Catching Up With the Networks: Host I/O at Gigabit Rates

Authors: Zubin D. Dittia, Jerome R. Cox Jr., and Guru M. Parulkar

The last few years have seen network data rates skyrocket from a few Mbps to a Gbps or more. However, a lack of integration of the host-network interface, the operating system, and network protocols has resulted in end-applications seeing only a small fraction of this total bandwidth being available for data transfer. The emergence of demanding applications in the realms of multimedia and virtual reality provides further impetus in the drive to overcome this problem. In this paper, we present the design of a high performance ATM host-network interface for workstations and servers that can support a bidirectional sustained data rate in excess of a gigabit per second. A prototype of the interface is being build at Washington University as part of an ARPA-sponsored gigabit local ATM testbed. Our interface design, which emphasizes seamless integration with the OS and network protocols, features: support for streaming data from I/O devices (e.g., cameras, disk arrays, etc.) to the network or vice-versa, as well as from device-to-device, while bypassing the main system bus; an ATM interconnect that...

Read complete abstract on page 2.
Catching Up With the Networks: Host I/O at Gigabit Rates

Complete Abstract:

The last few years have seen network data rates skyrocket from a few Mbps to a Gbps or more. However, a lack of integration of the host-network interface, the operating system, and network protocols has resulted in end-applications seeing only a small fraction of this total bandwidth being available for data transfer. The emergence of demanding applications in the realms of multimedia and virtual reality provides further impetus in the drive to overcome this problem. In this paper, we present the design of a high performance ATM host-network interface for workstations and servers that can support a bidirectional sustained data rate in excess of a gigabit per second. A prototype of the interface is being build at Washington University as part of an ARPA-sponsored gigabit local ATM testbed. Our interface design, which emphasizes seamless integration with the OS and network protocols, features: support for streaming data from I/O devices (e.g., cameras, disk arrays, etc.) to the network or vice-versa, as well as from device-to-device, while bypassing the main system bus; an ATM interconnect that extends to the desk-area; a zero-copy interface to system memory that is achieved through the use of page remapping techniques; full AAL-5 segmentation and reassembly; pacing control that provides for single-parameter bandwidth reservation; a high degree of scalability in terms of the number of I/O devices that can be simultaneously supported; low-cost (one ASIC); and multiprocessor support.

This technical report is available at Washington University Open Scholarship: http://openscholarship.wustl.edu/cse_research/333
Catching Up With the Networks: Host I/O at Gigabit Rates

Zubin D. Dittia, Jerome R. Cox, Jr. and Guru M. Parulkar

WUCS-94-11

April 1994

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis MO 63130-4899

This work was supported in part by the Advanced Research Projects Agency (ARPA), the National Science Foundation (NSF), and an industrial consortium of Ascom Timeplex, Bell Communications Research, Bell Northern Research, Gold Star Information and Communication, NTT, Italtel SIT, NEC America, Southwestern Bell, SynOptics Communications and Tektronix.
Catching Up With the Networks: Host I/O at Gigabit Rates

Zubin D. Dittia  Jerome R. Cox, Jr.  Guru M. Parulkar
zubin@dwarkin.wustl.edu  jrc@hobbs.wustl.edu  guru@flora.wustl.edu

Department of Computer Science
Washington University in St. Louis
St. Louis, MO 63130.

Abstract

The last few years have seen network data rates skyrocket from a few Mbps to a Gbps or more. However, a lack of integration of the host-network interface, the operating system, and network protocols has resulted in end-applications seeing only a small fraction of this total bandwidth being available for data transfer. The emergence of demanding applications in the realms of multimedia and virtual reality provides further impetus in the drive to overcome this problem. In this paper, we present the design of a high performance ATM host-network interface for workstations and servers that can support a bidirectional sustained data rate in excess of a gigabit per second. A prototype of the interface is being built at Washington University as part of an ARPA-sponsored gigabit local ATM testbed. Our interface design, which emphasizes seamless integration with the OS and network protocols, features: support for streaming data from I/O devices (e.g., cameras, disk arrays, etc.) to the network or vice-versa, as well as from device-to-device, while bypassing the main system bus; an ATM interconnect that extends to the desk-area; a zero-copy interface to system memory that is achieved through the use of page remapping techniques; full AAL-5 segmentation and reassembly; pacing control that provides for single-parameter bandwidth reservation; a high degree of scalability in terms of the number of I/O devices that can be simultaneously supported; low-cost (one ASIC); and multiprocessor support.

† This work was supported in part by the Advanced Research Projects Agency (ARPA), the National Science Foundation (NSF), and an industrial consortium of Ascen Timeplex, Bell Communications Research, Bell Northern Research, Goldstar Information and Communication, NTT, Italtel SIT, NEC America, Southwestern Bell, SynOptics Communications, and Tektronix.
1 Introduction

The rapid emergence of very high speed packet switched networks has opened up a whole range of possibilities for new and exciting distributed applications. But even as the first gigabit networks make their appearance, it is becoming increasingly evident that merely raising raw network bandwidth does not translate to improved performance for end-applications. The bottleneck is easily identified to lie within the end-hosts (i.e., the workstations or servers that run the end-applications). Within an end-host, a number of problem areas can be identified that contribute to this bottleneck, not the least of which is a lack of integration between the hardware architecture (particularly that of the host-network interface), the operating system, and network communication protocols. It is a well known fact that operations such as data copying, software checksumming, servicing of interrupts, and context switches are often responsible for poor performance. By carefully designing the network interface, OS, and protocols in a coherent and integrated fashion, it is possible to minimize, or in some cases to completely annul, the handicaps resulting from the use of these operations.

Solving the aforementioned problems would not, however, overcome the inherent limitations imposed by the peak system memory and bus bandwidths, which are often lower than the peak network bandwidth in a gigabit network. By rethinking the host I/O subsystem, and moving processing power closer to network I/O, it is possible to bypass the main system memory and bus altogether. This approach is especially well suited for continuous media data (e.g. video, audio, etc.), where the processing is usually of a very specialized nature (e.g. signal processing, compression, encryption, etc.) and needs to be carried out in real time. However, for most other types of data, and in particular for discrete media data (e.g. graphics, text, simulation data, etc.), it is still important to provide for a high speed pathway to the main system memory.

In this paper, we describe the design of an ATM host-network interface architecture for high performance multimedia workstations and servers. A prototype of the interface, which will allow for a sustained data transfer rate of 622 Mbps simultaneously in both directions (into and out from the network), is currently in the process of being built. This effort, which is slated for completion in 1995, is part of a bigger project (funded by ARPA), which involves the setup of a gigabit local ATM testbed at Washington University and the demonstration of some example applications.

