RICE UNIVERSITY

Improving Networking Server Performance with Programmable Network Interfaces

by

Hyong-Youb Kim

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Scott Rixner, Chair
Assistant Professor of Computer Science and
Electrical and Computer Engineering

Vijay S. Pai
Assistant Professor of Electrical and
Computer Engineering and Computer
Science

Alan L. Cox
Associate Professor of Computer Science

Houston, Texas

April, 2003

Improving Networking Server Performance with Programmable Network Interfaces

Hyong-Youb Kim

Abstract

Networking servers, such as web servers, have been widely deployed in recent years. While developments in the operating system and applications continue to improve server performance, programmable network interfaces with local memory provide new opportunities to improve server performance through extended network services on the network interface. However, due to their embedded nature, programmable processors on the network interface may suffer from inadequate processing power when compared to non-programmable application-specific network interfaces.

This thesis first shows that exploiting a multiprocessor architecture and task-level concurrency in network interface processing enables programmable network interfaces to overcome the performance disadvantages over application-specific network interfaces that result from programmability. Then, the thesis presents a network service on a programmable network interface that exploits the storage capacity of the interfaces to alleviate the local I/O interconnect bottleneck, thereby improving server performance. Thus, these two results show that programmable network interfaces can offset the performance disadvantages due to programmability and improve networking server performance through extended network services that exploit their computation power and storage capacity.

Acknowledgments

My utmost thanks go to my advisors Prof. Vijay S. Pai and Prof. Scott Rixner. Over the past year, they have guided me through research, inspired diligence and professionalism, and helped develop critical thinking. Without their help, this thesis would not have completed.

I thank Prof. Alan L. Cox for his interest in my research and valuable comments on this thesis. Mike Filippo at Advanced Micro Devices (AMD) facilitated donation of research machines from AMD and continues to provide valuable advice regarding AMD's microprocessors.

I also thank my parents for their support throughout my undergraduate years at the University of Rochester.

Contents

Abstract i

Acknowledgments ii

List of Illustrations vi

List of Tables ix

1 Introduction 1

1.1 Objective 3

1.2 Organization 6

2 Network Interface Operations 8

2.1 System Architecture 8

2.2 Network Stack 10

2.3 Device Driver/NIC Communication 11

2.4 Summary 17

3 Exploiting Task-level Concurrency in a Programmable Network Interface 19

3.1 Tigon Programmable Gigabit Ethernet Controller 21

3.1.1 Hardware Features 21

3.1.2 Event-driven Firmware 23
3.1.3 Released Tigon Firmware 25
3.2 Parallelization of Firmware 30
3.2.1 Parallelization Concepts 30
3.2.2 Parallelization Strategy 32
3.3 Evaluation of Parallelization Strategy 36
3.3.1 Firmware Implementation 36
3.3.2 Benchmarks 37
3.4 Experimental Results 39
3.4.1 Architectural Impact on Multiprocessor Performance 39
3.4.2 Throughput Improvements 44
3.5 Modern Programmable Network Interfaces 50
3.6 Scalability Issues 51
3.7 Summary 54

4 Network Interface Data Caching 56

4.1 Background and Related Work 57
4.1.1 Anatomy of a Web Request 58
4.1.2 Local Interconnect 59
4.1.3 Related Work 61
4.2 A Network Interface Data Cache 64
4.3 Network Interface Data Cache Design 74
4.3.1 Cache Architecture 74

4.3.2 Cache Management 76
4.3.3 Cache Interface 80
4.4 Evaluation Methodology 84
4.4.1 Web Traces 84
4.4.2 Prototype Implementation 86
4.4.3 Test Platform 88
4.4.4 Impact of the `Sendfile` System Call 90
4.5 Experimental Results 91
4.5.1 Local Interconnect Traffic and Server Throughput 91
4.5.2 PCI Bus Speed 95
4.5.3 Host Processor Speed 99
4.6 Discussion 106
4.6.1 Local I/O Bottleneck 106
4.6.2 Operating System Design Alternatives 107
4.6.3 Deployment in Other Applications 108
4.7 Summary 109

5 Related Work 112

6 Conclusions and Future Work 117

6.1 Conclusions 117
6.2 Future Work 119

Bibliography 122

Illustrations

2.1 The system architecture of a networking server 9

2.2 The Internet network stack in most Unix systems 10

2.3 Organization of buffer descriptors 11

2.4 Steps involved in sending a packet 15

2.5 Steps involved in receiving a packet 16

3.1 A block diagram of the Tigon controller 22

3.2 Organization of event-driven firmware 23

3.3 Illustration of overlapped execution of send and receive 24

3.4 Firmware with two independent event dispatch loops 31

3.5 Execution time of the event handlers 32

3.6 UDP send throughputs 44

3.7 Composition of the improved UDP send throughputs 44

3.8 UDP receive throughputs 46

3.9 Composition of the improved UDP receive throughputs 46

3.10 UDP bidirectional throughputs 47

3.11 Composition of the improved UDP bidirectional throughputs 47

3.12 Execution time of Send Data Ready and Receive Complete 47

4.1 Steps in processing an HTTP request 58

4.2 Steps in processing an HTTP request to a file in the network interface data cache 64

4.3 Potential reduction in HTTP content traffic using LRU caches 66

4.4 Potential reduction in HTTP content traffic using various replacement policies 69

4.5 Potential reduction in HTTP content traffic using various block sizes 70

4.6 PCI bus utilization for the server configuration 2200/33 72

4.7 PCI bus utilization for the server configuration 2200/66 72

4.8 Comparison of the HTTP content throughputs 72

4.9 Network interface data cache architecture 75

4.10 Steps in sending a response using the `sendfile` system call 78

4.11 Network interface API to support network interface data caching 80

4.12 Illustration of a packet transmission using network interface data caching . 83

4.13 Reductions in PCI traffic from network interface data caching for the server configuration 2200/33 93

4.14 Throughput improvements from network interface data caching for the server configuration 2200/33 93

4.15 Reductions in PCI traffic from network interface data caching for the server configuration 2200/66 96

4.16 Throughput improvements from network interface data caching for the server configuration 2200/66 96

4.17 PCI bus utilization for the server configuration 2600/33 100

4.18 Reductions in PCI traffic from network interface data caching for the server configuration 2600/33 101

4.19 Throughput improvements from network interface data caching for the server configuration 2600/33 101

4.20 PCI bus utilization for the server configuration 2600/66 103

4.21 Reductions in PCI traffic from network interface data caching for the server configuration 2600/66 104

4.22 Throughput improvements from network interface data caching for the server configuration 2600/66 104

Tables

3.1 Events in the Tigon 26

3.2 Shared data in the released firmware 29

3.3 Shared hardware resources in the released firmware 30

3.4 Static partitioning of events 36

3.5 Instruction throughput and memory performance of the Tigon 40

3.6 Macrobenchmark results 49

4.1 Web trace statistics 85

4.2 Prototype server configurations 87

4.3 Throughput improvements from the `sendfile` system call 91

4.4 Performance improvements from network interface data caching for the server configuration 2200/33 95

4.5 Performance improvements from network interface data caching for the server configuration 2200/66 98

4.6 Performance improvements from network interface data caching for the server configuration 2600/33 102

4.7 Performance improvements from network interface data caching for the server configuration 2600/66 105

Chapter 1

Introduction

Networking servers have become widely deployed in recent year, undoubtedly thanks to the continuing growth of the Internet and services like the World Wide Web. The performance of networking servers has significantly improved in recent years, mostly due to developments in microprocessors, applications and operating system software, and, to a lesser extent, increased processing power in network interfaces. Network interface cards (NICs) have had a well defined role as a peripheral device that allows the host processor to transfer data between the main memory and the network. Since the tasks of traditional network interfaces are fixed and well understood, most network interfaces are built using efficient application-specific integrated circuits (ASICs). Because traditional network interfaces only perform simple data transfers, they typically have limited computation power and small buffer memory that stores incoming and outgoing data temporarily.

Adding programmability and storage capacity to the network interface provides new opportunities to extend the set of network services provided by the network interface beyond simple data transfers between the main memory and the network. Thus, programmable network interfaces provide the flexibility to exploit their direct interaction with the network in order to improve networking server performance. Programmable network interfaces can exploit their storage capacity and computation power to offload network protocol processing, reducing the processing requirement on the host processor, or to provide novel network

services to address other performance issues in the system. The use of local storage, rather than simple buffer memory, allows programmable network interfaces to perform tasks that require a greater amount of state information for bookkepping information or data reuse.

A number of projects have used programmable network interfaces to reduce the processing requirement on the host processor to communicate with the network. Examples include checksum offloading, support for zero-copy I/O and user-level protocol processing, as well as partial implementations of network protocols on the network interface [9, 16, 20, 34]. Checksum offloading moves checksum computation for network protocols such as TCP/IP from the host processor to the network interface, thereby reducing expensive data-touching operations that a number of researchers have identified as a potential bottleneck [11, 13, 19]. Virtually all modern network interface cards now support checksum computation. Zero-copy I/O eliminates data copying between the user and kernel spaces, also reducing data-touching operations. Various user-level protocol processing mechanisms allow user-level applications to handle network protocol processing rather than the kernel of the operating system. Implementing network protocols on the network interface also moves network protocol processing away from the kernel. While these techniques exploit the computation power of network interfaces, they do not explore opportunities to use the storage capacity of network interfaces.

Besides the academic efforts mentioned above, industrial efforts are being made to offload protocol processing onto the network interface in order to improve storage server performance. Recently, iSCSI [35] has emerged as a popular protocol for network storage servers. iSCSI provides SCSI-like access to remote storage using the traditional TCP/IP protocols as their transport layer. By using inexpensive and ubiquitous TCP/IP network

rather than expensive proprietary application-specific networks, iSCSI is gaining popularity. Several iSCSI adapters aim to improve storage server performance by offloading the complete implementation of the TCP/IP protocols or a subset of the protocol processing on the adapters.

Using programmable network interfaces, it is easy to implement various services and upgrade them in the future. In contrast, it is difficult and costly to implement and upgrade complex protocols such as the TCP/IP protocols using ASICs. However, the performance of programmable network interfaces may be at a disadvantage when compared to ASIC-based network interfaces. Programmable network interfaces must spend time executing instructions that form their firmware software, whereas ASIC-based network interfaces implement their functions directly in hardware. Because programmable processors are embedded on a peripheral device, they must operate within technology constraints such as limited chip area and slow clock rate available to the device. These technology constraints can further excerbate the performance disadvantages of programmable network interfaces. Thus, most protocol offloading uses ASICs in spite of their reduced flexibility and greater design cost.

1.1 Objective

This thesis shows that programmable network interfaces can be used to implement extended network services and improve networking server performance by exploiting their storage capacity as well as computation power, without performance disadvantages over application-specific network interfaces. First, the thesis investigates a technique to over-

come the performance disadvantages of programmable network interfaces that result from instruction processing. Then, the thesis proposes a novel network service on programmable network interfaces that exploits both computation power and storage capacity of network interfaces in order to improve networking server performance.

This thesis first examines the Tigon, a programmable Gigabit Ethernet controller with two embedded processors [1]. Single processor performance is found to be far below that of a modern ASIC-based network interface. However, by exploiting the task-level concurrency found in send and receive processing, the parallelized firmware efficiently utilizes both processors and enables performance comparable to that of a modern ASIC-based network interface.

When the Tigon runs the firmware that utilizes a single processor, its throughput for bidirectional UDP traffic consisting of maximum-sized datagrams is 938 Mb/s, far less than the maximum throughput 1882 Mb/s of a modern ASIC-based network interface. The parallelization of firmware improves the Tigon throughput for the same workload by 65%, resulting in 1553 Mb/s. Throughput for real network services improves by 32–107%. This improved performace falls within 10–20% of a modern ASIC-based network interface. The Tigon was released in 1997. Considering it was manufactured using rather old process technology, the results indicate that programmable network interfaces can overcome the performance disadvantage due to programmability by exploiting multiprocessor architectures and parallelization techinques.

This thesis then proposes network interface data caching, an extended network service that can be implemented on programmable network interfaces with a modest amount of local memory. Network interface data caching directly targets server performance by al-

lowing frequently requested data to be stored in a data cache on the network interface. The network interface sends the frequently requested data stored in its data cache out onto the network directly from the cache. Thus, network interface data caching reduces data transfers, alleviating the contention on the local I/O interconnect. The cache would reside in a modest amount of on-board DRAM of a programmable network interface that can store and access data within the cache. The operating system on the host processor determines which data to store in the network interface data cache and for which packets it should use data from the cache. Thus, by exploiting the storage capacity as well as computation power of programmable network interfaces, network interface data caching reduces contention on local I/O interconnects, thereby improving server performance.

Most systems use a PCI bus as their local I/O interconnect. While the maximum bandwidth of a PCI bus has increased, transferring data over a PCI bus incurs significant overheads. For instance, experimental results reveal that in a web server, over 30% of available PCI bandwidth can be lost due to overheads such as addressing, memory stalls, and competition among multiple PCI devices over the use of the bus. This lost bandwidth along with inefficient use of the interconnect resulting from a large number of repeated transfers in a web server can lead to the interconnect bottleneck. Furthermore, simply increasing the PCI bus bandwidth using faster clock speeds does not eliminate the local I/O bottleneck. Unless main memory latencies improve proportional to the PCI bus clock speed, faster PCI buses will waste a greater fraction of their bandwidth due to memory stalls.

Experiments using a prototype implementation of a web server with network interface data caching shows that network interface data caching results in modest increases in server throughput. This prototype is a static content web server and uses a uniprocessor PC-based

server with a 64-bit/66MHz PCI bus and two Gigabit Ethernet links. Adding a cache with 16 MB of memory on each network interface shows 35–58% reductions in server PCI bus traffic for four out of five workloads used to evaluate the techique, resulting in throughput improvements of 6–25% for three workloads and peak throughput of 1583 Mb/s. Throughput improvements suggest that networking servers that utilize high-speed network links can benefit from network interface data caching.

Thus, this thesis shows that programmable network interfaces will be a viable platform to implement extended network services in order to address performance issues in networking servers. Programmable network interfaces can improve networking server performance by exploiting their storage capacity as well as computation power, the latter of which can be improved by exploiting multiprocessor architectures.

1.2 Organization

The thesis first shows that programmable network interfaces can overcome performance disadvantages over application-specific network interfaces that result from programmability. It then examines an extended network service on programmable network interface that can improve server performance. This thesis is organized as follows.

Chapter 2 describes the operations of network interfaces from three viewpoints: the system architecture, the traditional Internet TCP/IP network stack, and communications between the device driver and the network interface. Communications that must take place to send or receive a packet are described in detail in order to understand the workload of the network interface.

Chapter 3 introduces a strategy for parallelizaing the firmware of the Tigon, a programmable Gigabit Ethernet controller with multiple embedded processors. By exploiting the multiprocessor architecture and the task-level concurrency found in send and receive processing, the parallelized firmware alleviates the performance disadvantages due to programmability. This result suggests that properly architectured programmable network interfaces would be able to enjoy flexible programmability without sacrificing performance, facilitating the support of extended network services.

Chapter 4 presents network interface data caching, a novel technique to reduce local interconect traffic using a programmable network interface. The effectiveness of network interface data caching demonstrates that a programmable network interface can further improve networking server performance through extended network services.

Chapter 5 discusses several previous projects that have used programmable network interfaces to provide the host processor with extended functionality on the network interface. Most of these projects try to reduce the processing requirement on the host processor to send and receive packets by offloading components of network protocol processing onto the network interface.

Finally, chapter 6 concludes that programmable network interfaces are a viable option to improve networking server performance and warrant further research in computer architecture as well as system software. The chapter closes with possible network services that future programmable network interfaces can provide in order to improve networking server performance.

Chapter 2

Network Interface Operations

The traditional role of a network interface is to send and receive packets on behalf of the host CPU. This chapter describes operations of network interfaces from three viewpoints. First, in the system architecture, NICs are a peripheral device that transfers data between the main memory and the network. Second, NICs occupy the data link layer in the Open System Interconnect reference model. In the traditional TCP/IP network stack, NICs correspond to the layer below the IP layer, and they send and receive packets on behalf of the IP layer. Finally, from the viewpoint of systems software, NICs provide an interface to the host CPU so that the device driver running on the host CPU can communicate with the network.

The operations of NICs described in this chapter do not assume any particular implementation of a NIC and therefore apply to both programmable and application-specific nonprogrammable network interfaces. Programmable NICs implement these tasks by executing firmware code on one or more programmable processors, whereas application specific NICs implement them in hardware.

2.1 System Architecture

The system architecture of a typical networking server consists of one or more host CPUs, NICs, and disks. Individual components are connected through a local interconnect such

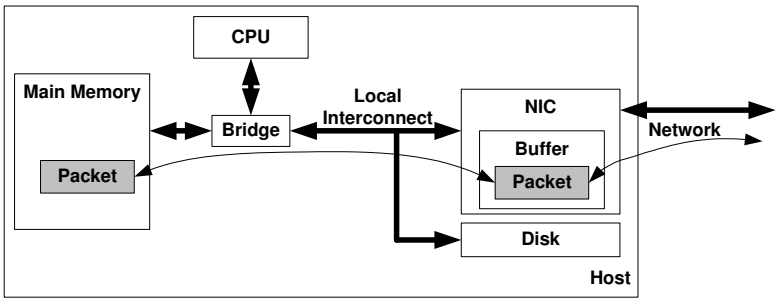

Figure 2.1 : The system architecture of an example networking server. The system shown in the figure is a uniprocessor system with a local interconnect that connects the main memory, NIC, and disk. The NIC transfers data between the main memory and the network.

as a peripheral component interconnect (PCI) bus. A uniprocessor system with single NIC and disk is illustrated in Figure 2.1. The main memory and peripherals, NIC and disk, are connected through a local interconnect. The NIC provides the host CPU the ability to exchange data between the local interconnect and the network.

To send packets, the host CPU creates and stores packets in the main memory. When instructed by the host CPU, the NIC transfers the packets from the main memory to the transmit buffer on the NIC through *direct memory access* (DMA) reads. The NIC then sends the packets out to the network using the link-level protocol for the network such as Ethernet. When packets arrive from the network, the NIC stores them in its receive buffer and transfers them to the main memory through DMA writes. Since the host CPU is in charge of the main memory, the NIC cannot write the packets into arbitrary regions in the main memory. Instead, the host CPU pre-allocates main memory buffers prior to actual receive, and the NIC must use only those pre-allocated main memory buffers to store the received packets. Once the newly arrived packets are stored in the main memory, the NIC notifies the host CPU of the packets, typically through an interrupt.

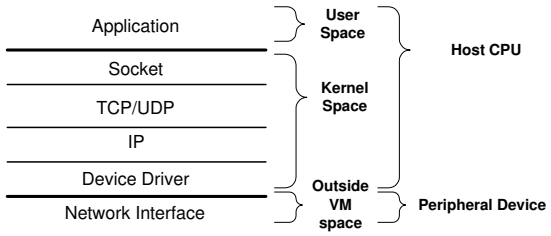

Figure 2.2 : The Internet network stack in most Unix systems. The kernel of the operating system implements the socket layer, TCP/UDP layer, IP layer, and device driver. The functionalities of the NIC are encapsulated by the device driver and exported to the IP layer. The NIC only interacts with the device driver and is invisible to the layers above the device driver.

2.2 Network Stack

The traditional Internet network stack in most Unix systems is shown in Figure 2.2. The stack consists of the application running in the user space, and the socket layer and the protocol stack implemented in the kernel of the operating system. Each layer communicates only with the layers directly above and below. The socket layer provides the application an abstraction of communication end points in the network. In order to send or receive data, the socket layer communicates with the protocol stack. The protocol stack is responsible for delivering data to and receiving from a remote host and consists of the TCP/UDP layer, the IP layer, and the device driver for the NIC. The TCP/UDP layer and the IP layer implement TCP/UDP and IP protocols respectively. As a part of the protocol implementation, the protocol stack traditionally computes various checksums for these protocols. The device driver encapsulates the send and receive funtionalities provided by the NIC.

