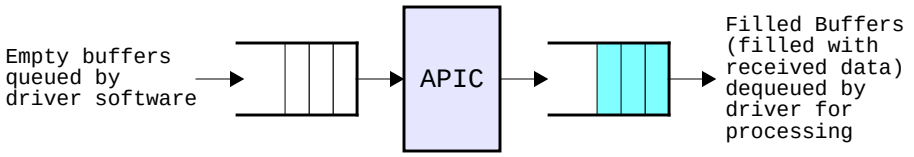

Figure 5.13: FIFO Queue Model for a Receive Descriptor Chain

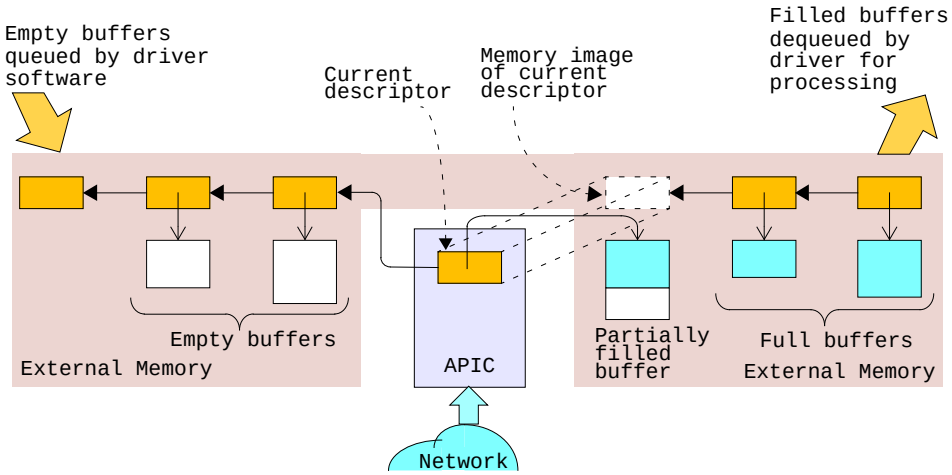

Figure 5.14: Receiving Data Using a Descriptor Chain

In the receive direction, descriptor chains can also be used in much the same way as in the transmit direction. The main difference is that in place of buffers containing data to be transmitted, the driver provides the APIC with empty buffers that have to be filled with received data. The desired FIFO queue model is shown in Figure 5.13, and its implementation using a descriptor chain is illustrated in Figure 5.14. Note the similarities and differences between Figures 5.11 and 5.14.

The APIC supports three different DMA modes: Simple DMA, Pool DMA, and Protected DMA. We now go on to describe these three mechanisms.

5.7.1. Simple DMA

Each channel that is configured to use *Simple DMA* is associated with a dedicated chain of descriptors, as shown in Figure 5.15. Since the APIC supports a maximum of 256 channels, there is on-chip storage for 256 current descriptors, one for each supported channel. Figure 5.16 shows the FIFO queue model that is supported by simple DMA.

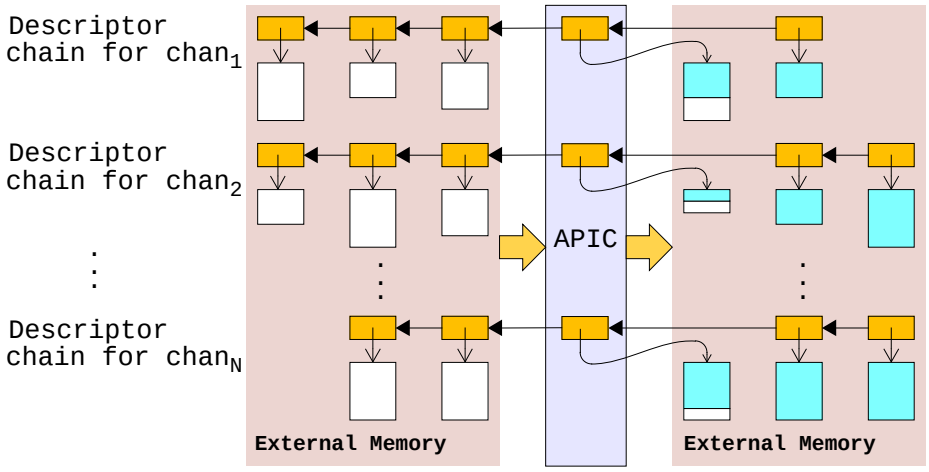

Figure 5.15: Illustration of Simple DMA

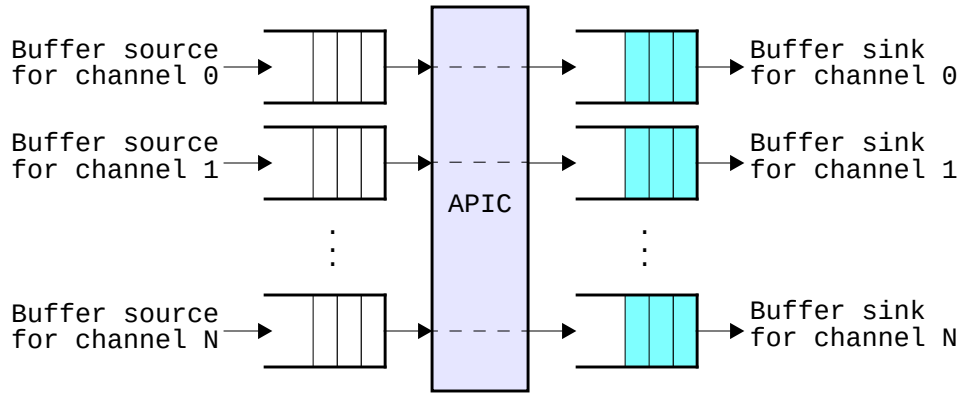

Figure 5.16: FIFO Queue Model for Simple DMA

Simple DMA can be used by kernel-resident (or trusted user-level server) protocol 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 otherwise accessible to the process). As mentioned earlier, the Protected DMA mode should be used for user-space driver implementations.

The APIC allows Simple DMA to be used in both the transmit and receive directions. However, Simple DMA can be very inefficient in its use of memory resources if it is used in the receive direction, especially if the number of open channels (and VCs) is large. This is because Simple DMA requires free (or empty) buffers to be dedicated to each open receive channel. This means that free buffers queued on one channel cannot be used by another. Since there is no way to anticipate when

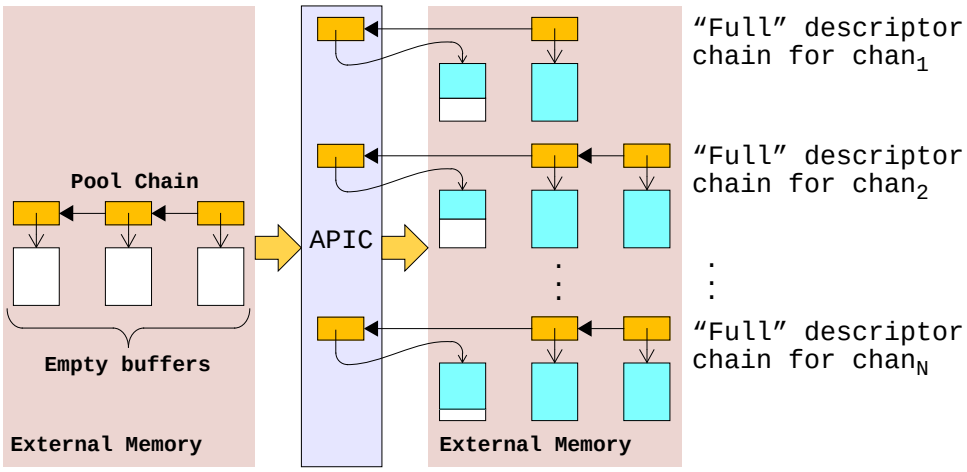

Figure 5.17: Illustration of Pool DMA

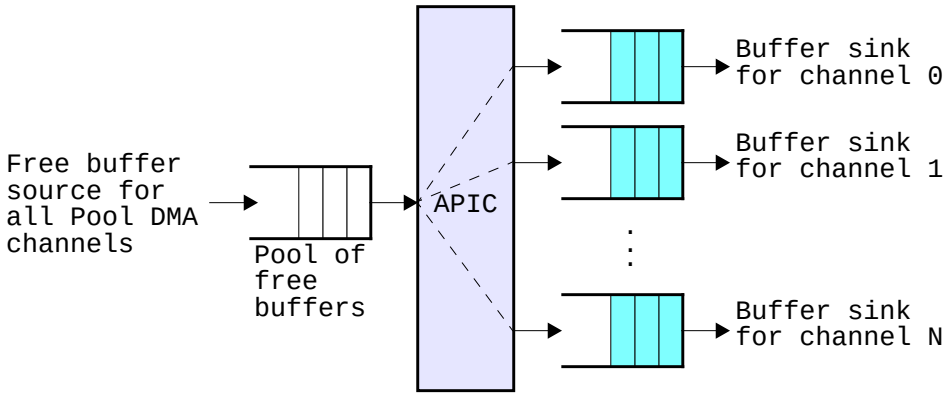

Figure 5.18: FIFO Queue Model for Pool DMA

data will arrive on a particular channel, the amount of buffer space needed for the free buffers can become very large. Pool DMA is designed to address this shortcoming of Simple DMA.

5.7.2. Pool DMA

Pool DMA allows a set of receive channels to all share the same pool of free buffers, as shown in Figure 5.17. The APIC fetches a new descriptor from a global *pool chain* whenever a channel needs an empty buffer to store newly arrived data. Before it writes the old “full” descriptor back however, the APIC fills in the next descriptor link field in that descriptor with the address of the new descriptor. This results in the demultiplexing of full buffers into separate chains of descriptors, one per channel. The FIFO queue model for Pool DMA is shown in Figure 5.18.

☞ It is important to keep in mind that while Simple DMA and Protected DMA modes can be used for both transmit and receive channels, Pool DMA can be used only by receive channels; it does not make sense and cannot be used for transmit channels.

Although we showed only a single pool chain for illustration purposes, the APIC actually supports a total of four (globally shared) pool chains. For each channel, it is possible to specify which pool chain to use. It is also possible to have data from resource management (RM) and operations and maintenance (OAM) cells deposited into buffers drawn from a pool chain that is different from the one used for the rest of the data. Also, as we will see later, it is possible to split frames into header, data, and trailer portions, and to use buffers drawn from different pool chains for these three portions of a frame.

5.7.3. Protected DMA

When we described the user-space control model, we mentioned that user-space drivers need to use the Protected DMA mode supported by the APIC. Protected DMA is in many ways similar to Simple DMA, but it enables certain protection checks to be carried out by the APIC on buffers and descriptors that are enqueued by untrusted code running in user-space. To enable this kind of checking, Protected DMA defines two types of descriptors: *kernel descriptors* and *user-descriptors*. Instead of having a single descriptor referencing each buffer, Protected DMA requires a pair of descriptors to refer to each buffer. One descriptor in the pair is a kernel descriptor, while another is a user descriptor. This is illustrated in Figure 5.19. Before the APIC can use a buffer from a Protected DMA channel, it always reads both descriptors for the buffer. The user descriptor contains values supplied by the user-space driver, and therefore cannot be trusted. The kernel descriptor on the other hand is initialized by the kernel driver and is not accessible to the user-space driver, so the values in a kernel descriptor are trustworthy. The APIC performs its protection checks by comparing values in the kernel descriptor against corresponding values in the user descriptor. The kernel descriptor is therefore used to certify the values supplied by the user-space driver in the user descriptor. Only if this validation succeeds can the APIC safely DMA data to or from the associated buffer.

It is important to recognize that the FIFO queue model implied by Protected DMA is identical to that for Simple DMA (see Figure 5.16). Both associate a single dedicated descriptor chain for

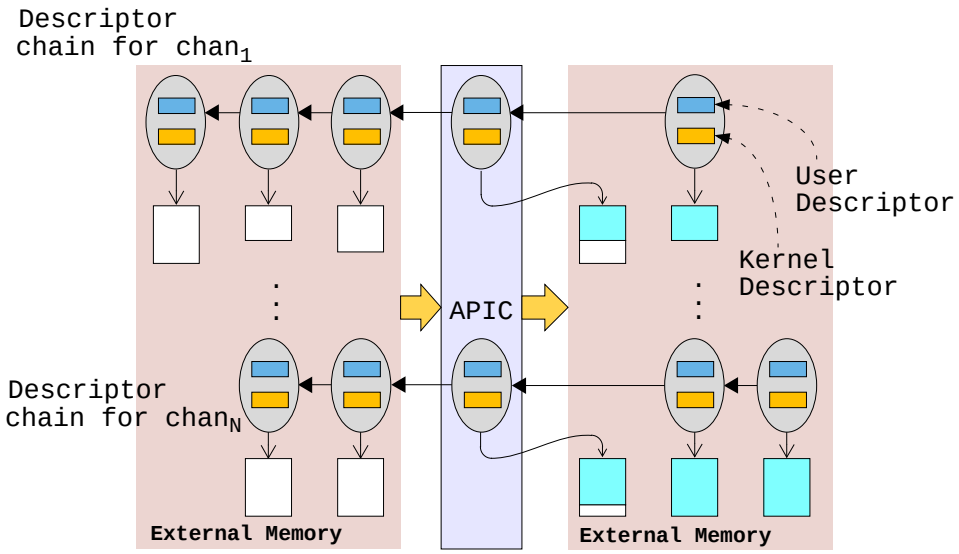

Figure 5.19: Illustration of Protected DMA

each channel. The difference is only in way the FIFO queues are implemented: in the Simple DMA case, there is a single descriptor per buffer and no protection checks made by the APIC, while for Protected DMA there is a pair of descriptors for each buffer, and implied protection checks.

To use Protected DMA, the user-space driver is required to allocate special communication buffers (using `malloc` or a similar utility), to wire these in memory (using a system call like BSD `mlock`), and to make a system call to the kernel driver to associate descriptor pair(s) with the buffer and initialize the kernel descriptor(s). These are all assumed to be control path operations, and should not need to be done very frequently. In fact, in most cases it may be sufficient to perform these actions for a set of communication buffers just once when the program is started. Note that a single communication buffer may span multiple non-contiguous physical memory pages, so it may be necessary to associate multiple descriptor pairs with the buffer (one pair for each page in the buffer).

Once the afore-mentioned control path operations are completed, the user-space driver is free to create chains of descriptor pairs pointing to the communication buffers by modifying fields in user descriptors only. It can enqueue portions of these buffers in any desired order onto the protected DMA channel's descriptor chain. No system calls are needed for these data path operations. The protection checks performed by the APIC ensure that accesses to illegal descriptors or buffers will be rejected. For each descriptor pair that is read by the APIC, two types of checks are made:

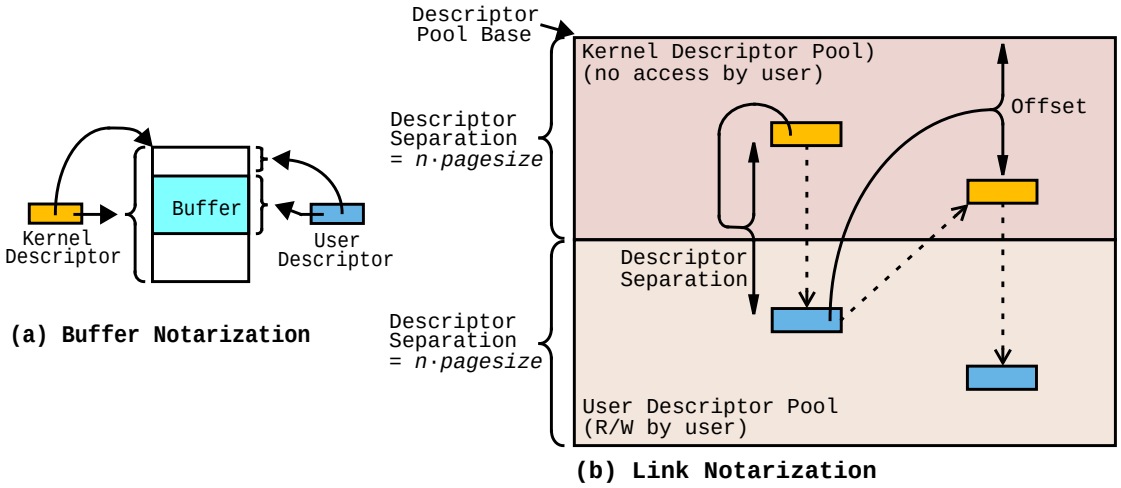

Figure 5.20: Notarization for Protected DMA

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 5.20(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*.
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 5.20(b) shows how link notarization works. Every protected DMA channel 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 channel. 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·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 a descriptor pair fails either notarization check, 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 descriptor pair passes notarization, the APIC creates a new *resultant descriptor* by combining values from the two descriptors (in the expected manner), and uses this resultant descriptor is then just like a normal (simple DMA) descriptor. 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.

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 a *channel 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.

Note that protected DMA requires data buffer pages private to an application to be wired in memory. While this is routinely done for kernel buffers, it can result in significantly poorer

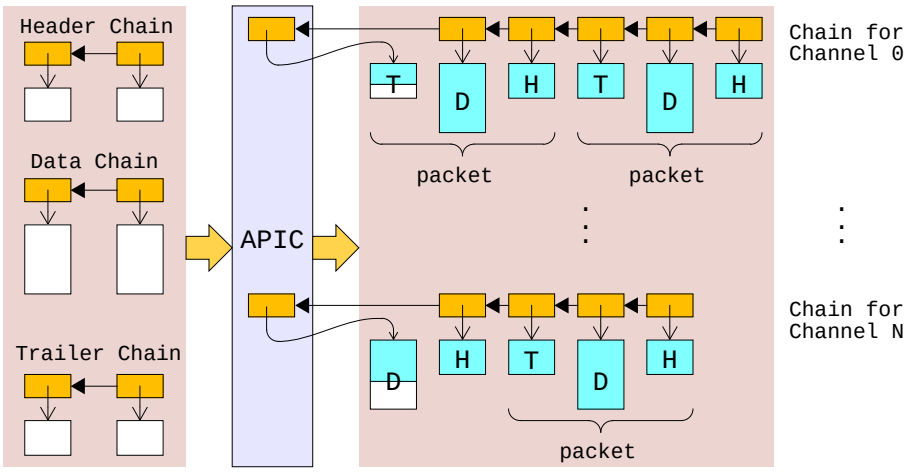

Figure 5.21: Pool DMA with Packet Splitting

utilization of memory resources if many user processes lock down large pools of physical memory for their own exclusive use. This is one of the drawbacks of using the protected DMA scheme for achieving zero-copy semantics. We now describe an alternate scheme called “packet splitting” which does not have this limitation, but can still support zero-copy semantics; however, this alternate scheme requires a kernel implementation and does not allow for direct control of the adapter from user-space.

5.7.4. Packet Splitting

For AAL-5 receive channels, the APIC includes support for a novel feature called *packet splitting*. As we shall see, this feature is useful in implementing zero-copy protocol stacks, and in distrbuted shared memory (DSM) implementations.

In normal operation (packet splitting turned off), the APIC completely fills buffers with data received on a channel before getting a new buffer from the descriptor chain. The only exceptions to this are when the end of an AAL-5 frame is reached, or when the type of data received on the channel changes (for example, if an OAM or RM cell is received). In these two cases, the APIC switches to a new buffer early, regardless of whether the current buffer has been completely filled or not. Note that the new buffer may be drawn from the channel’s dedicated chain if Simple DMA or Protected DMA is being used, and from one of the shared pool chains if Pool DMA is being used. Also note that an AAL-5 frame can span multiple buffers, but a single buffer cannot contain data from two or more frames.

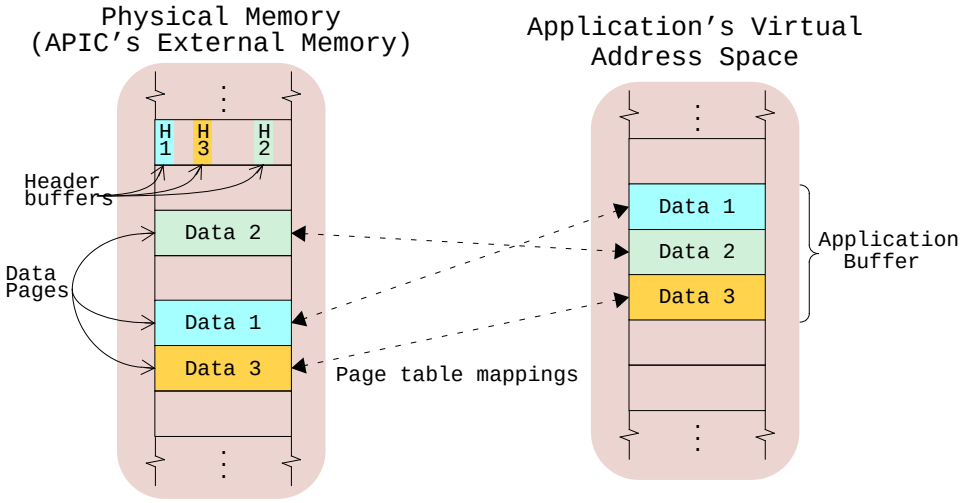

Figure 5.22: Zero-Copy Using Packet Splitting and Page Remapping

When packet splitting is enabled on a channel, the APIC will split a received AAL-5 frame into a header portion, a data portion, and a trailer portion, and write these portions into different buffers. In other words, the moment the APIC detects a header-data boundary or a data-trailer boundary in a received frame, it will switch to a new buffer. In the case of Simple DMA and Protected DMA, (as always) this new buffer is drawn from the channel's dedicated descriptor chain. For Pool DMA however, the shared pool chain that is to be used for each portion of the packet can be separately configured. Thus, for example, we could have all packet headers and trailers from arriving frames being deposited in buffers drawn from pool chain 0, while the data portions of the frames are deposited in buffers drawn from pool chain 1. In the most general case, we could use completely different pool chains for headers, data, and trailers. This is illustrated in Figure 5.21, where we have chosen to name the separate pool chains based on the parts of arrving frames that will be deposited in the corresponding buffers; thus, we have a header chain, a data chain, and a trailer chain.