Our proposed host-network interface architecture uses a unique daisy-chained ATM interconnect which can interface to various I/O devices, to the ATM link interface, and to the main system bus. This interconnect, which can be likened to a high speed I/O bus, serves to provide a communication pathway that would allow data on some connections (such as those carrying continuous media traffic) to skirt the main
system bus. The other innovative aspect of our design involves the interface to the main system bus—we show how it is possible, through careful integration with the network protocols and operating system, to achieve a unique zero-copy interface that minimizes or circumvents many of the penalties associated with data copying, checksumming, context switches, and interrupt processing. Our architecture also incorporates full AAL-5 segmentation and reassembly, and pacing\textsuperscript{1} control based on peak bandwidth allocation. Furthermore, it is highly scalable (in terms of the number of I/O devices that can be simultaneously supported), inexpensive (the prototype implementation will have only one ASIC), and its reach extends to external devices that lie within the desk-area.

Note that although our design assumes an underlying ATM-like network, most of our ideas are applicable (perhaps with some modification) to many other types of networks, including connectionless (datagram) networks.

Since our emphasis has been on achieving a high degree of integration between the network interface architecture, OS, and protocols, our description of the design will include frequent mention of issues relating to transport protocols and operating systems. However, these issues cannot be discussed in detail due to space limitations. The outline of the paper is as follows: Section 2 provides the motivation for a new host-network interface design, and Section 3 gives a brief overview of various existing network interface designs. Section 4 describes our proposed architecture. The heart of this architecture is a custom VLSI chip called the APIC (ATM Port Interconnect Controller). Section 5 details the internal design of this chip. The design of interface to the host system bus, and how it can be used to achieve zero-copy semantics, is reported in Section 6. Finally, Section 7 provides some concluding remarks and outlines our future plans.

2 Motivation

The research community has expended considerable effort toward the design of high speed networks and switching systems [4, 12]. Experiences from the five gigabit testbeds have shown that gigabit networking is not only possible, but can be very useful in a wide variety of application areas. At Washington University, we have an installed and operational 155 Mbps packet switched network [6], and we are currently in the process of building an ATM switching system [18] that will support data rates of 622 Mbps (SONET) and 2.4 Gbps (G-link) per port. Our primary objective in designing the host-network interface has been to support applications that can utilize these data rates, yet run on typical workstations and servers and deliver reasonably high performance.

\textsuperscript{1} Pacing (traffic shaping) is required to prevent bursty traffic sources from causing network congestion, and to avoid overwhelming a receiver that cannot handle a very bursty input.
The specific classes of applications targeted include point-to-point as well as multipoint multimedia and virtual reality applications. Multimedia applications are characterized by their requirement for transport of multiple synchronized media streams, possibly between a number of simultaneously participating and widely distributed hosts. Some of these streams (typically video streams) have high bandwidth and stringent real-time requirements. Multimedia applications involving collaboration or hypermedia navigation may also include a significant amount of user interaction. Virtual reality applications are characterized by very high data rates, a high degree of user interaction, and low latency requirements. Most of the applications we are considering require not only high bandwidth (hundreds of Mbps) and low latency, but also significant amounts of computing power. These compute requirements stem from operations such as compression/decompression, data encryption, image and speech processing, and computer graphics; in many cases, the required processing power can be in billions of operations per second.

The current state-of-the-art in workstation architectures (see Figure 1) typically includes: support for multiple processors; a main system bus that interfaces to the processors and main memory, and provides support for snooping based cache coherency protocols; a main memory that is organized as a single DRAM bank; and an I/O bus to which most of the I/O devices, including the network interface, are connected. This type of architecture is not well suited to the classes of applications we are interested in—the primary limitations arise from the bottlenecks imposed by the two buses\(^2\), and from the fact that the effective memory bandwidth is normally no more than 500 Mbps. To compound the problem, protocol stack implementa-

\(^2\) The effective bandwidth of the fastest buses in use today is no longer much higher than the network bandwidth (of emerging networks). With multiple bus traversals, the available bandwidth is typically much lower than the network bandwidth. Typical high performance workstation effective bus bandwidths range from 500 Mbps to 2 Gbps.
tions in existing operating systems require data to be copied multiple times (once from the network interface to kernel space, and then from kernel space to the application’s address space). Each such copy step increases the number of times the data has to traverse the system bus, and also increases the number of times memory is accessed. Furthermore, an application may need to access the data, possibly more than once, in order to perform some computation on the data, or just to copy it on a disk. Add to this the fact that multiple processors working on multiple media streams have to share the same bus, and it becomes apparent why many of the interfaces designed to support high speed networks fail to deliver even a small fraction of the network bandwidth to end applications. Clearly, there is a need to revise the host communication architecture and I/O subsystem, if we are to be able to meet the demands imposed by our choice of target applications.

3 Related Work

Several research groups have attempted to design and implement high speed host-network interfaces over the past few years [10]. One of the earliest efforts include the network adaptor board (NAB) [11], which was especially designed to support the VMTP transport protocol [5]. Important aspects of this interface include clever partitioning of protocol processing between the host CPU and a general purpose on-board microprocessor, and the use of a VRAM to allow concurrent protocol processing and transmission/reception of network data. The VRAM is also used to buffer packets in a store-and-forward manner for resequencing before data can be delivered to the host.

The Nectar communication accelerator board (CAB) [2] is a host-network interface that connects through a 10 Mbyte/sec VME interface to the host system. The CAB’s on-board processor runs a small kernel and is responsible for transport protocol processing. The unique feature of this interface is that an extra store-and-forward hop can be avoided by mapping the CAB’s on-board data memory into the address space of the node process.

Some of the more recent efforts have been concerned with the design of host-network interfaces for ATM networks [8, 17]. The Belfcore Interface [8] was designed for the DecStation 5000, and it connects to the hosts TURBOchannel I/O bus on one side and to a 622 Mbps SONET STS-12 link on the other. This interface places all data movement and per-cell operations in custom hardware. Control path functions including segmentation and reassembly are implemented using two on-board processors, thus making the interface very flexible. The University of Pennsylvania interface [17] was designed for the IBM RS/6000, and it interfaces to a SONET OC-3c link (155 Mbps). It features hardware implementation of the ATM segmentation and reassembly pipeline, and carefully partitions the protocol stack so that all per-cell functions can be carried out using dedicated hardware, while software is responsible for higher-level protocol pro-
cessing and for handling packet movement to/from host memory. We believe this approach affords maximum flexibility without compromising efficiency.

The Afterburner [7] interface design uses a VRAM to hold packets that have been received over the network. The VRAM is mapped into the host kernel's address space, so protocols can examine headers while the packets are still in the VRAM. Once it is determined where the packet is to be copied (into the application's address space), the copying is done by the CPU. Checksums can be computed in this copy loop. This interface design is highly flexible, scales easily, and can deliver high performance. The Medusa interface [3] is nothing but an Afterburner designed for a 100 Mbps FDDI network.

Washington University's Axon project [14, 15] also represents an attempt at designing a high performance host communication architecture for high bandwidth distributed applications. The proposed architecture allows processes to share their virtual memory address spaces, and 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 host-network interface 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 done in hardware. However, recent trends suggest that higher level protocol processing can be left to software for flexibility, without compromising performance. The Axon architecture has been simulated but has not been implemented.