The NIC communicates only with the device driver. To send a packet, the IP layer passes the packet to the device driver. The device driver then communicates with the NIC in order to send the packet out to the network. Receive occurs asynchronously when the

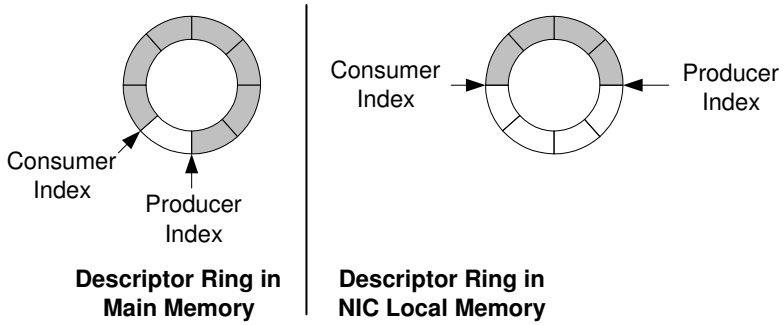

Figure 2.3 : Buffer descriptors are organized into a circular buffer (ring) structure. A ring has a producer index and a consumer index in order to account for valid or unprocessed descriptors. Both the main memory and the NIC local memory have a copy of the ring as well as the producer and consumer indices. In the figure, shaded blocks represent unprocessed (produced) descriptors while white blocks represent processed (consumed) descriptors. The ring in the NIC local memory may be inconsistent with the ring in the main memory, as shown in the figure. In the figure, both producer and consumer indices in the NIC local memory are different from those in the main memory.

NIC stores a newly received packet in the main memory and notifies the device driver of the packet. The device driver determines whether the received packet is valid and passes it to the IP layer.

2.3 Device Driver/NIC Communication

As mentioned earlier in the chapter, the operating system stores packets to and retrieves them from the main memory. The device driver and the NIC must communicate enough information with each other in order for the NIC to transfer packets between the main memory and the network. For example, the device driver must inform the NIC of the main memory address and the length of a packet to be sent. The device driver and the NIC communicate using buffer descriptors, memory-mapped registers, and interrupts.

A buffer descriptor is a data structure that specifies a contiguous region of memory.

It contains a main memory address, a length field, and additional flags to specify options or commands. The device driver and the NIC use buffer descriptors to specify memory regions, such as packets that are to be sent or have been received, in the main memory. The device driver and the NIC may define as many types of buffer descriptors as necessary. Buffer descriptors are typically organized into a circular buffer (ring) structure as illustrated in Figure 2.3. Associated with a ring of buffer descriptors are its producer and consumer indices. The producer index points to the most recently created buffer descriptor, while the consumer index points to the buffer descriptor that has been most recently processed and thus can be recycled. The buffer descriptors pointed by the producer and consumer indices mark the boundaries of the unprocessed buffer descriptors shown as shaded blocks in the figure.

Conceptually, there is one global buffer descriptor ring for each type of buffer descriptor. The device driver and the NIC cooperatively maintain the ring using the producer/consumer relationship. In the producer/consumer relationship, the producer only creates new buffer descriptors and updates the producer index, while the consumer only processes the buffer descriptors and updates the consumer index. Thus, a buffer descriptor ring can be thought as a one-way conduit through which the device driver passes information about packets (encapsulated in buffer descriptors) to the NIC or vice versa. Exact semantics of production and consumption of buffer descriptors depend on the implementation of the NIC. For example, the device driver may produce buffer descriptors to signify that there are packets to be sent, while the consumption of buffer descriptors by the NIC may signify that the packets have been sent out onto the network.

Because the main memory on the host and the local memory on the NIC are physically

separated, the global buffer descriptor ring is implemented using two copies of the ring; one in the main memory owned by the device driver, and another in the NIC local memory owned by the NIC. Each copy of the ring has its own producer and consumer indices as well as buffer descriptors as shown in Figure 2.3. To pass new buffer descriptors to the consumer, the producer first creates them in its own copy of the ring, updates its own producer index, and notifies the consumer of the updated producer index. Upon notification of the updated producer index, the consumer determines what buffer descriptors have been produced using its own producer index and the updated producer index. The consumer then fetches the new buffer descriptor and updates its producer index. Alternatively, the producer can store the new buffer descriptors in the consumer's copy of the ring. Finally, the consumption of buffer descriptors follows the opposite steps.

The two copies of the ring need not be indentical in size and state. Notification of production or consumption of buffer descriptors may be delayed arbitrarily. Thus, inconsistencies are allowed as long as each copy maintains the producer/consumer relationship both locally and with respect to the other copy. Suppose that the device driver is the producer, and the NIC is the consumer. First, the producer (the device driver) must ensure that its producer index is always ahead of the consumer index, and the consumer (the NIC) must do likewise. Second, the producer and consumer must ensure that the producer's (the device driver) producer index is not behind the consumer's (the NIC) producer index, and the consumer's (the NIC) consumer index is not behind the producer's (the device driver) consumer index. As long as these four conditions are met, the device driver and the NIC can use the producer/consumer relationship to pass buffer descriptors to each other. For example, the device driver produces buffer descriptors in the main memory to signify that

it wishes to send packets stored in the main memory. The device driver then notifies the NIC of the updated producer index. At this point, the two copies of the ring are inconsistent. However, they do not violate the producer/consumer relationship. It is up to the NIC whether and when to resolve the inconsistency by fetching the new buffer descriptors from the main memory. It is also up to the NIC to decide how many buffer descriptors to fetch. This situation is illustrated in Figure 2.3. In the figure, the producer index in the NIC local memory is behind the producer index in the main memory. Also the consumer index in the main memory is behind the consumer index in the NIC local memory, indicating that the consumer (the NIC) has consumed descriptors, but the producer (the device driver) has not updated its consumer index.

The NIC includes a number of memory-mapped registers, which the device driver can read and write through memory-mapped I/O. These registers are located at fixed local memory addresses on the NIC specified by the manufacturer of the NIC and are used to initialize the NIC. For instance, the NIC typically has a register that stores the base address of a buffer descriptor ring and other registers that indicate whether new buffer descriptors have been created. Using these registers, the device driver notifies the NIC of the location of buffer descriptors, and the NIC can fetch the buffer descriptors from main memory.

Finally, the NIC uses interrupts to alert the host CPU of asynchronous events. When interrupted, the host CPU invokes the device driver for the NIC and processes pending events. For example, the NIC typically interrupts the host CPU to signify that new packets have been received.

Actually sending or receiving a packet requires a series of communications using the mechanisms described above. Figure 2.4 shows the required steps involved in sending

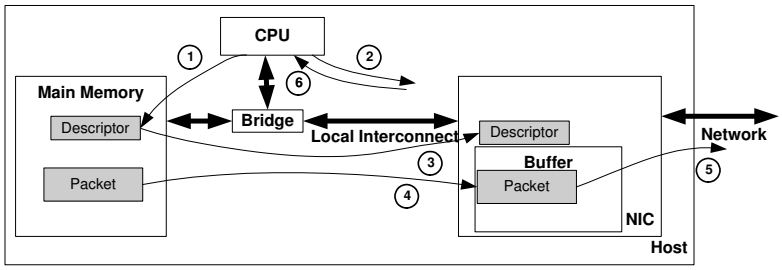

Figure 2.4 : Steps involved in sending a packet from the main memory out onto the network. Step 1: the host CPU creates a buffer descriptor for the packet to be sent. The buffer descriptor is stored in the main memory. Step 2: the host CPU notifies the NIC of the new buffer descriptor. Step 3: the NIC fetches the buffer descriptor through a DMA read. Step 4: using the information stored in the buffer descriptor, the NIC transfers the packet from the main memory to its transmit buffer. Step 5: the packet is sent out onto the network through a MAC unit. Step 6: the NIC notifies the host CPU that the buffer descriptor has been processed.

a packet. In step 1, the device driver creates a buffer descriptor that contains the starting memory address and length of the packet that it wishes to send. If the packet spans multiple discontiguous memory regions, then multiple buffer descriptors must be created, one for each continous region of memory. In step 2, the device driver writes to a memory-mapped register on the NIC with information about the new buffer descriptors. For instance, the producer index of the buffer descriptor ring suffices for the NIC to determine which buffer descriptors must be fetched. In step 3, the NIC initiates one or more DMA reads to transfer the descriptors to the local memory on the NIC. In step 4, using the starting memory address and length stored in the buffer descriptors that now reside in the local memory, the NIC initiates DMA reads to fetch actual packet data from the main memory into its transmit buffer. In step 5, the NIC sends the packet out to the network using its *medium access control* (MAC) unit, which implements the link level protocol such as Ethernet. Finally in step 6, the NIC informs the device driver that the descriptor has been processed, possibly

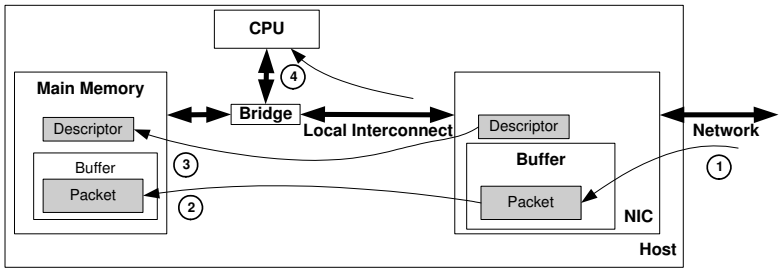

Figure 2.5 : Steps involved in receiving a packet from the network into the main memory. Step 1: a packet arrives from the network through a MAC unit. The packet is stored in the receive buffer on the NIC. Step 2: the NIC transfers the packet from the receive buffer into a main memory buffer through a DMA write. The main memory buffer has been pre-allocated by the device driver prior to prior to receiving the packet. Step 3: the NIC creates a buffer descriptor for the packet that is now stored in the main memory. The buffer descriptor contains the starting address and actual length of the packet. The NIC then transfers the buffer descriptor to the main memory through a DMA write. Step 4: the NIC notifies the host CPU of the new buffer descriptor.

by interrupting the host CPU.

Unlike send, receive is asynchronously initiated by the NIC. Because the host CPU cannot anticipate when packets will arrive from the network or the size of packets, the device driver must pre-allocate a pool of main memory buffers to store packets that may arrive in the future. As received packets consume these buffers, the device driver must frequently check the pool and replenish it when the pool is low on unused buffers. The device driver notifies the NIC of the location and size of the main memory buffers by following steps similar to steps 1 through 3 of Figure 2.4. For each main memory buffer, the device driver creates a buffer descriptor that stores the starting memory address and size of the buffer. It then writes into a memory-mapped register on the NIC with the information about the new buffer descriptor. The NIC transfers the buffer descriptor to its local memory through DMA reads. Now the NIC can find the location and size of the main memory

buffers.

The steps involved in receiving a packet from the network are shown in Figure 2.5. In step 1, a packet arriving from the network is received by the MAC hardware and stored in the receive buffer on the NIC. In step 2, the NIC initiates DMA writes to transfer the packet into an unused main memory buffer that has been pre-allocated by the device driver. In step 3, the NIC creates a buffer descriptor that stores the starting memory address and length of the received packet that now resides in the main memory. It then transfers the descriptor to the main memory through a DMA write. Finally, in step 4, the NIC alerts the device driver that the descriptor of the received packet has been produced, typically by interrupting the host CPU. At this point, the device driver may check the number of unused main memory buffers and allocate new buffers if necessary.

2.4 Summary

Network interface cards provide the host processor the ability to communicate with the network. Packets are stored in the main memory, and NICs transfer them between the main memory and network when instructed by the host processor. Thus, the primary purpose of NICs is data transfers between the main memory and network. The host processor and NICs communicate information regarding packets through buffer descriptors. Because the host processor and NICs are physically separated, sending or receiving a packet requires a series of communications with each other. The network interface processing then involves maintaining buffer descriptors, keeping track of outstanding data transfers, alerting the host processor of received packets, and finally coordinating these tasks according to the steps

required for sending or receiving packets.

Chapter 3

Exploiting Task-level Concurrency in a Programmable Network Interface

The steps for send and receive described in the previous chapter can be implemented using fixed state machines or programmable processors. A network interface with programmable processors is called a programmable network interface since the details of its implementation can be changed by modifying the software (firmware) running on the interface. Thus, programmable network interfaces can easily provide additional network services beside sending and receiving packets by implementing those services in software.

While programmable network interfaces have the flexibility to extend network services, their performance may suffer from instruction processing overheads associated with programmability. Overheads include instruction storage, fetching instructions, potentially high instruction counts required to implement particular tasks, and the achievable amount of parallel computation for the given microarchitecture. Due to technology constraints such as limited cooling area and power distribution on a PCI card, an embedded processor like a programmable network interface controller faces rather limited chip area and low clock speed compared to general-purpose processors. Instruction storage consumes scarce on-chip resources. Fetching instructions lowers effective memory bandwidth. Instruction counts and the achievable amount of parallel computation restrict the complexity of tasks that can be implemented in software. To achieve the target performance, programmable

network interfaces may require more hardware resources and faster clock rate than allowed by technology constraints. On the other hand, application-specific nonprogrammable network interfaces may be able to achieve the same or better performance with less hardware resources and lower clock speed.

Unless this performance disadvantage of programmable network interfaces is alleviated, programmable network interfaces will be at best a test platform for extended network services. This chapter examines the performance of send and receive processing of the Tigon programmable Gigabit Ethernet controller and introduces a parallelization strategy for Tigon firmware. The Tigon includes two programmable processors. However, when only one processor is utilized, the Tigon performs far worse than application specific network interfaces. Using a NIC based on the Tigon running firmware that utilizes only one processor, experimental results show that its maximum throughput for a bidirectional workload consisting of maximum-sized Ethernet frames is 938 Mb/s, far less than the maximum throughput 1882 Mb/s of a modern ASIC-based NIC for the same workload.

Fortunately, there is abundant parallelism in network interface processing. Exploiting parallelism using multiple processors can alleviate the performance disadvantage of programmble NICs due to instruction processing overheads. The parallelization of firmware improves the Tigon throuhgput for the above workload by 65% resulting in 1553 Mb/s. For the same workload except with minimum-sized Ethernet frames, the parallelization improves throughput by 157%. Accordingly, throughput for real network services increases by 32–107%. Finally, the improved performace falls within 10–20% of a modern ASIC-based NIC for real network services. This shortcoming of the Tigon even with the parallelized firmware seems to stem from the increasing contention on the memory of the Tigon

as the Tigon delivers over 1Gb/s throughput.

3.1 Tigon Programmable Gigabit Ethernet Controller

The Tigon is a programmable Gigabit Ethernet controller with a PCI host interface. A block diagram of the Tigon is shown in Figure 3.1. The Tigon has two 88 MHz single-issue, in-order embedded processors that are based on MIPS R4000. The processors share access to external SRAM. Each processor also has a private on-chip *scratch pad* memory, which serves as a low-latency software-managed cache. The firmware may use the scratch pads to store frequently accessed code regions or private data. However, shared data must be stored in the shared external SRAM since a processor cannot access the scratch pad of the other. Each processor also has a one-line (64-byte) instruction cache to capture spatial locality for instructions fetched from the SRAM. Similarly each processor has a one-line (8-byte) data cache.

3.1.1 Hardware Features

A hardware DMA controller enables the firmware to transfer data between the system's main memory and external SRAM on the NIC, while a hardware MAC unit enables the firmware to transfer data between the network and external SRAM. The DMA controller provides separate read and write channels. Each channel has an associated queue of DMA descriptors. Each DMA descriptor contains enough information to initiate one DMA transfer (such as host memory address, local SRAM address, and length). The PCI bus is half-duplex and can only support one transfer at a time, so the DMA controller serializes the transfers pending in these queues. Similarly, the MAC unit provides separate transmit and

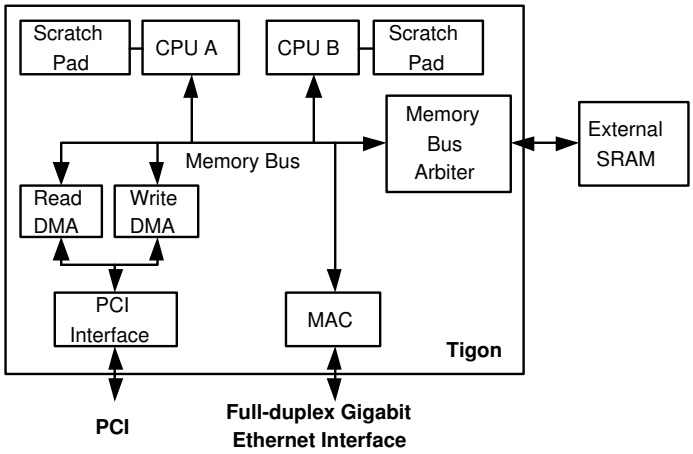

Figure 3.1 : A block diagram of the Tigon controller. Two RISC processors have private scratch pads and share access to the external SRAM. A DMA controller transfers data between the main memory and external SRAM. A MAC unit transfers data between the external SRAM and Ethernet.

receive channels, each with a queue of MAC descriptors that contain information about packets to be transmitted or that have been received. Unlike the PCI bus, Ethernet interface is full-duplex, and one send and one receive can occur simultaneously.

The Tigon provides three special-purpose registers to facilitate parallelization: a hardware semaphore register and an event register for each processor. The hardware semaphore register provides support for a single lock. The standard MIPS synchronization instructions (load-linked and store-conditional) are not supported, but additional locks can be implemented in software using the hardware semaphore. The event registers provide an efficient event notification mechanism to support event-driven firmware. Each bit in the event registers corresponds to a particular event; when a bit is set, it signifies that a particular event has occurred and has not yet been handled. The Tigon hardware reserves a number of bits for predefined hardware-generated events, and the firmware can use the rest of the bits for firmware-generated events.

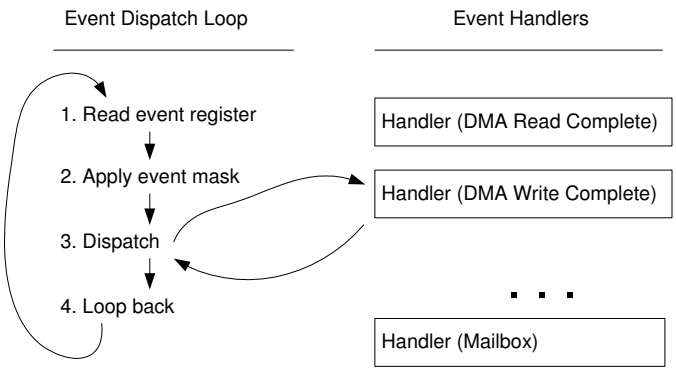

Figure 3.2 : Organization of event-driven firmware. Firmware consists of an event dispatch loop and a set of event handlers. Each handler processes an event (shown in parenthesis). The event names are those listed in Table 3.1.

3.1.2 Event-driven Firmware

As mentioned in the previous section, the Tigon is designed to support event-driven firmware. Event-driven firmware consists of an event dispatch loop and a set of event handlers as shown in Figure 3.2. Send and receive are implemented using a series of event handlers. Firmware normally spins in the event dispatch loop until an event occurs. Unwanted events are ignored by applying an event mask. Upon an event, the firmware executes the handler corresponding to the event. Unlike the asynchronous event notification mechanisms found in most operating systems, events in the Tigon do not interrupt the execution of handlers. Thus the handler executes to completion, upon which the control returns to the event dispatch loop.

The Tigon includes two special instructions `PRI` and `JOFF` to assist the event dispatch loop. `PRI` instruction first combines the contents of two source registers using logical AND and then produces the bit number of the most significant bit set. The event dispatch loop uses `PRI` instruction to apply event mask and compute the index into a jump table that

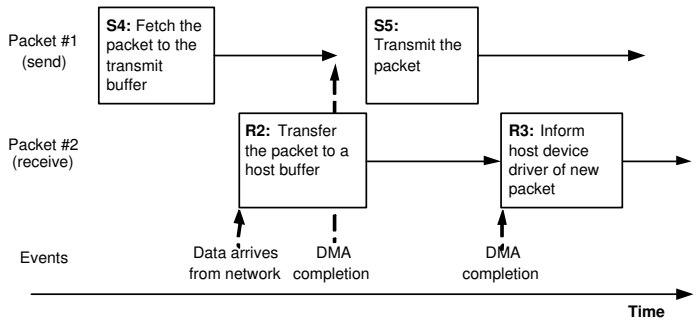

Figure 3.3 : Illustration of overlapped execution of the send and receive steps in Figures 2.4 and 2.5. Asynchronous events allow the firmware to overlap the execution of multiple steps of send and receive processing.

consists of jump instructions to event handlers. `JOFF` instruction left shifts the content of the first source register by one , adds the result to the content of the second source register, and then sets the program counter to the result of the addition. `JOFF` is used to compute the address in the jump table corresponding to the index produced by `PRI` instruction. The four steps of the event dispatch loop shown in Figure 3.2 consists of only four instructions using `PRI` and `JOFF`.

The hardware event mechanisms and descriptor queues of the Tigon enable firmware to efficiently handle multiple packets in transit and to overlap long latency operations, like DMA transfers, with other processing. For example, Figure 3.3 illustrates some of the processing for two packets; the numbers in the blocks correspond to the send and receive steps in Figures 2.4 and 2.5. Each block in the figure represents execution of an event handler. The first block in the figure corresponds to step 4 in sending packet #1, which initiates a DMA transfer of the packet. Rather than waiting for the DMA transfer to complete, the event handler for this block enqueues the appropriate DMA descriptor to the DMA controller and yields control to the event dispatch loop. While the DMA transfer of packet #1

is in progress, packet #2 arrives from the network. The arrival of the packet generates an event, upon which the firmware enqueues a DMA write to transfer the received data to the main memory.