How do we define the header, data, and trailer portions of an AAL-5 frame? In other words, how does the APIC detect header-data and data-trailer boundaries in an AAL-5 frame? First, we assume that the only trailer information in a frame is the AAL-5 trailer itself. Since the AAL-5 trailer contains a length field, that can be used to find the location of the data-trailer split. We assume that the header contains all protocol header information: this includes the network and transport layer headers. The APIC cannot automatically find the location of the header-data split,

so a header length value needs to be configured. This value can be independently specified for each channel, which means that the header length needs to be fixed for the lifetime of a connection.

☞ Even with variable length headers (eg., TCP/IP), the header length does not usually change during a connection's lifetime. The header length can either be negotiated during the connection setup phase, or the value from the first data packet can be used if we assume that the protocol software is written in such a way that it can recover in case the header length suddenly changes.

When used with Pool DMA, packet splitting provides a powerful technique for implementing zero-copy protocol stacks, and for efficient implementations of distributed shared memory (DSM). Figure 5.22 demonstrates how Pool DMA with packet splitting can be used in conjunction with the host's virtual memory page mapping hardware to implement a zero-copy kernel-resident protocol stack. We make the assumption that the protocol will use packets that carry enough data to fill one (or an integral multiple) of the receiving host's pages. This can be a parameter negotiated at transport protocol connection setup time. The left part of the figure shows physical memory into which the APIC DMAs received data. Frames received by the APIC are split into headers, data, and trailers: the headers and trailers can be written into small buffers drawn from one pool chain, while the data can be deposited into empty pages that are drawn from a different pool chain. Since the data part of the packets are assumed to be a multiple of the receiving host's page size, each frame will fill one or more pages completely with received data; no page will be left partially filled. As packets are received, the appropriate transport protocol can examine the header buffers and use the sequence number information contained therein to determine where the corresponding data pages should reside in the application's virtual address space. These protocols can then proceed to modify the virtual memory page table entries appropriately, so that the pages containing received data now appear contiguous in the application's address space, even though they may be discontiguous in physical memory. This avoids the need for copying data into the user's buffer. It is important to note that this scheme works even in the presence of network errors or loss: as and when a retransmitted packet is received correctly, its corresponding data pages can be mapped into the application's VA space at the appropriate location, thereby filling up any holes in the sequence number space that were caused because of lost packets.

☞ The cost of modifying pages tables can be quite high in some architectures. This overhead can usually 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*.

Above, we described the operation of a zero-copy protocol stack only in the receive direction. This is because zero-copy on the transmit side is trivially achieved by directly queueing application buffers, interspersed with kernel buffers containing headers and trailers, to the APIC on a Simple DMA channel. No page remapping is necessary, but it is important to wire the application buffer pages in main memory before they can be queued for transmission.

5.8. Interrupt Mechanisms

Interrupts are issued by the APIC to report the occurence of asynchronous events, such as completion of transmission and reception of a frame, or an error condition. The APIC includes three different mechanisms that are designed to improve interrupt response by reducing the frequency of interrupts, and the overheads associated with servicing an interrupt. In particular, Interrupt Demultiplexing, described below, can be used to address the receive livelock problem that plagues most high performance network interfaces.

5.8.1. Interrupt Demultiplexing

Because of the high overhead associated with servicing an interrupt, it is important to minimize the frequency at which interrupts are issued to the processor. There are three main components contributing to this overhead: the time taken by the kernel to field the interrupt and call the device driver's interrupt service routine (ISR), the time taken to process the interrupt in the ISR, and the indirect performance hit that results from cached data belonging to the current context being replaced in cache by new data as a result of servicing the interrupt. As mentioned in Chapter 3, the research community's efforts to reduce interrupt overhead have been targeted at achieving batching of multiple event notifications into a single interrupt, thereby amortizing the cost of an interrupt over many different events. Such event batching mechanisms work well for high bandwidth applications, which are usually not very sensitive to delay. For delay-sensitive applications however, such mechanisms can add significantly to end-to-end latency, because blind batching of events necessarily makes some events wait longer before they are reported to the processor, even if those events have

a greater urgency than other pending events. This can prove to be a problem in a mixed environment where both high-bandwidth and latency-sensitive applications have to coexist.

The APIC incorporates a mechanism called *interrupt demultiplexing*, which attempts to satisfy the conflicting requirements of both bandwidth-intensive and delay-sensitive applications, by adopting different event batching policies for different connections. Thus, the driver can choose to batch events for high bandwidth applications, but avoid batching and immediately deliver event notifications for latency-sensitive applications.

In the APIC, interrupt demultiplexing is implemented by including a one-bit flag in the state for every channel. This bit flag serves as a channel-specific *interrupt enable*: only if it is set will interrupts be issued in response to packet arrival (or completion of packet transmission) on the corresponding channel. The flag is initially set by the driver. When an interrupt is issued in response to an event on that channel, the APIC automatically disables more interrupts from occuring for that channel by clearing the bit, which remains cleared until the driver sets it again at some future time. The driver would usually not re-enable interrupts for that channel until it has finished completely processing all pending packets for the channel. In the meantime however, it may field interrupts for more urgent events, such as packet arrivals on latency-sensitive channels, impending underflow of a receive descriptor chain, or an error event.

If the frequency of interrupts is too high, then regardless of how efficiently we can batch interrupt events, the CPU will spend all its time servicing interrupts. This happens because with the way operating systems and protocol stacks are currently structured, interrupt processing has a higher priority than either protocol processing or application processing. Thus, all of the processing power would be used only for driver processing and interrupt overhead, and the protocols and applications would never get an opportunity to consume the received data. As described in Chapter 2, this condition is called receive livelock.

The interrupt demultiplexing feature can be used to delay the onset of livelock by a simple restructuring of the protocol stack architecture in BSD-like operating system architectures. This is illustrated in Figure 5.23. Part (a) of the figure shows the traditional way in which interrupts are handled: when packets are received on the interface, the interrupt service routine (ISR) is only responsible for taking the packet off the device's receive queue and placing it on a protocol input queue

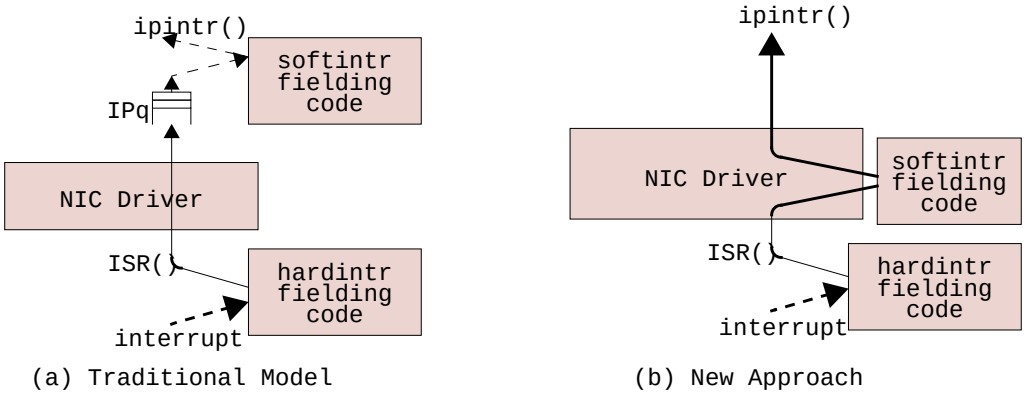

Figure 5.23: A Different Way of Structuring a NIC Driver

(e.g., the IP input queue for internet packets); before the ISR returns, it acknowledges the hardware interrupt, and schedules a software interrupt which will run at a later point in time, and will be responsible for performing protocol processing on the packet and for delivering it into the application's socket buffer. Because the software interrupt runs at a lower priority and can therefore be preempted by a hardware interrups, we can have a condition of receive livelock where if the rate of packet arrivals is high enough, most of the time is spent in the hardware interrupt service routine, and the software interrupt (and also the application itself) does not get an opportunity to do protocol processing on the received packets.

Part (b) of the Figure 5.23 shows how a simple restructuring of the driver, in conjunction with interrupt demultiplexing, can help with this situation. The ISR, instead of taking packets off the hardware receive queue, simply schedules a software interrupt, acknowledges the interrupt, and returns. Recall that with interrupt demultiplexing, the APIC automatically disables future interrupts from the channel on which the packet was received. The software interrupt is now responsible for pulling packets off the device receive queue, and directly making an upcall to the appropriate upper layer protocol to perform protocol processing and deliver the packet to the socket layer. The software interrupt can repeat this procedure for all pending packets received from that channel, before it re-enables interrupts on that channel. Notice that because of interrupt demultiplexing, the driver can continue to receive interrupts on other unrelated channels which may require more urgent service. This kind of structuring has the desirable effect that, even with a very high rate of packet arrival, the processor gets an opportunity to do protocol processing of all received packets. This is because protocol and driver processing now occur at the same priority, and future interrupts are

locked out (for the same channel) while the protocol processing is taking place. Thus, a lot more useful work is getting done than in the traditional architecture where only the ISR was getting an opportunity to run.

It is important to note that the scheme described cannot prevent livelock, only delay its onset. This is because although the protocol processing and driver interrupt handling are now occuring at the same priority, the application is still at a lower priority. Thus, with a high enough rate of arrival of packets, the application will not get a chance to consume the received data, resulting in livelock. It is also important to note that this scheme can only work if the device does not have scarce resources that can run dry while waiting for higher layer (protocol) processing to occur. In the context of the APIC, this is true because an APIC NIC does not have any on-board memory that can be quickly depleted with rapidly arriving packets. This could be used as an argument for using the main memory for all of the device's queues.

At this point, it should be clear that the real solution to solving the livelock problem is to make *everything* (driver processing, protocol processing, and application processing) in the data path happen at the same priority, thereby removing the possibility of one type of processing from locking out another. With conventional protocol stack structuring, this is not possible. However, if we bring the user-space control model into the picture, we have a complete solution. Because in that model the driver, protocols, and application all run in user space for data path processing, they all have the same priority, which is determined by the operating system's scheduler. Thus, there is no potential for any one type of processing to consume all CPU resources. Note however that this approach is practical only when used together with interrupt demultiplexing. Otherwise, one application disabling interrupts would result in interrupts being disabled for all other applications, effectively locking those applications out of using the network interface.

We now describe how we envision a complete solution to the problem of receive livelock would work in the context of the APIC. The driver's hardware interrupt service routine would have only one task: discover which channel the interrupt event is for by querying the APIC (using the notification list mechanism described below in Section 5.8.3.), find the application process responsible for that channel, and wake up the process if it is currently sleeping. The application process implements the user-space control model, using Protected DMA to directly move packets to and from the network; it is assumed to be in a continuous loop pulling packets off the network interface, and

doing protocol and application processing on those packets. As part of this loop it may also be talking to other devices, sending and receiving data. If the process runs out of work to do, it enables interrupts for the channel by using protected I/O to write to an APIC on-chip register, and then goes to sleep waiting on an event on a special file descriptor that refers to the appropriate receive channel on the APIC. If multiple devices or channels are involved, the process could wait on an event from any of them by doing a select system call on the set of file descriptors referring to those devices or channels. Because interrupts have been enabled, arrival of new data on a channel will cause an interrupt that will result in the process getting woken up. And because of the way interrupt demultiplexing works, all future interrupts will have been disabled on that channel until the process decides to re-enable them before it goes to sleep again in the future. Note that because of interrupt demultiplexing, a process servicing a latency sensitive application will get woken up and notified of an event on the corresponding channel even if a bandwidth-intensive application currently has interrupts disabled for its channel. Also note that the CPU scheduler now has full control over how processing resources are handed out to different applications, because all types of processing is occuring in the context of the corresponding user process. And, of course, interrupt livelock is impossible because there everything runs at the same priority within a user process.

5.8.2. Orchestrated Interrupts

The APIC also supports the concept of *orchestrated interrupts*, which are interrupts that are issued in response to an event that is expected and to which the processor assigns special significance. In the APIC context, this manifests itself as interrupts that would be issued when the APIC reads in a specially marked descriptor (i.e., a bit flag is set in the descriptor). This can be very useful in two situations. First, in the transmit direction, it can be used to signal an interrupt upon completion of transmission of a batch of packets: the last descriptor corresponding to the last packet that has been queued for transmission would be marked so that the APIC interrupts when it reaches that descriptor. Second, in the receive direction, orchestrated interrupts can be used to notify the processor when a descriptor chain underflow is imminent. For example, with Pool DMA, the driver can arrange for the APIC to issue an interrupt when it is close to running out of free buffers from a pool chain: this is signalled by a descriptor that is close to the end of the pool chain, and has been specially marked by the driver to cause the interrupt. The ISR would recognize this type of interrupt

and take appropriate action, which in this case would involve replenishing the pool chain with new free buffers.

5.8.3. Notification Lists

While interrupt demultiplexing and orchestrated interrupts serve to reduce the frequency of interrupts, they do not have any effect on the time taken to actually service an interrupt. The APIC includes an event reporting mechanism called *notification lists* to address this issue. One of the primary tasks of the ISR is to discover activity on various channels, and take appropriate action. For a latency-sensitive application, this may involve immediately processing data that has arrived on a channel, while for a high bandwidth application, the correct action would probably be to postpone processing data from the channel to a software interrupt. For a protected DMA channel, it may be necessary to wake up the corresponding user process if it is sleeping awaiting an event (eg., data arrival) on the channel. Clearly, it is important for the driver to be able to quickly discover what kinds of interrupt events have occured, and the channels that caused those events to occur. It is a high overhead proposition to poll every channel's registers to discover activity on the channels each time an interrupt is serviced. To avoid this, the APIC keeps track of the set of all channels that have had occurences of interrupt-related events (we call such channels *active channels*), and makes this set available to the controlling processor in the form of a list called the notification list. Every entry in the notification list contains the channel ID of an active channel, and a bit vector of the different kinds of events that have occured on that channel. The driver accesses this list by repeatedly reading a special APIC device register called the *notification register*. Each time this register is read, a new entry from the notification list is returned, and that entry is deleted from the list. When there are no more entries on the list, a value of zero is returned.

5.9. Miscellaneous Features

5.9.1. TCP Checksum Assist

As mentioned earlier, the APIC provides hardware assistance to the software for computing the TCP checksum for AAL-5 receive channels. Recall that because the APIC is a cut-through adapter, it is not possible to provide TCP checksum assist in the transmit direction (because the checksum field resides in the TCP header); however, on transmit the checksum computation can easily be rolled into the application-to-kernel copy loop.

How can the APIC compute the TCP checksum, given that it has no knowledge of TCP or IP packet formats? In other words, how can it know which parts of an AAL-5 frame correspond to a TCP packet? The answer to this question is that the APIC only provides assistance to the software in computing the TCP checksum; it doesn't calculate the checksum itself. This assistance is provided in the form of a checksum value that has been computed using the TCP checksum algorithm over entire AAL-5 frames, rather than just the portion of the frames corresponding to TCP packets. This value is made available to the software by writing it into the last descriptor for the frame. The software is then responsible for using this value to compute the actual TCP checksum over the packet. This is accomplished by computing the checksum over portions of the frame that are not part of the TCP packet, and "subtracting" the result from the checksum value over the entire frame (which has been computed in hardware by the APIC). Note that this "subtraction" operation is possible because the TCP checksum operation is both commutative and associative. Since the part of an AAL-5 frame that is not part of the corresponding TCP packet is usually quite small (comprising the AAL-5 trailer, some padding bytes, some IP header fields, and possibly an LLC/SNAP header) relative to the size of the TCP packet, the overhead of verifying the checksum in software is small compared to an all-software implementation of the TCP checksum.

5.9.2. Flow Control

In local and desk-area network environments, the APIC can operate in a practically loss-free manner through the use of one or both of two featured flow control mechanisms. These mechanisms are responsible for deasserting a *flow control grant* to upstream devices connected to both ATM ports, if the APIC finds that its on-chip cell store memory is close to full (the occupancy threshold is a configurable parameter called the *flow threshold*). An upstream device may choose to ignore the flow control signal, and continue sending cells (as it would if it were an ATM switch). However, if the upstream device is also an APIC chip (i.e., we are operating in a daisy chain environment), then it would respond to the flow control grant being deasserted by stopping further transmission of cells on the link. The APIC implements two kinds of flow control: a hardware-level flow control as defined by the UTOPIA specification, and a generic flow control (GFC) that works at the ATM layer.

UTOPIA Flow Control

The UTOPIA specification includes a special signal wire that can be used to assert flow control. This kind of flow control only works if the upstream device is connected to the APIC directly over UTOPIA. It cannot be used if there is an optical fiber connection between the APIC and its upstream device. Therefore, the hardware UTOPIA flow control feature is useful only in the following scenarios:

1. When the downstream device is an off-the-shelf optical link interface chip that uses the UTOPIA flow control signal to tell its upstream device (the APIC) to stop sending more data.
2. When multiple APICs are connected together on the same board, or by ribbon cable, without any intervening optical fiber.

Generic Flow Control (GFC)

When two APIC chips are connected by optical fiber carrying only ATM cells, the UTOPIA flow control mechanism cannot be used because the corresponding UTOPIA grant signal is not propagated over optical fiber. To address this problem, the APIC includes a second mechanism to assert flow control grants to an upstream APIC, which does not make use of any special hardware signal. This mechanism, which we call *Generic Flow Control (GFC)*, has to be specially enabled by a configuration pin on the chip. When GFC is enabled, the APIC includes a bit in the GFC field of every cell sent to the upstream APIC which signals to that APIC whether the flow control grant (for the corresponding downstream link) is asserted or not. If there are no cells to be sent to the upstream APIC, the APIC sends cells anyway on a specially designated *flow control VC*; the upstream APIC knows to extract the flow control bit from cells received on this VC and then discard these cells. Note that the GFC feature depends on the fact that ATM links usually come in pairs to support bidirectional data transfer: therefore, cells going in one direction can carry flow control information for cells flowing in the opposite direction.

5.9.3. Cache Coherent Bus Transfers

The APIC interfaces to the system processor over a PCI bus. All DMA transfers between the APIC and the system's main memory usually occur through a bridge chipset which is responsible for switching data between processors, memory, and devices on the PCI bus. Since the processor

caches work in cache-line sized blocks of data, the bridge must be able to handle partial reads and writes to cache-lines, while ensuring consistency of caches with contents in memory. For DMA transfers to and from memory, the APIC implements two special PCI bus transaction types that are intended to improve efficiency of large transfers of bulk data:

Memory Read Multiple: Under normal circumstances, if a PCI device wants to read data from a memory location, it would need to issue a vanilla "*Memory Read*" command. However, many bridge chipsets are designed to achieve maximum performance for large sequential transfers if they are allowed to prefetch one or more cache-lines, the intent being to keep their internal pipeline buffers full. The APIC uses a special PCI transaction, "*Memory Read Multiple*," which provides a hint to the bridge that the APIC intends to fetch more than one cache-line worth of data, and therefore it is acceptable to do read-ahead of cache lines for improved performance. The APIC uses this special transaction even if it does not intend to transfer a lot of data; this may have the effect of making some smaller transactions less efficient.

Memory Write Invalidate: Under normal circumstances, if a PCI device wants to write data to a memory location, it would need to issue a "*Memory Write*" command. If the word(s) being written to are part of a cache-line that is currently located in the processor's write-back cache, and if that cache-line is marked as dirty (i.e., it has been modified by the processor), then in order to maintain cache coherency the bridge chipset would need to read the cache-line from the processor cache (while simultaneously invalidating it), make changes to the appropriate word(s) being modified by the device, and then write the cache-line back to memory. If however, the bridge knew that the entire cache-line was going to be written to by the device, then it could simply invalidate the corresponding processor cache entry and write the new cache-line contents directly to memory. The APIC breaks all writes over the bus into an initial portion which is part of a cache line, followed by one or more complete cache lines, followed by a final portion which again is only part of a cache line. For the initial and final part-cache line transfers, the APIC uses the normal *Memory Write* transaction type. However, for the part in the middle which consists of whole cache-lines, the APIC uses a special PCI transaction type called "*Memory Write Invalidate*," which allows the bridge chipset to carry out the optimization described above. Use of *Memory Write Invalidate* transactions can significantly improve burst write speed over the PCI bus for most types of bridge chipsets.

Chapter 6

Internal Design of the APIC Chip

This chapter describes the functioning of the internals of the APIC. A functional block diagram of the chip is shown in Figure 6.1. Paths taken by data through the chip are shown using heavy arrows, while control paths are shown using thin arrows. Several control paths through the chip have been omitted from the figure for clarity — they will be introduced as necessary in the process of describing the operation of the chip.