All of the above interfaces connect to the host I/O bus and thus share the limitations mentioned earlier (see Section 2). These interfaces also require at least one data copy per packet from the interface buffer to application buffer. Finally, these interfaces are targeted for single processor machines and deliver all network data into the main memory; thus, they are not particularly suitable for multimedia multiprocessor workstations.

4 Proposed Architecture

A high level schematic of our proposed host-network interface architecture is shown in Figure 2. At the heart of this architecture is a daisy-chained interconnect comprising of a number of ATM Port Interconnect Controller (APIC) chips. Individual chips interface either directly to the main system bus, or through a dual ported memory (in this paper, we shall concentrate only on dual ported memories that are implemented as VRAMs) to an I/O device. One of the APIC chips on the interconnect is connected to the external ATM network through a link interface. ATM cells arriving from the network pass from one APIC to another until they reach an APIC chip that is directly connected to the destination device (or to the system bus). Similarly, data originating from a source device is passed as an ATM cell stream from the APIC directly connected to that device, down the APIC interconnect towards the link interface, and finally out to
the network. It is also possible for two local devices to communicate over the APIC interconnect, without
cells ever having to leave the desk area. Thus, for example, a video cell stream originating at a camera
could traverse the interconnect in order to reach the local display device. The APIC which directly
interfaces to the system bus can be used for traffic originating from, or destined for, the system's main memory
(we will have more to say concerning this interface in Section 6). All other traffic movement happens only
over the APIC interconnect, thus sparing the main system bus of this burden. In our prototype implementa-
tion, each APIC will interface to another APIC over two 622 Mbps links (one is used for incoming traffic, and
the other for outgoing traffic). The link to the ATM network switch port also operates at 622 Mbps. Thus,
our prototype interface will be capable of supporting an aggregate sustained bidirectional data rate in
excess of a gigabit per second. One of the major advantages of allowing full rate bidirectional traffic on the
interconnect is that each device can access the network in much the same way as if the device had exclusive
access to the network; the bandwidth reservation on the network can, in this case, be thought of as extend-
ing into the APIC interconnect, all the way to the device. This advantage would be lost if, for example, the
interconnect had a unidirectional ring topology in place of a daisy chained topology.

Note that our design is not merely that of a “host-network interface” in the traditional sense of the
term. We prefer to call it a networked-I/O subsystem, to emphasize the fact that it provides a way to directly
interface not only to the network itself, but also to various I/O devices residing in the desk area. Indeed, the
APIC interconnect bears a strong resemblance to the traditional notion of an I/O bus; we could think of it as a high speed packet-switched equivalent of an I/O bus, but without the usual drawbacks associated with bus-based architectures (for example, only one device can be using a bus at any point of time). Our architecture does not preclude the use of an I/O bus; in fact, it would be simpler and more cost effective to use an I/O bus for slow devices (such as keyboards, printers, etc.).

The APIC chip, which is central to our design, behaves like a $3 \times 3$ ATM switch, two of whose ports are connected to two other APIC chips, and the third port interfaces to external memory (possibly over a shared bus). One of the jobs of the APIC is to take specified data from the memory, create an ATM cell stream using that data, and forward the cell stream to a downstream APIC\(^3\). In the reverse direction, the APIC is responsible for doing a VPI/VCI virtual channel lookup on every arriving cell to determine if it is destined for the local device. If so, the cell payload is placed in memory, else it is forwarded to a downstream APIC. The APIC is also responsible for implementing full AAL-5 segmentation and reassembly; thus, frames up to 64 Kbytes in length can be reassembled. Entire frames are not reassembled within the APIC itself; instead, the APIC gradually builds a frame in the external memory by writing the payload from a few cells at a time. Finally, the APIC is also responsible for pacing outgoing cells. This serves two purposes: first, it ensures that the APIC's local device does not get an unfair share of the ATM interconnect bandwidth, and second, it prevents bursty sources from causing network congestion or from swamping the receiver with a burst that cannot be handled (by the receiver).

In the remainder of this section, we describe: how to configure and control the APIC interconnect; how to interface an APIC to an I/O device; the different options available for doing media processing on data that is sourced or sinking at a device on the interconnect; and how our approach differs from other similar approaches. We have chosen to make no mention in this section of the APIC's interface to the main system bus. As a matter of fact, it is this interface that roughly corresponds to the traditional notion of a "host-network interface". Our design of this interface allows us to provide zero-copy semantics to applications; this enables us to potentially achieve very high application throughput. Section 6 is dedicated to describing the important features of this design.

### 4.1 Configuration of APIC and Control of APIC and Local Devices

Individual APIC chips in the interconnect can be remotely configured at the time of initialization by using

---

\(^3\) We refer to downstream and upstream APICS several times in this paper; while no such distinction can be made out of context, we typically use these terms when discussing a specific APIC chip, to refer to its two neighboring APIC chips. In the context of a cell arriving at the APIC chip that is the focus of the discussion, we refer to the originating neighbor as the upstream APIC. On the other hand, when a cell is forwarded from the APIC in question to another APIC, then the latter will be referred to as the downstream APIC.
special control cells. The operating system running on the host CPU(s) issues these control cells by providing appropriate commands to the APIC that directly interfaces to the main system bus. Once an APIC has been initialized, its control can be taken over by a local control processor that is responsible for controlling both the APIC as well as the locally connected device (the control processors are not shown in Figure 2; in Section 4.2, we shall describe how the APIC interfaces with the DPM, control processor, and I/O device). There are several advantages to following this approach of dedicating a control processor for each device:

- Data can "flow" from network-to-device (or vice-versa), as well as from device-to-device, without any intervention from the main host CPU.
- It is easier to provide real-time support for devices if they are controlled by a dedicated processor.
- The operating system running on the host CPU(s) can be relieved of having to service interrupts and perform context switches every time new data arrives, or needs to be sent from a local device.

Note that we are not advocating that one processor should be dedicated to every I/O device in the machine. We would use control processors only for devices that reside on the APIC interconnect, and since these devices would typically be very high bandwidth (and possibly expensive) devices, it is justifiable to have a low-cost processor available for their control. For all other devices, the method of interfacing to the host would be no different from what it is today—they would either reside on the main system bus, or on a separate I/O bus. Also, note that a control processor need not be very powerful, since it does not have to touch the data.

4.2 Interfacing Devices to the APIC

In Figure 3, we show how a continuous media I/O device can be interfaced to an APIC through the use of a VRAM. The bus transceiver is merely an analog buffer that serves as a directional switch; depending on its state, it allows data on the stream bus to go onto the local bus, or vice-versa, or it could be open (no connection). When the APIC is ready to place some data in the VRAM, it issues a read transfer cycle on a row address to the VRAM controller over the Local Bus (the transceiver is configured to allow the address to pass from the Stream Bus to the Local Bus). This results in data from the specified row of the VRAM being read into the serial access shift register within the VRAM. Following this, the APIC can use the Stream Bus to clock data into the shift register over the serial port of the VRAM. The transceiver would behave like an open switch during this phase of the transfer. Once the serial transfer is completed, the APIC would issue a write transfer cycle with the row address so that the contents of the shift register get written back into the appropriate row of the (random access memory in the) VRAM. The transceiver would again be in the "Stream bus to Local bus" state during this brief transaction. The device can also read data from the VRAM over the Stream Bus, using a similar sequence of operations. Note that the Local Bus interfaces to the ran-
dom access port of the VRAM, thereby enabling the control processor to read or write the contents of the VRAM; this is required in order for the processor to perform control related actions. Also, the control processor can issue commands to the APIC over the Stream Bus by placing the transceiver in a state where it allows data movement from the Local bus to the Stream bus. Note that in most cases, the continuous media data need not move over the Local Bus, so this bus does not need to achieve high performance.