During the processing of step R2, the DMA transfer for packet #1 completes. Upon the completion of a DMA transfer, the Tigon generates an event. As noted previously, events do not interrupt firmware execution. Once an event handler is invoked, it is run to completion, even if other, possibly more urgent, events occur. Thus, the NIC does not initiate the processing of step S5 until the processing for step R2 completes. When step R2 completes, the firmware enters the event dispatch loop and executes the event handler for the pending event. The event handler for step S5 initiates the packet transmission and yields control to the event dispatch loop. Finally, when the DMA write completes for packet #2, the NIC informs the device driver of the new packet in the main memory.

3.1.3 Released Tigon Firmware

Alteon Websystems, the original manufacturer of the Tigon, released several versions of the Tigon firmware as open-source software [2]. This released firmware is event-driven, consisting of an event dispatch loop and a set of event handlers as described in Section 3.1.2. Using the `PRI` instruction, the event dispatch loop prioritizes events according to the order of the corresponding bits in the event register. Table 3.1 describes the hardware-generated and firmware-generated events used by the firmware to send and receive packets.

The firmware defines three types of buffer descriptors: *send* for packets to be sent, *receive* for pre-allocated buffers in the main memory into which the NIC may later store received packets (as described in Section 2.3, and *receive return* for received packets after

Hardware-generated Event	**Description**
Mailbox	The host CPU has written into a memory-mapped Tigon mailbox register.
DMA Read Complete	A DMA read transfer has completed.
DMA Write Complete	A DMA write transfer has completed.
Receive Complete	A packet has been received.
Firmware-generated Event	**Description**
Send Buffer Descriptor Ready	The device driver has produced send buffer descriptors. The firmware needs to fetch them into the local memory.
Send Data Ready	Send buffer descriptors have been fetched into the local memory. The firmware needs to fetch packets specified by the descriptors.
Update Send Consumer	The firmware has finished processing the buffer referenced by a send buffer descriptor and has updated the consumer index of its copy of the send ring. The firmware needs to transfer the index to the main memory.
Receive Buffer Descriptor Ready	The device driver has produced receive buffer descriptors. The firmware needs to fetch them into the local memory.
Update Receive Return Producer	The firmware has produced a receive return buffer descriptor in the main memory and updated the producer index of its copy of the receive return ring. The firmware needs to transfer the index to the main memory.

Table 3.1 : Hardware-generated and firmware-generated events in the Tigon for send and receive processing.

they are stored in the main memory. Send and receive buffer descriptors are produced by the host device driver and consumed by the NIC, while receive return buffer descriptors are produced by the NIC and consumed by the host device driver. Buffer descriptors are organized in the ring structure as described in Section 2.3. There is a separate buffer descriptor ring for each type of buffer descriptors. The device driver and firmware cooperatively manage the buffer descriptor rings by having two copies of the ring; one in the main memory

and the other in the NIC local memory.

The high level steps of send and receive processing described in Section 2.3 are implemented through multiple events. Sending a packet specifically consists of the following sequence of events.

1. Mailbox – Read the producer index of the device driver's copy of the send buffer descriptor ring from a *mailbox register*. Raise the Send Buffer Descriptor Ready event.
2. Send Buffer Descriptor Ready – Enqueue a DMA descriptor to transfer the newly produced buffer descriptor(s).
3. DMA Read Complete – Determine that the completed DMA has transferred send buffer descriptors. Update the producer index of the firmware's send ring based on the number of fetched descriptors. Raise the Send Data Ready event.
4. Send Data Ready – Enqueue one or more DMA descriptors to transfer a packet into the transmit buffer. Also enqueue a MAC descriptor to inform the MAC to transmit the packet when it arrives in the transmit buffer.
5. DMA Read Complete – Determine that the completed DMA has transferred packet data. Update the consumer index of the firmware's send ring. If the number of send descriptors consumed is greater than the interrupt coalescing threshold, raise the Update Send Consumer event.
6. Update Send Consumer – Enqueue a DMA descriptor to transfer the consumer index of the send ring to the host.
7. DMA Write Complete – Determine that the completed DMA has transferred the consumer index. Interrupt the host CPU.

The pre-allocation of main memory buffers for future received packets follows steps 1–3 above, but using the receive buffer descriptor ring and Receive Buffer Descriptor Ready, instead of the send ring and Send Buffer Descriptor Ready. Steps 4 and 5 are eliminated, as there is no data to fetch. Transferring the consumer index of the receive buffer descriptor ring is also not needed, as it is stored as a field of the receive return buffer descriptors that are produced by the NIC. When the device driver retrieves a receive return buffer

descriptor from the main memory, it can extract the consumer index from the descriptor. For the receive return buffer descriptor ring, the firmware only maintains its producer index because the device driver ensures that the receive return buffer descriptor ring is never full when the receive buffer descriptor ring is not full. Thus, as long as there are unused main memory buffers, the receive return buffer descriptor ring is not full, and checking the consumer index is unnecessary. The actual receive into pre-allocated main memory buffers consists of the following events.

1. Receive Complete – Enqueue a DMA descriptor to transfer the packet into the buffer described by the next buffer descriptor in the receive ring. Also create a receive return buffer descriptor with information about the packet, and enqueue a DMA descriptor to transfer the new buffer descriptor. If the number of receive return descriptors produced is greater than the interrupt coalescing threshold, raise the Update Receive Return Producer event.
2. Update Receive Return Producer – Enqueue a DMA descriptor to transfer the new producer index of the receive return buffer descriptor ring.
3. DMA Write Complete – Determine that the packet has been transferred to the host. No action is needed.
4. DMA Write complete – Determine that the receive return buffer descriptor has been transferred to the host. No action is needed.
5. DMA Write Complete – Determine that the producer index of the receive return ring has been transferred to the host. Interrupt the host CPU.

The various events used to send and receive packets must share certain data structures such as the send or receive buffer descriptor ring to maintain global state information. The events also share hardware resources such as DMA channels. Table 3.2 and 3.3 list the data and hardware resources shared by the event handlers and the cause of sharing in the released firmware. Data is typically shared by subsequent steps in the send or receive sequence, or between the last step in transferring descriptors for the pre-allocated main memory buffers and the first step in the actual receive. Events related to sending or pre-allocating main

Shared Data	Sharing Events	Cause
Producer index of the device driver's send buffer descriptor ring	Mailbox Send Buffer Descriptor Ready	The device driver notifies the NIC that it has produced send buffer descriptors by updating this index and writing it in a mailbox register. The firmware must then use this producer index to fetch the new buffer descriptors.
Producer index of the firmware's send buffer descriptor ring	DMA Read Complete Send Data Ready	When the DMA unit completes a transfer of buffer descriptors, it must update the number of buffer descriptors stored in the local memory. Send Data Ready then uses these locally-copied descriptors to fetch packets.
Consumer index of the firmware's send buffer descriptor ring	DMA Read Complete Update Send Consumer	The firmware has completely consumed a send packet when the DMA unit completes the transfer of packet data. At this point, it updates the send consumer index and must then transfer this index to the host.
Producer index of the device driver's receive buffer descriptor ring	Mailbox Receive Buffer Descriptor Ready	The device driver notifies the NIC that it has produced receive buffer descriptors by updating this index and writing it in a mailbox register. The firmware must then use this producer index to fetch the new buffer descriptors.
Producer index of the firmware's receive buffer descriptor ring	DMA Read Complete Receive Complete	When the firmware has fetched the buffer descriptors that will later be used for received packets, it updates the number of descriptors stored in the local memory. The Receive Complete function later uses these when choosing the buffer into which it should transfer a received packet into the main memory.
Producer index of the firmware's receive return buffer descriptor ring	Receive Complete Update Receive Return Producer	The Receive Complete function produces a receive return buffer descriptor for a received packet. The Update Receive Return Producer transfers the producer index of the receive return ring to the main memory to notify the device driver of the received packets.

Table 3.2 : Shared data in the released firmware.

memory buffers share the read DMA channel, while the write DMA channel is primarily shared among the events related to the actual receive process. The only exception is

Shared Hardware	Sharing Events	Cause
DMA read channel	DMA Read Complete Send Buffer Descriptor Ready Send Data Ready Receive Buffer Descriptor Ready	The device driver produces packets to be sent and send/receive buffer descriptors. The NIC transfers these to its memory through DMA reads.
DMA write channel	DMA Write Complete Update Send Consumer Update Receive Return Producer Receive Complete	The NIC passes received packets, updated ring indices, and the receive return buffer descriptors that it produces back to host memory through DMA writes.

Table 3.3 : Shared hardware resources in the released firmware.

that Update Send Consumer, which is a part of send processing, requires the write DMA channel.

3.2 Parallelization of Firmware

The released firmware for the Tigon only makes use of a single processor. However, the single processor is not capable of processing bidirectional traffic at rates comparable to modern ASIC-based NICs. Furthermore, technology limitations on a PCI card, including limited cooling area and power distribution, constrain the clock speed of a processor on the card, so performance cannot be gained by simply increasing operating frequency. Therefore, the firmware must be parallelized across multiple processors in order to approach the performance of special-purpose NICs.

3.2.1 Parallelization Concepts

Effective parallelization requires identifying the concurrent tasks, partitioning those tasks among processing elements for load balance, and avoiding sharing of data or hardware

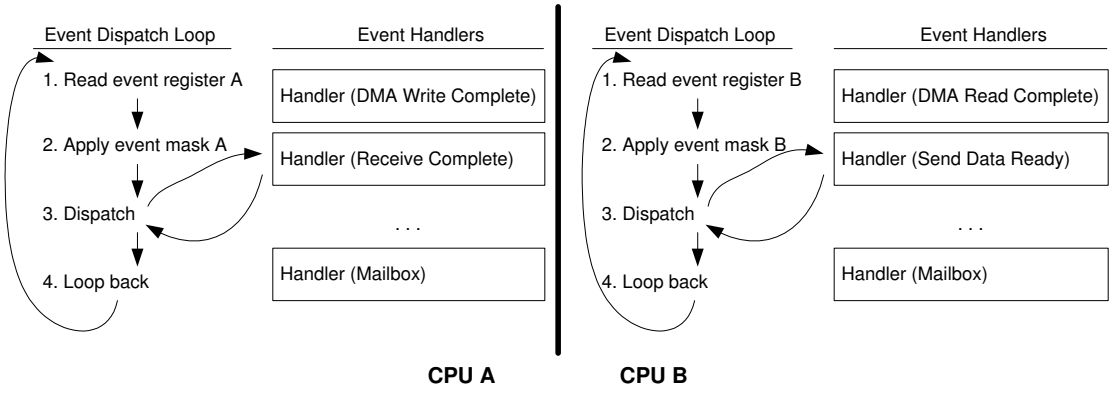

Figure 3.4 : Firmware with two independent event dispatch loops. Each processor in the Tigon contains an event register. Events can be partitioned and assigned to specific processors by having separate event dispatch loops and event masks.

resources between processing elements. In an event-driven system like the Tigon firmware, the fundamental unit of concurrency is an event handler. Thus to exploit parallelism, the events may be partitioned and assigned to specific processors.

Since the Tigon contains one event register per processor, partitioning events can be accomplished by having independent event dispatch loops as shown in Figure 3.4. Each processor has its own event register and event mask. Thus each processor can handle different sets of events by using separate event masks.

However, partitioning event handlers across the processors may require synchronization. As described in Section 3.1, event handlers always run to completion without being interrupted. Thus, sharing of data or hardware resources between event handlers that always run on the same processor requires no synchronization. An effective partition should consider the sharing patterns described in Table 3.2 and 3.3 and place events that share data or resources on the same processor whenever possible in order to reduce sharing across the processors.

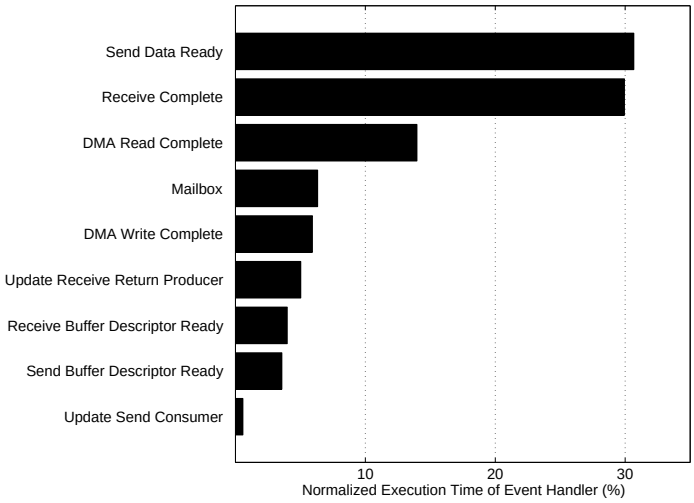

Figure 3.5 : Execution time of the event handlers normalized to the total execution time of all send and receive events.

Some forms of parallelization may enable better use of the Tigon hardware mechanisms. For example, data or event handler code that is exclusively used by one processor may be placed in that processor's private scratch pad, allowing much faster access than the external SRAM. Event handler code may be replicated in one or both scratch pads, but executing an event handler solely on one processor facilitates better scratch pad efficiency.

Finally, the cooperation between the firmware and the device driver provides additional support for parallelization. In particular, a firmware operation that impedes parallelization may simply be eliminated if the host device driver can implement it efficiently instead.

3.2.2 Parallelization Strategy

The events may be partitioned statically by assigning one set of events to one processor and the remaining events to the other. Statically partitioning event handlers must target specific workloads in order to achieve a good balance of load between processors. The

amount of work done by a processor is a function of three factors: the events handled by the processor, the frequency of those events, and the amount of processing required for the event handler corresponding to each event. Since distinct workloads utilize handlers differently, a particular partition of the event handlers may achieve a good load balance for one workload but could result in an imbalance of load for another workload. The target workload of the parallelization strategy described is bidirectional traffic with equal emphasis on send and receive. This workload is used to evaluate impact of parallelization on maximum throughput because it leads to maximum throughput achievable using a full-duplex Ethernet interface.

Figure 3.5 profiles the execution time of the event handlers in the released firmware for a workload consisting of simultaneous sends and receives of minimum-sized (64-byte) Ethernet frames. The send and receive rates are equal and are chosen such that no packets are dropped by the Tigon to ensure that the Tigon processor is not saturated. The Y axis shows the events. The X axis shows the processor execution time consumed for each event handler as a fraction of the total execution time of all send and receive events. The scratch pads are disabled during the measurement to eliminate any bias for the particular subset of code and data that the Tigon manufacturer chose to place in the scratch pad for the released firmware.

The Send Data Ready and Receive Complete handlers clearly dominate execution time at roughly 30% each. Among the other event handlers, DMA Read Complete is the next largest at 14%. It is over twice as expensive as the DMA Write Complete handler because the DMA Read Complete handler uses the newly read data to initiate other actions. Each of the other handlers consumes 4–6% of the total, except for Update Send Consumer as the

least significant at 0.6%.

Since they collectively account for 60% of the execution time, a static partition of event handlers for this workload should place Receive Complete and Send Data Ready on separate processors in order to balance the load on the processors. The other event handlers should be selected based on both load balance and the sharing patterns shown in Table 3.2 and 3.3. DMA Read Complete shares data with both of the dominant event handlers, but only shares hardware with Send Data Ready. Thus, DMA Read Complete should be colocated with Send Data Ready. However, this particular colocation is only valuable if it eliminates synchronization for this hardware resource – the DMA read channel. Thus, Send Buffer Descriptor Ready and Receive Buffer Descriptor Ready should also be put on the same processor. Both of these handlers share data with Mailbox, but placing that handler on the same processor would push this processor's load well above 50% of the total. DMA Write Complete shares no data with any other event, but should be placed on the same processor as Receive Complete since they share hardware. Update Receive Return Producer shares both data and hardware with Receive Complete, and thus should be placed on the same processor.

The only remaining handler is the least significant, Update Send Consumer. It shares data with DMA Read Complete but shares the DMA write channel with the handlers on the other processor. To resolve this conflict, the Update Send Consumer handler is modified to update only the local copy of the send ring consumer index without transferring it to the main memory using DMA. Instead, the device driver is modified to read this index through memory-mapped I/O. This eliminates sharing of the DMA write channel, thus allowing Update Send Consumer to be colocated with DMA Read Complete.

The only examples of interprocessor sharing resulting from the above partition are the data sharing between DMA Read Complete and Receive Complete and the data sharing between the Mailbox and the Send/Receive Buffer Descriptor Ready handlers; sharing of hardware shown in Table 3.3 is eliminated. The only shared variables are the producer indices for the driver's send buffer descriptor ring and for both copies of the receive buffer descriptor ring. All other variables may be placed in the scratch pads. The sharing pattern of each shared variable is such that only one handler writes it and the reader may see an old value without causing any correctness problems. Consequently, these variables may be shared without mutual exclusion. With the static partitioning described so far, the two processors split the original execution time for this workload by 53% to 47%.

Table 3.4 summarizes this partitioning. Although the partitioning process described above is based on load balance and sharing, the processors are split so that CPU A handles the Mailbox event and events related to receiving into pre-allocated main memory buffers, while CPU B handles all events related to sending or pre-allocating main memory buffers for receive except for the Mailbox event. This partition arises because the send and receive sequences are roughly symmetric, and most sharing of data and hardware resources takes place within a single sequence. Load balancing drives the apparent misallocation of the Mailbox event, leading to two shared variables. This decision stems from the fact that DMA Read Complete requires twice as much processing as DMA Write Complete. DMA Read Complete updates data structures related to send and receive buffer descriptors, and fetching of packet data whereas DMA Write Complete simply interrupts the host CPU when appropriate.

This static partitioning also allows better utilization of the scratch pads for instruction

Processor	Events
CPU A	Mailbox DMA Write Complete Receive Complete Update Receive Return Producer
CPU B	DMA Read Complete Send Buffer Descriptor Ready Send Data Ready Receive Buffer Descriptor Ready Update Send Consumer

Table 3.4 : Static partitioning of events

storage. In particular, the size of the event handlers in the released firmware is roughly 21 KB, which is larger than CPU A's scratch pad (16 KB). However, all of the event handlers can fit in the combined scratch pads (16 KB for CPU A and 8 KB for CPU B), so the static partition eliminates access to longer latency external SRAM for instruction access.

3.3 Evaluation of Parallelization Strategy

The effects of parallelizing network interface firmware is evaluated using 3Com 710024 Gigabit Ethernet interface cards. These cards are based on the Tigon controller and have 1 MB of external SRAM. This section describes the network interface firmware code used in this study, along with the microbenchmarks and macrobenchmarks used to test firmware performance.

3.3.1 Firmware Implementation

Several versions of the Tigon firmware are used to show the effects of parallelization. RELEASE is revision 12.4.13 of the released Tigon firmware, which was made open-source by

the manufacturer and is described in Section 3.1.3. BSD is the firmware that is distributed with the FreeBSD operating system, which is a modified version of revision 12.4.11 of the released Tigon firmware and is only available as object code. PARALLEL is parallelized firmware which statically partitions event handlers and data according to the strategy described in Section 3.2.2. To understand the effects of the scratch pads, PARALLEL has SRAM-CODE, SRAM-DATA, and SRAM-BOTH variants. These variants show the effects of storing all of the code, data, and both code and data in the external SRAM, respectively. For comparison, RELEASE also has an SRAM-BOTH version.

Several Ethernet extensions that can be supported by the Tigon are not considered in performance evaluation. Jumbo frames (which allows frame sizes up to 9000 bytes instead of the standard Ethernet limit of 1518 bytes) and VLAN-tagging (which allows virtually separate local area networks to share the same physical network) are disabled or unsupported in all firmware versions. Furthermore, checksum offloading is also disabled in all firmware versions because of a hardware bug in the Tigon, so the host CPU calculates checksums for all firmware versions. Even though most Gigabit network interfaces allow the host CPU to offload TCP, UDP, and IP checksum computations onto the interface, experiments using fast modern host CPUs (1.8 GHz and above) show no noticeable benefits from checksum offloading.

3.3.2 Benchmarks

The performance of various versions of the Tigon firmware are first evaluated using microbenchmarks. The microbenchmarks generate and receive packets directly in the kernel to isolate the network interface's performance as much as possible from other factors. The

microbenchmarks test UDP send, UDP receive, and simultaneous UDP send/receive for datagram sizes varying from 18 bytes (leading to minimum-sized 64-byte Ethernet frames after accounting for 20 bytes of IP headers, 8 bytes of UDP headers, 14 bytes of Ethernet headers, and 4 bytes of Ethernet CRC) to 1472 bytes (leading to maximum-sized 1518-byte Ethernet frames).