Before we describe the operation of each module in the chip, identify the two input UTOPIA ATM ports on the top left corner of the figure, the two output UTOPIA ATM ports on the top right corner, and the PCI bus port at the bottom of the figure. Along with a few configuration pins, these ports represent all of the signal pins presented by the chip to the outside world. We number the two ATM ports as *port 0* and *port 1*, and the bidirectional bus port as *port 2*. Taken together, input port 0 and output port 0 comprise one bidirectional ATM port pair, while input port 1 and output port 1 comprise another. In a daisy-chained DAN interconnect, each port pair is connected to another APIC, or to an optical link interface device.

☞ It is possible for input port 0 and output port 0 to connect to different devices (and similarly for input port 1 and output port 1). An example is the perfect shuffle topology shown in Figure 5.3. One of the repercussions of doing this is that the GFC flow control feature of the APIC can no longer be used, since that feature assumes a bidirectional ATM link between pairs of APICs. Another side effect relates to the routing of cells on the transit path; this will be discussed below.

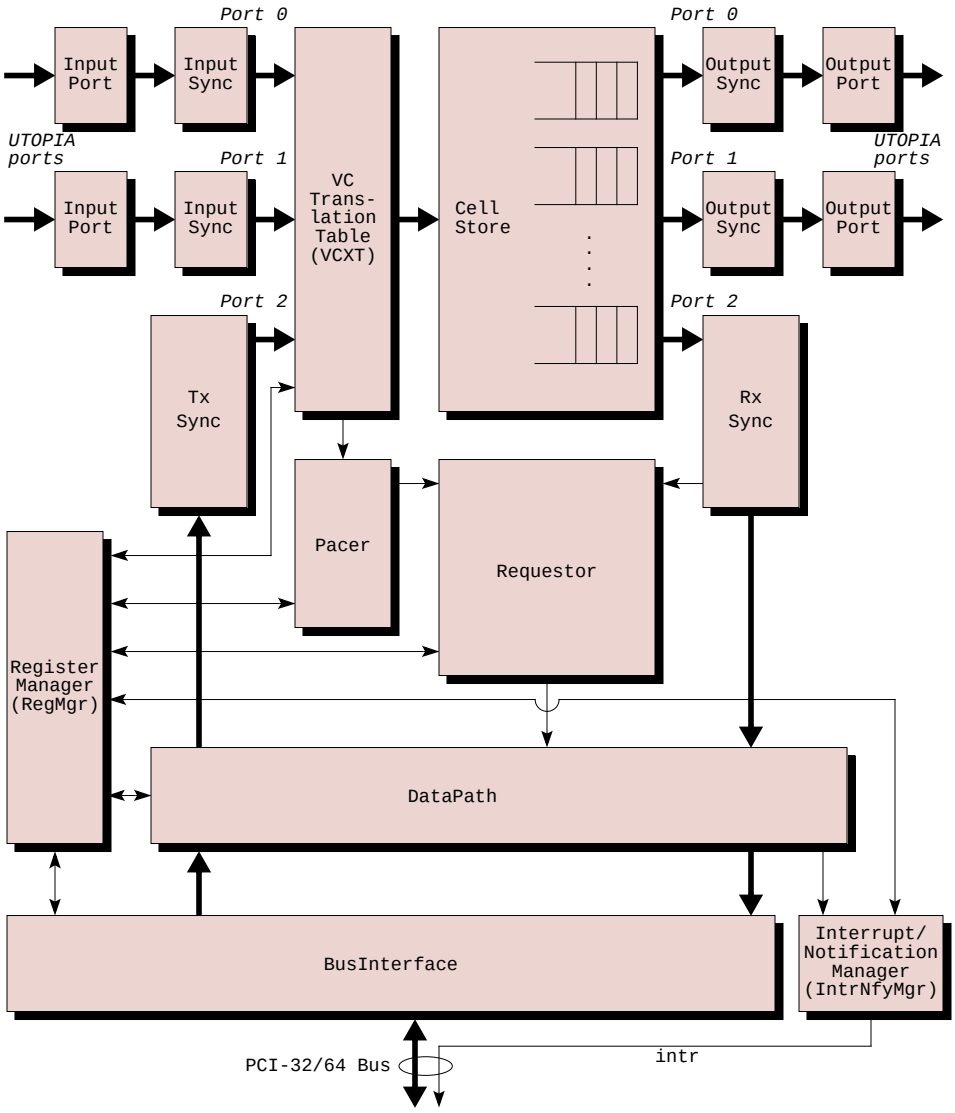

Figure 6.1: Functional Block Diagram of APIC Internals

6.1. Clock Regimes

The chip operates in six different clock regimes, shown as shaded areas in Figure 6.2. Each of the two input and two output ATM ports operates with its own separate clock (regimes A, B, C, and D in the figure). Clock regime E is clocked from the PCI bus, which is nominally 33 MHz but can be lower in some machines. Clock regime F in the figure is known as the APIC's *internal clock* domain. The frequency of this clock is fixed at 85 MHz to enable operation of all ATM ports at the maximum possible rate (1.2 Gb/s), regardless of the clock rates for individual ports or for the PCI bus.

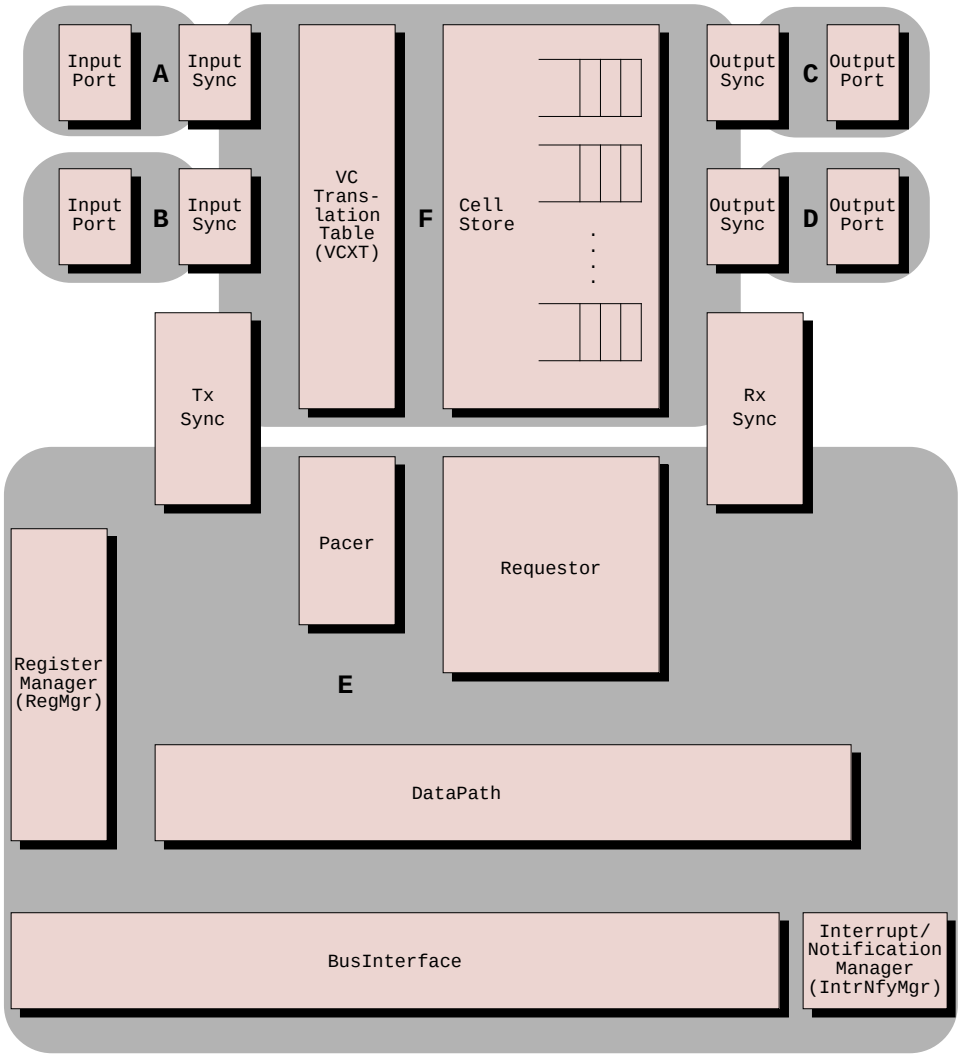

Figure 6.2: APIC Clock Regimes

6.2. Module Functions and Paths Taken by Cells Through the Chip

6.2.1. Synchronization Modules

For every signal that passes from one clock regime to another, there needs to be synchronization logic that is responsible for meta-stability resolution. Along the data paths, this synchronization is handled by the *InputSync*, *OutputSync*, *TxSync*, and *RxSync* modules. Notice that each of these modules straddle two different clock regimes.

6.2.2. Input and Output Ports

Returning to Figure 6.1, the *InputPort* and *OutputPort* modules are responsible for implementing the UTOPIA interface protocol. The InputPorts also verify the header error check (HEC) of incoming cells, and discard the cell if it is incorrect. If the HEC verifies correctly, the HEC byte is stripped and the cell is passed on to the corresponding InputSync module. The OutputPorts are responsible for computing and inserting the HEC byte in all cells before they are output on the link. Finally, the InputPort and OutputPort modules are responsible for generating and reacting to flow control signals (both UTOPIA and GFC flow control).

6.2.3. BusInterface

The *BusInterface* module implements the PCI bus protocol. It implements registers for the PCI configutation space, and handles slave accesses to these configuration registers internally. When the APIC is the target of a memory-mapped slave access, it forwards the request to the *RegisterManager* (aka *RegMgr*) module, which is responsible for handling the corresponding register access. The BusInterface module also implements bus mastering: it accepts read and write transaction requests from the *DataPath* module, arbitrates for the bus, and completes the specified transaction once it is granted ownership of the bus (i.e., when it becomes bus master).

6.2.4. RegisterManager

As mentioned above, the *RegisterManager* module is responsible for handling accesses to all on-chip control/status registers (except for PCI configuration registers, which are handled internally by the BusInterface module). The operation of the RegisterManager is illustrated in Figure 6.3. Register accesses can originate from one of two places: from the BusInterface module if the register is being accessed using memory-mapped I/O on the PCI bus (along control path ① in the figure), and from the DataPath module if the register is being accessed using a control cell (control path ②). The registers themselves are distributed throughout the chip in five different modules (shown with a deeper shade in the figure): the RegisterManager itself, the VC Translation Table ③, the Pacer ④, the Requestor ⑤, and the Interrupt/Notification Manager ⑥. These are the only programmable modules in the APIC; all other modules can be considered to be “dumb”. The RegisterManager forwards register access requests from the BusInterface or DataPath to the appropriate target module, using the address of the register that is being accessed as a demultiplexing key.

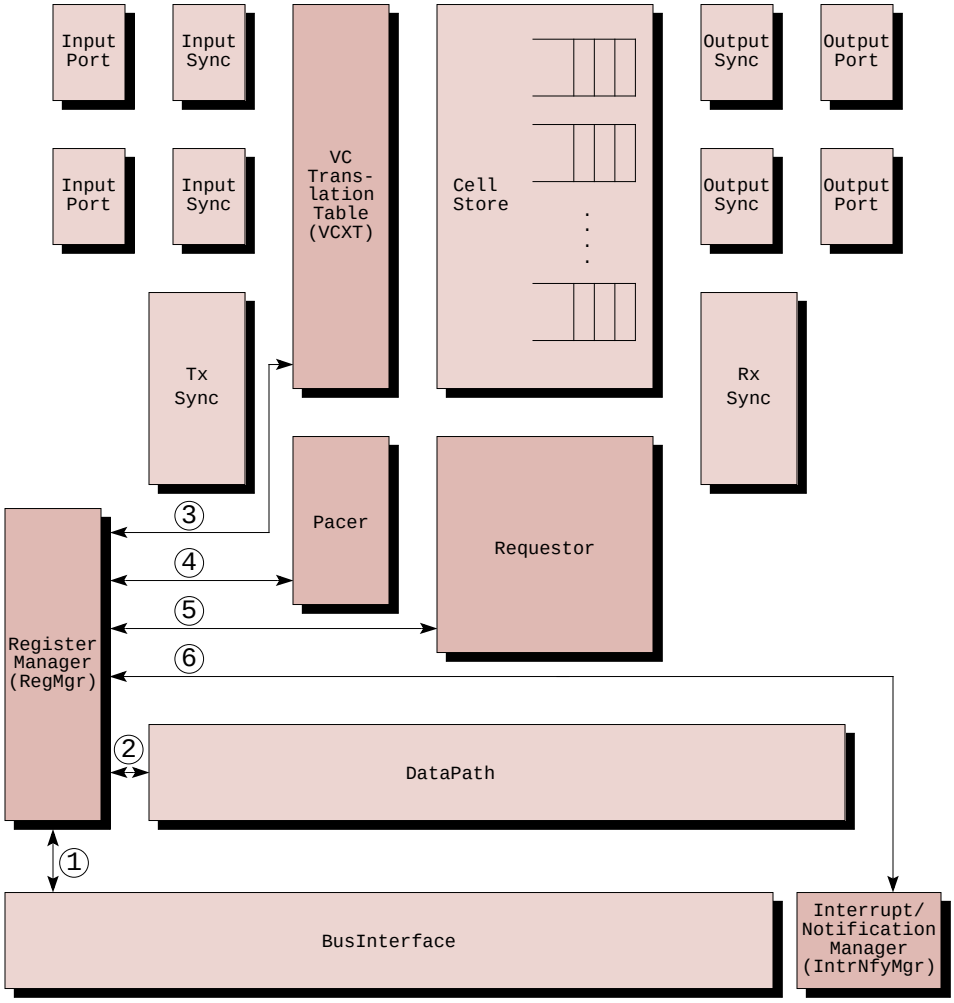

Figure 6.3: Operation of the RegisterManager Module

Transit Path

It is easiest to explain the operation of the remaining modules in the design by tracing the various paths taken by data through the chip. In Figure 6.4, we show the transit path. To simplify the figure, we have omitted several modules that are not involved in the transit path (i.e., all modules in the bus clock regime). The red arrow traces the path taken by a transit cell received at input port 0.

6.2.5. VCXT

The *VC Translation Table* module, which we will henceforth refer to as the *VCXT,* holds a table containing entries for all open receive VCs; this table is filled in by the APIC's controlling processor

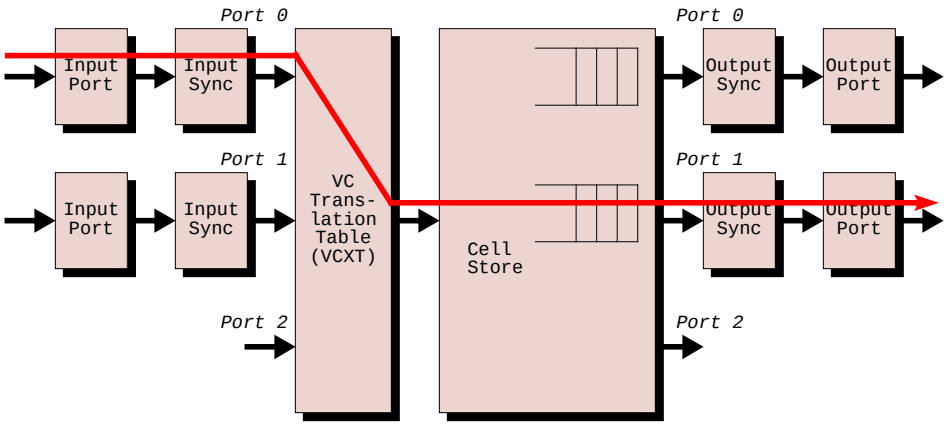

Figure 6.4: Transit Path Forwarding

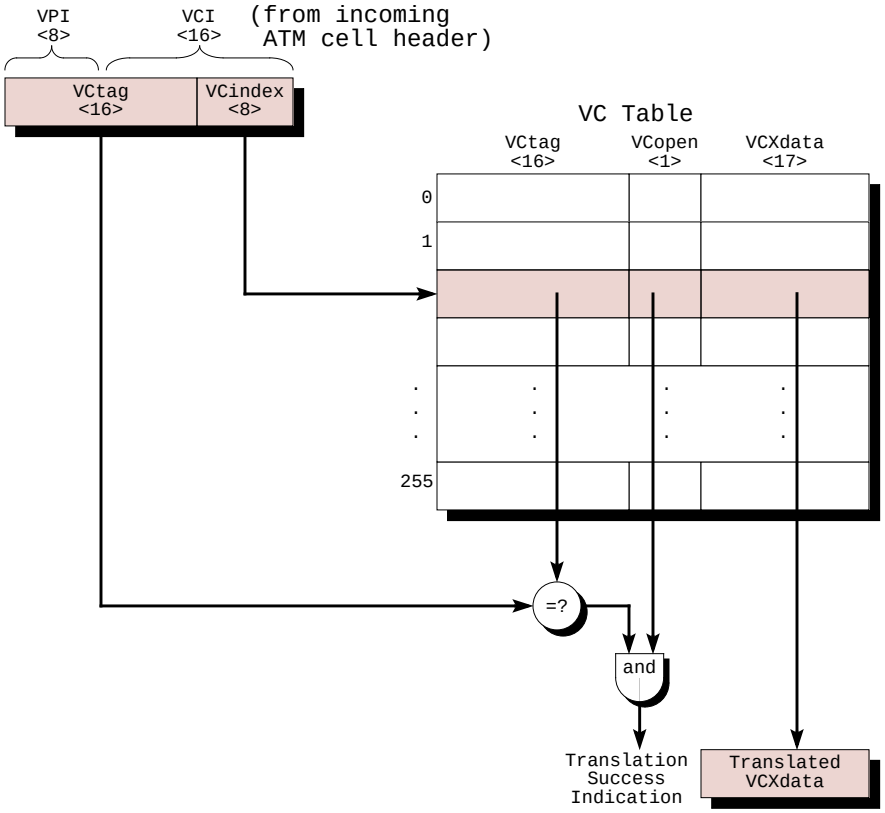

Figure 6.5: VC Translation Process in the VCXT Module

by writing to appropriate APIC registers. For every cell received on input ports 0 and 1, the VCXT uses the VPI/VCI fields from the cell to perform a lookup in this table. If there is a matching entry in the table for the corresponding VC, and if that entry is marked as "*open*" (or valid), the data from the entry is used to decide what to do with the incoming cell.

Figure 6.5 illustrates the lookup mechanism used in the VCXT. The low order 8 bits of the VCI, which we shall call the *VCindex*, is used to index into the VC table. The latter contains 256 entries corresponding to the 256 receive VCs that are supported by the APIC. The 8 bits of the VPI and the high order 8 bits of the VCI account for the remaining 16 bits from the VPI/VCI fields in the cell; we refer to these 16 bits as the *VCtag*. These bits are compared against a `VCtag` that is part of the entry resulting from the table lookup. If they match, and if the table entry is marked as valid (i.e., the *VCopen* bit is set), then the incoming cell is said to have been *successfully translated*. For such cells, the VC translation data (*VCXdata* for short) field from the corresponding table entry is used to decide what to do with the cell. The `VCXdata` field contains several subfields (such as: the set of output ports to which the cell should be directed, whether the cell should be treated as low delay or not, etc.).

☞ Clearly, the lookup mechanism used in the VCXT implies that no two receive VCs supported by the APIC can share the same value for the low order 8 bits of the VCI (the `VCindex`). This is true even if the VCs use different input ports of the APIC. In a single APIC environment, the usual case would be to use VCIs ranging from 0 to 255.

With multiple APICs in a daisy chain, it is possible for two receive VCs that terminate on different APICs in the chain to share the same `VCindex`, so long as they differ in the `VCtag` part of the VPI/VCI. In fact, for local communication between APICs in a daisy chain, it is recommended (though not required) that the `VCtag` be set equal to the 16-bit value of the `APICid` (which is pin configured never to be the same for any two APICs on the interconnect). In this case, the VPI/VCI allocation problem for local communication (within the interconnect) becomes the localized decision of choosing an unused `VCindex`.

If an incoming cell is *not* successfully translated (i.e., there is no matching valid entry in the VC table), then the VCXT treats the cell as a transit cell. Such cells should be forwarded to the "other" ATM port (i.e., the ATM port opposite to the one on which the cell arrived). In the example in Figure 6.4, the transit cell came in on port 0, so it should be sent out on port 1. Had the cell entered the APIC on port 1, it would leave on port 0.

Thus, for incoming cells that are transit cells, the VCXT knows the output port for the cell based on the port it came in on. For non-transit cells, it obtains information about the set of output ports from the VC table lookup. If this set of output ports includes only port 2, then the corresponding

VC is termed a *receive VC*, and cells belonging to this VC follow the *receive path* (which is described later). If the set of output ports contains more than one port, then we have a multipoint VC; cells belonging to such VCs may follow more than one path through the chip (for example, the transit path as well as the receive path).

Note that just because a cell follows the transit path through the chip does not mean that it is a transit cell. A cell may be successfully translated in the VCXT, and the resulting set of output ports may contain just the single port that is opposite to the one the cell came in on. In this case, the cell would behave very much like a transit cell, and it would follow the transit path through the chip. A cell can even be made to loopback to its entry port by setting the VC table entry appropriately. This kind of behavior is important for architectures other than the daisy chain (for example, the perfect shuffle of Figure 5.3), where the default transit port of the APIC may not be the right place to forward an incoming "transit" cell.