The theoretical peak transfer rate into the serial port of a typical 40 MHz VRAM over a 64-bit Stream Bus is 2.56 Gbps. If we take into account the fact that each data unit would have to traverse the Stream Bus twice, and allowing for overhead due to bus arbitration, start-up, and transfer control, there is enough bandwidth left over to easily accommodate the entire 622 Mbps link rate. If the device is full duplex and can simultaneously source and sink up to 622 Mbps, then the design can be slightly modified to use a triple-ported VRAM (one having two serial ports and one random access port). This would ensure that data movement between the APIC and the VRAM does not interfere with data movement between the VRAM and the device.

4.3 Media Processing

We now address the problem of how to do media processing on data originating from or destined for a device on the interconnect. We have three approaches for tackling this problem. One simple way is to rel-
egrate all media-specific processing to the local control processor of a device. Clearly, this approach cannot be used when the data rate is very high; the limiting factor would be the bandwidth of the random access port of the VRAM (see Figure 3). A second approach is to have a special media processor residing as a separate “device” on the interconnect. This method would work especially well when the processing device is an encryption/decryption engine, for example. Data destined for the network could be made to “pass through” the encryption device on its way to the link interface. Similarly, data arriving from the network could be made to pass through this device so the data can be decrypted before being forwarded to the destination I/O device. A third way to support media processing in our design is to embed a signal processor on the data path between a continuous media device and the VRAM (see Figure 3). This method suffers from the drawback that the processor cannot be easily shared by multiple devices. However, in some cases this would be the approach of choice, such as, for example, when the media processor is a hardware compression/decompression engine. This ensures that uncompressed data would not have to traverse the interconnect, which is important because the data rate of the uncompressed stream might be a significant fraction of, or more than, the maximum bandwidth of the interconnect.

4.4 Alternative Approaches

Before we conclude this section, we would like to contrast our approach with other similar approaches. As mentioned earlier, our design bears some similarity to the concept of an I/O bus. There are several advantages to using I/O buses: I/O buses can be long, can have many types of devices connected to them, and normally follow a bus standard. In contrast, CPU-memory buses are short, cater to very few different types of devices, can change frequently to track changes in technology, and are designed to optimize the performance of the CPU, cache, and memory.

Our design of the APIC interconnect allows it to inherit all of the usual advantages attributed to an I/O bus. By fully specifying the interface between APIC and memory, it becomes easy for different vendors to market devices that can interface with the APIC interconnect. Further, by choosing a standard physical interconnection between APIC chips, we can make the interconnect long. For example, our prototype design uses the UTOPIA standard [1] for connecting two APIC chips within a single “box”. In going from one box to another within the desk-area, we would switch to an optical medium (622 Mbps SONET). An off-the-shelf chipset can be used to translate from electronic (UTOPIA) to optical (SONET) medium, and vice-versa. The link interface to the ATM network can also be built using this chipset; thus, our entire host-

4 UTOPIA, which is an acronym for Universal Test & Operations PHY Interface for ATM, is a standard proposed by the ATM forum for the data path interface between the ATM layer (performing segmentation and re-assembly), and the physical (PHY) layer components of an ATM subsystem. It can support a wide range of speeds: from sub-100 Mbps to 155 Mbps (with an 8-bit wide data path), and up to 622 Mbps (with a 16-bit wide data path).
network interface requires that we design only one ASIC (the APIC).

Note that while our design retains all the nice properties attributed to I/O buses, it is also able to eliminate a number of drawbacks one normally associates with an I/O bus. Predominant among these is speed: I/O buses are usually asynchronous (to allow for long buses connecting many dissimilar devices), and their speed is typically much lower than that of CPU-memory buses. Furthermore, only one device can be using an I/O bus during the period of a single transaction. In contrast, the APIC interconnect allows multiple devices simultaneous access, and it is fast. As a matter of fact, it is faster than most CPU-memory buses in use today.

An architecture in which a generalized packet switched interconnect is used to connect processors, memories, and devices has widely come to be known as a "desk area network" (DAN). The concept of a DAN was first introduced by David Tennenhouse [16]. We believe that desk area networks hold promise for the future of computer architecture. Clearly, the APIC interconnect architecture itself falls into the class of DAN-based architectures. If we generalize the APIC so that it becomes a full-fledged ATM switch that can interface simultaneously over different switch ports to a number of devices (including CPUs and memory modules), then we arrive at a more futuristic DAN-based architecture. While a system based on a such an architecture would likely deliver higher performance than an APIC based architecture, it does not come without some drawbacks, at least in the short term. Among these arc hardware costs—the APIC solution comes out ahead both in terms of the number of chips that would need to be designed, as well as in the number of chips that would actually be used in an implementation of comparable size. Furthermore, the APIC scheme provides an incremental cost solution: only as many APICS need to be used as there are devices in the host. The APIC interconnect can also span long distances by using optical fiber for connecting widely separated APIC chips.

5 Internal Design of the APIC

Figure 4 shows a high level block diagram of the internal design of the APIC chip. To understand this figure, first identify the three shaded blocks labelled Input Pads, Output Pads, and Host Pads. The inset at the bottom left of the figure shows how the APIC interconnect would appear if we were always to show the input pads on the left and the output pads on the right, as is the case in the block diagram. Note that the host pads are used to interface to the system bus or to the serial port of a VRAM.

Figure 4 is best explained by identifying the various paths that can be taken by data passing through the APIC. First, we consider the simplest case: a cell from a downstream APIC being forwarded to an upstream APIC. The cell is not destined for the local device, so it will not be sent to the host pads. When the cell arrives at one of the two input pads, it is in a 16-bit wide UTOPIA format. The incoming data stream
is synchronized with the chip's internal clock (in the block labelled Sync), following which the cell is multiplexed with traffic from the other input port, as well as traffic originating from the local device (the latter is read from the transmit FIFO, TxFifo). The Mux performs simple time-division multiplexing; it reads one cell from each of its three inputs in turn in a round-robin fashion. The cell is then written into a SRAM that serves as a cell store. The address of a free cell slot into which the cell can be written is obtained from the FreeFifo. A copy of the cell header, along with the address in the cell store where the cell resides, are passed to the virtual circuit translator (VCX), where a table lookup based on the virtual path and channel identifiers reveals that the cell is not intended for the local device, and should therefore be forwarded to the upstream APIC. The VCX conveys this information to the cell dispatcher, which is responsible for scheduling cells to be read out from the cell store. The Read logic is responsible for performing this read operation, and for returning the address of the newly freed cell slot to the FreeFifo. The demultiplexer (Demux) forwards the cell to the synchronization logic, following which the cell leaves the chip through the output
pads.