The firmware versions are then tested using macrobenchmarks. The thttpd web server and the Click software IP router are used. thttpd is a light weight and high performance event-driven web server [33]. A modified version of thttpd 2.22beta4 is used. The modifications include the use of the `sendfile` API, support for HTTP pipelined persistent connections, and other generally applicable optimizations. `sendfile` system call implements zero-copy I/O, which reduces data copying between the user and the kernel spaces. The server is accessed by two synthetic clients that replay web traces from Rice University's Computer Science department (CS), a NASA site (NASA), and the 1998 Soccer World Cup site (WC) as fast as the server can handle. The NASA and World Cup traces are made publicly available by Arlitt and Williamson [3]. Click is a modular software IP router implemented as a loadable kernel module [25]. Two client machines replay IP packet traces from Advanced Network Services (ADV), NASA Ames to MAE-West (AIX), and the University of Memphis (MEM). These traces were made available by the National Laboratory for Applied Network Research as a part of the Network Analysis Infrastructure project [22].

The experimental testbed consists of a server and two client machines. The server includes an Athlon 2600+ CPU, 2GB DDR SDRAM, a 64bit/66 Mhz PCI bus, and a 40GB IDE disk (none of the workloads are disk-intensive). Each of the client machines has an

Athlon 2200+ CPU, 512MB of DDR SDRAM, a 40GB IDE disk, a 64bit/66MHz PCI bus, and an Intel PRO/1000 MT Server Gigabit Ethernet Adapter. The server has one 3Com 710024 NIC for the microbenchmarks and thttpd and has two NICs for Click. The machines are connected through one or more Gigabit switches on isolated networks, as appropriate. In the microbenchmarks, one client machine is used to receive packets from the Tigon-based NIC being measured in the server, and the other client machine is used to send packets to it.

3.4 Experimental Results

As mentioned earlier, the parallelized versions of the Tigon firmware may increase the utilization of the scratch pads. This section first examines the performance characteristics of the scratch pads and the external SRAM, and their implications on the multiprocessor performance. Then, the experimental results from the microbenchmarks and macrobenchmarks are analyzed to evaluate the parallelization strategy.

3.4.1 Architectural Impact on Multiprocessor Performance

To measure the architectural impact on multiprocessor performance, a benchmark based on the LMbench benchmark suite is implemented in the Tigon [23]. It measures the instruction throughput of a processor in the Tigon using sequences of addition instructions, the load latency of the scratch pad and external SRAM using dependent pointer-chasing loads to fully expose memory latencies, and the load and store throughput of the scratch pad and external SRAM using independent stride-1 accesses to allow maximum memory pipelining. Store latency is not measured since stores return no value to the processor, making it

	CPU A Only		CPU B Only		CPU A and B Together			
	Code Storage		Code Storage		Code Storage			
	SRAM	Scratch	SRAM	Scratch	SRAM		Scratch	
CPU	CPU A	CPU A	CPU B	CPU B	CPU A	CPU B	CPU A	CPU B
Million Instructions/second	65.12	85.91	65.36	85.91	65.07	65.07	85.91	85.91
Load Latency (nanosecond)								
Dest: SRAM	81.90	81.63	81.90	81.63	82.24	82.24	81.63	81.63
Dest: Scratch Pad	37.79	46.75	37.91	35.11	36.62	36.62	46.75	35.11
Load Bandwidth (MB/s)								
Dest: SRAM	55.79	56.64	55.76	56.68	54.88	54.88	56.68	56.68
Dest: Scratch Pad	144.84	110.24	144.84	162.28	145.76	145.76	110.24	162.28
Store Bandwidth (MB/s)								
Dest: SRAM	132.56	169.88	132.80	170.28	87.76	87.76	170.28	170.28
Dest: Scratch Pad	230.12	162.28	228.56	162.24	225.56	225.56	162.28	162.28

Table 3.5 : Instruction throughput and memory performance when the benchmark code is stored in the external SRAM or the scratch pads (indicated by SRAM and Scratch in the table). Dest: SRAM and Dest: Scratch Pad indicate the destination of load and store instructions. The CPU row shows which processor executes the benchmark code. The left portion of the table shows the results when the benchmark runs only on CPU A. The middle portion of the table shows the results when the benchmark runs only on CPU B. The right portion of the table shows the results when the benchmark runs concurrently on both CPU A and CPU B.

impossible to form a sequence of dependent stores.

Table 3.5 shows instruction throughput and memory performance of the Tigon. The left two columns of the table show the results when the benchmark code runs on CPU A, and CPU B is halted. Storing the benchmark code in the scratch pad increases the instruction throughput from 65 million instructions per second to 86 million instructions per second, closely matching the clock rate 88 MHz of Tigon processors by virtue of avoiding several pipeline stalls on each instruction cache miss to external SRAM (one miss per 16 instructions since the instruction cache is 64 bytes). The memory bus of the Tigon is 64-bit wide and runs at 50 MHz. The external SRAM has the maximum bandwidth of 800 MB/s. The load latency of the external SRAM is about 82 nanoseconds or about 7 processor cycles (the processors run at 88 MHz). Thus each miss per 16 instructions results in the processor

utilization of roughly 70% (16 cycles/23 cycles) or 62 million instructions/s.

The scratch pad provides a lower latency than the external SRAM. However, the load latency of the scratch pad is about 38 nanoseconds or 3 processor cycles when the benchmark code is stored in the external SRAM. From the instruction throughputs, it is certain that fetching an instruction from the scratch pads takes one cycle. Since the benchmark executes dependent pointer-chasing loads, it seems that the Tigon processors do not implement full bypassing. When the benchmark code is stored in the scratch pad, the load latency increases by one processor cycle, leading to a latency of 47 nanoseconds. Because the scratch pad now serves both instructions and data, instruction fetch and load operations seem to compete for the access to the scratch pad. The one cycle increase of the load latency indicates that the processor favors instruction fetch over load.

The scratch pad provides roughly twice the load bandwidth of the external SRAM. The load bandwidth of the external SRAM indicates that loads are not pipelined. Since the SRAM operates at 50 MHz, it should be able to provide 200 MB/s if all loads are pipelined. Instead, 56 MB/s of load bandwidth implies that all independent loads to the external SRAM pay roughly 7 processor cycle latency (88 MHz/7 * 4B = 50 MB/s). Similarly, the load bandwidth of the scratch pad indicates that independent loads to the scratch pad are not pipelined, and each load pays the full latency (3 processor cycles when the benchmark code is stored in the external SRAM and 4 processor cycles when the code is stored in the scratch pad).

The store bandwidth of the scratch pad is roughly twice the store bandwidth of external SRAM only when the code is stored in the SRAM. The store bandwidth of the scratch pad when the benchmark is stored in the external SRAM is about 230 MB/s. Since the instruc-

tion throughput in this case is 65 million instruction/s, the processor seems to pipeline most independent store instructions (65 million stores/s * 4B = 260MB/s). When the benchmark code is stored in the scratch pad, the store bandwidth of the scratch pad drops to about 162 MB/s. As with the load latency, it appears that instruction fetch and store compete for the access to the scratch pad. From the instrcution throughput (86 million instructions/s), the fully-pipelined store bandwidth should be 344 MB/s. However, instruction fetch also occurs every cycle. Since 162 MB/s is roughly half of 344 MB/s, it seems that instruction fetch and store altenate each cycle.

The store bandwidth of the external SRAM is only about 133 MB/s when the benchmark code is stored in the external SRAM. Since the memory bus operates at 50 MHz, the maximum store bandwidth is 200 MB/s. However, the processor utilization is roughly 70% (due to one miss per 16 instructions), so the store bandwidth of the external SRAM is theoretically 140 Mb/s close to the actual bandwidth of 133 MB/s. The store bandwidth of the external SRAM when the benchmark code is stored in the scratch pad is about 170 MB/s, 85% of the fully-pipelined store bandwidth of 200 MB/s. These store bandwidths indicate that stores are underpipelined.

The middle two columns of Table 3.5 show the benchmark results when the benchmark runs only on CPU B. Except the scratch pad performance, the performance characteristics are roughly same as those when the benchmark runs only on CPU A. When the benchmark runs on CPU B, storing the benchmark code in the scratch pad does not increase the load latency of the scratch pad (38 nanoseconds versus 35 nanoseconds). Recall that when the benchmark runs on CPU A, storing the code in the scratch pad increases the load latency of the scratch pad. Similarly, the load bandwidth of the scratch pad slightly improves when

the code is stored in the scratch pad (145 MB/s versus. 162 MB/s). These results indicate that CPU B's scratch pad is faster than CPU A's scratch pad, likely due to its smaller size (8 KB versus 16 KB). The CPU B's scratch pad may have dual read ports. However, the store bandwidth of CPU B's scratch pad is same as that of CPU A's scratch pad. This store bandwidth indicates that CPU B's scratch pad only provides faster read accesses.

The memory performance and instruction throughput of each processor remain roughly same even when the benchmark code runs concurrently on both CPU A and CPU B. The right four columns of Table 3.5 show instruction throughput and memory performance when both processors execute the benchmark code. The instruction throughput, load latency, and load bandwidth of each processor are the same as those when the benchmark runs on the respective processor alone, indicating no interference between the processors. Theoretically, a parallel version of the Tigon firmware should be able to achieve double the performance of a sequential version. When the benchmark code is stored in the scratch pads, each processor gets the same store bandwidth as it would when only one processor executes the benchmark. However, when the benchmark code is stored in external SRAM, the store bandwidth of the SRAM for each processor is about 88 MB/s, 45 MB/s lower than the store bandwidth of the SRAM when only CPU A runs the code (133 MB/s). These store bandwidths of the external SRAM when the code is also stored in the SRAM indicate that either the memory bus or the SRAM experiences contention.

The above results suggest that the scratch pad is useful in most cases both for improving performance and for reducing contention related to dual processor accesses. These observations tend to support the PARALLEL parallelization, as it stores all event handlers and most data in the scratch pad.

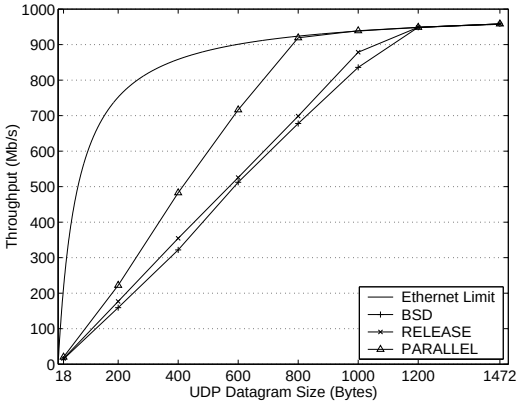

Figure 3.6 : UDP send throughputs achieved by different firmware versions with various datagram sizes.

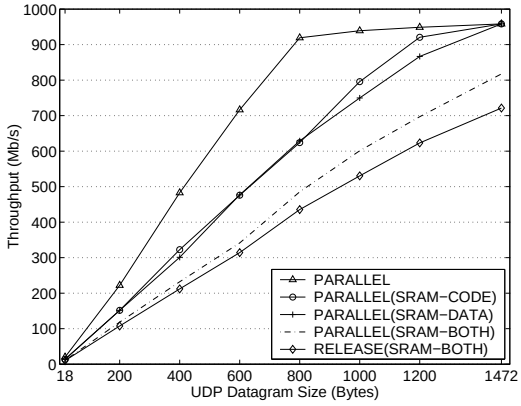

Figure 3.7 : Differences in UDP send throughputs due to the use of the scratch pads.

3.4.2 Throughput Improvements

Microbenchmark results show that parallelized firmware improves both unidirectional throughput (UDP send and receive) and bidirectional throughput. The improvements result not only from the greater computational power enabled by utilizing both processors, but also from the better utilization of hardware resources; an example of the latter is the increased utilization of the scratch pads, which is a by-product of the parallelization.

Figure 3.6 shows the UDP send throughputs of different versions of the firmware. The X axis shows UDP datagram sizes varying from 18 bytes (leading to minimum-sized 64-byte Ethernet frames) to 1472 bytes (leading to maximum-sized 1518-byte frames). The Y axis shows throughput in Mb/s of UDP datagrams, excluding network headers. The Ethernet Limit curve represents the theoretical maximum data throughput of the UDP/IP protocol running on Ethernet for a given datagram size; protocol overheads, including headers and required inter-frame gaps, prevent the full utilization of 1 Gb/s for data.

PARALLEL outperforms both BSD and RELEASE across all datagram sizes and delivers up to 32% more throughput than both (with 800-byte datagrams). As the datagram size decreases from the maximum 1472 bytes, throughput diverges from the Ethernet limit and starts decreasing linearly starting at 1000-byte datagrams for BSD and RELEASE and 800-byte datagrams for PARALLEL. The packet processing rate in packets per second corresponds to the slope of the throughput curve. The linear decrease indicates that the firmware handles a constant rate of packets regardless of the datagram size, and that the processors in the Tigon are saturated.

As discussed in Section 3.2.2, PARALLEL utilizes both processors to send packets and increases scratch pad efficiency by storing all event handlers in the scratch pads. Figure 3.7 shows the contributions from the parallelization and increased utilization of the scratch pads by comparing the base PARALLEL with the SRAM-CODE, SRAM-DATA, and SRAM-BOTH versions, and with the SRAM-BOTH version of RELEASE. The throughput difference between RELEASE(SRAM-BOTH) and PARALLEL(SRAM-BOTH) shows that the improvement due to parallelization alone is at most 13% for this workload, because only one of the send events (Mailbox) is processed by CPU A. PARALLEL delivers up to 89% more throughput than PARALLEL(SRAM-BOTH) with 800-byte datagrams, showing that increased utilization of the scratch pads contributes most to the improvement of the parallelized firmware over the sequential firmware for this workload.

The parallelized firmware also improves UDP receive throughput. Figure 3.8 shows the UDP receive throughputs achieved by various versions of the firmware. PARALLEL achieves up to 19% more throughput than BSD, which in turn outperforms RELEASE for this workload. As datagram size decreases, the throughput of BSD falls below the Ether-

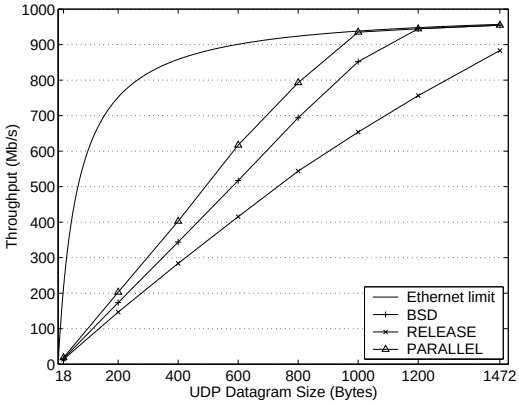

Figure 3.8 : UDP receive throughputs achieved by different firmware versions with various datagram sizes.

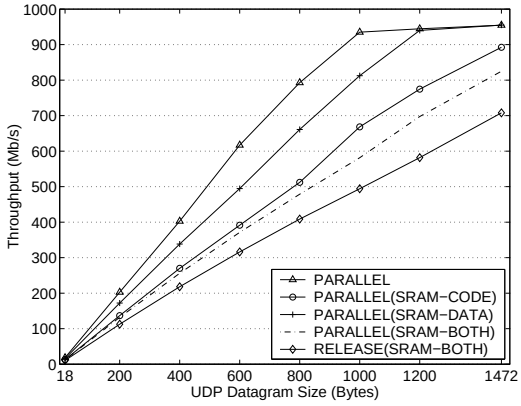

Figure 3.9 : Differences in UDP receive throughputs due to the use of the scratch pads.

net limit for datagrams less than 1200 bytes, indicating that the processor is saturated by packet processing. In contrast, PARALLEL does not fall below the limit until the datagram size falls below 1000 bytes. The UDP receive throughput of RELEASE never reaches the Ethernet limit.

The improvement of PARALLEL again comes from both parallelization and increased utilization of the scratch pads. Figure 3.9 shows throughput increases due to the parallelization and increased utilization of the scratch pads. Receive benefits from the parallelization since CPU B handles the Receive Buffer Descriptor Ready and DMA Read Complete events that are used to inform the NIC about main memory buffers pre-allocated by the device driver. PARALLEL(SRAM-BOTH) provides roughly 18% greater throughput than RELEASE(SRAM-BOTH), showing the improvement due to the parallelization alone. PARALLEL delivers up to 66% more throughput than PARALLEL(SRAM-BOTH) indicating that, like UDP send, most of the improvement of PARALLEL for this workload

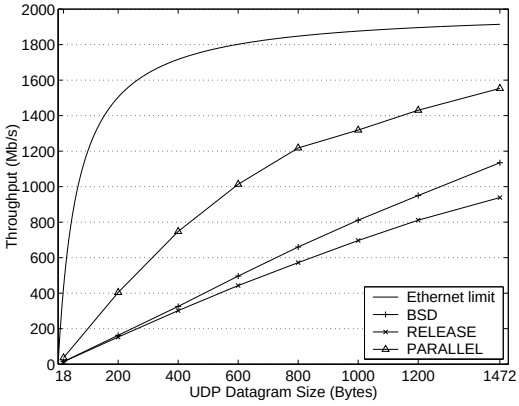

Figure 3.10 : UDP bidirectional throughputs achieved by different firmware versions with various datagram sizes.

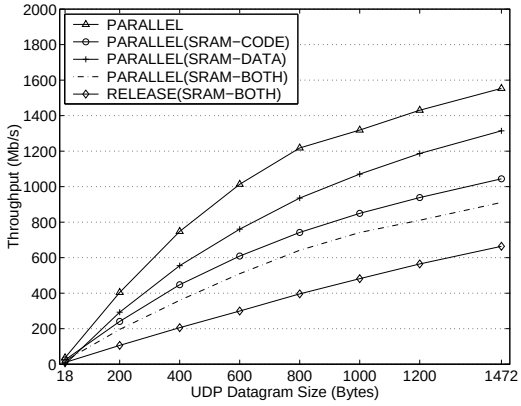

Figure 3.11 : Differences in UDP bidirectional throughputs due to the use of the scratch pads.

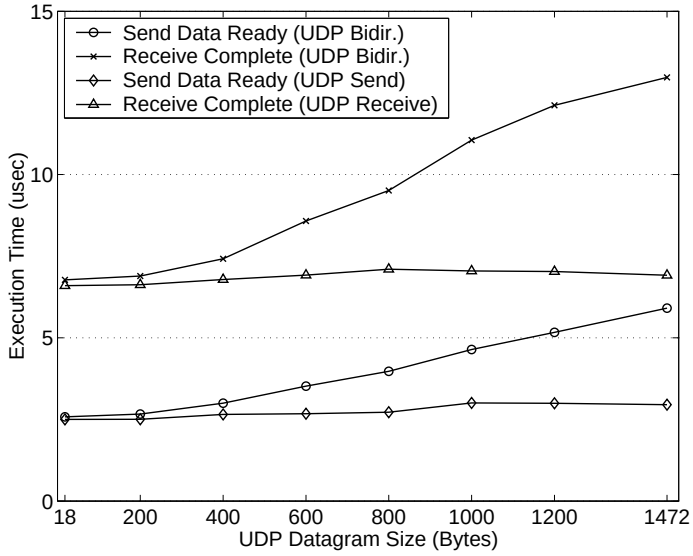

Figure 3.12 : Execution time in microseconds of each invocation of the event handlers for Send Data Ready and Receive Complete in PARALLEL firmware. Benchmarks used while measuring execution times are shown in parenthesis.

is again due to the increased utilization of the scratch pads.

The unidirectional UDP workloads suggest that the parallelized firmware benefits primarily from better scratch pad utilization. These workloads see few benefits from utilizing

both processors because the static event partition only targets load balance for bidirectional traffic that equally emphasizes send and receive. Figure 3.10 shows the bidirectional UDP throughputs achieved by different versions of the firmware. The Ethernet limit is now doubled since the network links are full-duplex. Compared to RELEASE, PARALLEL increases throughput by 65% with 1472-byte datagrams and 157% with 18-byte datagrams. Compared to BSD, PARALLEL increases throughput by 37% with 1472-byte datagrams and 146% with 18-byte datagrams. For datagram sizes less than 600 bytes, PARALLEL delivers over 100% more throughput than either BSD or RELEASE.