The VCXT encapsulates various information derived from the VC table in an *internal header* which is tagged on to the cell before forwarding it to the *CellStore* module. For transit cells, the internal header is derived based on default values and on the port the cell came in on, rather than from a table entry. Among other information, the internal header contains the set of output ports for the cell, whether the cell should be given low delay processing priority, etc.

6.2.6. CellStore

The *CellStore* module contains the main on-chip *cell buffer* that will be used to temporarily hold the ATM cells pending availability of the appropriate output port(s) (including port 2 — the bus port). The capacity of this cell buffer is 256 cells. The CellStore module also implements several FIFO queues that contain pointers (8-bit indices) to cells stored in the cell buffer. Using this technique, the 256 cells worth of storage in the cell buffer can be shared amongst many FIFO queues. One of these FIFO queues is called the *free queue*; it holds pointers to free slots in the cell buffer. The remaining FIFO queues are called *busy queues*; they hold (pointers to) cells awaiting output on one or more of the three output ports. Whenever the CellStore receives a cell from the VCXT, it stores the cell in the slot that is pointed to by the entry at the head of the free queue. This entry is then taken off the free queue and put onto one or more of the busy queues. Since we are currently focusing on the transit path, we introduce only the busy queues for ports 0 and 1; the busy queues for port 2 will be discussed when we cover receive path processing. As shown in Figure 6.6, the

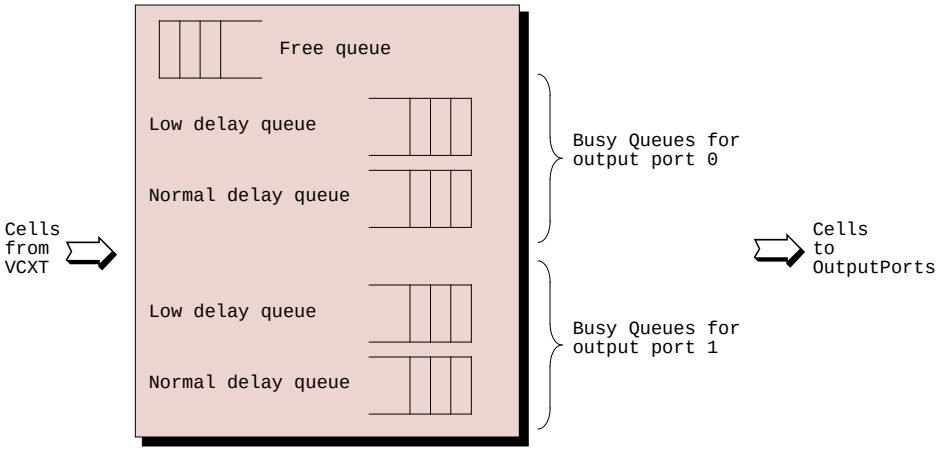

Figure 6.6: FIFO Queues in the Cell Store (Port 2 queues not included)

CellStore module implements two busy queues each for ports 0 and 1: one for low delay traffic, and the other for normal delay traffic. All transit cells are automatically categorized by the VCXT as low delay (this is indicated by a bit in the internal header that is attached to the cell), so these cells will be always be put on the *low delay queue* for the appropriate output port.

☞ As we will see later, the low delay busy queue can, in addition to transit traffic, contain cells from low delay transmit channels and from low delay receive VCs. The normal delay queues can contain cells from best-effort and paced transmit channels and from normal delay receive VCs. Note that the normal delay queues for ports 0 and 1 never carry any transit traffic.

Each OutputPort reads cells from the corresponding busy queues in the CellStore and transmits these cells at the maximum link rate, so long as it has a grant from the downstream device (using either UTOPIA or GFC flow control). The low delay busy queue is always given priority over the normal delay busy queue. In other words, no cells will be transmitted from the normal delay queue for a port until the low delay queue for that port has been completely drained.

Receive Path

Now that we have finished describing the transit path, we move on to the *receive path* which is shown in Figure 6.7. An incoming cell is considered to be a receive cell if it is successfully translated in the VCXT, and if the set of output ports from the resulting `VCXdata` field of the VC table includes port 2 (the bus port). In this case, the cell will follow the receive path through the chip.

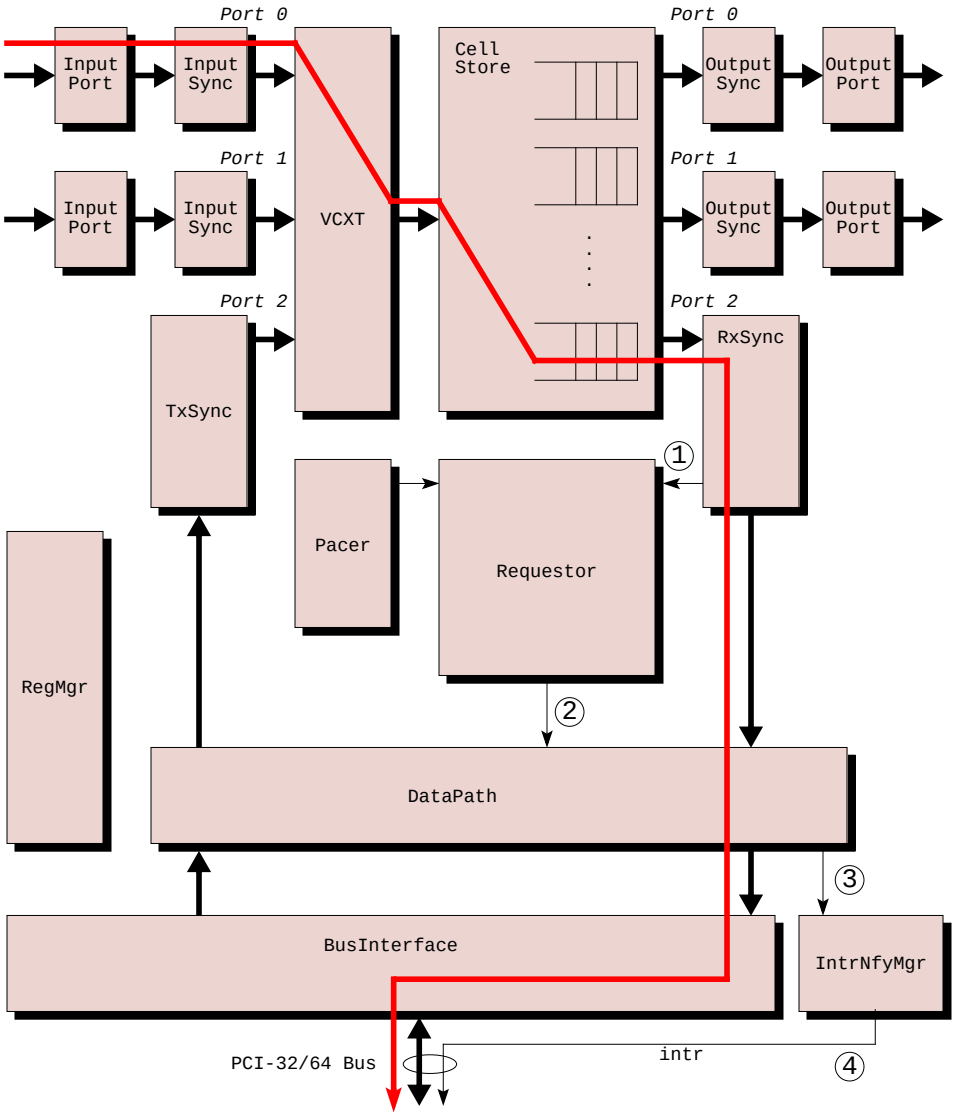

Figure 6.7: The Receive Path

When a receive cell arrives at the CellStore, it is put onto one of the busy queues for port 2. Figure 6.8 expands on Figure 6.6 to include the set of busy queues for port 2. As for ports 0 and 1, there is a low delay busy queue for port 2. However, while ports 0 and 1 were associated with a single normal delay busy queue, the CellStore implements 256 normal delay busy queues for port 2, one for each of the 256 receive VCs that are supported by the APIC. This kind of per-VC queueing is required to support cell batching (see Section 5.5.5), as will be evident once we describe the service discipline used to empty these queues.

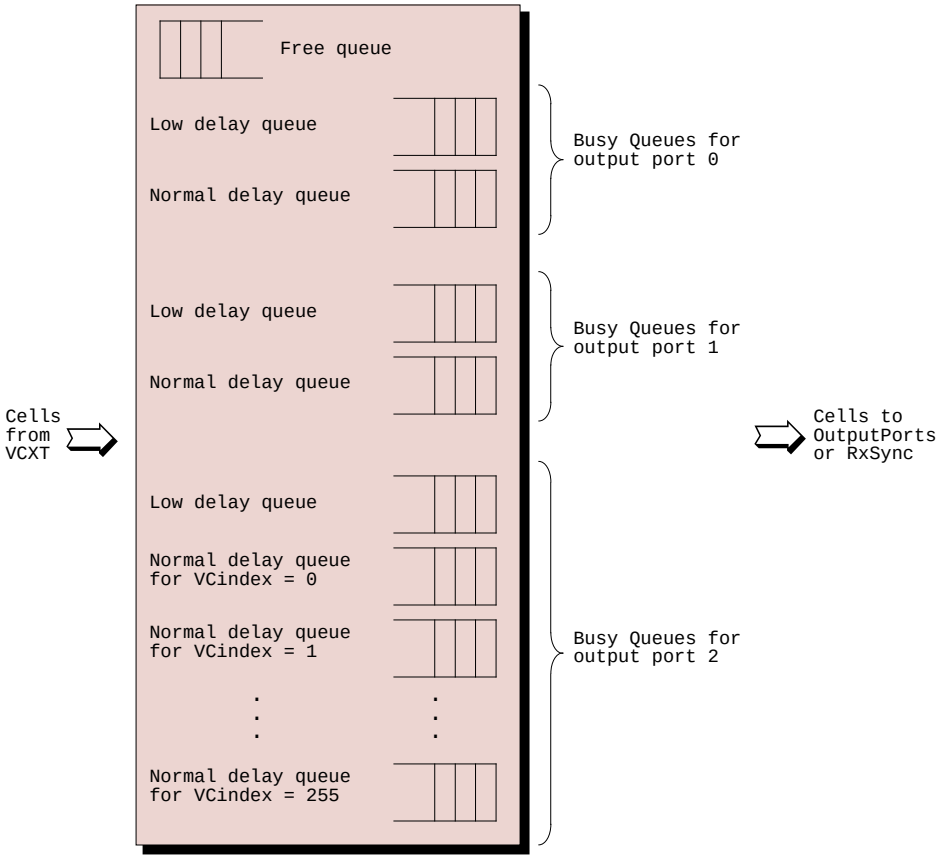

Figure 6.8: FIFO Queues in the CellStore

☞ Note that all low delay cells, regardless of their VC, are put into the single low delay busy queue. This implies that there is no per-VC queueing for low delay VCs, and consequently, low delay VCs cannot benefit from cell batching. This is acceptable because cell batching helps improve throughput, and our assumption that low delay VCs carry small amounts of data infrequently implies that they do not also require very high throughput.

On the output side, the *RxSync* module reads receive cells from the CellStore and creates batches of one or more cells for processing by the Requestor/DataPath subsystem. The following rules are used to decide which busy queue the next cell should come from:

1. The low delay queue is always given priority. In other words, if the low delay queue is not empty, it will always be serviced in preference to the normal delay queues.
2. If the low delay queue is empty, then cells will be drawn from one of the normal delay queues. The CellStore keeps track of the set of normal delay busy queues that are not empty. This set

is maintained as a list of `VCindex` values. Whenever a receive cell is placed in a normal delay queue that is not already in the set, that queue is added to the set by appending the corresponding `VCindex` to the tail end of the list. This list is used to service the normal delay busy queues. The queue represented by the `VCindex` at the head of the list is serviced first, and it is completely drained before moving on to the next queue in the list. This policy guarantees that all the busy queues will be serviced eventually. Also, since the busy queue for each normal delay VC is completely drained before moving on to another busy queue, the batching of cells belonging to the same VC is maximized.

☞ Some may argue that this policy is unfair because, in overload conditions, it would give all of the bandwidth available on the bus to a single VC. While this is true, we believe that it is better for at least one VC to get its data through during congestion, as opposed to losing some data from all VCs.

Another argument against such a policy is that it could result in data from one VC being significantly delayed by data from another VC. This is true only in overload conditions; otherwise, the interval between when a queue is drained completely and when it can be serviced again in the future is limited by the size of the cell buffer (256 cells). If better delay characteristics are desired, low delay VCs should be used.

6.2.7. RxSync

Continuing our discussion of the receive path (Figure 6.7), we already mentioned that the RxSync module, in addition to serving as a synchronization module between clock regimes, also performs the task of creating batches of cells belonging to the same VC. To aid in this task, the RxSync has a small amount of internal storage (8 cells worth).

6.2.8. Requestor

The *Requestor* module is responsible for deciding when a batch from the RxSync should be processed ①, and for figuring out how to process these batches. The Requestor contains most of the per-channel state for both transmit and receive channels. Also, a large chunk of the "brains" of the APIC (i.e., the control logic) are concentrated in the Requestor module. Some of the tasks of the Requestor are:

- To arbitrate between requests to transmit cells that are generated by the Pacer module, and requests from the RxSync to process received cells ①.

- To do all of the DMA processing for the different DMA modes, and to figure out where in external memory data should be written to (or read from).

- To break up transmit or receive requests from the Pacer and RxSync modules into appropriate sized transactions that can be issued on the bus, taking into account things like the host's cache line size, etc.

- To report interrupt events to the Interrupt/Notification Manager module.

6.2.9. DataPath

The Requestor does not itself move any data; it implements the control logic that decides when data should be moved, where it should be moved, and how much to move. It provides this information to the *DataPath* module ②, which is responsible for actually moving the data. On the receive path, the DataPath takes transaction requests from the Requestor and processes these requests by issuing commands to the BusInterface that result in data being read out of the RxSync and written to external memory. While the data is in flight, the DataPath module is also responsible for data path operations such as computing the AAL-5 CRC and TCP checksum over received frames.

The Requestor and DataPath modules form a pipelined subsystem: while the DataPath is moving data corresponding to a transaction request, the Requestor is computing the next transaction request that should be issued. This pipelining ensures that the APIC can keep up with the full rate of the PCI bus.

6.2.10. IntrNfyMgr

As cells are received and the data from these cells is written to external memory, various interrupt events may occur. These events are recognized by the Requestor, and are passed on to the DataPath module along with the corresponding transaction request ②. The DataPath blindly forwards these event notifications to the *Interrupt/Notification Manager* (*IntrNfyMgr* for short) module, as soon as the transaction is completed ③. The IntrNfyMgr module decides whether or not the interrupt event should result in an actual interrupt to the processor. If so, it raises the bus interrupt line

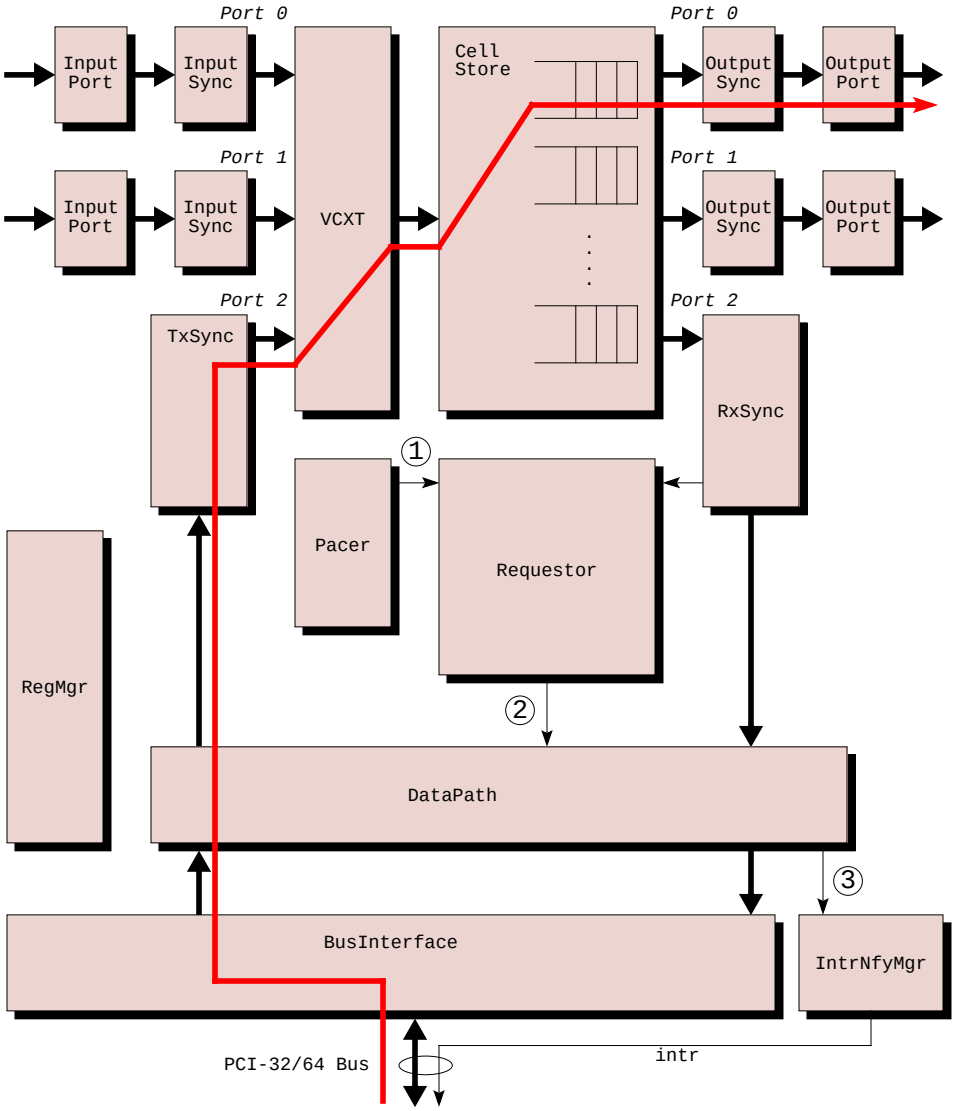

Figure 6.9: The Transmit Path

④, and also resumes the remote interrupt channel (if one has been configured). Finally, the IntrNfyMgr module also maintains the APIC's notification list.

Transmit Path

We now go one to describe the transmit path, shown in Figure 6.9. We begin our discussion with the *Pacer* module, which is responsible for generating requests to transmit batches of cells. The Pacer generates these requests based on the currently active set of low delay, paced, and best-effort channels. Note that for paced channels, the Pacer generates requests periodically, based on

the channel's pacing rate. The Requestor arbitrates between requests from the Pacer ① and requests from the RxSync (for processing of received cells). The arbitration algorithm used by the Requestor is based on the following simple priority order:
control cell > low-delay Rx > low-delay Tx > normal-delay Rx > all other Tx
If the winning request is from the Pacer (i.e., it is for Tx), then the Requestor breaks up the request for the batch of cells into appropriate sized transaction requests, which it issues to the DataPath module ②. The process is very similar to that used on the receive path, except that in this case the transaction requests result in data being read from external memory and encapsulated in cells that can then be transmitted. During the process of issuing transactions to read data from external memory, the Requestor may recognize interrupt events, which it passes (through the DataPath ②) to the IntrNfyMgr ③. This is again very similar to the receive path interrupt and notification processing.

The transmit cells, once constructed in the DataPath, are tagged with an internal header that specifies the output port for the cell (the internal header can come directly from external memory if the channel uses AAL-0). These tagged cells are forwarded through the TxSync to the VCXT. For transmit cells that do not have to be looped back to port 2, the VCXT merely forwards the cell unchanged to the CellStore. The latter puts the cell on one of the busy queues for the appropriate output port, from where it can be read out by the corresponding OutputPort module and transmitted on the link. The choice of busy queue in the CellStore (low delay or normal delay) is specified in the low delay bit that is part of the internal header; this bit is set if the cell came from a low delay transmit channel, and is cleared if it came from a paced or best-effort channel.

Control Cell Path

Figure 6.10 shows the path taken by a control cell and the corresponding response cell through the APIC. Note that the response cell leaves the APIC on the same port pair that the control cell entered. In the example shown, the control cell enters from input port 1, so the corresponding response cell will leave from output port 1. This is because the response cell should be returned to the sending entity. Contrast this with the transit path, where a cell leaves the APIC from the port opposite to the one it came in on.