The next data path we shall examine is for the receipt of a cell that is intended for the local device. In other words, a cell following this path would arrive at one of the input pads, and its payload would eventually need to be placed in external memory that can be accessed from the host pads. The discussion of the blocks traversed on this data path follows that of the earlier example, until we reach the virtual circuit translation table (VCT). Here, it is determined (based on the VCI/VPI) that the cell is destined for the local device, and this information (along with the address of the cell in the cell store) is passed to the dispatcher. The dispatcher schedules the cell to be read out at an appropriate time; the cell slot is so chosen that the demultiplexer would forward the cell to the CRC verification circuit. The dispatcher also places a tag\(^5\) identifying the connection on which the cell arrived in the BatchFifo. This is later read by the HostData/Addr block and used to associate data from the RxFifo with the connection to which it belongs. Note that only the cell payload is placed in the RxFifo; the header is removed by the Header Delete circuit. The HostData/Addr block is responsible for generating an appropriate address for the VRAM or system memory where the data would be placed, and for issuing appropriate read or write transactions on the bus connected to the host pads.

The last data path of interest, namely that corresponding to a cell originating at the locally connected device and bound for a downstream APIC, will not be described here. Suffice it to say that it is very similar to the other examples above, and can easily be guessed by the reader.

There are two blocks in the APIC design that deserve extra attention—the cell dispatcher and the HostData/Addr block; these are described in the following subsections.

### 5.1 The Dispatcher

The dispatcher is primarily responsible for scheduling the reading of cells from the cell buffer. It is also responsible for the partial reassembly of cells. The need to do partial reassembly of cells stems from a requirement for high performance when using the serial port of a VRAM. The shift register that forms the serial access memory (SAM) of a VRAM is usually much larger than the payload of one ATM cell (a typical SAM size for a 64-bit wide VRAM is 512 bytes). If we are to achieve a high enough average rate on the serial port of the VRAM, then it is important to minimize the overhead associated with VRAM transfer cycles—the minimum overhead results if we can completely fill the SAM before a write transfer cycle. Another reason to do partial reassembly of cells arises from the need to reduce the number of times we arbitrate for the sys-

\(^5\) The “tag” encodes the VCI/VPI pair for the connection. Since an APIC chip typically supports only a few connections that would actually originate from or terminate at the local device, we require fewer bits to identify the connection locally.
tem bus—a number of others have built high performance network interfaces only to later find out that bus arbitration overhead can be a major killer of overall performance. By parking on the bus for the entire time it takes to transfer a partially reassembled cell batch, we can significantly reduce the bus arbitration overhead (over the case where we arbitrate once for each cell’s worth of data). In our prototype design, the dispatcher uses a content-addressable memory (CAM) that is keyed on the connection identifier to batch cells belonging to the same connection into groups of 1, 2, 4, or 8 cells. After the entire batch has been read out from the cell store, a single tag corresponding to the batch is placed in the BatchFifo. If fewer cells than the specified batch size for the connection have been received within a preset time interval, then tags for the cells that have so far been received are placed in the BatchFifo anyway.

5.2 The HostData/Addr Block

The HostData/Addr block is responsible for generating addresses for the external memory from which data is read or written, and for issuing the appropriate signals to affect these read or write transactions. It also collaborates with the cell dispatcher for pacing outbound cells. We shall defer the discussion of the way in which the HostData/Addr block generates addresses and interfaces to the external memory to Section 6.

The pacing control is based on a single parameter—the peak bandwidth. While this may sound like an overly simplistic approach at first, it does allow for a satisfactory level of fine-grained traffic shaping. For coarser-grain shaping, such as that deriving from the use of burst length (for example, that resulting from a transport level rate control scheme), we contend that software implementations would suffice. The advantages and drawbacks of using hardware to do such coarse grain pacing are not apparent, and its discussion is beyond the scope of this paper. Note that the only place where the pacing directly translates to bandwidth reservation is on the APIC interconnect itself; indeed, we do not want to lose cells even before they reach the network! The HostData/Addr block affects the pacing by scheduling the data corresponding to one batch of cells to be read from the external memory once every $T$ seconds, where the value of $T$ determines the peak rate of the connection. No cells are read when the local cell store is almost full; thus flow control under congestion conditions is achieved by back pressure rather than by discarding cells.

In order to deal with the different types of system buses and VRAMS to which the APIC may have to interface, the pin out of the APIC is designed to work with a typical multiprocessor system bus and VRAM interface. When interfacing to a different type of bus or VRAM, a programmable logic device such as an EPLD and possibly a few standard packages can be used to affect the translation between the two types of interfaces. Differences between two such interfaces are typically small enough that a reasonable sized EPLD should suffice.
5.3 Feasibility

At this juncture, some mention of the size and performance estimates for the prototype APIC chip are in order. We have assumed a 622 Mbps external link rate. With UTOPIA, this translates to a 16-bit wide data path that is clocked at close to 40 MHz. The output of the multiplexer within the APIC would need to be clocked at more than thrice this data rate if we retained the same data path width. In the prototype APIC design, the output of the multiplexer has a data path width of 32 bits, and allowing for the overhead involved with reading and writing cells from the internal cell store, an internal rate of 80 MHz suffices, and is well within the limits imposed by technology. A preliminary (and elementary) analysis of the storage requirements within the APIC, assuming the use of 32 input channels and 32 output channels (each channel is an ATM connection that originates or terminates at the APIC in question), resulted in a requirement of about 140 Kbits. While this is rather a large chip, it is well within the bounds of the fabrication technology we shall be using.

6 Design of a Zero-Copy Interface to System Bus

The APIC has two different types of interfaces to the host: one is the interface to the main CPU-memory bus, while the other is to the serial port of a VRAM that connects to an I/O device. There are a number of issues relating to memory management, interrupt generation, and protocol processing that need to be addressed in both types of interfaces in order to achieve desired levels of performance. In addition, the study of the first of the aforementioned interfaces involves issues relating to caching, cache coherence, virtual memory paging, context switching, and OS scheduling. Careful design of this interface, which constitutes the subject of this section, is of paramount importance if we are to achieve high end-to-end application throughput. The design of the interface to the VRAM serial port (when the local connection is to an I/O device) is relatively less complex. Due to space limitations, we shall defer its detailed discussion to a future paper.

6.1 Goals

Our design of the interface to the system bus is geared toward achieving high application throughput, low end-to-end latency, and minimal use of the system bus (so that the CPU is not starved). In meeting these objectives, we had to place minor constraints on network protocols and the operating system. We believe this is necessary and worthwhile because it leads to significantly higher performance and allows applications to fully exploit available network bandwidth. However, one of our secondary goals has been to ensure that our interface design performs no worse than existing state-of-the-art interfaces, when used with current protocols and operating systems with no modifications. Finally, we felt that it was important for end-applications to see a clean and easy-to-use interface, and one which would allow for easy adaptability
to existing programming techniques.