The sublinear increase of throughput achieved by PARALLEL starting at 600-byte datagram size and 1 Gb/s of achieved data throughput indicates that the packet processing rate is starting to decrease for larger datagram sizes. Since the firmware includes no size-dependent overheads, this decrease indicates that hardware resources that see per-byte overheads (such as the DMA and MAC controllers, the PCI and Ethernet interfaces, or the external SRAM) are seeing increased contention as they attempt to serve over 1 Gb/s of network data throughput. Figure 3.12 shows execution time of Send Data Ready and Receive Complete events in PARALLEL firmware. Measurement is taken for those event occurrences that result in a particular execution path in the event handler for Send Data Ready or Receive Complete. Thus each invocation of Send Data Ready or Receive Complete event handler does same amount of processing. The Y axis shows execution time in microseconds. As in other figures, the X axis shows UDP datagram sizes. With the UDP send or receive tests, the execution times remain constant. With UDP bidirectional test, as UDP datagram size increases, execution times of both events start increasing noticably at 400 byte datagram size. 400 byte datagram size coincides with the start of the sublinear

		BSD	RELEASE	PARALLEL	Intel
thttpd (Mb/s)	CS	659	616	932	925
	NASA	689	636	944	942
	WC	587	550	702	870
Click (Thousand pkt/s)	ADV	134	113	150	166
	AIX	133	115	173	198
	MEM	133	119	247	276

Table 3.6 : Throughput achieved by the thttpd web server and Click software IP router with the Tigon running various firmware versions and with a nonprogrammable Intel PRO/1000 MT Server NIC.

increase in throughputs shown in Figure 3.10. The event handlers for both events have no size-dependent processing and do not wait for any other hardware resources than memory. The ratio of the number of event occurrences for which the executime time is recorded to the total number of event occurrences also remains constant across UDP datagram sizes. Thus, the increases in execution time of Send Data Ready and Receive Complete indicate that accesses to the external SRAM on the Tigon experiences contention when it serves over 1Gb/s throughput, and the processors waste cycles due to increasing memory access latency. Since control data structures such as buffer descriptor rings are stored in the external SRAM, accesses to the external SRAM are unavoidable.

Figure 3.11 shows that the performance gains of PARALLEL for UDP bidirectional traffic stem from both parallelization and increased utilization of the scratch pads. Parallelization increases throughput by 37–95%, far greater than 13–18% for the unidirectional workloads. The scratch pads further provide up to 70% increase in throughput.

The macrobenchmarks also benefit from the parallelized firmware. Table 3.6 shows the HTTP content throughput in Mb/s achieved by the thttpd web server and the packet routing throughput in thousands of packets per second achieved by the Click software IP

router. For comparison, the final column shows the performance of one of the fastest non-programmable NICs, the Intel PRO/1000 MT Server Adapter. The PARALLEL firmware increases HTTP content throughput by 20–41% over BSD and by 48–73% over RELEASE. The Tigon running PARALLEL performs slightly better than the Intel NIC for the CS and NASA traces and only 20% worse for the WC trace. Similarly, PARALLEL increases packet routing throughput by 12–86% over BSD and by 32–107% over RELEASE. The Click throughput achieved by the Tigon running PARALLEL is only 10–13% lower than that of the Intel NIC.

These results indicate that parallelizing the network interface firmware enables a programmable NIC to achieve throughput competitive with modern ASIC-based NICs for real network services. Combined with the underlying potential to extend the functionality of network services, parallelization thus makes programmability a viable and attractive option for high-performance network interface design.

3.5 Modern Programmable Network Interfaces

As network link speeds scale to 10 Gb/s and beyond, future programmable network interfaces will require substantially greater processing power than existing interfaces in order to deliver link speed throughput. Future programmable network interfaces are likely to operate at a higher clock rate than 88 MHz of the Tigon. The Tigon controller was released in 1997. Its core runs at 88 MHz and contains 24 KB of on-chip memory. Today's (2003) process technology would allow the same architecture to improve performance by simply employing a faster clock speed and a greater capacity of the on-chip memory. In 2001, Broad-

com released programmable Gigabit Ethernet controllers that are based on the Tigon. These newer controllers include two RISC processors, 32 KB scratch pad memory, and on-chip 96 KB buffer packet buffer memory in addition to optional external SRAM. Although neither the chip specification nor firmware source code is available publicly, the processors are likely to run at a higher clock rate than 88 MHz given that other programmable controllers operate at a higher clock rate. For instance, LANai 9 programmable controllers released in 2001 that are used in various Myrinet/PCI interfaces can operate at up to 200 MHz.

Although technology trends would help improve performance of programmable NICs, embedded processors cannot scale in frequency or complexity as aggressively as general-purpose processors due to technology constraints such as limited power and cooling area available to a PCI card. Thus, programmable network interfaces must rely on multiprocessor architectures to target exponentially increasing network link speeds.

3.6 Scalability Issues

As mentioned in the previous section, technology constraints would force future programmable network interfaces to rely on multiprocessor architectures to target network link speeds as they scale to 10 Gb/s and beyond. However, such rapidly growing link speeds will likely require more than just two processors, making it difficult to achieve high performance using only statically-exposed task-level concurrency.

The static partition of event handlers described in Section 3.2 effectively balanced the workload across two processors. However, a greater number of processors makes it more difficult to balance load in the same fashion, given that the event handlers have different

processing requirements. Instead, the firmware may dynamically choose the processor to handle a given event. While this may improve load balance, it creates additional overheads by requiring mutual exclusion to prevent simultaneous execution of an event handler or simultaneous access to data and resources shared across event handlers that may now run on any processor. Similarly, dynamic partitioning also degrades locality for shared variables, private variables that persist across handler invocations, and event handler code, since any processor may access these data or code regions in the future. The specific implications for the Tigon are contention for the single hardware semaphore and less efficient utilization of the scratch pads, likely offsetting the performance benefits of load balancing. However, other architectures may see greater benefits from dynamic partitioning.

The parallelization scheme discussed in this paper exploits task-level concurrency in which the unit of concurrency is an event handler. Thus, the granularity of load balancing is limited by the distribution of processing times for the execution of an individual handler, even with dynamic partitioning. For instance, a distribution that is greatly skewed by some event handlers that require significantly more processing than others leads to a poor balance of load. To mitigate this imbalance, the firmware may parallelize the execution of the most demanding handlers. One approach to reducing the granularity of concurrency in this fashion would be to exploit parallelism across independent packets, allowing a single handler to process unrelated packets simultaneously on multiple processors. As with dynamic partitioning, parallelization of individual handlers may introduce further synchronization overheads due to sharing of variables and hardware resources.

Static partitioning, dynamic partitioning, and parallelizing particular event handlers all have advantages and disadvantages as discussed above. With more than two processors

available, an effective parallelization would benefit from the use of all three approaches as they complement each other. More advanced hardware can also provide features to address some of the problems caused by each approach to parallelization.

For instance, hardware can help reduce synchronization requirement by providing a set of DMA channels for each processor and multiple locks. In the Tigon, sharing of the DMA channels is not related to data sharing. Rather, the sharing is an artifact of the limited number of channels available in the Tigon. If the Tigon had provided three logically separate DMA read channels, each for send buffer descriptors, receive buffer descriptors, and packet data, then the firmware can use one channel for each type of data obviating the need of synchronization regardless of the partition of the events. Multiple locks allow fine-grained locking without introducing contention on a single hardware lock.

A hardware support for atomic manipulation of event registers can facilitate dynamic partitioning scheme. In dynamic partitioning, processors must ensure that an event is not processed multiple times by different processors, and no event is lost. Since events occur frequently, the use of a global lock to synchronize accesses to event registers would degrade performance.

Finally, intelligent hardware-managed queues would help multiple processors exploit packet-level parallelism. To exploit packet-level parallelism, multiple processors can handle different packets simultaneously. However, the processors need to synchronize the actual transmission of the packets so that the packets are transmitted in the order that the host processor initially intended. Otherwise, the arbitrary ordering of packets at the network interface would degrade the performance of reliable network protocols such as TCP that are sensitive to the order of packet arrivals, even though unreliable network links such as Eth-

ernet do not guarantee the ordering of packets. To reduce synchronization, hardware could provide MAC queues that accept requests of packet transmissions in an arbitrary order but transmits the packets in the order specified by the device driver. With these queues, processors would be able to handle packets and schedule transmissions without synchronization operations to enforce a particular order. Similar DMA queues would furthur reduce interprocessor communication and synchronization that are required to enforce ordering of packets.

3.7 Summary

The performance disadvantage of programmable network interfaces that result from instruction processing overheads can be alleviated by exploiting task-level concurrency in network interface processing. This chapter introduces and analyzes a parallelization strategy for the Tigon firmware. The handlers that process various components of network interface processing are carefully partitioned such that the processors on the Tigon achieve load balance with minimal sharing and no synchronization. The parallelization of firmware increases throughput by 65% for bidirectional UDP traffic of maximum-sized packets, 157% for bidirectional UDP traffic of minimum-sized packets, and 32–107% for real network services. This parallelization enables performance within 10–20% of a modern ASIC-based network interface for real network services. Given that the Tigon was manufactured using the process technology available in 1997, the current process technology would enable the same architecture to run at a faster clock rate and employ greater on-chip memory. With greater processing capacity, a similar multiprocessor architecture running parallelized

firmware would perform much better than modern ASIC-based network interfaces.

Future Ethernet speeds are expected to continue growing; experimental controllers already implement a 10 Gigabit Ethernet specification. Although clock speeds of programmable network interface controllers will grow above the 88 MHz rate of the Tigon, the clock rate is unlikely to grow at the same unrestricted pace as general-purpose CPUs because of the limited power and cooling area available to any I/O device. For instance, the maximum power consumption of any PCI board is limited to 25 watts [30], whereas recent Pentium 4 processors can consume over 80 watts [18]. This frequency constraint would eliminate the possibility of uniprocessor programmable NICs supporting future Ethernet wire speeds. However, exploiting concurrency in network interface processing proves effective. Thus using multiprocessor architectures, programmable network interfaces will be able to serve as a platform for high performance network interfaces that provide extended network services in order to improve network server performance.

Chapter 4

Network Interface Data Caching

The previous chapter shows that with proper architectures and software programmable network interfaces can overcome the performance disadvantage that result from instruction processing. Through the flexibility of programmability, programmable network interfaces allow easy implementation of various services that can improve networking server performance. A few previous projects have implemented extended network servies on programmable network interfaces as they will be discussed in Chapter 5. These projects primarily focus on improving general networking performance, both send and receive, by offloading some of network processing onto the network interface. This chapter proposes a new service, network interface data caching, that directly targets networking server performance by exploiting programmable network interfaces for both their computation and storage features.

Network interface data caching is a technique to allow frequently transmitted data to be cached on the network interface. The network interface then sends cached data directly from the interface without fetching them from the main memory. Thus by eliminating repeated transfers, this technique reduces local interconnect traffic for application workloads such as web servers that exhibit locality in stream of transmitted data. Consequently, when a networking server experiences significant contention on local interconnect, reducing interconnect traffic can lead to improvement in server performance.

The data cache on the network interface resides in on-board DRAM and is managed entirely in software, requiring no additional hardware for cache management. The operating system running on the host processor keeps track of the content in the cache and issues commands to the network interface to control the cache. Thus the operating system decides which data to store in the cache and for which packets it should use cached data. The network interface then acts as a slave to the operating system taking appropriate actions on the cache. Cache contents may be appended to packet-level and application-level headers generated by the host processor and then sent over the network.

Since web servers are one of the most common networking servers, network interface data caching is evaluated in the context of web servers. This chapter first examines the use of local interconnect in web servers and evaluates performance impact of networking interface data caching on their performance.

4.1 Background and Related Work

Web servers utilize various system components to service user requests, including the network interface, local interconnect, main memory, and host processor. Most previous research on improving web server performace addresses the efficiency of the host processor and memory utilization through various techniques such as zero-copy I/O and scalable event notification mechanisms [4, 14]. These developments in software and advanced hardware has improved web server performance to the point that contention on the local interconnect may become a bottleneck.

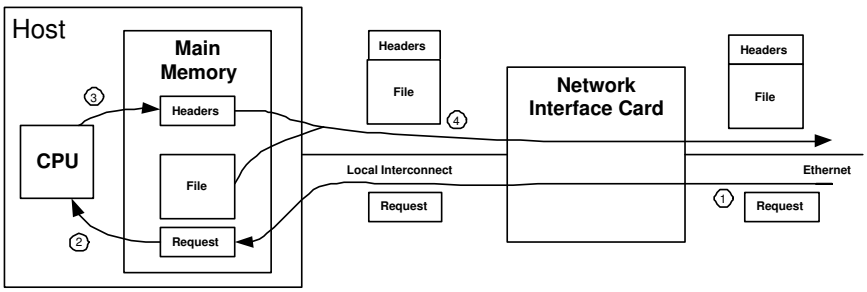

Figure 4.1 : Steps in processing an HTTP request to a file that currently resides in the OS file cache. Step 1: the HTTP request is transferred from the Ethernet through the network interface to the main memory. Step 2: the kernel reads the request, processes it, and passes it on the web server. Step 3: the web server generates HTTP headers and tells the kernel which file to send. The kernel generates TCP, IP, and Ethernet headers, and finds the file in its file cache (if the file is not in the file cache, the file system initiates a disk read). Step 4: the headers and file are transferred to the Ethernet through the network interface.

4.1.1 Anatomy of a Web Request

To illustrate the performance issues in web servers, this section considers the flow of a request and response through a system that supports zero-copy I/O and includes a network interface that supports checksum offloading. Operating systems that support zero-copy I/O use special APIs or memory management schemes to avoid copying data between the kernel and user space of main memory or between different subsystems of the kernel (such as the file cache and network buffers) [14, 26, 29]. As described in Section 2.2, network interfaces that support checksum offloading reduce load on the host CPU by directly calculating IP header checksums and TCP packet checksums so that the operating system running on the host CPU need not compute checksums. Both techniques aim to reduce load on the CPU and memory bandwidth utilization by eliminating expensive memory accesses.

Figure 4.1 shows the steps taken by a typical web server to process an HTTP request for a static page and produce a response, assuming the requested file resides in the file

cache. In step 1, the HTTP request packets arrive on the Ethernet link connected to this server's network interface. The network interface card initiates a DMA across the local interconnect, through which the device takes control of the interconnect and writes data into the system's main memory. Reception of the HTTP request packets follows the steps of receive processing described in Section 2.3. In step 2, the CPU reads the request packet data from the main memory so that the TCP/IP stack of the kernel and the application-level HTTP server can take appropriate actions based on the request. In step 3, the application responds by creating a set of HTTP headers corresponding to the requested file and writing them into the main memory. The application then initiates a TCP/IP transmission of the file through a system call. On a file cache hit, the kernel passes a reference to the data in the file cache buffers to the TCP/IP stack. (On a miss, the file system would first read the file from disk.) The TCP/IP stack then writes the protocol headers to the main memory and has the device driver alert the NIC of a new transmission. Transmission of the HTTP response packets follows the steps of send processing described in Section 2.3. In step 4, the NIC initiates DMA transfers of the TCP/IP headers, HTTP headers, and HTTP content from the system memory to the network interface buffers. Finally, the network interface calculates checksums for each packet and sends the data out onto the network.

4.1.2 Local Interconnect

As described in the previous section, the web server must transfer every response across the local interconnect, which can lead to inefficient use of the interconnect if a large fraction of transfers are repeated transfers. Moreover a PCI bus, which is used in most systems, suffers from overheads associated with data transfers. Consequently, inefficient use of the

PCI bus exacerbates the overheads.

Transferring data over a PCI bus involves several phases, and each phase may incur certain overhead [30]. The first phase is arbitration in which the device that wishes to transfer data to or from main memory must acquire the PCI bus. If the bus is already acquired by another device, the device must wait until the access is granted. Once the device acquires the bus, it sends memory address to main memory (addressing phase). Modern main memory built using DRAM spends cycles preparing for a pending transfer and cannot start transferring data immediately upon request. During these memory stall cycles, the device must wait. Finally, actual data transfer begins after main memory becomes ready (transfer phase). Even during actual data transfer, either the main memory or the device may need to abort the transfer. For instance, the device may run out of buffer or main memory may need to service urgent requests from host processor. In case of an abort, the device needs to restart the entire transfer in future.

A standard 64-bit/33Mhz PCI bus provides a peak bandwidth of 2 Gb/s. Since a web server typically only utilizes the half-duplex bandwidth of 1 Gb/s, the volume of outbound traffic (HTTP responses) dominates that of incoming traffic (HTTP requests). A web server as described in Figure 4.1 can theoretically use a 64-bit/33Mhz PCI bus to support two Gigabit Ethernet NICs at full transmit bandwidth. However, the PCI bus cannot deliver this theoretical maximum throughput due to overheads associated with data transfers. In fact, the PCI bus in a web server can lose over 30% of bandwidth due to the overheads. Thus, the PCI bus can become a performance limiter as the server begins to utilize a significant fraction of the Ethernet bandwidth.

4.1.3 Related Work

A number of studies developed techniques to improve web server performance by reducing processing requirement on the host processor. Examples of techniques developed to improve processor efficiency include zero-copy I/O, scalable event notification mechanisms, persistent connections, and asymmetric event-driven systems. Using these techniques, modern web servers have the ability to deliver over gigabit/s rate of content throughput, which puts substantial stress on local interconnect bandwidth.

As described previously, zero-copy I/O avoids data copying between the user and kernel spaces or subsystems of the operating system through special APIs or memory management mechanisms. Zero-copy I/O aims to reduce load on processor by eliminating expensive memory accesses. Druschel and Peterson developed a cross-domain transfer facility called *fast buffers* (fbufs) [14]. fbufs allows zero-copy data transfers across different protection domains by exploiting immutable buffers. The fbufs are primarily for network subsystem and their experimental results indicate increases in UDP/IP throughput. Nahum et al. studied performance issues in web servers using several web servers with various operating system optimizations [26]. In particular, they emphasize importance of zero-copy I/O support in web servers. They report that a zero-copy `sendfile` system call implemented in IBM AIX workstations increased web server throughput by up to 51 %. Pai et al. developed a unified I/O buffering and caching system called IO-Lite [29]. Like fbufs, IO-Lite is based on immutable buffers and provides zero-copy I/O across protection domains. Unlike fbufs, IO-Lite provides buffering and caching across applications and all subsystems of operating system including network, filesystem, file cache, and interprocess communication

subsystems. Their prototype web server using the IO-Lite shows 40% increase in throughput. Network interface data caching essentially extends zero-copy I/O to the final crossing in the server, the main memory and the NIC, by caching data at the NIC.

While zero-copy I/O support in the operating system reduces expensive memory accesses, scalable event notification mechanism and asymmetric multi-process event-driven system allow host processor to efficiently handle a large number of concurrent user requests. Banga et al. analyzed the scalability problem in `select` system call and developed an event-delivery mechanism that scale with the number of file descriptors [4]. Typically applications use `select` system call as event-notification mechanism. The authors identify that `select` is not event-based but rather state-based, which requires the kernel to maintain the current state of each file descriptor. The new event-delivery mechanism is strictly event-based allowing applications to dequeue new events rather than acquiring the current state of every file descriptors. The authors show a proxy server using the new event-delivery mechanism scales far better than the same proxy using `select`. Lemon designed and implemented Kqueue on the FreeBSD operating system[21]. Kqueue is an event notification facility similar to the one developed by Banga. Kqueue supports notification of events for asynchronous I/O operations as well as for regular file descriptors. It also supports events to report changes in file states. Using Kqueue, an application would first register events that it wishes to monitor. It then can collect new events without having to re-register the events. The author uses a modified version of the thttpd web server that uses Kqueue and shows that Kqueue is able to sustain relatively constant server response time as the number of idle connections increases.

Pai et al. [28] introduced asymmetric multi-process event-driven (AMPED) system in

which a server utilizes both event-driven loop and multiple threads (processes). Traditional event-driven systems consist of a single thread. However, any blocking operation such as disk accesses can stall the system. Multi-threaded systems overcome stalling due to blocking operations by using multiple threads. While one thread blocks, other threads can continue executing. Multi-threaded systems however suffer from overheads associated with maintaining threads. The AMPED system takes advantages of both paradigms. It normally runs as fast event-driven but spawns threads to handle potential blocking operations. The authors evaluate AMPED using a prototype web server named Flash and show that AMPED Flash outperforms both event-driven and multi-threaded systems.

The HTTP standard itself introduced a feature to improve web server performance. The HTTP persistent connections allow multiple content transfers per connection [15]. With non-persistent connections, each HTTP connection can only service one user request. By reusing connections, persistent connections amortize overheads associated with TCP connection setup/teardown, thereby reducing the load on server processor.