Control cells arrive on the special control VC that is characterized by `VCindex` = 0x21, and `VCtag` = `APICid`. Here, the `APICid` is a 16-bit pin configured value that is different for every

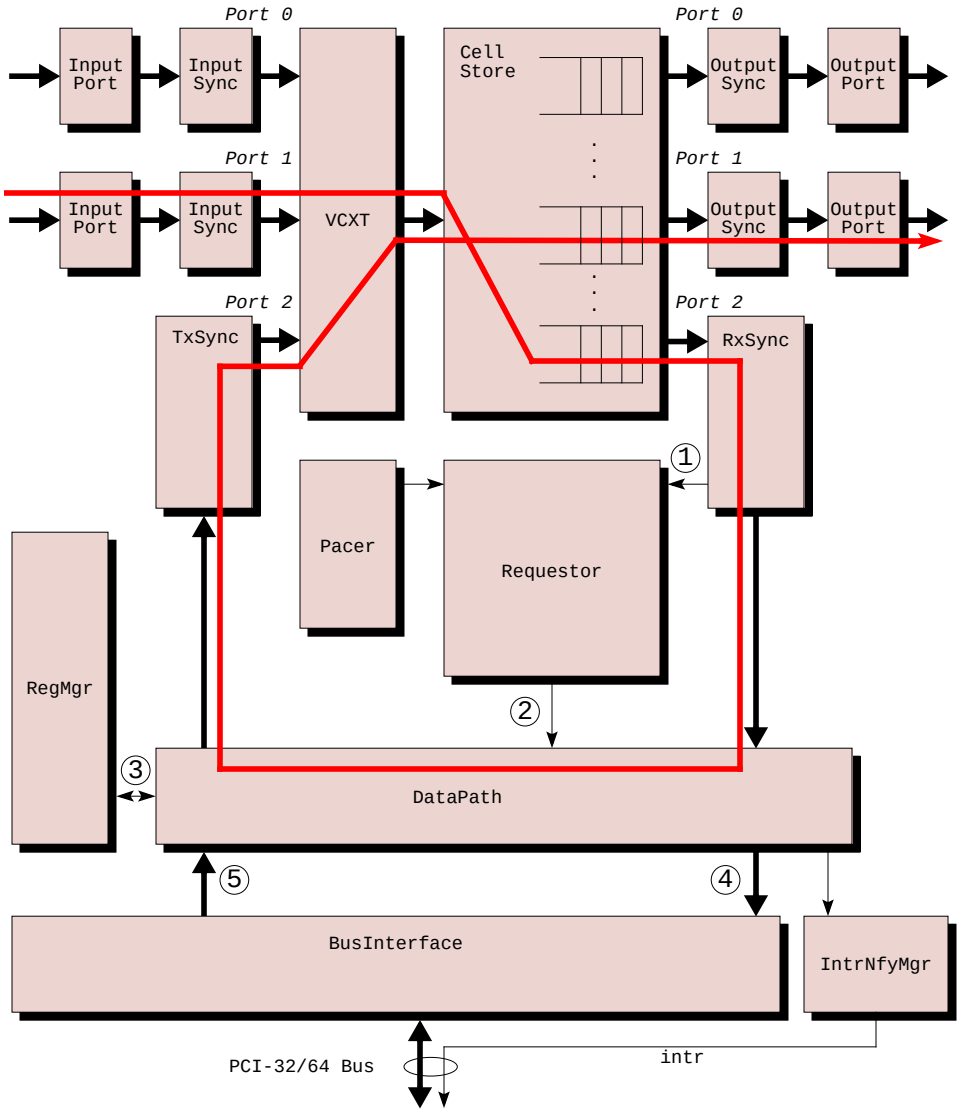

Figure 6.10: Control and Response Cell Path

APIC along the path traversed by the control cell enroute to its target APIC. Think of this in the context of a daisy chain of APICs. In effect, the APICid serves as an address for the target APIC. The VCXT recognizes incoming control cells based on this encoding of the VPI/VCI. It forwards such cells to the CellStore with an internal header that specifies:

1. that the cell is a control cell,
2. that it should be output to port 2, and
3. that it should be treated as a low delay cell.

Consequently, the cell ends up in the low delay busy queue for port 2 in the CellStore. As was the case for the receive path, the RxSync reads the cell out from this queue, and generates a received cell request to the Requestor ①. Recognizing the cell as a control cell, the Requestor issues a special "control cell transaction" to the DataPath module ②.

All of the control cell processing is carried out within the DataPath module. First, the contents of the cell are read out from the RxSync. Next, the CRC for the cell is verified, and the cell is dropped if the CRC is incorrect. Following this, the DataPath compares the sequence number bit in the cell to the expected sequence number (as per an alternating bit protocol). If they match, the operation specified in the cell is carried out and a new response cell is generated. Otherwise, the contents of the cell are ignored and the previous response cell (i.e., the one corresponding to the previous successful control cell operation) is used. In both cases, a CRC is computed and inserted in the response cell. Once this is done, the cell is treated as a transmit cell and forwarded with the appropriate internal header to the TxSync, from where it makes its way along the usual transmit path to the appropriate output port. It is important to note here that the ATM header used in the response cell is derived from the contents of the corresponding control cell.

As discussed earlier, control cell operations can be either internal accesses to the APIC's on-chip registers, or external accesses to memory or another device that can be accessed over the PCI bus. For internal accesses, the DataPath carries out the operation by issuing requests directly to the RegisterManager to read or write the specified register ③. For external accesses, the DataPath issues write ④ or read ⑤ transaction requests directly to the BusInterface module.

Multipoint and Loopback Paths

We have so far discussed intra-chip paths for point-to-point connections only. The transit, receive, and transmit paths can be combined in various ways to yield multipoint intra-chip paths.

Figure 6.11 shows one possible multipoint receive path. Cells belonging to a VC that originates at one of the two ATM ports can be directed to any combination of the three output ports. The set of output ports for such a VC is controlled by a subfield in the `VCXdata` field of the VCXT table. For the example in the figure, all three output ports would belong to this set.

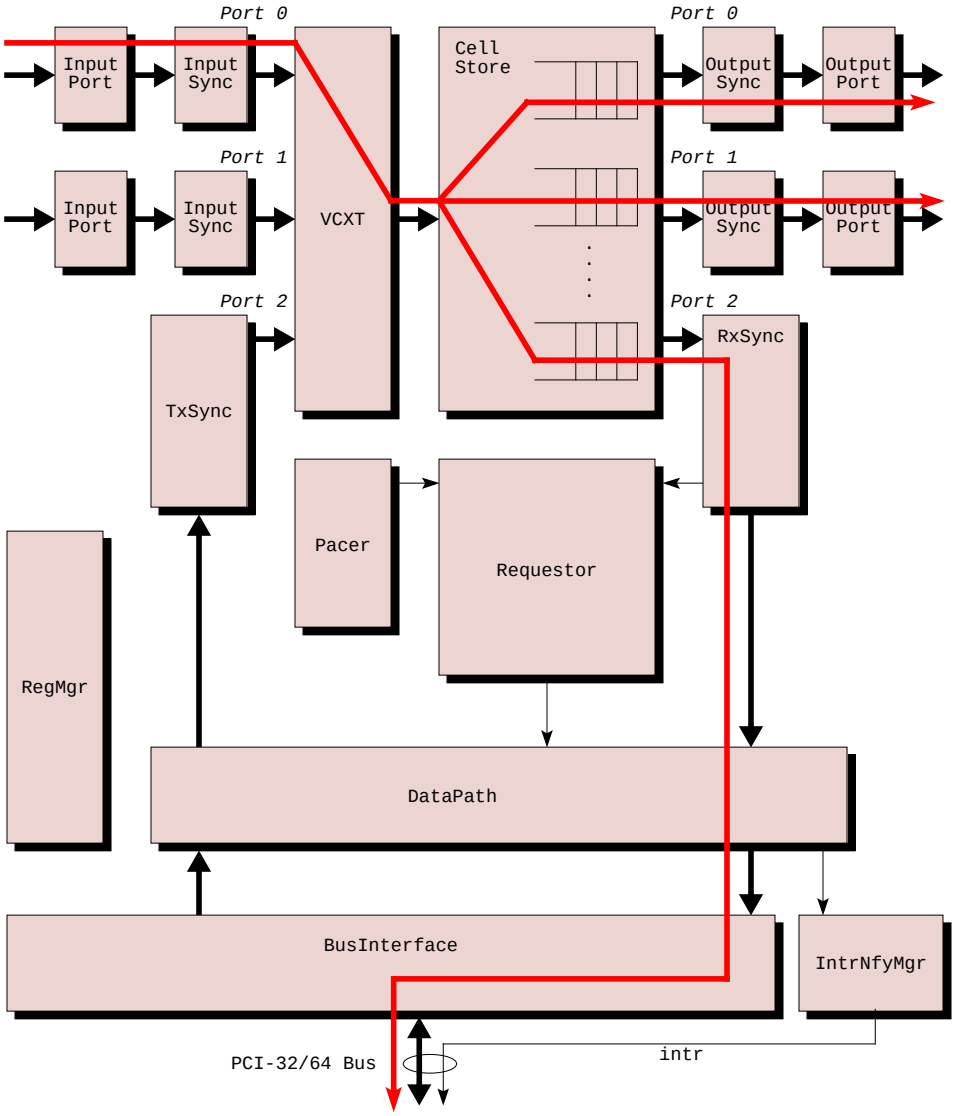

Figure 6.11: A Multipoint Receive Path

A note about the implementation of the multipoint capability within the CellStore module is in order. The information about the set of output ports for a cell is forwarded to the CellStore along with each cell, as part of the cell's internal header. Although Figure 6.11 shows three copies of the cell being made in the CellStore module, in reality these are logical copies only. Each incoming cell occupies only one cell slot in the CellStore's cell buffer, but a pointer to this cell slot can be stored in multiple busy queues (one busy queue per destination output port). Whenever a cell reaches the head of a busy queue and the corresponding output port becomes available, the cell is output to that port, and the pointer entry is removed from the head of the busy queue. However, the

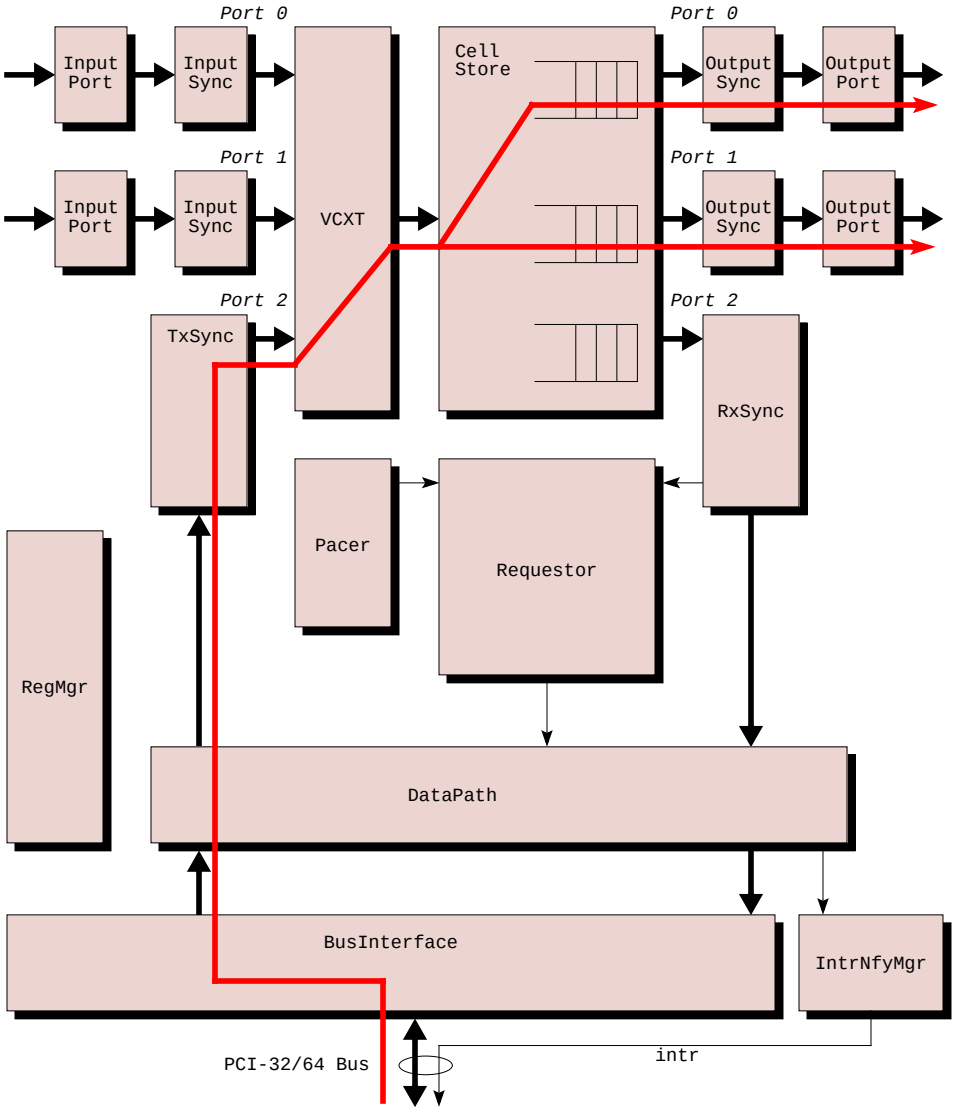

Figure 6.12: A Multipoint Transmit Path

cell slot containing the cell cannot be returned to the free queue until the cell has been dequeued from all of the busy queues on which it is enqueued. The CellStore therefore maintains a reference count for each slot in the cell buffer; a (pointer to a) slot is not returned to the free queue until the reference count reaches zero.

Figure 6.12 illustrates a multipoint transmit path. This is identical to the transmit path through the chip, except that a copy of the cell is sent to both output ATM ports. One example of the use of this path would be to implement broadcast in a daisy chain LAN environment.

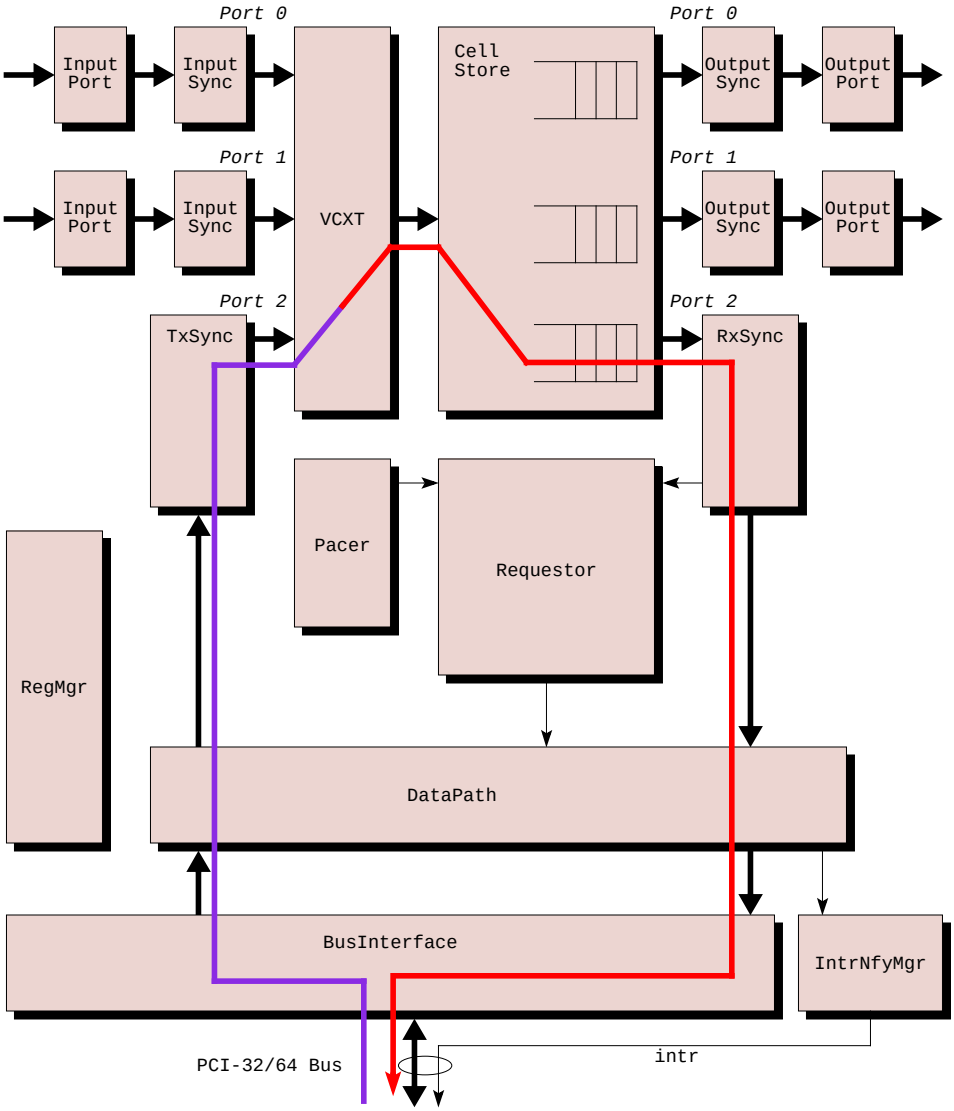

Figure 6.13: Loopback Path

☞ As in the case of the multipoint receive path, the CellStore needs to enqueue cells following this path on the busy queues for both ATM ports, and to do this it looks in the internal header of the cell to discover the set of output ports for the cell. For the multipoint receive path, the internal header is tagged onto the cell in the VCXT, based on information derived from the VC table lookup. For the multipoint transmit path however, the internal header is already tagged to the cell before it even reaches the VCXT — for AAL-0 channels the internal header is read along with the cell data from external memory, while for AAL-5 channels it is attached (along with the ATM cell header) to the cell payload by the DataPath module.

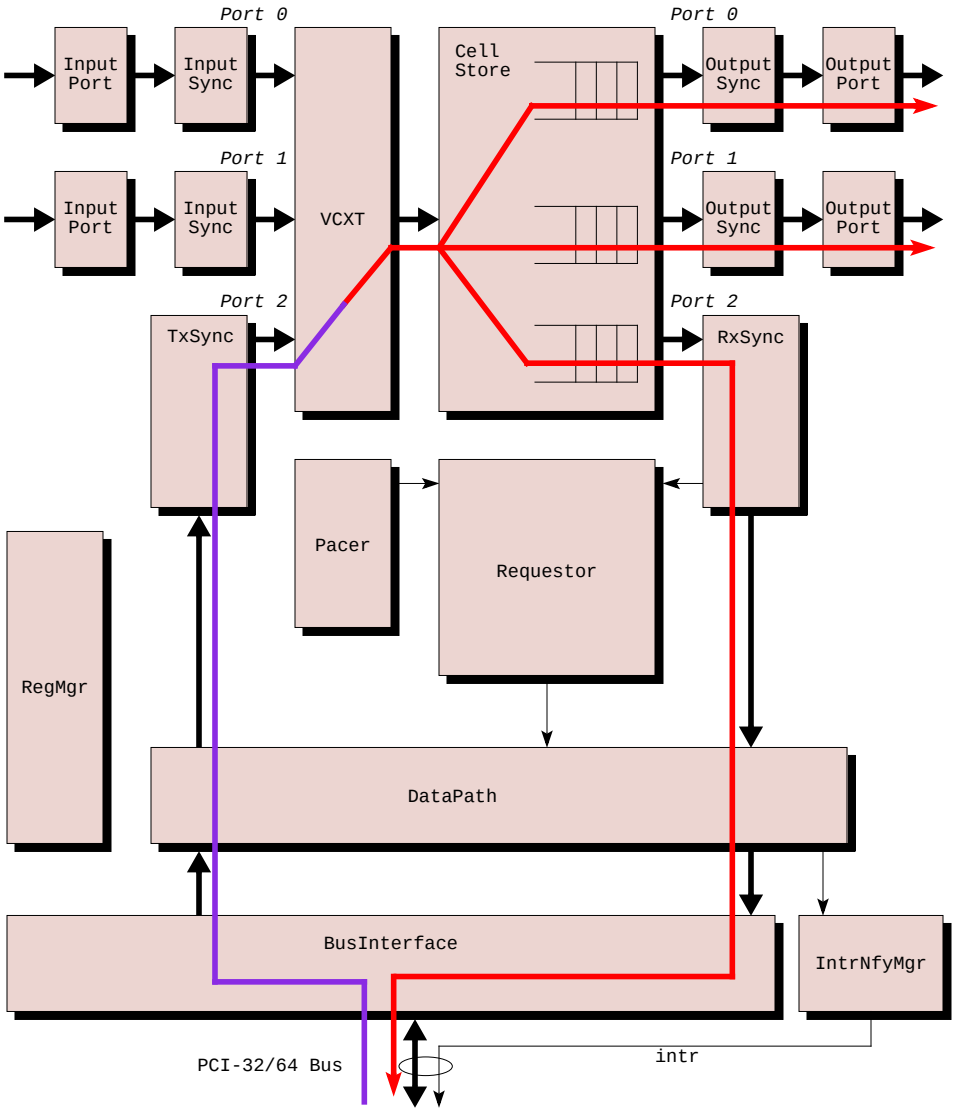

Figure 6.14: A Multipoint Loopback Path

In Figure 6.13, we illustrate the path taken by cells that are to be looped back to external memory. Since ATM VCs are unidirectional entities, the loopback feature requires setting up both a transmit and a receive VC. A loopback cell transitions from the transmit path to the receive path in the VCXT, as indicated in the figure by the different colors used for the two segments of the loopback path. Usually, cells from transmit channels do not require a table lookup in the VCXT. However, if the set of output ports for a transmit channel include port 2, then it is a loopback channel,

and cells belonging to such channels undergo a table lookup in the VCXT. From that point on, these cells are treated exactly as if they were cells that were received on one of the input ATM ports.