The above goals helped isolate the following properties that the interface needs to have:

1. All protocol processing above the ATM adaptation layer that did not involve touching data should be done in software. Recent results for TCP, based on work done by Van Jacobson (see page 240 in [12]), have shown that the fixed cost (not including costs for handling data) of transport level protocol processing can be as low as 150 RISC instructions per packet, for both sending and receiving. This is fast enough for most applications on any workstation, and so there is no justification for losing the flexibility afforded by a software implementation by choosing to put protocols in hardware instead.

2. Application data belonging to an application data unit (ADU) should appear contiguous in the application’s address space. Most current networking software supports this type of an application interface. There have been some recent research efforts that propose the use of an abstract data type layered above one or more noncontiguous buffers which would enable applications to view these buffers as a single aggregate “contiguous” buffer (see Figure 2 and Section 5.2 in [9]). While such an approach can be used to build up a very flexible (and high performance) I/O buffer management facility, we believe that it may not be suitable to many applications. In particular, an application would find it very difficult to map multidimensional arrays or a complex data structure onto the aggregate abstraction introduced in [9], and would usually be forced to copy the data into contiguous space thus compromising performance. In our approach, an ADU does appear contiguous in the application’s address space. However, we believe that a scheme similar to the fbuf scheme (introduced in the same paper cited above, [9]) is crucial in order to achieve high performance for subsequent cross-domain data transfers.

3. The interface should support zero-copy semantics. A zero-copy interface is defined as one where data moves directly from an application’s address space to the network, or vice-versa. This is to be distinguished from a single-copy interface, where the data in a packet is copied once from the application’s address space to a packet buffer that is resident on the interface, and then from there to the network. In a zero copy interface, data belonging to a transport level packet will be in transit in the network even before the contents of the entire packet have been read out from the main memory. Zero-copy semantics can result in very low end-to-end latency; this is especially important for transaction oriented traffic (for example, NFS), and for achieving faster response time to user interaction.

---

6 An ADU is the unit used by applications for sending or receiving data, and typically corresponds to some logical block of data that the application looks upon as a single entity. An ADU can possibly consist of several transport level packets (also called TPDUs, or transport protocol data units).

7 All of the interfaces described in Section 3 involve one or more copies; the fastest of these are usually single-copy interfaces.
(for example, in virtual reality applications). The end-to-end throughput that can be achieved using zero-copy semantics is also very high, since the operating system does not need to be tied down doing data copying. To the best of the authors’ knowledge, this is the first time an attempt has been made to design and build a zero-copy interface. This unique feature was achieved through careful integration of the interface with protocols and the operating system.

6.2 Architectural Choices

Figure 5 shows three possible approaches in which the interface to the system bus could be realized. In the first approach, shown in Figure 5(a), the APIC has direct memory access (DMA) capability built in, so it can read and write data directly from/to the system’s main memory. Since the APIC would be the bus master during such transfers, the bus should be able to support multiple masters. Moreover, it is important that the bus provide support for cache-coherent transactions. This is not a problem, since most state-of-the-art workstation architectures now available are multiprocessor machines that implement some form of snooping-based cache-coherency protocol on the bus, and the trend is likely to continue for future machines. Without cache-coherent transactions, the processor would have to invalidate the cache in software after the APIC has written into main memory, and it would have to flush the cache before the APIC can read from main memory. Software cache invalidation and flushing is typically inefficient and can cause a significant degradation in performance.

In Figure 5(b), we show an alternate scheme for interfacing the APIC to the system bus. In this case, the APIC reads/writes data from/to the serial port of a VRAM, the random access port of which is connected to the system bus. The VRAM serves only as a reassembly buffer; once a complete frame has been reassembled, the CPU can copy it over from the VRAM into the main memory. A similar approach has been used in the Afterburner [7] and Medusa [3] interface designs. If this approach is used, and given that DRAMs usually have cycle times in excess of a 100ns, the extra copy step involving two DRAMs (one is the main memory, the other is within the VRAM) can adversely affect performance. Also, data has to traverse the system bus at least twice before it can be placed in main memory (once from the VRAM to the CPU cache, and then from the cache to main memory). Using DMA to transfer data from the VRAM to the main memory does not help alleviate this problem, since both memories need to be separately addressed over the bus. Finally, it is impossible to support zero-copy semantics with this type of an architecture. Based on this last observation, we chose not to use this approach for our interface design.

In the third approach, shown in Figure 5(c), the system’s main memory is now a VRAM, the serial port of which is connected to the APIC. Although this approach may sound preposterous at first, the fact that VRAM prices are seldom more that 1.5 times the price of DRAMs of the same size, makes it a possibility
that should not be ignored. The appeal of the approach lies in the fact that APIC-memory traffic would not interfere with CPU-memory traffic. However, things are not as rosy as they may seem. Cache coherency can pose a significant problem. One solution is to use software cache flushing and invalidation techniques, but as mentioned before, this would result in inefficiencies that would offset any gains gotten otherwise. A second solution is to use cache invalidation transactions (if supported by the system bus) every time the APIC writes data into the VRAM (main memory). A connection between the APIC and the system bus through a transceiver can be used for this purpose, as shown in the figure. On the sending side, software cache flushing can be avoided by using write-through caches, but most modern multiprocessor workstation buses only support write-back caches since these normally yield better performance when used with small cache lines and large cache sizes. However, it is our contention that the performance gains accrued by letting the network data skirt the main system bus altogether will more than offset any loss in performance resulting from the use of write-through caches and cache invalidation transactions, especially when the cache line size is large. The trade-offs depend on several parameters and require further investigation.
Based on some of the observations we have made above, we arrived at the conclusion that the first approach (shown in Figure 5(a)) was most suitable for the prototype design in terms of meeting our goals. We would like to point out, however, that most of our ideas in the rest of this paper apply equally well if we had selected the third approach (shown in Figure 5(c)). Our choice did result in imposing a few restrictions on the design environment, as pointed out earlier. Based on some of these constraints, our final choice of target workstation was a SPARC\textsuperscript{8} based architecture that uses a high performance circuit-switched system bus called the MBUS [13]. The MBUS interface is well specified and simple enough that our chosen approach can be successfully implemented. Appendix A describes some of the relevant MBUS details, and shows how the APIC can use the MBUS to affect DMA transfers to main memory.

6.3 Implications of Design Goals on Network Protocols

Given that we shall be using DMA over the system bus for transferring data between main memory and the APIC, we can arrive at a few constraints on higher layer (above the adaptation layer) network protocols that follow directly from the need to meet our design goals (in the following discussion, we have underlined key phrases that identify each such constraint that we isolate):

1. Since one of our objectives is to allow data belonging to an ADU to appear contiguous in an application’s address space, and given that packet headers and data appear interspersed on the network, it is clear that the APIC would need to support scatter gather DMA functionality if we desire a zero-copy interface. In particular, the APIC should, at the very minimum, be able to distinguish packet headers from application data, and be able to split them in memory. Then, if data from consecutive packets is written next to each other, there is hope that it might be possible to get it to appear contiguous in the application’s virtual address space. A simple solution to efficiently recognizing packet headers in hardware is to use fixed length headers throughout the life of a connection.