The techniques discussed above improve server performance by reducing load on the host processor. Recently, Yocum and Chase proposed payload caching for network intermediaries (e.g., firewalls and routers) [39]. A payload cache stores incoming packet payload data directly in the network interface to avoid sending such content back to the network interface for retransmission, while imposing few constraints on the NIC processor and using the host processor to generate headers. Payload caching uses storage on the NIC as a short-term holding buffer and associates that storage with information specific to a packet in transit. Although payload caching reduces routing latency by eliminating the need to transfer the same packet payload twice across the local interconnect, the technique

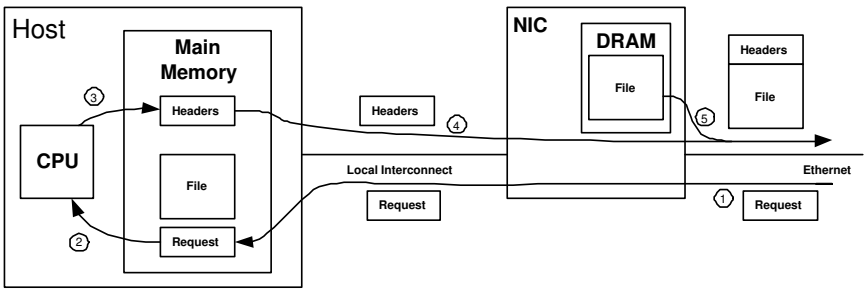

Figure 4.2 : Steps in processing an HTTP request to a file that currently resides in the network interface data cache. Steps 1–3 are unchanged from the original system, shown in Figure 4.1. In step 4, after the headers are generated, they are sent to the network interface without the file. The network interface then retrieves the file from its cache and transfers the headers and file out onto the Ethernet in step 5.

has severely limited applicability. Because packet payloads are cached on the network interface that receives packets, network intermidiaries such as routers only benefit from payload caching if those received packets are routed out to the same network interface. In most cases, the primary task of network intermediaries is to route packets arriving from one interface to different interfaces. Thus payload caching would help network intermediaries only if network interfaces consist of multiple ports, each of which connects to different networks. Network interface data caching adapts the payload caching concept to the server domain by caching content that originates locally rather than caching incoming network traffic.

4.2 A Network Interface Data Cache

As web server throughput improves, contention on the local interconnect increases and becomes a bottlenck in the system. Local interconnect traffic can be reduced by eliminating repeated transfers. Reducing interconnect traffic in turn reduces contention on the inter-

connect and can improve server throughput. Adding a data cache directly on the network interface allows the interface to capture repeatedly transferred files. By storing frequently requested files in this cache, the server will not need to send those files across the interconnect for each request. Rather, the server can simply generate the appropriate protocol and application headers, and the network interface card can combine those headers and the file data to be sent out over the network. Referring back to Figure 4.1, the web server normally transfers requested files across the local interconnect to the network interface (step 4 in the figure). Storing copies of files in a cache on the network interface eliminates this final transfer of file data from the system memory to the local memory on the network interface, reducing the bandwidth demands on both the local interconnect and main memory.

Figure 4.2 shows the stages in processing a web request in a system with a network interface data cache. As in Figure 4.1, requests arrive on the Ethernet, and the NIC transfers them to the main memory. The host CPU reads the request and generates the appropriate headers (steps 1–3). In step 4, however, if the kernel of the operating system determines that the file being sent is currently cached within the network interface, then only the headers and the location of the file in the NIC's local memory are transferred to the network interface via DMA. In step 5, the network interface then finds the data in its local memory, appends this data to the header sent by the kernel, calculates the required checksums, and sends the response over the network. A system with a network interface data cache reduces traffic on the local interconnect by transferring only relatively short header information when possible, rather than transferring entire files for every HTTP request. While current zero-copy I/O systems avoid copying at the host CPU and main memory, this scheme further extends zero-copy I/O to the server's network interface.

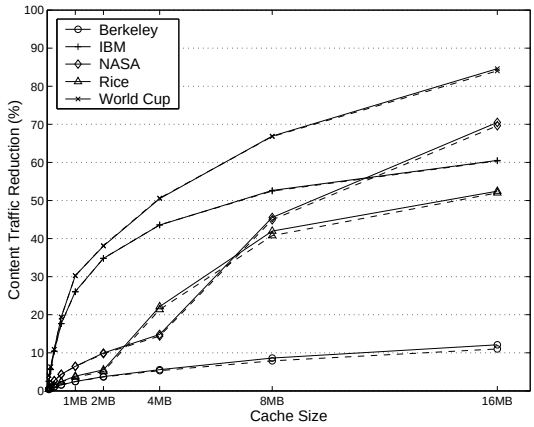

Figure 4.3 : Potential reduction in HTTP content traffic using LRU caches. The X-axis depicts cache sizes from 64 KB to 16 MB, and the Y-axis shows the percentage of response content bytes eliminated from the local interconnect for each of the five traces. The solid and dashed lines show the content traffic reduction possible with one cache and two independent caches, respectively.

The network interface data cache may store parts of a file if necessary. When the kernel finds that parts of the requested file reside in the network interface data cache, it informs the NIC of the location of each part in the cache in addition to the location of the remainder that resides in the main memory. The NIC then transfers the remainder from the main memory and assembles complete packets. Otherwise the operations of the cache described previously remain same.

Due to space, power, and cost constraints, the memory on a NIC is far more limited than the server's main memory, so the network interface data cache will be much smaller than the operating system's file cache. For instance, 256 MB DIMM (Dual In-line Memory Module) DRAMs consume about 8 watts, 32% of the maximum power allowed for any PCI device (25 watts). Therefore, for effective caching, web server requests must have significant data locality. Figure 4.3 shows the percentage of the HTTP content traffic that

would be eliminated from the local interconnect by network interface data caches of varying sizes. This figure was generated using a cache simulator that simply plays back a web trace and determines what portion, if any, of each successive requested file is currently in the cache. The web traces are from Berkeley's Computer Science department, IBM, NASA, Rice's Computer Science department, and the web site for the 1998 Soccer World Cup. These traces are described in detail in Section 4.4.1. The figure shows the results for caches sized from 64 KB to 16 MB, which could easily be supported by DRAM on a NIC, with 4 KB blocks using least-recently used (LRU) replacement. The solid lines show the potential traffic reduction for a single cache. The dashed lines show the potential traffic reduction if two caches of the same size are used with the trace split evenly across the two, simulating a system with independent caches on two network interfaces. Even though this situation doubles the total cache size, the traffic reduction is slightly lower since splitting the traces reduces temporal locality. The figure shows that even modestly sized data caches can significantly reduce traffic from HTTP content, indicating substantial potential main memory and local interconnect bandwidth savings.

As shown in the figure, the World Cup and IBM traces have small working sets, so even 2 MB caches reduce HTTP content traffic by over 35%. The NASA and Rice traces begin to show significant benefit with 8 MB caches, with content traffic reductions of 40–45%. The potential traffic reduction for these four traces continues to improve as cache size increases, with 52–84% of HTTP content traffic eliminated from the local interconnect given dual 16 MB caches. The Berkeley trace, however, has a large working set, and therefore caches as large as 16 MB only reduce HTTP content traffic by 12%.

The choice of replacement policy and cache block size makes little difference in the

potential reduction in HTTP content traffic. Figure 4.4 shows potential reduction in HTTP content traffic using network interface data caches with 4 KB blocks and least-recently used (LRU), least-frequently used (LFU), and first-in first-out (FIFO) block cache replacement policies. As with Figure 4.3, the X-axis shows cache sizes ranging from 64 KB to 16 MB. The Y-axis shows the percentage of HTTP content traffic bytes that are removed by using network interface data caches. The solid and dashed lines show potential reduction in content traffic using one network interface data cache and two independent caches respectively. LRU replacement policy results in slightly greater potential reduction than FIFO replacement policy (1–5% greater reduction using a 16 MB cache). LFU replacement policy again performs slightly better than LRU replacement policy (2–6% using a 16 MB cache). As previously noted, two caches results in slightly lower potential reduction because HTTP request stream is split equally into two caches.

Potential HTTP content traffic reduction using LRU caches with various block sizes is shown in Figure 4.5. The X-axis shows cache sizes from 64 KB to 16 MB. The Y-axis shows percentage of HTTP content bytes that are eliminated using one network interface data cache with block sizes from 512 B to 8 KB. The figure reveals that the block size has very little impact on potential reduction. While caches with smaller blocks perform better, the improvement over caches with largest block (8 KB) is at most 5% with 16 MB cache (World Cup trace).

Figure 4.6 shows the utilization of the PCI bus during the execution of each workload on an actual PC-based web server. The server includes a single AMD Athlon 2200+ CPU, two Gigabit Ethernet NICs, a 64-bit/33 MHz PCI bus, and a VMETRO PBT-615B PCI bus analyzer for passively measuring PCI bus traffic. Section 4.4 gives additional details about

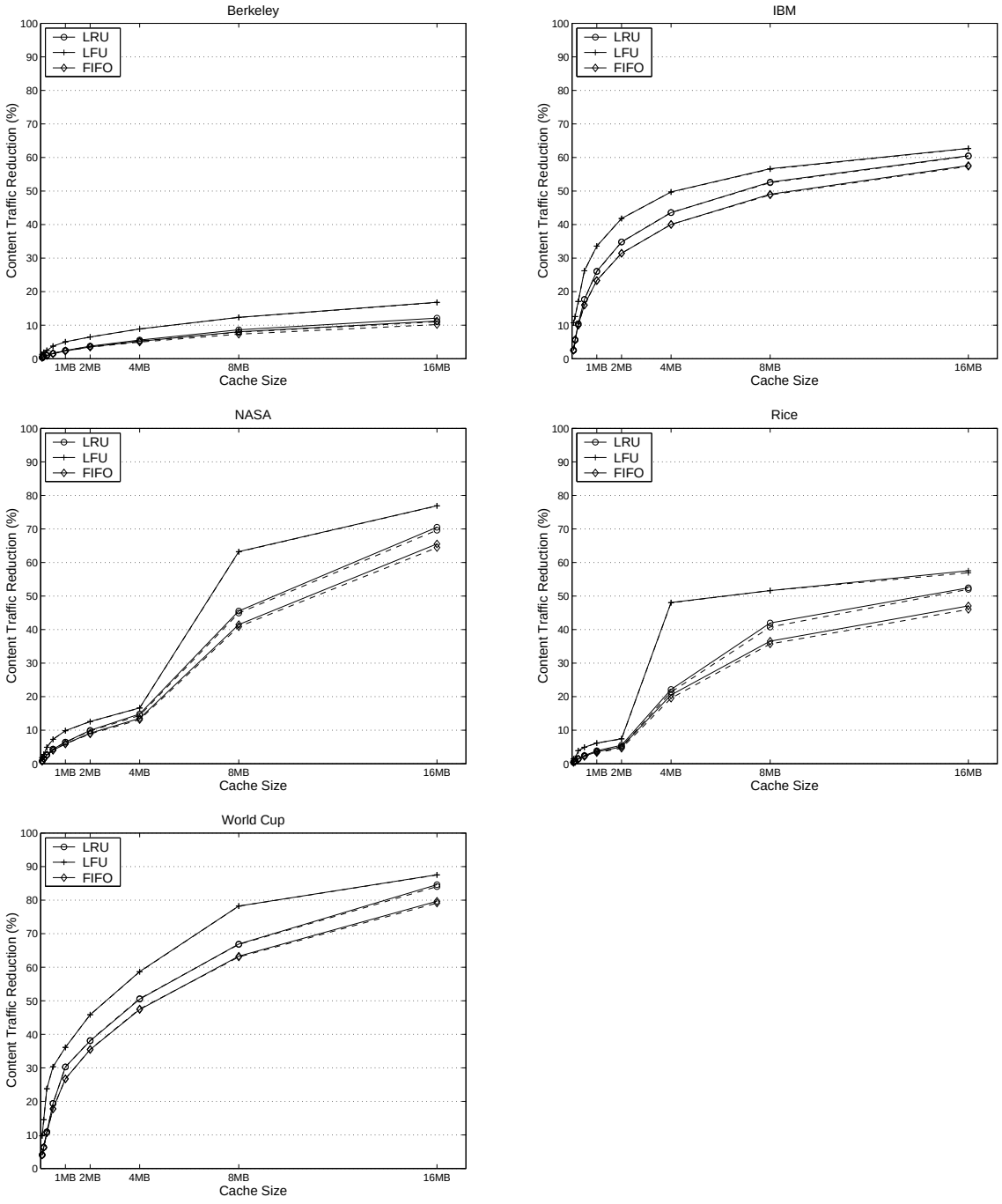

Figure 4.4 : Potential reduction in HTTP content traffic for the five web traces using LRU, LFU, and FIFO caches with 4 KB blocks. Each plot shows potential traffic reduction for one trace. The corresponding trace name is shown at the top of each plot. The axes are same as those in Figure 4.3. The X-axis shows cache sizes ranging from 64 KB to 16 MB. The Y-axis shows the percentage of HTTP response content bytes removed from the local interconnect using different replacement policies. The solid and dashed lines show the content traffic reduction possible with one cache and two independent caches, respectively.

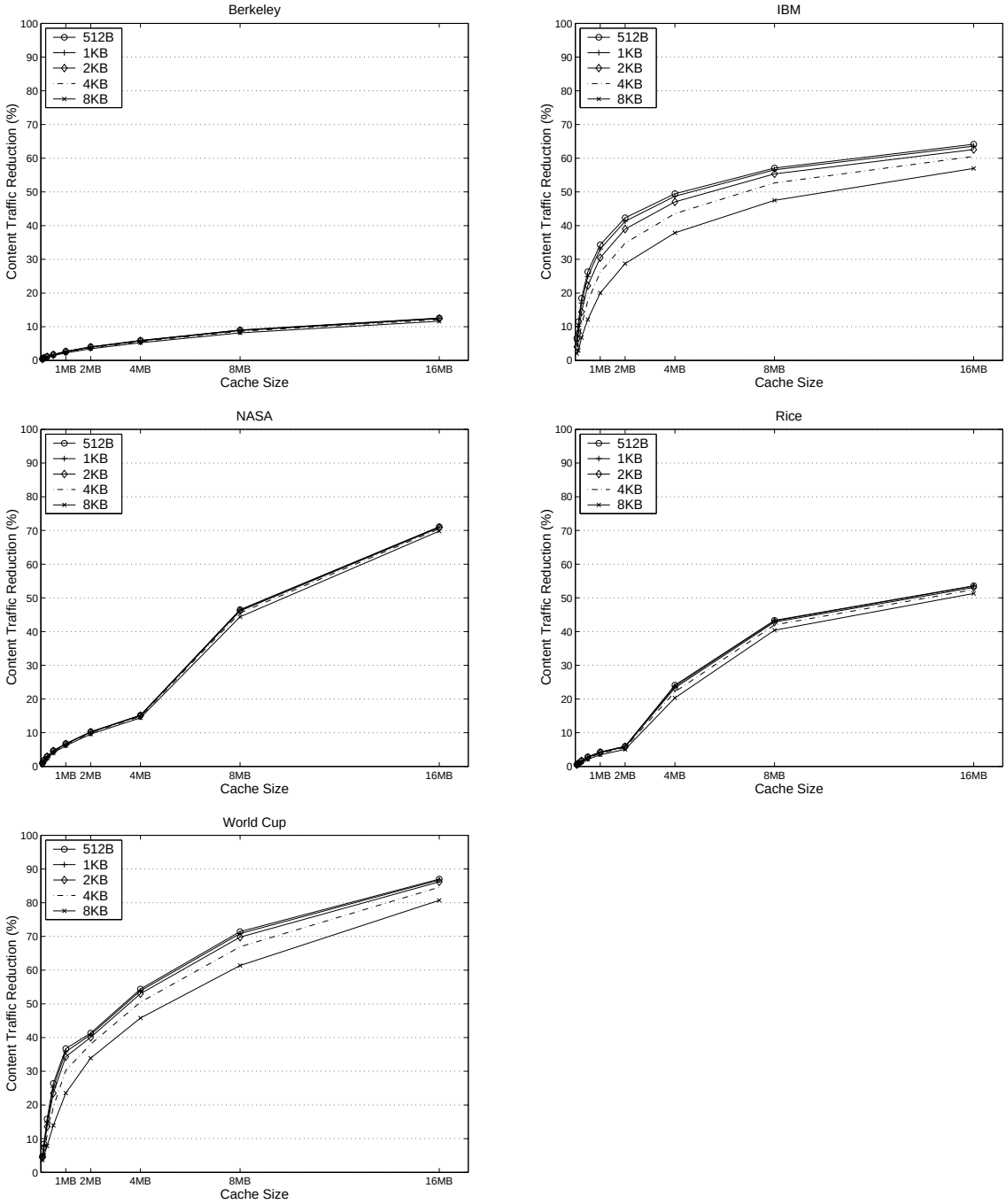

Figure 4.5 : Potential reduction in HTTP content traffic for the five web traces using LRU caches with various block sizes. The axes are same as those in Figure 4.4. Each plot shows potential traffic reduction for one trace. The corresponding trace name is shown at the top of each plot. The X-axis shows cache sizes ranging from 64 KB to 16 MB. The Y-axis shows the percentage of HTTP response content bytes removed from the local interconnect using different block sizes. One network interface data cache is assumed.

the experimental methodology. The utilization shown in the figure represents the ratio of actual PCI traffic during the execution of the workload to the peak theoretical traffic for the server's PCI bus during the same time period. The figure categorizes the different sources of PCI utilization on the server, including HTTP response content, PCI overhead, networking headers (TCP/IP and Ethernet headers from the server), HTTP response headers, and other PCI data (including HTTP request headers, TCP/IP packets from the client, and traffic from other peripherals). PCI overhead accounts for bus cycles spent on transfer-related overheads such as address cycles, wait cycles caused by initiators or targets that are not ready to transfer data, or wait cycles on reads switching the bus from the address phase to the data phase. The NASA and Rice traces lead to around 90% PCI bus utilization, causing severe contention on the bus. The average sizes of requested files for the IBM and World Cup traces are quite small compared to the other traces. The server CPU seems to saturate before the PCI bus. The server can only achieve enough throughput to utilize 43% and 63% of the PCI bus on the IBM and World Cup traces, respectively. The Berkeley trace requires heavy disk accesses due to its large working set size, so disk latency becomes a bottleneck. The HTTP content and PCI overhead account for around 60% and 30% of all PCI traffic, respectively. Thus, even at 94% utilization, a 64-bit/33 Mhz PCI bus only transfers about 1.2 Gb/s of HTTP content on these traces (1188 Mb/s HTTP content throughput for the NASA trace).

Figure 4.7 shows the PCI bus utilization during the execution of each workload on the same web server decribed above except that it uses a 64-bit/66 MHz PCI bus, instead of 64-bit/33 MHz PCI bus. Theoretically, the 64-bit/66 MHz PCI bus can provide twice the bandwidth of the 64-bit/33 MHz PCI bus. However, the overall bus utilization is similar to

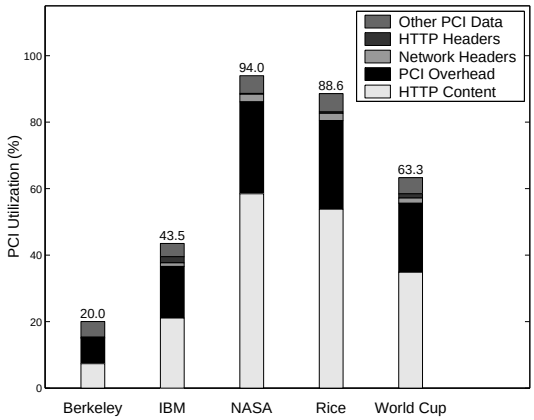

Figure 4.6 : Measured PCI bus utilization in a web server during the execution of the five web traces. The server uses a 64-bit/33 MHz PCI bus. The Y-axis shows the PCI bus utilization for each trace, split into categories for HTTP content, PCI overhead caused by addressing and stalls, networking headers related to TCP/IP and Ethernet, HTTP-level headers, and other PCI data (e.g., traffic from other peripherals). The X-axis shows the trace names.

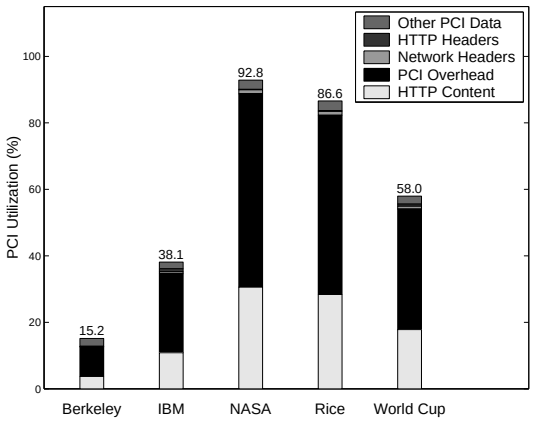

Figure 4.7 : Measured PCI bus utilization in a web server during the execution of the five web traces. The server uses a 64-bit/66 MHz PCI bus. The bus utilization is shown in the same format as in Figure 4.6.