The last path through the chip that we describe is the multipoint loopback path, illustrated in Figure 6.14. This is identical to the loopback path, except that the VC table lookup in the VCXT yields more than one destination output port. It is very important to note that the set of output ports actually taken by cells along this path is a property of the receive VC, and not of the transmit channel. Indeed, depending on how the receive VC is configured, a transmit cell originating from a loopback transmit channel may not even be looped back — it could, for example, simply be sent to one of the output ATM ports (although there doesn't seem to be much point to such an exercise, since the same effect could be achieved by just a transmit channel alone).

6.3. Pacer Design

While most internal blocks of the APIC have straightforward implementations, the Pacer block deserves special mention because of the unique hardware architecture it uses to enable pacing of traffic over many simultaneous channels with differing pacing rates. As mentioned earlier, the novel *d*-heap pacing algorithm used by the Pacer allows the APIC to provide QoS in a very flexible manner to various types of applications.

Naive approaches to implementing pacing, such as parallel search of departure times of cells from various channels do not scale, and can be excessive in their space and power requirements when the number of channels to be supported is large. Most commercial ATM network interfaces employ such naive schemes, which is why they cannot support more than a very small number of paced channels. In order to scale to large numbers of channels, a brute force approach does not work, and cleverer techniques are called for. When searching for an appropriate algorithm, we have to keep in mind both area and power constraints, as well as being able to meet the timing needed to issue pacing grants once every cell time. It took several trials and iterations until we hit upon the solution which is described below, and which is scalable to very large numbers of connections at modest cost.

Every paced channel in the APIC has an associated "pacing interval", which represents the time interval between successive cell transmissions. The pacing rate for the channel is simply the inverse

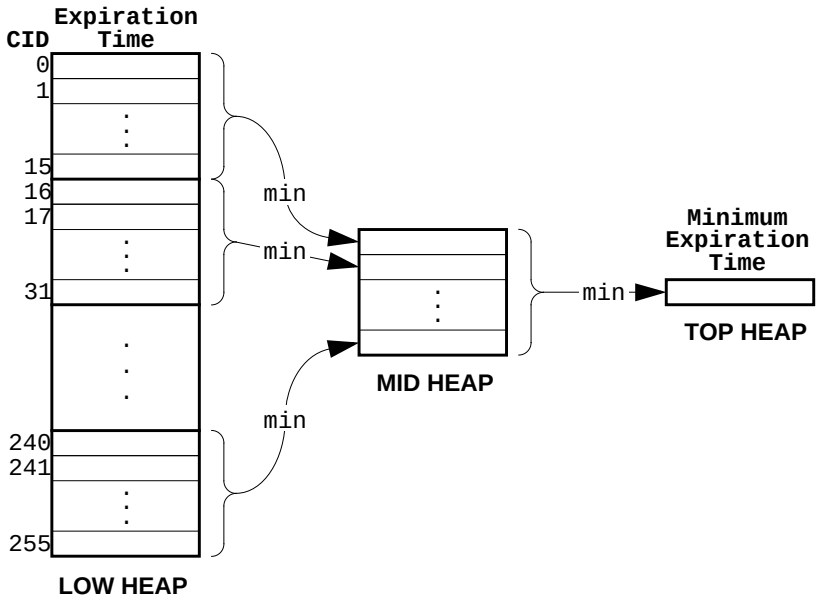

Figure 6.15: d-Heap Based Pacing

of this quantity. Each channel is also associated with an "expiration time", which is the time at which the next cell is scheduled for transmission on that channel. All of the channels are maintained on an event queue which is keyed on the expiration time. Whenever the current time is equal to or greater than the minimum expiration time of any channel, a cell is transmitted from the corresponding channel, and the channel's expiration time is incremented by the pacing interval, and reinserted into the event queue. This simple algorithm can implement pacing over large numbers of connections. The only problem is: how do we implement an event queue in hardware that is keyed on expiration times of large numbers of connections?

Our solution to this problem is to use a *d*-heap data structure, with a large value of *d*. The way we formulate the data structure is specially geared towards hardware implementation. Figure 6.15 shows a *d*-heap for 256 channels (each of which is indexed with its channel id, CID). The low heap contains the expiration times of all the channels. The low heap is divided into 16 blocks, and the minimum expiration time from each block is stored in the mid heap. Thus, the mid heap has only 16 entries. Finally, the top heap contains the minimum expiration time from all the entries in the mid heap; it is the minimum expiration time over all channels. When ready to transmit, the Pacer module in the APIC issues a requerst to transmit a cell from the channel corresponding to the top heap entry. It then adds the pacing interval to that channel's expiration time, and recomputes the

minimum from the corresponding block in the bottom heap and then recomputes the minimum of all the entries in the mid heap.

With this solution, note that d is 16, and so only a single 16 wide comparator tree is needed to do all the comparisons. The lower levels of the d-heap are larger, so an implementation could choose to put them in more compact but slower memories (eg., DRAM), while placing higher levels of the d-heap in bigger (in terms of area) but faster memories (SRAM or registers). In the current APIC prototype, the low and mid levels are in SRAM, while the top level is a register.

With this pacer algorithm, the lookup time is $O(\log_d n)$, where n is the number of channels. Clearly, by making d large, the lookup time can be reduced to very reasonable small constants. The cost of increasing d is a larger comparator tree (in terms of area). With $d = n$, this scheme reduces to the brute force approach mentioned earlier.

Note that the size of the heap data structure is $O(n)$, which is optimal. Also note that very large numbers of channels can be supported by increasing the depth of the heap, and increasing d. For example, 32K channels can be supported by using a four level heap with $d = 32$.

There is one issue that we would like to point out here with regards to the pacing intervals: we get finer granularities of pacing bandwidth at lower bandwidths, which is quite a nice property to have, except that the granularity at high bandwidths is too coarse if the pacing interval is an integer. For example, with integer pacing intervals, it is not possible to get any bandwidth between 0.5 and 1.0 of the maximum (link) bandwidth. To overcome this limitation, the pacing interval needs to be specified with a fractional part, as a fixed point real number. In the APIC context, we found that using a 24 bit value for the pacing interval, with a 16 bit integer part and an 8 bit fractional part gave us sufficient granularity at high bandwidths, while allowing us to be able to specify very low bandwidths at the other end of the spectrum (in the range of a few Kb/s).

Chapter 7

APIC Software

In this chapter, we cover the software framework in which the chip is intended to operate, and describe the implementation of the APIC's kernel driver, which is an important piece of that framework.

7.1. Overall Software Framework

Figure 7.1 shows a proposed software framework for the APIC chip. In order to limit the scope of work, not all components in this software framework have been implemented. The components with darker shading represent those that we have implemented in our NetBSD test platform.

☞ The NetBSD operating system was selected because it is open source, features an industry standard Internet protocol stack, and is in use in several other projects at our site.

The TCP/IP stack and the socket layer are components that are standard in NetBSD. Many of the remaining components remain unimplemented at the time of this writing. It is expected that future users of the APIC chip from other projects, as well as from the Gigabit Kits initiative, will invest the time to implement most of the other pieces of this framework.

The bold lines in the figure represent different data paths through the system. Light lines represent control paths.

The APIC kernel driver implements all of the in-kernel functionality needed to support communication using the APIC. It makes use of the facilities provided by an ATM device-dependent

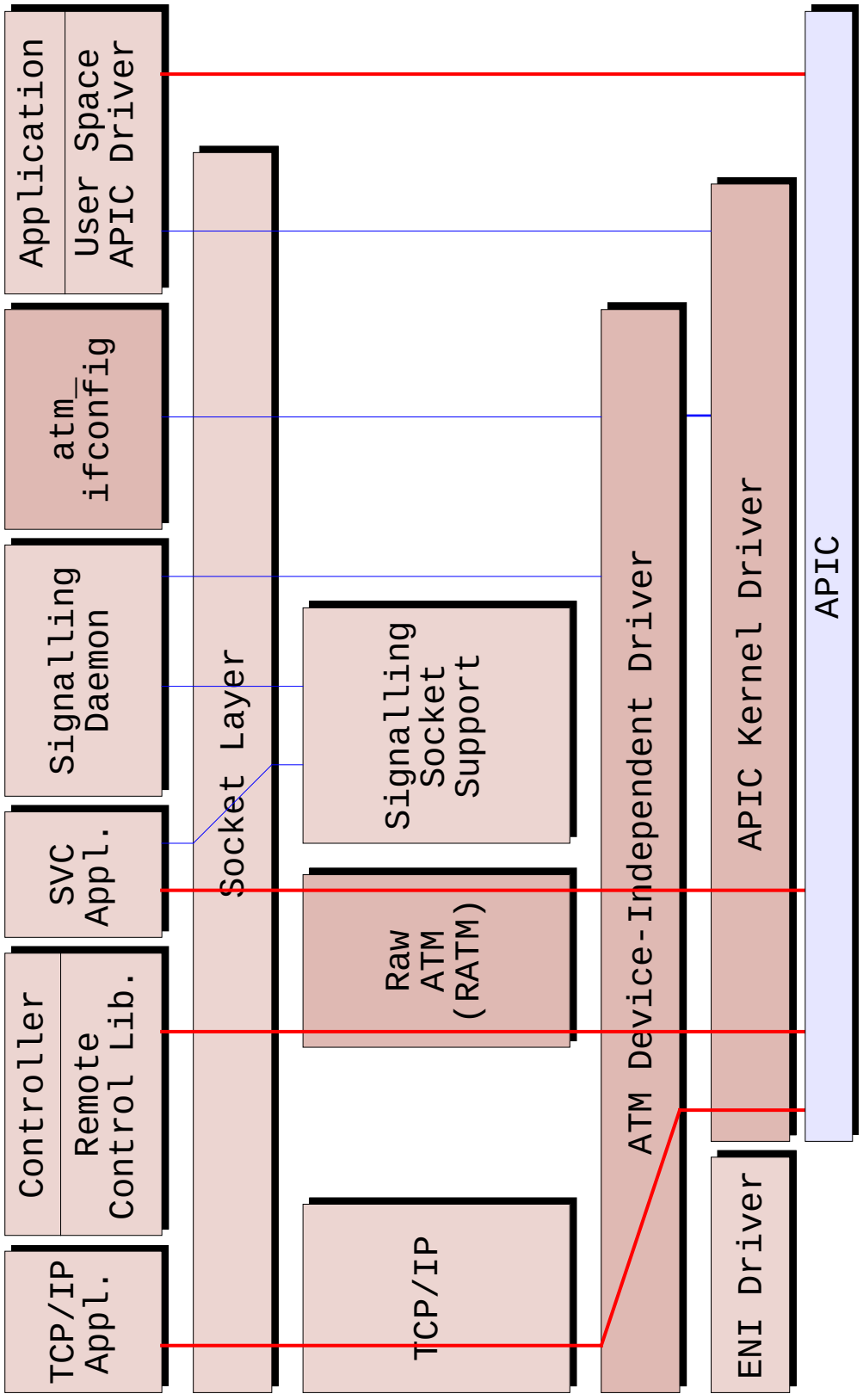

Figure 7.1: Software Framework for the APIC

driver that has been designed to abstract out functionality common to all ATM network interfaces. Thus, for example, the Efficient Networks (ENI) ATM driver could share this section of the driver code with the APIC. The driver interfaces directly to the system's TCP/IP stack, thereby allowing legacy TCP/IP applications to use the APIC for communications over the Internet.

The Raw ATM (RATM) layer provides AAL-0 and AAL-5 transmission and reception capabilities directly to applications residing in user-space. It uses the operating system's socket layer to export this service to these applications.

Before any communication can occur over the APIC, the on-chip channels and connections that will be used need to be configured. This can be done in one of two ways. The `atm_ifconfig` user-space utility can be run either manually or in a batch script; it takes a number of parameters as arguments and makes the appropriate ioctl system calls to the driver to configure these parameters into the chip. It can be used to setup or teardown connections, assign pacing rates to channels, etc. This utility is typically used to setup permanent virtual circuits (PVCs) that originate or terminate at the APIC.

ATM switched virtual circuits (SVCs) can be setup using a signalling daemon, which is a user-space program that, in addition to implementing the signalling protocols, makes ioctl system calls to the APIC driver to configure connections in a manner similar to `atm_ifconfig`. Different SVC applications make requests to the signalling daemon to setup and teardown end-to-end virtual circuits; these applications can use special signalling sockets to communicate with the signalling daemon. Once a connection has been setup, these applications can make socket calls to the RATM stack to send and receive ATM AAL-5 frames.

The remote controller process shown in the figure can be used to control remote APICs that reside on the local desk-area network. The controller implements a device driver for the remote APIC in a special remote control library. It may also implement a driver for the remote device. Since all communication with the remote APIC and device is through control, response, and interrupt cells, the controller needs the ability to be able to send and receive raw AAL-0 cells using the local APIC; it can use socket calls to the RATM protocol layer to achieve this.

The figure also shows an application using the APIC's user-space control model. In this case, the kernel driver is used only for the control path, such as for setting up a Protected DMA / Protected

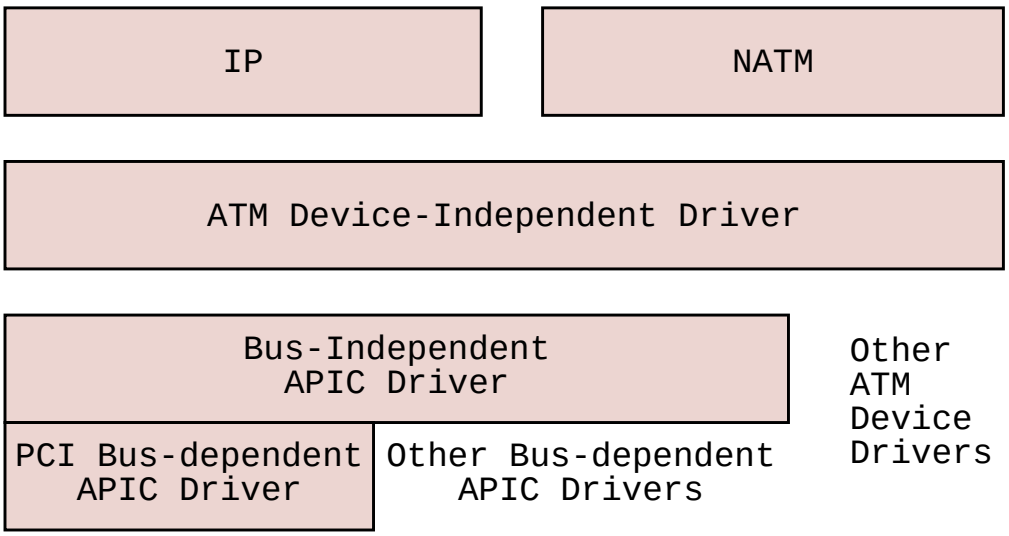

Figure 7.2: APIC Kernel Driver Structure

I/O channel. The user-space driver, which is implemented as a library linked with the application, is responsible for all data path communication using Protected DMA and Protected I/O to talk directly to the APIC chip. In this context, in addition to other calls already supported by the kernel driver for in-kernel protocol implementations, the following additional calls would, at a minimum, need to be supported:

- A system call for allocating a protected DMA channel. The kernel driver would allocate the protected DMA descriptor pool for the channel, and map the user descriptors portion of it into the address space of the calling process. A file descriptor would be allocated and used as a handle for referencing the channel.

- A system call for binding a protected DMA channel to an ATM VC. The corresponding per-channel registers would be setup in the APIC by this system call, and the APIC's interrupt service routine would be configured to wake up the user process if it is sleeping (for example, waiting on a `select()` call) awaiting an event on the file descriptor that references the protected DMA channel. The call would additionally setup the protected I/O *access mask register* for the channel, and map the user-access region of the per-channel registers into the caller's address space and return a pointer to it.

In addition, for easy portability of the user-space driver between different operating systems and environments, the kernel driver would need to support a system call to wire user-space buffers into physical memory (most operating systems already contain such a system call).

7.2. Kernel Driver Structure

The kernel driver is responsible for all control related operations for the APIC chip, as well as for supporting data movement for kernel resident protocols (e.g., TCP/IP, RATM, etc.). Figure 7.2 shows the modular structure of this driver. The PCI bus-dependent portions of the code have been abstracted out to enable easy porting of the driver to future revisions of the chip that are built for a different I/O bus. Most of the driver functionality that was deemed to be common to all ATM network interfaces was also abstracted out into an ATM device-independent driver module. The intention is to allow other ATM network adapter drivers to share this code with the APIC. The bulk of the APIC-specific code is contained in the bus-independent portion of the driver, which comprises the remainder of the code.

The ATM device-independent driver exports a common view of all ATM network interfaces to the higher layer protocols. This includes what VCs are supported by the device, and other such device capabilities. It also maintains a list of all active VCs and the device-independent state for each VC (for example, the AAL type and traffic parameters). This portion of the driver also handles AAL framing and LLC/SNAP encapsulation of packets, address resolution (i.e., translation of an IP address to an outgoing ATM VC), and address binding to allow routing table entries to be inserted for PVCs to different destinations.

The bus-independent portion of the kernel driver is responsible for recognizing the presence of an APIC device at bootup time, and for registering the device with the kernel's autoconfiguation code. Following this, it maps the global access and kernel-access per-channel registers of the APIC into the kernel's virtual address space. This part of the driver is also responsible for registering the interrupt service routine with the kernel's interrupt fielding code, and for setting up the endianness registers in the APIC based upon the machine's native endianness.

The bus-independent portion of the driver is responsible for almost everything else; some of the tasks it handles are:

- chip reset and initialization
- implementation of ioctls for setting up VCs and controlling other device parameters
- descriptor allocation and management
- transmit and receive processing
- interrupt handling
- page mapping for protected I/O
- setup of protected DMA channels

The current version of the driver uses simple DMA for transmit channels, and pool DMA for receive channels, and supports both IP and RATM as higher layer protocols.

7.2.1. Interaction with IP

All IP communcation is assumed to take place using ATM PVCs which have been pre-configured in network switches. First, the APIC needs to be assigned a local IP address using the `ifconfig` program; for example:

```
ifconfig apic0 128.252.169.100 netmask 0xffffffff up
```

Next, the PVC is setup using the `atm_ifconfig` utility, which talks directly with the ATM device-independent portion of the APIC's driver. The usage of the `atm_ifconfig` program is shown below:

```
atm_ifconfig apic0 [tx|rc] <vpivci> [aal0|aal5]
   [llc | nollc <protocol>] [lowdelay | paced <rate> | besteffort]
```

Finally, a route needs to be added to the system's routing table in order to allow the packets to be routed over this PVC. This is done using the system's `route` program; for example:

```
   route add -iface 128.252.169.200 -link apic0:0x000010
```

```
struct sockaddr_ratm sa;
struct vcparams vcparams;

/* Create socket */
sock = socket(AF_RATM, SOCK_DGRAM, 0);

/* Create and connect to a new VC */
sa.sratm_family = AF_RATM;
sprintf(sa.sratm_if, “apic0”);
sa.sratm_vpivci = 0x000010;
bind(sock, &sa, sizeof(sa));

/* Setup VC parameters */
vcparams.aal = AAL5;
vcparams.do_llc = 0;
vcparams.traftype = PACED;
vcparams.rate = 10*1024;
ioctl(sock, SIOCSETVCPARAMS, &vcparams);

/* Now can send packets on socket, will get
sent as AAL-5 frames at 10 Mbps */
write(sock, buf, sizeof(buf));
```

Figure 7.3: Example Code to Illustrate RATM Access to the APIC

This command inserts an entry into the system’s routing table which tells the system that all packets destined for host 128.252.169.200 are to be sent to device apic0, and that they should be sent out over the virtual circuit with VPI = 0x00 and VCI = 0x0010.

7.2.2. Interaction with RATM

The RATM (raw ATM) protocol implementation allows user-space applications to use the familiar socket API to transmit and receive AAL-0 or AAL-5 frames with the APIC. Figure 7.3 is an example code segment that binds a RATM socket to an APIC virtual circuit, and sets various

paramters associated with the VC before it starts sending data on the socket using standard socket system calls such as `sendto()` or `write()`.

Chapter 8

Experimental Results

The APIC has been successfully fabricated and tested in its first spin, with the exception of a couple of small bugs. The chip is currently in use in several projects, some of which utilize novel features such as the remote control capability. This chapter describes some of the experiments we performed on the APIC using the NetBSD kernel driver, and the results of those experiments.

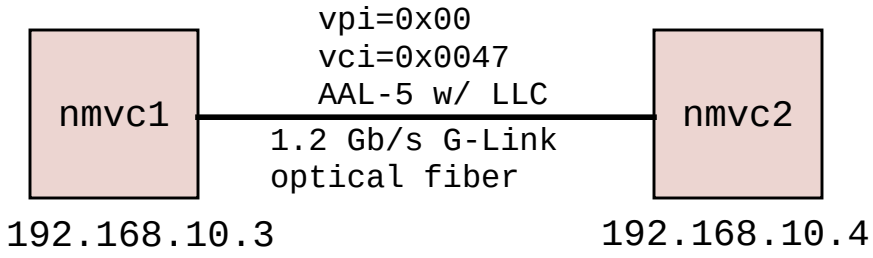

Figure 8.1: Experimental Setup

Figure 8.1 shows the experimental setup used in all our tests. It consists of two machines, `nmvc1` and `nmvc2`, each with an APIC in it. The two APICs are linked using an optical fiber with 1.2 Gb/s G-link. In most experiments, a virtual circuit with VPI=0 and VCI=0x47 is setup and used for communication in both directions. Both machines are 450 MHz Pentium II PCs with 128 MB of memory each. Both machines run NetBSD and use the locally developed APIC driver. Table 8.1 lists some of the performance metrics of these machines, as measured using the `lmbench` benchmark suite.