2. Since most workstation buses including the MBUS utilize physical addresses, and since data in consecutive physical pages would usually not be contiguous in the application’s virtual address space, the APIC’s DMA engine should be able to scatter-gather across different pages in physical memory. Furthermore, if any of the pages to which an APIC writes received data (except boundary pages of an ADU) are partially filled, then the only way of making the data appear contiguous in the application’s virtual address space would be by copying, which would defeat our objective of achieving zero-copy semantics. From this, we conclude that the APIC should completely fill all but the two boundary pages corresponding to an ADU. Moreover, if data from packets did not end on page boundaries, then it would not be possible to avoid copying data if a corrupted packet is received, or if a packet is lost.

\textsuperscript{8} SPARC\textsuperscript{TM} is a registered trademark of SPARC International, Inc.
If we can guarantee that data from packets will indeed always begin and end on page boundaries, then we should be able to use the virtual memory system to remap pages into the correct order and thereby avoid copying. Based on the above observations, we come to the conclusion that the length of the application data within a packet should be a multiple of the receiving host's page size. When there are multiple receivers, as would be the case in a multipoint connection, the length of the application data within a packet should be a multiple of the largest of the receiving hosts' page sizes (assuming that the page size is always a power of 2).

3. Checksumming data in software can adversely affect performance, since the CPU would have to read the data from memory, thus introducing an extra data touch and an extra bus traversal. Thus, by depending on software checksumming, we would be moving away from our objective of achieving a true zero-copy interface in which neither the operating system nor any protocols resident in software would ever need to touch the application's data. However, transport level checksumming is recognized as an essential component of any protocol that guarantees reliable data transport. Clearly, we would need to support checksumming in hardware if we are to meet our objectives. Hardware checksumming cannot be achieved in a zero-copy interface unless the checksum is positioned right at the end of the transport level packet. Clearly, this precludes the use of many existing transport protocols, including TCP, since they do not use trailing checksums.

The requirements imposed by the first two constraints identified above can easily be met by using a transport protocol that negotiates header and data length that will be used in packets at connection setup time (or when an endpoint is added or dropped, in the case of a multipoint connection). While none of the constraints stated above are essential for the operation of our interface, they are necessary in order to demonstrate true zero-copy behavior, and to allow the data from an ADU to appear contiguous in the receiving application's virtual address space. If one or more of the constraints does not hold, then our design would still work, but less efficiently.

We are currently in the process of defining a high speed multipoint connection-oriented internetwork transport protocol that we believe overcomes many of the drawbacks usually associated with existing protocols such as TCP, and which would work well with a variety of network interfaces, including ours. Among other desirable features, this protocol uses a packet format with trailing checksum, fixed header length, and data length that depends upon the receiver(s)'s page size(s).

We would again like to emphasize that the aforementioned transport protocol features are desirable,

---

9 Application oriented error control strategies have been suggested that can provide a variable grade of reliability to end-applications based on their requirements, but even such schemes rely on an end-to-end checksum to decide what percentage of error or loss is tolerable.
but not necessary—while our interface would still work with protocols such as TCP/IP, it would not be as
efficient (it would not be inefficient either; we have ensured that our interface will perform at least as well
as any state-of-the-art host-network interface design, when used with existing transport protocols).

6.4 Details of the Zero-Copy Interface

The design of our zero-copy interface is best explained through reference to Figure 6, which focuses only
on the receive side. The design for the sending end is relatively much simpler; we have chosen to defer its
description. In the figure, we have shown ATM cells belonging to three different connections arriving over
the same link to an APIC that interfaces to the main system bus. Those belonging to connection #3 are not
destined for main memory, so they simply pass through the APIC to another downstream APIC. Cells from
connections #1 and #2 are destined for main memory. Notice that cells from different connections can be
interspersed on a single link. However, ATM networks guarantee ordered delivery of cells belonging to the
same connection. Hence, the APIC can be in the process of reassembling at most one AAL-5 frame per con-
nection, at any time.

All of the constraints mentioned in the Section 6.3 about packet formats suitable to our interface are
presumed to hold. We further make the following assumptions purely from the point of view of simplifying
the task of explaining the figure: (1) each packet has a fixed header length that is the size of the payload
of a single ATM cell (48 bytes), and (2) the length of application data within a packet is exactly equal to the
size of a single page in memory. Note that none of these assumptions are necessary for the interface to
work as desired.

We shall now focus our attention on connection #1. We have shown cells holding the contents of
three different packets (AAL-5 frames) arriving on this connection. To help distinguish them, cells contain-
ing data from the same packet have all been given the same shade of gray, while cells containing data from
different packets are shaded differently. The first cell of every packet contains the packet header, and the
remaining cells contain application data. Although we have shown only one or two cells containing data
per packet, in reality there will be many more, since typical page sizes are much bigger than the size of the
payload of a single ATM cell. As the APIC receives cells on the connection, it DMAs their payloads into main
memory. First, the packet headers are stored in free header buffers that are drawn from a pool of special
buffers laid aside for headers by the operating system. These header buffers all reside in pages that are per-
manently mapped into the kernel's address space; this ensures that kernel-resident protocols can examine
or modify the headers. The payloads from consecutive cells containing application data belonging to the
same packet are DMA’d into consecutive locations within a single page, which is drawn from a pool of free
physical page frames that has been made available to the APIC by the operating system. Note that these free
pages need not be mapped into the kernel's address space (or any other address space, for that matter). All the data belonging to a packet would eventually fill in a single page frame in physical memory. The APIC maintains the association between packet headers and corresponding pages in main memory (from where it can be read by protocols in operating system), or through the use of on-chip FIFOs (see Appendix B). The CPU is informed of the arrival of data through the use of interrupts; more details can be found in Section 6.6. The transport protocol, and possibly other higher layer protocols, would then examine the contents of packet headers and make a decision about where to place data from the packet in the application’s address space. Once this has been determined, a request would be made to the operating system to modify page tables so that the new virtual memory mappings reflect the correct ordering of received pages within the application’s virtual address space. The result of this exercise is that the data from multiple packets now appears contiguous from the application’s perspective, and this was achieved without the kernel or protocols ever having to touch application data.
Appendix B contains a short description of the data structures that could be used by the APIC in order to implement the scheme outlined above.

So far, we have described only the “receive” part of our interface. The “send” part is much simpler; the operating system merely gives the APIC a pointer to a list of <address, size> pairs, one per buffer, to be sent out on a specified connection. A packet is constructed using many chained buffers, the first of which would normally contain the header while the remaining may point to pages belonging to the application that have been locked down in physical memory. The APIC DMA’s data from memory on demand, based on grants from the pacing logic. The wired-down pages can be made swappable again once the packet has been sent.