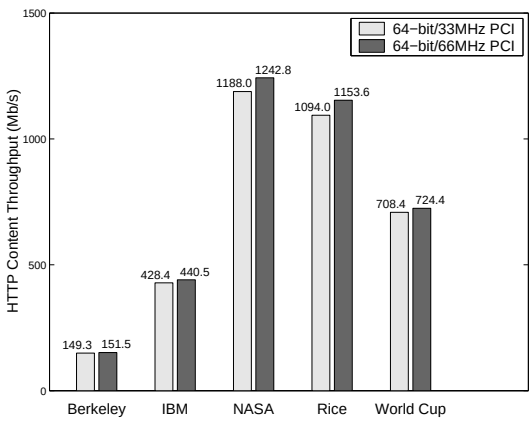

Figure 4.8 : Comparison of the HTTP content throughputs achieved in the web server with a 64-bit/33 MHz PCI bus against those achieved in the web server with a 64-bit/66 MHz PCI bus.

that shown in Figure 4.6. The NASA and Rice traces result in the highest bus utilization factors 92% and 86% respectively, again causing severe contention on the PCI bus. With

the IBM and World Cup traces, the server CPU again seems to saturate , and the web server utilizes 38% and 58% of the PCI bus. The Berkeley trace still requires heavy disk accesses and results in a low 15% bus utilization. The PCI overhead now accounts for roughly 62% of all PCI traffic, almost twice the PCI overhead in the server with the 64-bit/33 MHz. Most PCI overheads are incurred as a result of main memory stall latencies that are proportionally longer for faster buses. The HTTP content account for only about 30% of all PCI traffic. As a result, even though the web server utilizes 92% of the PCI bus during the execution of the NASA trace, the server throughput is about 1243 Mb/s, little improvement over 1188 Mb/s achieved with 94% utilization of the 64-bit/33 MHz PCI bus. Figure 4.8 compares HTTP content throughputs achieved with the web servers using the 64-bit/33 MHz and 64-bit/66 MHz PCI buses. With the 64-bit/66 MHz PCI bus, the server throughput improves by at most 60 Mb/s (for the Rice trace). Thus, little throughput improvements and minimal decreases in bus utilization indicate that simply increasing the bandwidth of the PCI bus does not eliminate the interconnect bottleneck.

Network interface data caching directly targets the HTTP content. With the 64-bit/33 MHz PCI bus, it is the largest component of the PCI traffic and is the second largest component with the 64-bit/66 MHz PCI bus. Reductions in HTTP content traffic lead to reductions in PCI overhead, since the system will now handle fewer transfers. Network interface data caching aims to reduce the two components that account for roughly 90% of all PCI traffic for both types of the PCI buses. Also it is expected to achieve large reductions in HTTP content traffic with reasonable storage capacity. Thus, network interface data caching should provide substantial reductions in overall PCI traffic for these workloads.

4.3 Network Interface Data Cache Design

A network interface data cache utilizes a network interface with a programmable processor and a modest amount of on-board DRAM. The cache stores data that may be appended to packet-level and application-level headers generated by the host CPU and then sent out over the network. The operating system running on the host CPU determines which data to store in the network interface cache and for which packets it should use data from the cache.

4.3.1 Cache Architecture

The network interface data cache is simply a region of local memory on the network interface card. Upon initialization, a programmable network interface must allocate storage for transmit and receive buffers, firmware code, and temporary storage needed by computations used to transmit and receive packets on the network. Any additional space can be used for the network interface data cache. The programmable processor on the NIC notifies the operating system of the size of the network interface data cache after it has been allocated. Ideally, the network interface data cache is as large as possible, but, as shown in Figure 4.3, even modestly sized caches of a few megabytes can significantly reduce interconnect traffic.

Since the network interface data cache resides in the local memory of the network interface card, only the processor on the NIC may access the cache. However, the network interface data cache is controlled entirely by the operating system on the host processor. The NIC processor acts as a slave to the operating system. It inserts and retrieves informa-

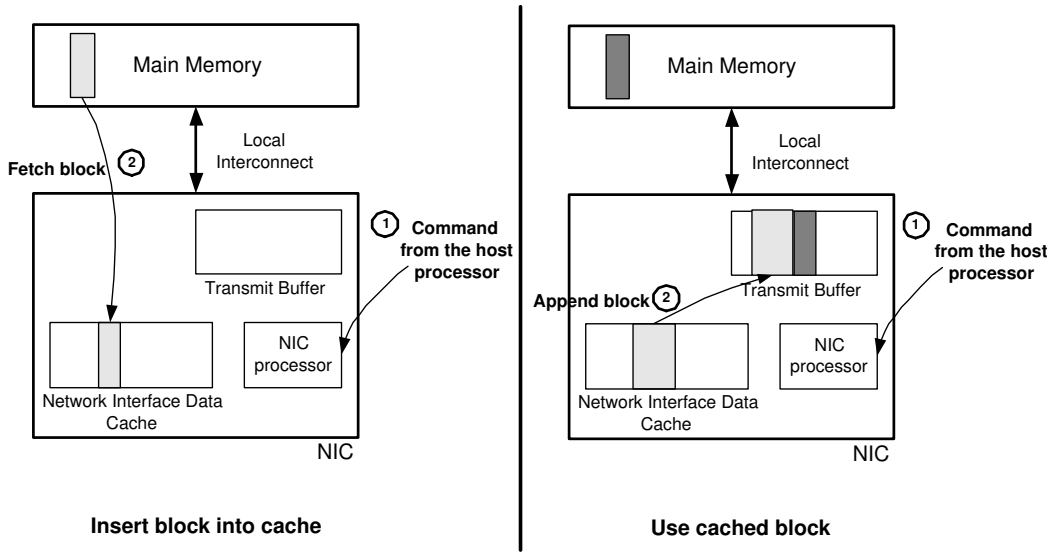

Figure 4.9 : Network interface data cache architecture. The data cache resides in the local memory of the NIC and is completely controlled by the operating system running on the host processor. The processor on the NIC inserts data into the cache or reuses them when instructed by the host processor. The left portion of the figure illustrates insetion of data into the cache. The right portion of the figure illustrates reuse of cached data.

tion from the cache only at the direction of the host.

Figure 4.9 illustrates insertion of data into the network interface data cache and reuse of cached data. When adding new data to the network interface data cache, the operating system instructs the NIC to fetch that data from the main memory and store the data at a particular offset in the cache (step 1 in the left portion of the figure). The NIC then fetches the specified data from the main memory into the cache (step 2). When the operating system decides to use data within the network interface data cache for a packet, it simply instructs the NIC to append data from the cache to the packet by giving the offset and length of the desired data in the cache (step 1 in the right portion of the figure). The NIC appends the specified cached data to the packet in the transmit buffer (step 2). In this way, the operating system can use any data in the network interface data cache for any outgoing

packet. For example, the data can be a subset of a block that was previously inserted into the cache or can straddle multiple cached blocks.

4.3.2 Cache Management

Since the processor on the NIC does not interpret the data in any way, the host processor must establish policies for allocation, replacement, and use of data in the network interface data cache. Additionally, the host processor must resolve the cache coherence problem that arises on modifications to the main memory copy of content replicated on the NIC local memory. The operating system implements all policies for these cache management tasks.

When allocating storage in the network interface data cache, the operating system caches content at the granularity of a file block. Caching blocks instead of packets allows the TCP/IP stack to structure packet contents differently for different responses, if necessary, and also simplifies cache management by using fixed size objects. The operating system also manages a directory of the contents stored within the network interface data cache. The directory entries contain information relating a block in the network interface data cache to the original file that contains the block. An entry contains the file identifier, the offset within the file, the file revision number (maintained by the operating system to track changes in files), and any required status information associated with the data stored in the network interface data cache. A system with multiple NICs has separate directories for each network interface data cache, since the NICs have separate storage.

The operating system attempts to use data from the network interface data cache in response to the `sendfile` system call from the application. `Sendfile` is a commonly-implemented API for zero-copy I/O in servers. Although a server can also transfer data

using the `read` and `write` system calls, the use of user-level data buffers in those system calls commonly causes the kernel to copy data from the kernel file cache to user space on a `read` and from user space to the kernel network buffers on a `write`. Such copying increases CPU and memory load in performing the copies and memory pressure in storing multiple copies of the same information. In contrast, `sendfile` allows for a straightforward implementation of zero-copy I/O since it refers to file content through a descriptor rather than a user-level buffer. As will be shown later, the use of `sendfile` system call can improve web server throughput by up to 47%.

Although the system benefits from zero-copy I/O, it is not a requirement for the network interface data caching. The network interface data caching can use any API as long as the operating system can relate each piece of data that is being sent out onto the network to the original file and supply the cache directory with information required in directory entries.

Figure 4.10 depicts the actions taken by the operating system in response to `sendfile`. If a call to `sendfile` specifies a portion of the file that resides in the operating system file cache, then the operating system creates a set of small memory buffers (called `mbuf`s in FreeBSD) to hold control information and a pointer to the data in the file cache. Each `mbuf` specifies a contiguous region of memory. In the figure, each `mbuf` points to a page of the file to be sent. In step 1, the operating system annotates these `mbuf` structures with the original file identifier, the offset into the file (page offset), and the file revision number. The `mbuf` chains created by `sendfile` are transformed into packets by the networking stack. In step 2, the process of forming packets may split the `mbuf`s so that they reference subranges of pages because each `mbuf` can reference at most one contiguous region of memory. The networking stack then passes the `mbuf` chain for each packet to the

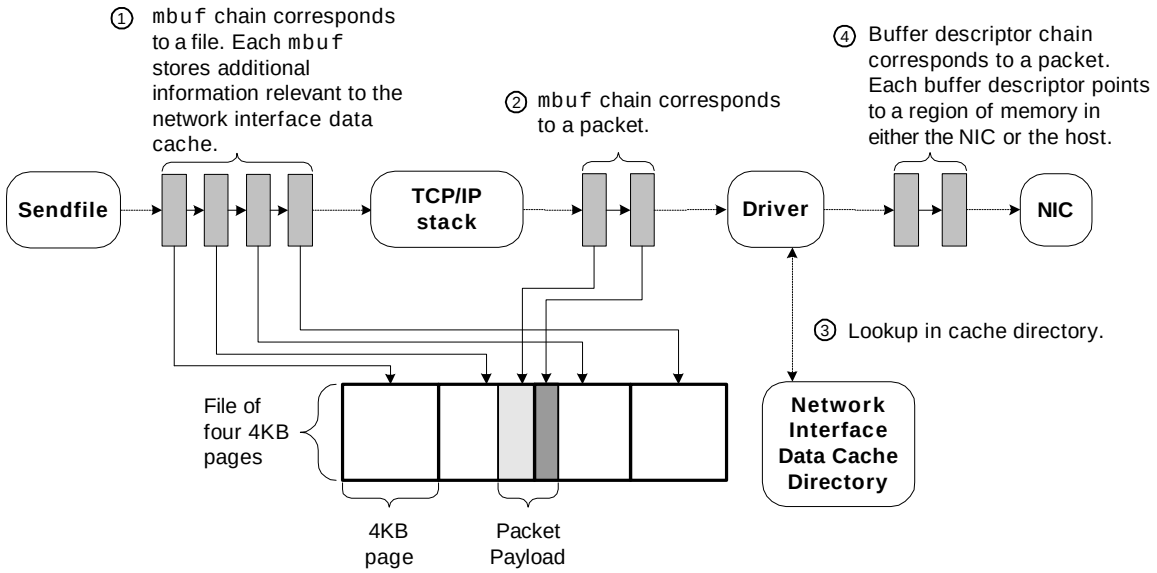

Figure 4.10 : Steps in sending a response using the `sendfile` system call. In step 1, `sendfile` creates a chain of `mbuf`s pointing to the pages in the file. In step 2, the TCP/IP stack splits the pages of the file into packets, creating a new `mbuf` chain for each packet. In this step, each `mbuf` points to a subrange of a page. There is also an `mbuf` at the head of the chain (not shown) for the TCP/IP headers. In step 3, the NIC's driver consults the network interface data cache directory to determine if any parts of the packet are cached. Finally, in step 4, the driver creates a chain of buffer descriptors for each packet. Each buffer descriptor points to data in either main memory or the network interface data cache.

device driver for the NIC that will be used for transmission.

In step 3 in the figure, the device driver looks up each referenced block in the network interface data cache directory. If the block is already present, then the driver informs the NIC of the offset and length of the content within the network interface data cache. If the block is not present, the driver seeks to allocate a block in the cache, using an LRU replacement policy to evict old blocks if no space is available. As described in Section 2.3, the device driver creates buffer descriptors to send packets. Without network interface data cache, buffer descriptors are used to pass main memory location of packets to the NIC. To support the network interface data caching, in step 4, the device driver creates a set of

buffer descriptors to pass the relevant information about the cache as well as location of packets. With network interface data cache, packet data may reside in both main memory and NIC local memory. Thus a buffer descriptor now either points to a buffer in the main memory or a region of the network interface data cache.

The device driver and the NIC follow communication steps described in Section 2.3 in order to pass buffer descriptors to the NIC. In short, the CPU notifies the NIC that it has created new buffer descriptors by writing to a memory mapped register of the NIC. The NIC then retrieves the buffer descriptors using DMA and uses the information contained within them to initiate the necessary DMA transfers to retrieve the data from main memory. After completing the requested operation, the NIC interrupts the CPU to inform it that the buffer descriptors have been consumed. In a system without a network interface data cache, these buffer descriptors always require the NIC to transfer packet data from main memory using DMA. In a system with a network interface data cache, however, buffers pointing to cached content do not require a DMA transfer for packet data.

The file revision field stored in each directory entry enables a straightforward mechanism to keep the cached blocks coherent with the objects stored on the server's main storage system. When looking up blocks in the network interface data cache directory, the device driver lazily invalidates blocks for which the current revision identifier does not match the cached revision. Note that multiple NICs do not present any additional cache coherence problems, since each network interface data cache operates independently.

To further simplify the coherence implementation, the operating system keeps the main memory file cache strictly inclusive of the network interface data cache. With this guarantee of inclusion, information mapping file blocks to network interface data cache storage need

- `nic_cache_init()`
  Allocate and initialize the data cache on the programmable network interface and return its size.
- `nic_cache_insert(mmaddr, ncoffset, len)`
  The buffer descriptor contains `mmaddr`, `ncoffset`, and `len`. Use DMA to transfer `len` bytes starting at main memory address `mmaddr` into the network interface data cache starting at offset `ncoffset`.
- `nic_pkt_append_mm(mmaddr, len)`
  The buffer descriptor contains `mmaddr`, `len`, and additional flags. Use DMA to transfer the `len` bytes starting at main memory address `mmaddr` into the network interface's transmit buffer. This function is unchanged from the original operation of the NIC.
- `nic_pkt_append_cache(ncoffset, len)`
  The buffer descriptor contains `ncoffset`, `len`, and additional flags. Copy `len` bytes from the cache starting at offset `ncoffset` into the transmit buffer.

Figure 4.11 : Commands supported by the programmable processor on the network interface and invoked by the operating system. These commands are passed to the network interface through the buffer descriptors, which are currently used to control DMA transfers.

not persist beyond the replacement of a block from the main memory file cache. This inclusion property provides two further benefits. First, it allows caching even for NICs that do not support checksum offloading, since all content on the NIC also resides in the main memory and can thus have checksum calculations performed by the host CPU. Second, inclusion simplifies network retransmits in the event of replacements from the network interface data cache, since the the operating system always keeps data in the main memory file cache for retransmits.

4.3.3 Cache Interface

The operating system manages the network interface data cache using an API that consists of the four functions listed in Figure 4.11. The table only shows the functions used to send packets; the NIC must also support functions to receive packets and perform other actions,

but those need not be modified since the cache does not address those tasks. Since the host processor cannot directly call functions on the network processor, these API functions are actually implemented using existing mechanisms to communicate from the host processor to the NIC. In particular, the operating system uses flags in the buffer descriptor data structure to indicate which command to invoke, and additional fields within those buffer descriptors to pass arguments to the NIC.

The API for the network interface data cache includes functions to initialize the cache, to copy data from main memory to the network interface data cache, to append a block of main memory to the current packet, and to append a cached block to the current packet. All other NIC functions remain unchanged. The initialization function, `nic_cache_init`, allocates space in the NIC's local memory and notifies the operating system of the amount of memory that has been allocated so that the operating system may construct and manage the cache directory. Data is added to the cache using the `nic_cache_insert` function. Data is transferred from the main memory to the network interface data cache using DMA. As with all DMA transfers, a single buffer descriptor can only describe a contiguous buffer. So, if disjoint memory regions are to be added to the cache, the operating system must call `nic_cache_insert` multiple times.

The API of a conventional NIC effectively only includes the `nic_pkt_append_mm` function to construct and send packets. As described in Section 2.3, packets are transmitted over the network by generating a list of buffer descriptors that are then sent to the NIC. Each buffer descriptor points to a region of main memory that the NIC should append to the current packet by using DMA to transfer that block of memory from the host to the transmit buffer. This is accomplished by having each buffer descriptor invoke the

`nic_pkt_append_mm` function on the NIC. Additional flags in the buffer descriptor are used to indicate to the NIC if that block is the first or last block in the packet, if a particular function should be performed before sending the packet (such as checksum offloading or other future services), or any additional information required by the NIC to process the packet.

The last API function is `nic_pkt_append_cache`. This function resembles `nic_pkt_append_mm` but copies data to the transmit buffer from the indicated offset in the network interface data cache instead of using a DMA from main memory. Note that even this internal copy could be eliminated if the engine that transfers the data out onto the network could gather the packet from disjoint memory regions on the NIC. As with `nic_pkt_append_mm`, additional flags in the buffer descriptors are used to indicate if the block is the last block in the packet or if additional processing should occur.

Referring back to Figure 4.10 in Section 4.3.2, the buffer descriptors of step 4 convey the network interface data cache commands of Figure 4.11. If the data represented by an `mbuf` is not present in the network interface data cache, the driver inserts the data into the cache using `nic_cache_insert`. The driver then sends a series of buffer descriptors have contain command `nic_pkt_append_cache` or `nic_pkt_append_mm`, as appropriate, to transmit each packet. Headers for TCP/IP, Ethernet, and HTTP are transferred to the NIC using `nic_pkt_append_mm` and are never inserted in the network interface data cache. The processor on the NIC concatenates the cached data with the headers fetched from the host memory before the packets are transmitted onto the network.

An illustration of a packet transmission using the API is shown in Figure 4.12. A portion of packet data is assumed to reside in the network interface data cache. Furthur assume

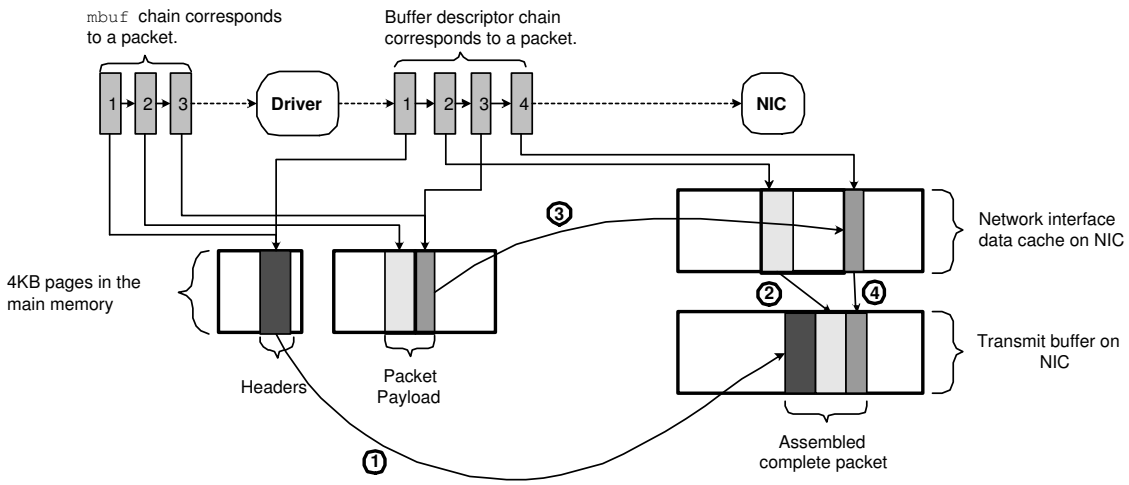

Figure 4.12 : Illustration of a packet transmission using the API shown in Figure 4.11. When the device driver creates buffer descriptors, the packet data pointed by the third `mbuf` is not yet in the network interface data cache. Thus the device driver creates the third buffer descriptor that contains command `nic_cache_insert` to insert that data into the cache. The first buffer descriptor points to the headers and contains command `nic_pkt_append_mm`. The second and fourth buffer descriptors point to data in the cache thus contain command `nic_pkt_append_cache`. The NIC processes the buffer descriptors in order and assmebles a complete packet.

that the device driver maintains a policy of always caching packet data if it is not found in the cache. The first `mbuf` points to a buffer in the main memory that stores headers for TCP/IP, Ethernet, and HTTP. Since headers are never cached, the first buffer descriptor points to that main memory buffer and contains command `nic_pkt_append_mm`. The `mbuf` points to the portion of packet data that already resides in the network interface data cache. The second buffer descriptor then points to a block in the cache and contains command `nic_pkt_append_cache`. The third `mbuf` points to the rest of packet data which is not in the cache. Thus the device driver creates the third buffer descriptor that points to that packet data in the main memory and contains command `nic_cache_insert` to insert the data into the cache. The last, fourth, buffer descriptor points to the block in the

cache that will contain the uncached portion of packet data. The NIC processes each buffer descriptor in order, thereby implicitly concatenating headers to packet data, and sends the complete packet out onto the network. Specifically, in step 1, the NIC transfers the headers from the main memory into the transmit buffer. In step 2, the NIC copies the data in the cache that is pointed by the second buffer descriptor to the transmit buffer. In step 3, the NIC transfers the data in the main memory that is pointed by the third buffer descriptor into the cache. In step 4, the NIC copies the data that has just been inserted into the cache.