Table 8.1: Performance metrics for NetBSD on PCs used in experiments

CPU Memory read bandwidth	2.2 Gb/s
CPU Memory write bandwidth	1.4 Gb/s
CPU Memory copy bandwidth	1 Gb/s
Null system call latency	2 μs
Context switch latency	11 μs

8.1. Best-effort TCP Throughput

Out first test was to measure the TCP throughput for best-effort traffic. In this test, specially written client and server programs were used to transfer a 16 MB buffer 25 times over a TCP/IP socket connection. The socket buffer size was set to 64 KB. The observed throughput was about 300 Mb/s. While this may sound low, it is very competitive with some of the highest numbers reported in the literature for communication in this class of machine. It is important to note that because the standard TCP/IP stack is being used, the number of data touches from this stack is 5. In addition, because of a serious APIC bug, the driver is forced to re-order the words in a received packet; this results in an additional two data touches, bring the total number of data touches to 7. So a performance number of 300 Mb/s despite this high number of touches is encouraging.

8.2. Pacing Test for UDP Traffic

The next test we ran was intended to test the APIC's pacing logic. The VC used in the tests was setup as a paced channel with a variable pacing rate, and 25 datagrams of 9 KB each are transmitted over this VC. The socket buffer size was set to 240 KB, the maximum possible value in NetBSD. Note that all of the transmitted data can fit within the socket buffer; this is important because UDP has no builtin feedback, and since software is able to generate data faster than can be consumed by the APIC, sending more data than the socket buffer in a single send call would result in packet loss. The receiver records the time at which the first datagram is received, and also the time at which the last datagram is received. The difference is used to compute the rate at which data was received over the connection. In Figure 8.2, the throughput rate is plotted as a function of the rate parameter specified to the to the `atm_ifconfig` program. The latter is used in a special formula to compute the pacing parameter actually programmed into a register on the APIC chip, and it is intended that the

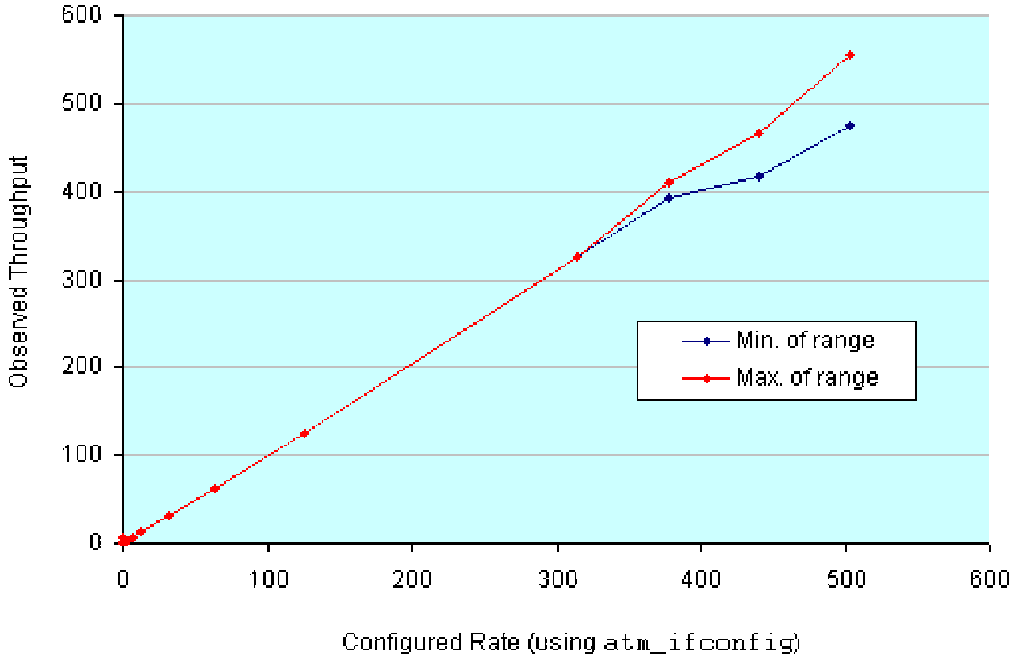

Figure 8.2: Throughput vs. Specified Pacing Rate for UDP Traffic

specified rate should closely match the rate at which the APIC actually sends out data on the interface. Each test was repeated a number of times, and the minimum and maximum observed throughputs from different test runs are plotted separately in the figure. As can be seen from the plot, the specified throughput does match the observed throughput, up until about 330 Mb/s, at which the observed throughput, which lies between the minimum and maximum curves, begins to level off. This is also the point at which the minimum and maximum rates diverge. The reason is that beyond these levels the machine's limitations in terms of CPU and bus utilization begin to affect the actual throughput that the APIC can achieve in reading and transmitting data over the connection. The uncertainty introduced by this interference also results in widely varying results on different runs of the experiment; the super-linear behavior of the curves between 450 Mb/s and 500 Mb/s can be attributed to this noisiness in the data collected. It is interesting to note, however, that throughputs as high as 550 Mb/s have been achieved, which is remarkable given the theoritical maximum is the PCI bus rate of 1 Gb/s, and given the number of data touches for each UDP packet is 7 (because of the APIC bug mentioned earlier). If the specified rate is increased beyond about 550 Mb/s, we begin to see packet loss (this is not shown in the figure), which is to be expected given that the transmitter is capable of sending at a higher rate than the receiving APIC can sink.

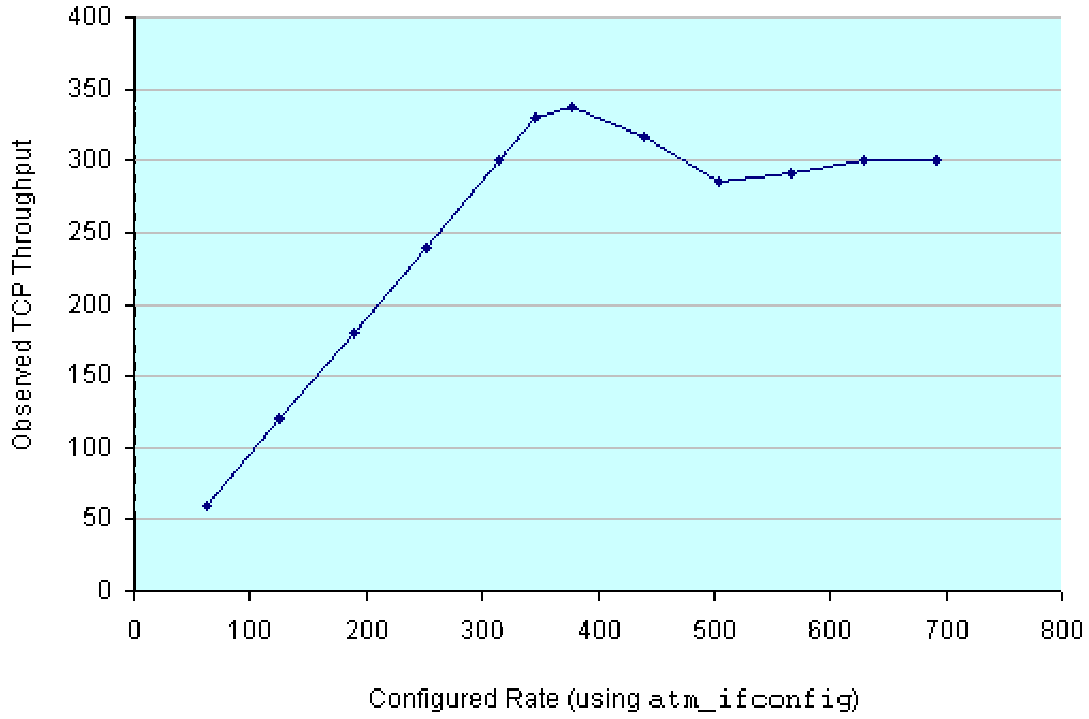

Figure 8.3: Throughput vs. Specified Pacing Rate for TCP Traffic

This test demonstrated that the APIC's pacing scheme works correctly and reliably for specified rates that lie within the realm of the capabilities of the machine and device.

8.3. Pacing Test for TCP Traffic

A pacing test was also applied to TCP traffic to ensure that the reliability and windowing functions of TCP do not hamper the ability to specify a pacing rate for a connection. A 16 MB buffer is transmitted 25 times over a TCP/IP socket connection with a 64 KB socket buffer. As before, the transmit channel's pacing rate is varied, and the observed throughput is plotted as a function of the specified rate. The result is shown in Figure 8.3. Notice that the throughput closely tracks the specified rate until about 330 Mb/s, after which it falls a little before stabilizing at 300 Mb/s for very high specified rates. This is consistent with the rate that was observed for best-effort TCP throughput. The dip in throughput can be explained by packet loss causing TCP to adjust its sending rate in order to match the network capacity available to it; the fact that the dip is so small is an indication of just how well the TCP flow control adjusts to network conditions.

8.4. End-to-end Delay and Driver Performance

In order to measure the end-to-end delay in a way that does not include operating system or protocol specific delays, we formulated an experiment that measures the total round-trip time for a single cell ping packet and subtracts the components of the delay that are spent in the operating system and protocols on both sides hosts. The experimental setup is illustrated in Figure 8.4. Probes are strategically placed at points in the protocol stacks of both machines, which use the Pentium's cycle counter to record the time for events corresponding to the packet reaching those points.

☞ Pentium processors have a register called the cycle counter which increments once every clock cycle, and can be read by a special instruction to obtain precise measurements of elapsed time.

The entry and exit points marked in the figure represent the probes, and correspond to the locations in the protocol stack listed in Table 8.2.

The quantities $(exit_1 - entry_1)$ and $(exit_3 - entry_3)$ represent the amount of time spent in the transmit portion of the APIC driver, and was measured to be an average of 2.7 μsec.

The quantities $(exit_2 - entry_2)$ and $(exit_4 - entry_4)$ represent the amount of time spent in the receive portion of the APIC driver, and was measured to be an average of 4.8 μsec.

The quantity $((entry_4 - exit_1) - (exit_3 - entry_2))$ represents the round-trip time between the APICs, exclusive of driver or protocol processing time; it can be assumed to be twice the delay from when the packet is committed to the sending APIC, to when the receiving CPU is interrupted to inform it of the packet's arrival. This round-trip time was measured at an average of about 30 μsec, or about 15 μsec of one-way delay. This one-way delay number includes the following hardware components: two uncached register writes (one for signalling channel attention on transmit, the other for acknowledging the receive interrupt), bus delays on either end, APIC on-chip delays, and propagation delay over about 3 meters of fiber. It also includes the software delay involved with fielding the interrupt when a packet is received. The latter can be a significant component of the one-way delay; independent third-party experiments have shown that fielding a null interrupt in NetBSD can take upto 10 μs on a Pentium class machine.

The round-trip time including driver processing delays can be computed as approximately 2*2.7 + 2*4.8 + 30 = 45 μsec, and is representative of the total round-trip times that would be seen

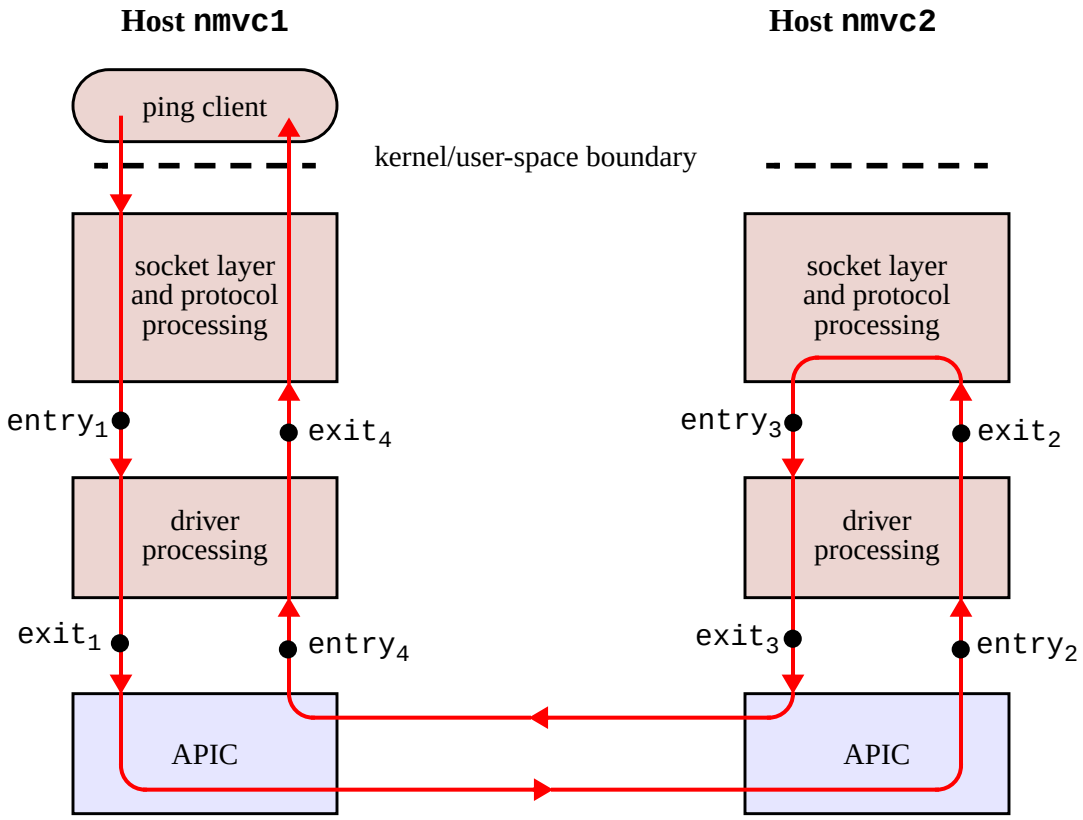

Figure 8.4: Measuring APIC delay and round-trip time performance

Table 8.2: Probe points in the protocol stack

Probe name	Description	Function entry point
$entry_1$ $entry_3$	Time at which the packet enters the transmit side of the kernel driver	`apic_devoutput()`
$exit_1$ $exit_3$	Time at which the packet is committed to the APIC by writing to the channel attention register	-
$entry_2$ $entry_4$	Time at which the driver's interrupt service routine is called upon packet arrival	`apic_intr`
$exit_2$ $exit_4$	Time at which the packet is delivered to higher layers by the driver	`atmc_input`

by an application if it were using the user-space control model to exchange data. Of course, the intervening switches in the network can add their own components to the end-to-end delay, which

should be taken into account, but the number illustrates the fact that the APIC is capable of efficiently supporting applications requiring end-to-end delays of as low as 22.5 μsec

8.5. Protected DMA Throughput and Delay Performance

Two experiments were designed to test the performance of user-mode access to the APIC using Protected DMA. In the first experiment, a packet is ping-ponged between a pair of user-space processes running on the two hosts in our experimental setup. With 100,000 back-to-back ping-pongs, the rate (number of ping-pongs per second) gives us a measure of the round-trip delay for user-space applications. The results of this test are representative of the overhead associated with a RPC call (a null RPC) implemented between the two user-space processes. This test was repeated in three different scenarios:

1. Using protected DMA. The packet is never copied (zero-copy), and there is no kernel involvement in the data path.
2. Using simple DMA. In this test, we use simple DMA from a privileged user-space process in exactly the same way that we would use protected DMA, again with no data copying. This is an "artificial" test, since simple DMA could not normally be used in this capacity unless the application process were privileged. However, this test would yield the highest possible performance for data transfer between two user-space processes, and therefore will serve as a baseline reference for comparison with the other cases.
3. Using UDP socket I/O. There are system calls involved on both receive and transmit, and there is one copy each on transmit and receive (to and from kernel space).

All three scenarios were run with two different packet sizes: a single cell packet, and a packet which results in an AAL-5 frame containing one full physical page worth of data (4 KB). The results are shown in Table 8.3. The round-trip time (RTT) for a single ping-pong is shown in the third column of this table, and demonstrates that protected DMA affords us an improvement of 570% over legacy UDP socket-based I/O for small packets, and about 360% for large packets. In other words, the performance of a latency-sensitive distributed application can be improved by a factor of 3 to 5 by using protected DMA to do direct transfers between user-space and the network adapter. The difference in performance between simple DMA and protected DMA gives us a measure of the cost incurred by the additional mechanisms required in the protected DMA scheme to

Table 8.3: Results of Ping-Pong Test

Packet size	Transfer Type	RTT
40 bytes (single cell)	Simple DMA from kernel	21.7 µsec
40 bytes (single cell)	Protected DMA	27 µsec
1 byte (single cell)	UDP socket	123.5 µsec
4072 bytes (one page)	Simple DMA from kernel	109.4 µsec
4072 bytes (one page)	Protected DMA	113 µsec
4000 bytes (one page)	UDP socket	393 µsec

Table 8.4: Results of User-Space Throughput Test

Packet size	Transfer type	Data Throughput
40 bytes (one cell)	Simple DMA	125 Mb/s
40 bytes (one cell)	Protected DMA	89.7 Mb/s
4 KB (one page)	Simple DMA	644 Mb/s
4 KB (one page)	Protected DMA	630 Mb/s

allow for protected access to the network card by multiple processes. The table shows that this "delay cost" is very small: under 24% for small packets, and only 3.3% for large packets.

The second experiment was aimed at discovering the throughput performance using protected DMA. Throughput was measured by sending 2048 packets back-to-back at full rate between the two machines. The experiment was carried out with direct access to the adapter from user-space processes running on the two machines, using both simple DMA (unprotected) and protected DMA. In both cases, there was no kernel involvement and the transfers were zero-copy from the user-space buffers. The experiment was repeated with both small (single cell) and large (4 KB) packets. The results are shown in Table 8.4. Notice that with large packets, it is possible to achieve data rates of well over 600 Mb/s using protected DMA; this is close to the theoritical peak rate of 1 Gb/s supported by the PCI bus. It is important to note that the latter is a theoritical number only, and can never be achieved in practice because of bus overhead (address and turnaround cycles), and other overheads from software, DMA, and SAR. The difference in performance between simple and protected DMA is a measure of the cost of adding protection support to the DMA mechanisms of

the APIC; in these experiments, this cost was 28% for small packets and only 2.2% for large packets. This demonstrates that with very little additional overhead, protected DMA can support very efficient user-space access to the network adapter.

Chapter 9

Conclusions

In this thesis we addressed the issues involved with architecting a network interface device for a high-speed ATM network. In this context, we identified a number of problems with conventional approaches to network interface design, and have proposed effective solutions to these problems.

9.1. Contributions

The specific contributions made in conjunction with the research effort described in this thesis include:

- A novel daisy-chained desk-area network architecture that can remove most constraints on the number of high-bandwidth multimedia devices that can be used within a host. This is achieved by using more memories and a cell-switched interconnect to get around memory bandwidth and I/O bus limitations. The interconnect features extremely low stage-to-stage latencies of under 10 μsec. The introduction of the concept of remote control for our network interface was an enabling technology for this architecture, which has been validated through the use of the APIC chip in several projects utilizing this feature.

☞ In particular, the Smart Port Card (SPC) project at Washington University has used the APIC's remote control facility to perform inline download and bootup of the SPC from a remote network node.

- Development of an implementation strategy for user-space control of a network interface. This strategy introduced the novel concepts of Protected I/O and Protected DMA, which

when taken together enable user-space control of a device without the need for a system-provided or on-board I/O MMU. The Protected I/O scheme enables protected access to individual registers on a chip by user-level processes; this was achieved by overloading the functionality already supported by a host's virtual memory protection hardware. The Protected DMA mechanism allows buffers to be enqueued for transmission or reception directly by a user process without kernel intervention, thereby providing for a very low delay communication mechanism for applications. Experiments have indicated that end-to-end delays as low as 22.5 μsec are achievable by such applications. The user-space control model also allows for a zero-copy architecture, which can improve throughput performance by decreasing the number of data touches.

- Introduction of the concept of interrupt demultiplexing. This scheme is simple to implement, and allows bandwidth-intensive and latency-sensitive applications to co-exist without one adversely affecting the other. When coupled with the user-space control model, interrupt demultiplexing provides a complete solution to the problem of receive livelock. Significantly, this has been achieved without requiring any complex software interaction between the network adapter and the operating system.

- An innovative Pacer design that allows the network interface to efficiently quality of service. This is the first hardware-based design we know of that can scale to support pacing of large numbers of connections with different pacing rates for each connection. The feasibility of an ASIC implementation of the d-heap algorithm used by our design has been confirmed, and experimental results have validated this approach.

- A complete and successful implementation of our proposed architecture in the form of a sophisticated ASIC developed using 0.35 micron technology, and comprising several hundred thousand gates (see Figure 9.1). The ASIC development was done using the VHDL hardware description language, and in addition our design process included development of a detailed behavioral simulation of the chip in C++, which was used for both functional correctness testing as well as for verification of the VHDL design.