6.5 An Optimization
The performance of the scheme outlined in the previous subsection is mainly governed by the cost of page remapping, which can be significant in some systems. A variant of the scheme, which we now present, can almost completely eliminate page remapping, and thereby improve performance (both in terms of throughput and latency) considerably. In the new scheme, we use the connection-oriented aspect of ATM—the decision regarding which pages will hold data from an incoming packet is made based on the connection on which the packet arrived. Since each connection is associated with an application, we can choose to place data from arriving packets directly in pages that are already mapped into the application’s virtual address space. Data from consecutive packets would be placed in pages that would be contiguous in the virtual address space. So long as there are no errors or packet loss, there is no need to do any page remapping. If an error occurs, or if a packet is lost, then it would be necessary to remap pages so that the ADU still appears contiguous in the application’s address space. This scheme performs best when the ADU size is small. Clearly, it would perform much better than the scheme outlined in the previous subsection most of the time, and it would perform no worse than the previous scheme in the presence of errors or loss.

6.6 Other Issues
Due to space constraints, a lot of details have been left out of our discussion of the interface. Conspicuous among these are software related issues. While we hope to cover most of these missing details in future papers, we would like to draw attention to a few of them that we felt were especially important:

Lazy Updating of Page Tables: In many systems, the overhead of page remapping is often comparable to the cost of copying a page. Hence, we propose a scheme we are calling “lazy updating of page tables”, whereby we wait for as much data to arrive as the application had requested (for reading), before doing any page remappings. Clearly, using this scheme would have repercussions on the design of the interprocess-communication (IPC) application interface; in particular, it would make it
blocking (possibly with a time-out). The scheme works well only if the receiving application knows how much data to expect in a single read operation; since most high bandwidth applications fall into this category, the performance gains that accrue from using it could be significant.

**Interrupts:** Reducing the number of interrupts the APIC would issue to the CPU is very important from a performance standpoint. We believe that the optimal scheme involves interrupting only when the full header queue (see Appendix B) goes from empty to non-empty (i.e., it has at least one entry). This would ensure that the CPU gets interrupted only once per burst of packets.

**Supporting Other Protocols:** To support existing protocols like TCP, the APIC would DMA data into kernel buffers from where a conventional approach to implementing protocols in the kernel can be used. These approaches usually copy data between kernel and application space, and also do software checksumming, so there would be a degradation in performance.

### 7 Conclusions and Future Work

We presented the design of a high performance network interface architecture for multiprocessor multimedia workstations and servers. By allowing continuous media traffic to use the APIC interconnect and thereby skirt the main system bus, it was shown that very high network data rates (over a Gbps) can be easily achieved. In addition, a high speed pathway is also provided for data transfer between different devices resident in the same desk area (host).

An interface design to the main system bus that could support zero-copy semantics was demonstrated. The zero-copy property was achieved without sacrificing other desirable features such as data continuity (in an application’s address space), transport level checksumming, etc.

We are currently in the process of refining our design, and we plan to undertake the prototyping of the APIC chip using a VHDL tool. We shall also be prototyping APIC interconnect based interfaces for a few target workstations and servers that will include I/O devices such as video cameras, high resolution displays, video storage, and possibly a video wall. This work is part of a bigger effort sponsored by ARPA that is aimed at the design and deployment of a gigabit local ATM testbed at Washington University. In addition to prototyping, we are also planning to use simulation and analytical techniques to evaluate several other trade-offs that were mentioned in this paper.
References


Appendix A: Relevant MBUS Details

While details of the prototype design environment are not important for the purposes of this paper, we felt it was important to give the reader a brief overview of the MBUS and relevant transaction types it supports, with a view to familiarizing him/her with the state-of-the-art in workstation buses, and to permit a better understanding of some of the other topics that were touched on in the paper. The MBUS has a 64-bit wide data path, on which both address and data are multiplexed. All addresses used on the MBUS must be physical addresses. At 40 MHz, the MBUS has a peak bandwidth of 2.56 Gbps, and an effective bandwidth of 840 Mbps. The MBUS supports four types of transactions that can be used to maintain cache consistency. All of these transactions have a transaction size of 32 bytes, which is the size of a cache line in MBUS based systems. Two of these are relevant to our discussion: the Coherent Read (CR) transaction, and the Coherent Write and Invalidate (CWI) transaction. With reference to Figure 5(a), if the APIC issues a CR transaction, and if the data which is to be read is not in cache, or if it is in one or more caches but has not been modified, then the data is read from main memory and returned to the APIC. If, on the other hand, the data is in cache and has been modified, then the MBUS cache coherency protocol ensures that there is one (and only one) cache that is the “owner” of the latest version of the corresponding cache line, and it is this cache that will respond by placing the requested data on the bus. In either case, the APIC would always read the most up-to-date version of the data.

We next consider the case where the APIC has some data that it wants to write to memory. If the APIC issues a CWI transaction, then the data gets written into memory, and all caches snooping on the bus that have a cache line for which there is an address hit will invalidate that cache line. Thus, the next time a processor tries to read from the address that was written to, it will access the fresh data that was placed in memory by the APIC. It is not difficult to see that by the use of these two transaction types, the APIC can DMA data to/from main memory while ensuring that cache coherency is always enforced. Furthermore, all such transfers of data, which usually occur in transaction units of 32 bytes, involve a single bus traversal. Accounting for bus arbitration overhead, this still allows the APIC to achieve transfers to memory at close to the maximum memory bandwidth (assuming that there is not much activity between the CPU and memory).
Appendix B: Data Structures for the Page-Remapping Scheme

Figure 7: Data Structures Used by the APIC for the Receive Interface.

Figure 7 shows the data structures that are used by the APIC to implement the scheme outlined in Section 6.4. Within the HostData/Addr block of the APIC, there are two arrays indexed on the connection identifier (this is the same as the “tag” used by the APIC as a short form for the VCI/VP; see Section 5). The first array holds header lengths for connections, while the second contains a pointer to physical memory where new data arriving on a connection is to be placed (using DMA). The figure also shows four queues that can reside either on the APIC itself, or in main memory from where they can be read by the APIC. Each entry in these queues contains a pointer to a header buffer or page frame, and also may contain some extra information such as a connection identifier or checksum. The operating system sets up the two queues on the left, which contain pointers to free header buffers and page frames. When the APIC receives a new cell that is destined for main memory, it checks to see if it is the first cell of an AAL-5 frame. If so, it uses a new buffer obtained from the free header buffer queue to hold data arriving on that connection, up to a maximum of the header length for that connection. After that, it switches to the free page frame queue, and as pages fill up, it keeps drawing pointers to new free pages from that queue until the end of the AAL-5 frame. The APIC uses the “current address” array to store the next address to be written; this value is typically incremented whenever new data is written to memory, and it gets overwritten when a pointer to a new buffer or page is drawn from one of the “free” queues. As soon as a header buffer is ready or a page frame has been filled, the APIC places a pointer to it, along with the corresponding connection identifier, and possibly a checksum which has been computed in hardware, on the appropriate “full” queue. The APIC device driver within the operating system would then read the full header and buffer queues and inform higher layer protocols so they can do protocol processing on the received packets. Note that this design supports different header lengths on different connections, and it also allows data from a packet to span multiple pages.