- A complete kernel driver implementation for the APIC in the NetBSD operating system, including support for both TCP/IP and raw ATM protocols.

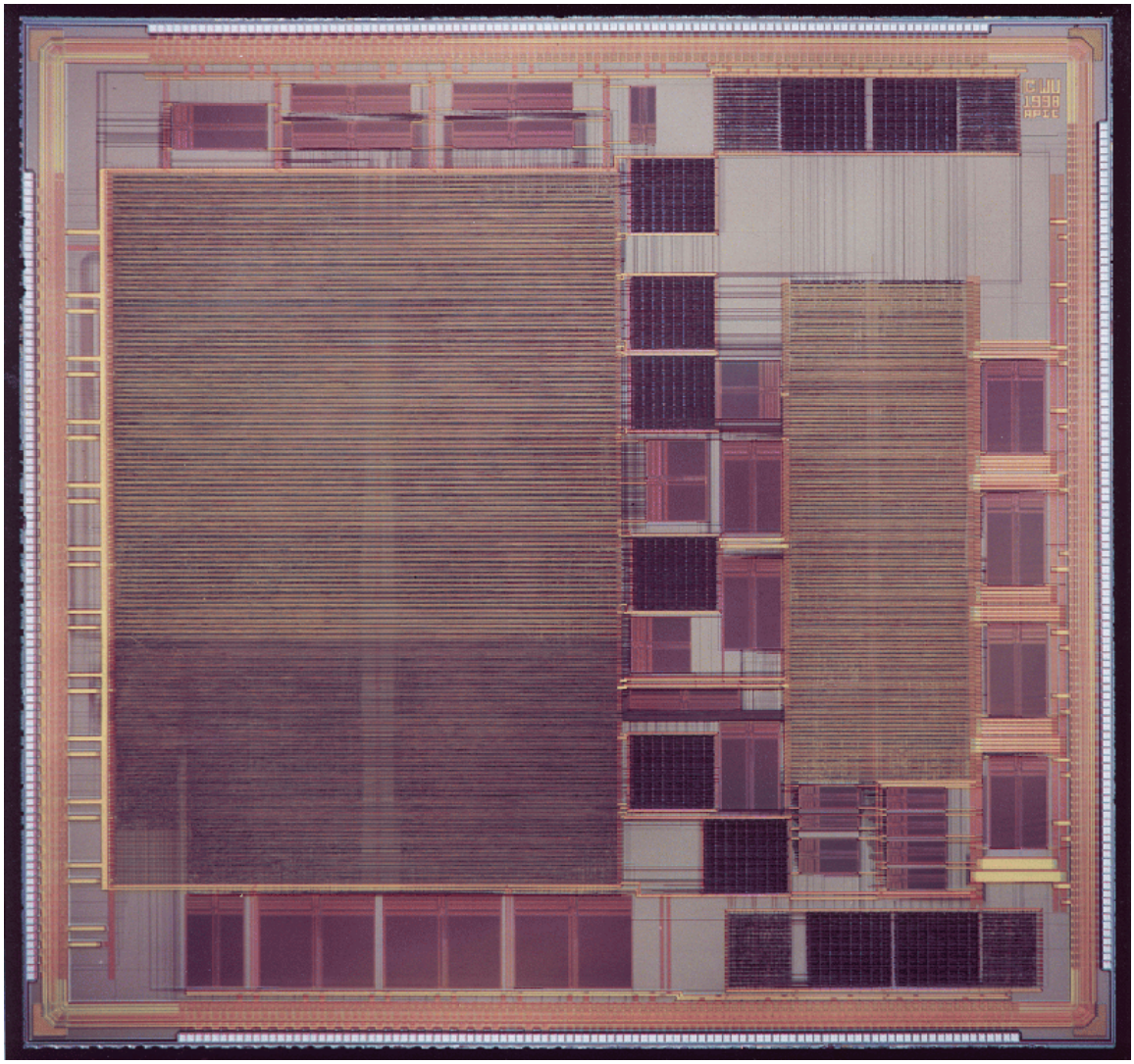

Figure 9.1: APIC Internal Layout

The APIC has achieved throughputs in excess of 300 Mb/s, despite a serious bug in the chip which necessitates two additional data touches for all received packets. This by itself is a significant illustration of the validity of our techniques; to put this in perspective, here is an excerpt from an article drawn from a very recent press article titled “Gigabit Ethernet hits second gear” (dated March 20, 2000):

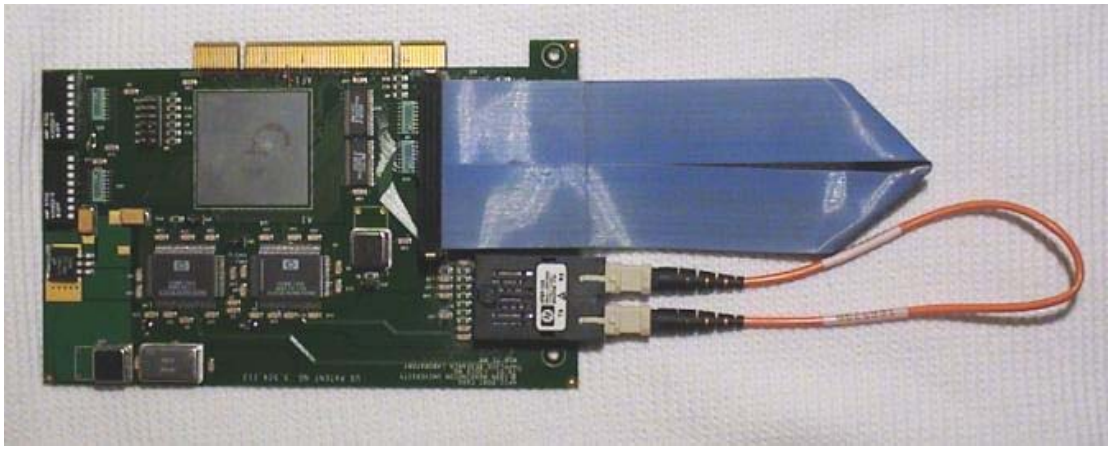

Figure 9.2: The APIC Network Interface Card

> In our previous round of testing, the best Gigabit Ethernet performance we saw was 29 Mbps measured in a file transfer between a Windows NT 4.0 server and Windows 95 client. This amounted to a meager 3% bandwidth utilization over Gigabit Ethernet.
>
> Using second-generation Gigabit Ethernet products and Windows 2000, the best real-world throughput was 158 Mbps, which sets bandwidth utilization at 16 percent. On average, we measured performance ranging from 137 to 145 Mbps. (See Table 1). These results bode well for connecting server and high performance workstations directly to Gigabit Ethernet - something we couldn't recommend previously.

By most metrics the APIC effort has been successful. It is currently in use in several research projects both at Washington University and elsewhere. It has seen widespread distribution as a research vehicle under the NSF-sponsored Gigabit Kits initiative. Because of its open architecture and open source drivers, the chip has attracted developers who are using it to further their own agendas; for example, we know of at least two independent efforts to port the APIC driver to the Linux operating system. Although time limitations did not permit us to fully explore all of the features of the APIC, it is expected that many of these features will be exercised and several of our innovations will see validation and proof of concept through the continuing work of other developers.

9.2. Future Work

There are several areas, especially related to software support for the APIC chip, where significant contributions could be made:

- A full implementation of a user-space driver in library form would help validate the user-space control model, and allow for better evaluation of the gains to be achieved by using this model.

- A new API that is different from the socket API is needed in order to efficiently support zero-copy semantics. The design of such an API, that can work both with the user-space control model as well as with kernel-based zero-copy schemes, would make for a very interesting project for future research.

- A comprehensive remote control library that could be used to develop generalized user-space device-drivers for devices in a desk or system area network can be a very useful future contribution that could pave the way for greater use of such networks.

- On the hardware side, quality of service support in network interfaces for packet-based networks could be a useful area of future work. The pacing algorithms described in this thesis do not cover variable length packets, and efficient hardware implementation of schemes such as weighted fair queuing (WFQ) for network adapters is an open area for research.

- Many of the schemes described in this thesis have made use of the fact that ATM data received by the adapter can be easily demultiplexed based on the VC. Similar techniques cannot be directly applied to packet based adapters unless an efficient hardware packet classification engine is developed which can classify packets into end-to-end flows (which can then be treated in a manner similar to ATM VCs). The design of such packet classifiers is currently a hotly researched topic, but most of these efforts are targeted at classifiers used in routers, rather than network interfaces.

9.3. Closing Remarks

With the ASP (application service provider) model becoming ubiquitous in the Internet, more and more applications are migrating from the desktop to large servers. These servers will need to connect to the Internet over very high bandwidth links, and be able to efficiently serve content and applications to thousands, and even millions of end points. With multimedia services gaining in

popularity, the servers serving up such content will have to scalably provide quality of service for all the streams they handle. In this environment, there is ample opportunity to design new and improved network interfaces that can handle not only very high data transfer rates, but also be able to perform more complicated QoS functions such as pacing or WFQ over very large numbers of connections. The APIC effort represents only a first step in that direction, and the future will likely see many more challenges and advances in this very exciting field.

References

[1] Arnould, Emmanuel et al; “The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers,” *ASPLOS-III (ACM SIGOPS Operating Systems Review)*, Vol. 23, New York, April 1989, pp. 205-216.

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

[3] 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.

[4] Cheriton, David R.; “VMTP: A Transport Protocol for the Next Generation of Computer Systems,” *Proc. ACM SIGCOMM ‘86*, Vol. 16, No. 3, 1986, pp. 406-415.

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

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

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

[8] 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.

[9] Dalton, C.; Watson, G.; Banks, D.; Calamvokis, C.; Edwards, A.; and Lumley, J.; “Afterburner,” *IEEE Network*, Vol. 7 No. 4, July 1993, pp. 36-43.

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

[11] Demers, A.; Keshav, S.; and Shenker, S.; “Analysis and Simulation of a Fair Queueing Agorithm,” *Proc. ACM SIGCOMM 87*, Austin, Texas, Sep. 1987.

[12] 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.

[13] Druschel, P.; and Banga, G.; “Lazy Receiver Processing (LRP): A network subsystem architecture for server systems,” *Proc. Second USENIX Symp. on Operating Systems Design and Implementation*, pp. 261-276, October 1996.

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

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

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

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

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

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

[24] 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.

[25] Greaves, David J.; McAuley, Derek; and French, Leslie J.; “Protocol and Interface for ATM LANs,” *Proc. 5th IEEE Workshop on Metropolitan Area Networks*, Taormina, Italy, May 1992.

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

[27] 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.

[28] 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.

[29] IEEE Journal on Selected Areas in Communications, Vol. 11 No. 2,” February 1993.

[30] Indiresan, Arti;, Mehra, Ashish; and Shin, Kang G.; “Receive Livelock Elimination via Intelligent Interface Backoff,” `http://rtcl.eecs.umich.edu/outgoing/ashish/end-rll.ps`

[31] Jacobson, Van; “Efficient Protocol Implementation,” *ACM SIGCOMM 90 tutorial*, Sep. 1990.

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

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

[34] Mogul, J., and Ramakrishnan, K.K., “Eliminating receive livelock in an interrupt-driven kernel,” ACM Trans. Computer Systems, vol. 15, No. 3, Aug. 1997.

[35] Neufeld, Gerald W.; Robert Ito, Mabo; Goldberg, Murray; McCutcheon, Mark J.; Ritchie, Stuart; “Parallel Host Interface for an ATM Network,” I*EEE Network*, July 1993.

[36] Partridge, C., “Gigabit Networking,” *Addison Wesley*, 1994.

[37] 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.

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

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

[40] Thekkath, C., Nguyen, T., Moy, E., and Lazowska, E. “Implementing Network Protocols at User Level,” *Proc. ACM SIGCOMM ‘93*, Sep. 1993.

[41] 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.

[42] Turner, Jonathan S.; “Efficient Cell Pacing in ATM NICs,” *presentation made in the Department of Computer Science at Washington University*, July 14, 1997.

[43] “Virtual Interface Architecture Specification,” (Compaq, Intel, Microsoft): http://www.viarch.com.

[44] Zhang, Lixia; “Virtual Clock: A New Traffic Control Algorithm for Packet Switched Networks,” *ACM Trans. on Computer Systems*, Vol. 9, No. 2, May 1991, pp. 101-124.

Vita

Zubin D. Dittia February 4, 2001

zubin@dittia.com

WORK EXPERIENCE

2000-2001: **Jibe Networks, Inc.**
Co-Founder and Chief Scientist

2000: **Cisco Systems, Inc.** (through acquisition of Growth Networks, Inc.)
Hardware Engineer, HFR Business Unit
Important architectural contributions to Cisco's next generation multi-terabit router fabric. Developed from scratch the performance simulation environment (in C++) for the HFR system, which is currently being extended and used by 5-10 engineers in Cisco.

1998-2000: **Growth Networks, Inc.**
Founding Engineer
Key architect of Growth Networks' ten-terabit switching fabric chips. Also designed and implemented a clock-tick accurate performance simulator for these chips, and worked with customers to address requirements and tune system performance to their needs.

EDUCATION

1993-2001: **D.Sc. in Computer Science**

Applied Research Laboratory, Washington University, St. Louis.

Architecture, design, and implementation of a 1.2 Gb/s network interface chip and supporting operating system software. Chip is currently in use at 30+ universities and labs for high performance networking research.

1990-1992: **M.S. in Electrical Engineering**

Washington University, St. Louis.

Implementation and evaluation in the SunOS Unix kernel of TCP extensions for high bandwidth-delay product networks (RFC 1323).

1986-1990: **B.Tech. in Electrical Engineering**

Indian Institute of Technology, Bombay, India.

OTHER PROJECTS

Tree Bitmap

An algorithm for high-speed IP address lookups and packet classification. Currently being implemented by *Cisco Systems, Inc.* in a line card chip for their next generation multi-terabit router (HFR).

Router Plugins

A kernel-based software architecture for access routers. Implemented in the NetBSD kernel. Joint project with ETH Zurich. Technology licensed by *Ascom AG* and *Inalp AG*.

REFERENCE CONTACTS

Upon request

PATENTS

Decasper, Dan S.; and Dittia, Zubin D.; “Intelligent Content Precaching,” App. No. 09/566,068, filed 05/05/00 (Jibe Networks, Inc.).

Decasper, Dan S.; and Dittia, Zubin D.; “Personalized Content Delivery Using Peer-to-Peer Precaching,” App. No. 09/660,991, filed 09/13/00 (Jibe Networks, Inc.).

Decasper, Dan S.; and Dittia, Zubin D.; “Infrastructure for On-Site Service Providers (OSP),” App. No. 60/203,375, filed 05/09/00 (Jibe Networks, Inc.).

Dittia, Zubin D.; and Turner, Jonathan S.; “Method and Apparatus for Accumulating and Distributing Traffic and Flow Control Information in a Packet Switching System”, (Cisco Systems, Inc.).

Turner, Jonathan S.; and Dittia, Zubin D.; “Method and Apparatus for Controlling Input Rates Within a Packet Switching System,” (Cisco Systems, Inc.).

Dittia, Zubin D.; Fingerhut, Andrew J.; and Lenoski, Daniel E.; “Method and Apparatus for Distributing Packets Across Multiple Paths Leading to a Destination,” App. No. 09/519,715, filed 3/7/00 (Growth Networks, Inc.).

Dittia, Zubin D., Eatherton, William N.; Fingerhut, Andrew J.; Galles, Michael B.; and Turner, Jonathan S.; “Accumulating and Distributing Flow Control Information via Update Messages and Piggybacked Flow Control Information in Other Messages In a Packet Switching System,” App. No. 09/521,278, filed 3/7/00 (Growth Networks, Inc.).

Turner, Jonathan S.; and Dittia, Zubin D.; “Method and Apparatus for Scheduling Packets Being Sent from a Component of a Packet Switching System,” App. No. 09/519,721, filed 3/7/00 (Growth Networks)

Peris, Vinod Gerard John; Turner, Jonathan S.; and Dittia, Zubin D; “Method and Apparatus for Delaying Packets Being Sent from a Component of a Packet Switching System,” App. No. 09/520,685, filed 3/7/00 (Growth Networks, Inc).

Lenoski, Daniel E.; Dittia, Zubin D.; Fingerhut, Andrew J.; and Eatherton, William E.; “Method and Apparatus for Reducing the Required Size of Sequence Numbers Used in Resequencing Packets,” App. No. 09/519,716, filed 3/7/00 (Growth Networks, Inc.).

Turner, Jonathan S.; Dittia, Zubin; and Fingerhut, Andrew J.; “Telecommunications Interconnection Network With Distributed Resequencing,” App. No. 09/520,684, filed 3/7/00 (Washington University).

Eatherton, William E.; and Dittia, Zubin D.; “Data Structure Using a Tree Bitmap and Method for Rapid Classification of Data in a Packet Switch or Router,” App. No. 09/371,907, filed 8/10/99 (Washington University).

PUBLICATIONS

Dittia, Zubin D.; Parulkar, Guru M.; and Cox, Jerome R.; “The APIC Approach to High Performance Network Interface Design: Protected DMA and Other Techniques,” *Proc. IEEE Infocom 97*, Kobe, Japan 1997.

Decasper, Dan S.; Dittia, Zubin D.; Parulkar, Guru M.; and Plattner, Bernhard; “Router Plugins: A Software Architecture for Next Generation Routers,” *IEEE/ACM Trans. on Networking*, February 2000. Also appeared in *Proc. of ACM SIGCOMM ‘98*, Vancouver, Canada, September 1998.

Eatherton, William E.; Dittia, Zubin D.; and Varghese, George; “Tree Bitmap: Hardware/Software IP Lookups with Incremental Updates,” April 1999.

Decasper, Dan S.; Waldvogel, Marcel; Plattner, Bernhard; Adiseshu, Hari; Dittia, Zubin D.; and Parulkar, Guru M; “A Toolkit for Integrated Services over Cell-Switched IPv6,” *IEEE ATM 97*, Lisboa, Portugal.

Dittia, Zubin D.; Cox, Jerome R.; and Parulkar, Guru M.; "Design and Implementation of a Versatile Multimedia Network Interface and I/O Chip," *Proc. 6th Intl. Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, April 1996.

Dittia, Zubin D.; Cox, Jerome R.; and Parulkar, Guru M.; "Design of the APIC: A High Performance ATM Host-Network Interface Chip," *Proc. IEEE INFOCOM 95*, Boston, 1995, pp. 179-187.

Krchnavek, Robert R.; Chamberlain, Roger D.; Barry, T.; Malhotra, V.; and Dittia, Zubin D.; "Optical Interconnect Design for a Manufacturable Multicomputer," *2nd International Conference on Massively Parallel Processing using Optical Interconnections*, October 1995.

Dittia, Zubin D.; Cox, Jerome R.; and Parulkar, Guru M.; "Washington University’s Gigabit ATM Desk Area Network," *Proc. of 9th Annual IEEE Workshop on Computer Communications*, October 1994.

Dittia, Zubin D.; Cox, Jerome R.; and Parulkar, Guru M.; "Using an ATM Interconnect as a High Performance I/O Backplane," *Proc. of HOT INTERCONNECTS 94*, August 1994.

Turner, Jonathan S.; ARL staff; and ANG staff; "A Gigabit Local ATM Testbed for Multimedia Applications: System Architecture Document for Gigabit ATM Switching Technology," Document prepared by: Dittia, Zubin D.; and Fingerhut, Andy; December 1994.

Dittia, Zubin D.; Cox, Jerome R.; and Parulkar, Guru M.; "Catching Up With the Networks: Host I/O at Gigabit Rates," *Washington University Technical Report WUCS-94-11*, March 1994.

Parulkar, Guru M.; Buddhikot, Milind M.; Cranor, Charles D.; Dittia, Zubin D.; and Papadopoulos, Christos; "The 3M Project: Multipoint Multimedia Applications on Multiprocessor Workstations and Servers," *Proc. of IEEE Workshop on High Performance Communication Systems*, September 1993.

Anderson, Jim M.; Parulkar, Guru M.; and Dittia, Zubin D.; "Persistent Connections in High Speed Internets," *Proc. GLOBECOMM*, December 1991.

Gong, Fengmin; Dittia, Zubin D.; and Parulkar, Guru M.; "A Recurrence Model for Asynchronous Pipeline Analysis," *Washington University Technical Report WUCS-91-14*, 1991.

Dittia, Zubin D.; Kumar, P.S.; Mundkur, P.Y.; and Desai, Uday B.; "A Parallel Scheme for Adaptive 2D Parameter Estimation," *Proc. Workshop on Parallel Processing*, Bhabha Atomic Research Center, Bombay, February 1990.

Mundkur, Prashanth Y.; Dittia, Zubin D.; and Desai, Uday B.; "A Systolic Architecture for Adaptive 2D Parameter Estimation," *Proc. Workshop on Signal Processing, Communications and Networking*, Indian Institute of Science, Bangalore, July 1990. Communication Systems, September 1993.

PROFESSIONAL ACTIVITIES

Publicity co-chair for ACM Multimedia 94. Reviewer for papers submitted to the IEEE/ACM Transactions on Networking, the Journal of Computer Communications, ACM Transactions on Internet Technology, ACM SIGCOMM, ACM ASPLOS, IEEE INFOCOM, IEEE Network magazine, IEEE Multimedia, NOSSDAV, ACM PODC, and IEEE ICNP.