Note that packets can bypass the network interface data cache by only using `nic_pkt_append_mm`. Such packets are useful in circumstances where the driver chooses not to cache a file block or where a transmission does not refer to a file block (such as dynamic content, `ping`, or `telnet`).

4.4 Evaluation Methodology

4.4.1 Web Traces

Five web traces are used to evaluate network interface data caching. The NASA and World Cup traces are access logs collected at a NASA web site and the 1998 World Cup web site, respectively. These were made public by Arlitt and Williamson, who studied these traces in order to derive invariants of web access patterns [3]. The IBM trace was donated privately by Erich Nahum at the IBM T.J. Watson Research Center. The Rice trace is one of the access logs collected at a web site for the Computer Science department at Rice University. The Computer Systems Group at Rice University initially acquired the Berkeley trace and made it available to the author. The trace is an access log collected at a web site for the

Statistic	Berkeley	IBM	NASA	Rice	World Cup
Trace Duration	1 month	4 days	1 month	1 month	2 weeks
Year	2000	1998	1995	2000	1998
Total Response Bytes (MB)	82,545	44,487	96,069	8,588	72,869
Average Response Size (B)	25,920	2,857	56,983	34,936	6,847
Data Set Size (MB)	6,664	997	271	1,191	93
Total Requests	3,184,540	15,568,217	1,685,904	245,820	10,641,170
Distinct Requests	97,766	39,363	4,690	15,528	5,163

Table 4.1 : Basic statistics of the five web traces used to evaluate network interface data caching.

Computer Science department at the University of California at Berkeley.

Table 4.1 lists several statistics of the five web traces that are most relevant to the evaluation of network interface data caching. All traces contain only those requests to static contents that were successfully transferred. Each column shows statistics of a particular trace. The trace duration is the elapsed time between the first request and the last request in a trace. The total response bytes is the number of bytes transferred by the server. The total requests are the number of all requests in a trace, and the distinct requests are the number of requests without repeated requests. The average response size is then computed the total response byte divided by the total requests. Finally, the data set size is the total size of all distinct files in a trace.

The Berkeley trace has the largest date set size, over 6 GB. All other traces have data set sizes around or less than 1 GB. The NASA and Rice traces have the largest average response sizes about 57 KB and 35 KB respectively. These are considerably higher than the average response sizes of the other traces. Large reponses tend to reduce networking overheads such as HTTP connection setup/teardown. For all traces, distinct requests are only a fraction of total requests. Thus, caching is expected to be very effective. Arlitt

and Williamson [3] present detailed analysis of frequently transferred files and predict that caching would be effective.

4.4.2 Prototype Implementation

A prototype implementation of network interface data caching is built using a PC-based server and 3Com 710024 Gigabit Ethernet interface cards. The prototype implements an LRU block cache with lazy invalidation, a block size equal to the page size of the operating system (4 KB), and a requirement of inclusion in the main memory file cache. As discussed in Section 4.2, the choice of replacement policy and block size makes little impact on the potential reduction in HTTP content traffic. The LRU replacement policy with 4 KB block size is chosen to simplify implementation. The server has 2 GB of DDR SDRAM, two 36 GB SCSI disks, two 3Com 710024 copper Gigabit Ethernet NICs, and a VMETRO PBT-615B PCI bus analyzer. The server has various configurations of the host processor and PCI bus in order to evaluate how technology affects performance of network interface data caching. Table 4.2 shows the configurations. For all configurations, the two Gigabit NICs are plugged in 64 bit PCI slots. The SCSI controller is plugged in a 32 bit PCI slot. The 3Com 710024 NIC is based on the programmable Tigon Gigabit Ethernet controller described in Section 3.1. Each NIC also has 1 MB of on-board memory. The 3Com NICs in the server run a modified version of the PARALLEL firmware described in Section 3.3.1. The modified firmware implements the API commands of Section 4.3.3. The PCI bus analyzer passively measures the actual PCI bandwidth utilization and injects no traffic onto the PCI bus.

The server runs a slightly modified FreeBSD 4.6 operating system. The `sendfile`

Configuration	Host Processor (Clock Speed)	PCI Bus (Maximum Throughput)
2200/33	AMD Athlon XP 2200+ (1.8 GHz)	64-bit/33 MHz PCI Bus (2.1 Gb/s)
2200/66	AMD Athlon XP 2200+	64-bit/66 MHz PCI Bus (4.2 Gb/s)
2600/33	AMD Athlon XP 2600+ (2.1 GHz)	64-bit/33 MHz PCI Bus (2.1 Gb/s)
2600/66	AMD Athlon XP 2600+	64-bit/66 MHz PCI Bus (4.2 Gb/s)

Table 4.2 : Server configurations used to evaluate network interface data caching. The remainder of this chapter uses the configuration names to concisely describe server hardware.

system call is extended to use network interface data caching. The `mbuf` structure has five new fields, and `mbuf` manipulation routines are modified to handle the new fields appropriately. Finally, the device driver for the NIC is modified to maintain cache directories and generate modified buffer descriptors that contain the API commands for network interface data caching. All these kernel changes require about 150 modified lines in the `sendfile` system call and `mbuf` manipulation routines, and roughly 850 lines of new code in the device driver.

The 3Com NIC only has 1 MB of on-board memory. Roughly three-fourths of the on-board memory are allocated to storage for the firmware code, transmit buffer, and receive buffer. Thus, the 1 MB of on-board memory is insufficient for evaluation purposes. Instead, network interface data caches of various sizes are emulated using the following modifications to the API shown in Figure 4.11. Upon receiving a `nic_cache_insert` command, the prototype fetches the specified data and discards it instead of adding it to the cache. On a `nic_pkt_append_cache` command, the prototype simply increments the pointer to the end of the transmit buffer by the specified length, using whatever data is currently in the buffer. The other API functions behave as specified. The NIC with these simplifications generates the same amount of PCI bus traffic and Ethernet traffic as a fully functional NIC

that actually stores cached blocks and reuses them on appropriate commands. However, the lack of copying in `nic_pkt_append_cache` ignores the overhead of copying. This copying is unnecessary for a NIC that supports gather I/O and can transmit packets consisting of discontiguos memory regions. Another problem with `nic_pkt_append_cache` is that packets that include cached data have invalid checksums. The Tigon checksumming hardware is integrated into the DMA engine, so only data that is transferred between the host and the Tigon may be checksummed. A slight modification to the Tigon architecture to allow data stored in local memory to be run through the checksumming hardware can solve the checksum problem. For instance, the DMA engine could allow a DMA transfer to read from the local memory, compute checksums, and then discard the data. These additional features are simple and would not require substantial implementation costs. Thus despite the simplifications in `nic_cache_insert` and `nic_pkt_append_cache` commands, the prototype should accurately emulate network interface data caching and its impact on the server performance.

4.4.3 Test Platform

The performance testbed consists of the prototype web server and two client machines connected via a Gigabit Ethernet switch. Each client machine has an AMD Athlon MP 2000+ processor and two Intel Pro/1000 MT Server Adapters. The client machines run FreeBSD 4.7. The server and clients are connected through two Netgear GS508T Gigabit Ethernet switches isolated from the rest of the network, so there is no background traffic during the experiments. The experiments use two private subnets, and each machine in the testbed has one network interface on each subnet.

The network interface drivers on the synthetic clients have been modified to accept packets from the prototype server containing cached data despite the invalid checksums discussed in Section 4.4.2; to support this distinction, the server marks such packets with an artificial time-to-live field in the IP header. These modifications would not be needed if the network interface had sufficient memory and additional checksumming hardware to properly support `nic_pkt_append_cache`.

The server runs the same thttpd web server described in Section 3.3.2 that is used to evaluate the parallelization technique for the Tigon firmware. Server throughput is measured using a trace replayer tool, which takes a web trace as input and simulates multiple users by opening multiple simultaneous TCP connections to the server, each of which corresponds to a single user. The trace replayer uses an infinite-demand model, issuing requests as fast as the server can sustain responses. Requests that came from the same anonymized client IP address within a fifteen second period in the original access log are treated as a single persistent connection. Within a persistent connection, requests that arrive less than five seconds apart in the original log are grouped, and requests within each group are pipelined. Each client machine runs two instances of the trace replayer, which connect to the server through different subnets. The traces are split equally among all four replayers. As described in Section 4.4.1, the web traces are from Berkeley's CS department, IBM, NASA, Rice's CS department, and the web site for the 1998 Soccer World Cup. The traces include between about 245 thousand (Rice) and 15 million (IBM) requests.

4.4.4 Impact of the `Sendfile` System Call

This section presents throughput improvements from the zero-copy `sendfile` system call over the traditional `write` system call that copies data from the user space to the kernel space. Two versions of the thttpd web server are used to compare web server throughputs. One version uses the zero-copy `sendfile` system call, and the second version uses the `write` system call. With the `write` system call, thttpd must first read file data into the user space, and then the kernel copies the data to the kernel space. The thttpd web server uses the `mmap` system call that maps pages of a file into the user's virtual memory space. The use of the `mmap` system call eases the buffer management task of the server since it does not need to manually allocate buffers to store file data. The `sendfile` version of thttpd does not use `mmap` since reading file data into the user space is unnecessary. These versions do not use one optimization that is implemented in the version of thttpd used for the evaluation of network interface data caching. Specifically, this optimization caches file information such as the file size so that the web server does not have to query the kernel for each requested file. The use of `sendfile` makes the implementation straighforward. Since the optimization is excluded from both the `sendfile` and `write` versions of thttpd, the exclusion of the optimization does not favor one version over the other.

Table 4.3 shows throughputs achieved in the prototype server using the two versions of thttpd. The server configuration is 2600/66 in Table 4.2. Since this server has the fastest hardware among the four server configurations (see Table 4.2), copying operations should be least expensive. For all workloads, the `sendfile` version of thttpd improves server throughput by 21%–81% over the thttpd that uses the `write` system call. The throughput

Web Trace	`sendfile` thttpd	`write` thttpd	Improvement
Berkeley	142 Mb/s	118 Mb/s	21%
IBM	424 Mb/s	331 Mb/s	28%
NASA	1278 Mb/s	914 Mb/s	40%
Rice	1148 Mb/s	635 Mb/s	81%
World Cup	737 Mb/s	599 Mb/s	23%

Table 4.3 : Throughput improvements from the `senfile` system call. The `sendfile` thttpd uses the zero-copy `sendfile` system call to transmit data to clients. The `write` thttpd uses the `write` system call instead. The `write` system call copies data from the user space to the kernel space.

improvements of the `sendfile` version of thttpd come from both using zero-copy I/O and eliminating the use of the `mmap` system call. Since `sendfile` enables both benefits, it is clearly advantageous for thttpd to use `sendfile`.

4.5 Experimental Results

As shown in Figure 4.3 of Section 4.2, there is significant locality of access to web pages in a typical server. Data caches of 16 MB per network interface have the potential to eliminate 52–84% of HTTP content traffic from the local interconnect for four of the five web traces and 12% for the Berkeley trace. This section discusses the interconnect traffic reductions achieved in the prototype system and their effect on server throughput. This section then examines how improving technology for the CPU and PCI bus affects the effectiveness of network interface data caching.

4.5.1 Local Interconnect Traffic and Server Throughput

Figure 4.13 shows the impact of network interface data caching on PCI traffic for the server configuration 2200/33 in Table 4.2. The figure shows four bars for each trace. The leftmost

bar represents traffic without caching, and the right three bars represent traffic with cache sizes of 4 MB, 8 MB, and 16 MB per network interface. All measures of PCI traffic are normalized to the traffic without caching. As in Figure 4.6, the bars are each split into five categories. As can be seen in Figure 4.13, network interface data caching reduces the amount of HTTP content transferred across the local interconnect substantially for all workloads except the Berkeley trace, as predicted by Figure 4.3. Reducing the bus traffic from HTTP content also reduces the associated PCI bus overhead by eliminating some addressing cycles and transfer stalls.

The third and fourth columns of Table 4.4 show the actual reductions in PCI bus utilization from HTTP content traffic and PCI overhead for each trace and cache size listed in the first two columns. Because the PCI overhead stems from data transfers across the bus, removing data transfers also reduces the PCI overhead associated with those transfers. As the content traffic reduction increases, the PCI overhead decreases, leading to further reductions in PCI traffic. With the 64-bit/33 MHz PCI bus, the PCI overhead was originally the second largest part of PCI utilization, at roughly 30% (see Figure 4.6 in Section 4.2).

With 16 MB data caches on each network interface, the server reduces HTTP content traffic on the bus by 51–83% for the IBM, NASA, Rice, and World Cup traces. These numbers closely match the predictions in Figure 4.3 despite being measured on a real system, which may reorder requests and responses due to various latencies in the system and the different ways of splitting the traces. The PCI overhead accordingly decreases by 13–33%, combining for overall PCI bus traffic reductions of 33–54%. Since only HTTP content is cached on the network interface, network interface data caching does not change the other types of PCI traffic such as HTTP and network headers.

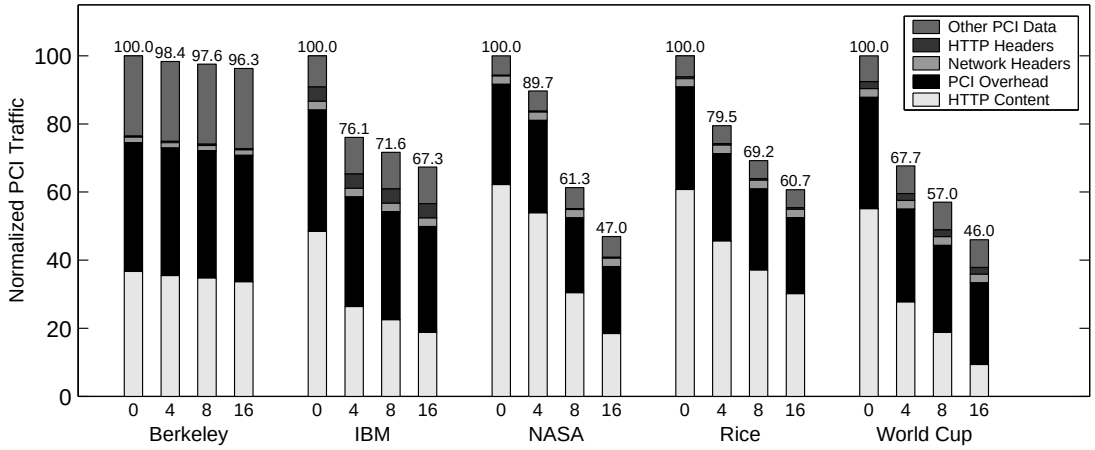

Figure 4.13 : Comparison of PCI traffic with and without network interface data caching for the server configuration 2200/33 in Table 4.2. The cache sizes are 4, 8, and 16 MB per network interface. The cache size 0 MB means that caching is not used.

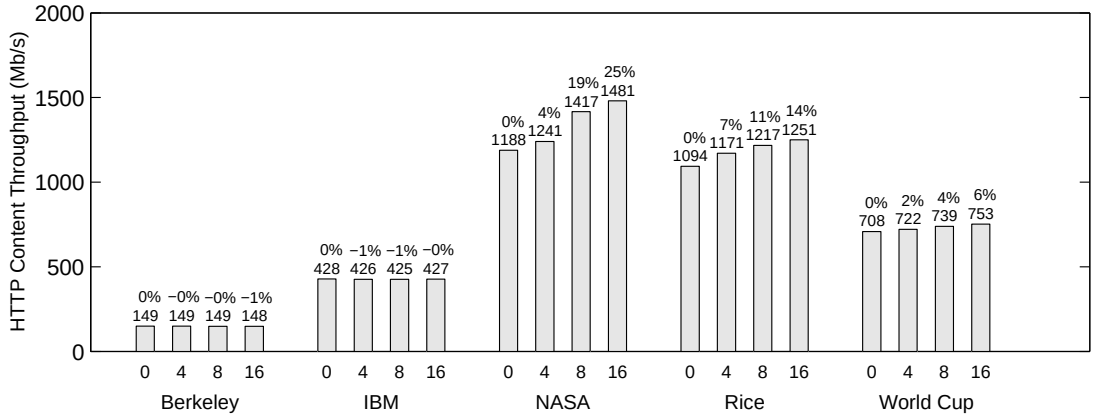

Figure 4.14 : Improvements in HTTP content thoughput from network interface data caching using the same server configuration as in Figure 4.13. Server throughputs are shown on top of the bars. The percent numbers show the throughput improvement over the 0 MB case (no caching).

The Berkeley workload shows minimal reduction in overall PCI traffic because its large data set size (over 6 GB, as shown in Table 4.1) allows network interface data caching to eliminate only 9% of the HTTP content traffic and 2% of the PCI overhead from the bus. More intelligent replacement policies may provide additional benefits for such a work-

load [8, 10]. Additionally, predicting reuse patterns could allow for reducing cache pollution by bypassing the cache entirely for some data, as has been studied in other contexts [38].

Figure 4.14 shows the throughput improvements achieved by network interface data caching for the server configuration 2200/33. As shown in Figure 4.6, the NASA and Rice workloads result in the highest PCI bus utilization without caching. Therefore, they benefit the most from network interface data caching. Figure 4.14 shows that the server achieves a 25% throughput improvement on the NASA trace and a 14% improvement on the Rice trace using 16 MB caches in the network interfaces. This enables the server to achieve a peak throughput of 1481 Mb/s on the NASA trace and 1251 Mb/s on the Rice trace. Network interface data caching is most effective at capturing the locality of the World Cup trace, with 16 MB caches reducing 83% of the bus traffic for HTTP content. However, since PCI utilization is only 63%, this translates to a smaller throughput improvement (6%) than on the NASA and Rice traces. Table 4.4 tabulates actual reductions in HTTP content and PCI overhead as well as throughput improvements.

As expected, the network interface data caching does not improve the server throughput for the Berkeley and IBM traces. As discussed in Section 4.2, the disk latency limits server throughput for the Berkeley workload, and the server CPU seems to limit throughput for the IBM trace. The PCI utilization is thus low for both traces. Consequently the contention on the bus is not a bottleneck in the system. Any reduction in PCI traffic does not improve throughput. Additionally, the overhead of managing the cache and using the cache commands causes a slight performance degradation.

Trace	Cache Size per NIC	Reduction in HTTP Content	Reduction in PCI Overhead	Throughput (Mb/s)	Request Rate (req/s)	Cache Benefit
Berkeley	none			149	767	
	4 MB	4%	1%	149	764	0%
	8 MB	6%	1%	149	763	0%
	16 MB	9%	2%	148	762	-1%
IBM	none			428	19443	
	4 MB	44%	10%	426	19314	-1%
	8 MB	52%	11%	425	19309	-1%
	16 MB	60%	13%	427	19380	0%
NASA	none			1188	2733	
	4 MB	13%	8%	1241	2855	4%
	8 MB	51%	25%	1417	3259	19%
	16 MB	70%	33%	1481	3408	25%
Rice	none			1094	4257	
	4 MB	26%	15%	1171	4565	7%
	8 MB	40%	21%	1217	4748	11%
	16 MB	51%	26%	1251	4873	14%
World Cup	none			708	13551	
	4 MB	49%	17%	722	13805	2%
	8 MB	66%	22%	739	14142	4%
	16 MB	83%	27%	753	14395	6%

Table 4.4 : Improvement in server performance from caching. The server configuration is 2200/33 in Table 4.2. The measured throughput includes only actual HTTP content, not HTTP or networking headers.

4.5.2 PCI Bus Speed

As discussed in Section 4.2, the use of a faster 64-bit/66 MHz PCI bus results in vastly increased PCI overheads and little server throughput improvements. This section examines the impact of the faster 64-bit/66 MHz PCI bus on the effectiveness of network interface data caching and compares it to the results in the previous section that are based on the server with a 64-bit/33 MHz PCI bus.

Figure 4.15 shows the PCI traffic reductions from network interface data caching for

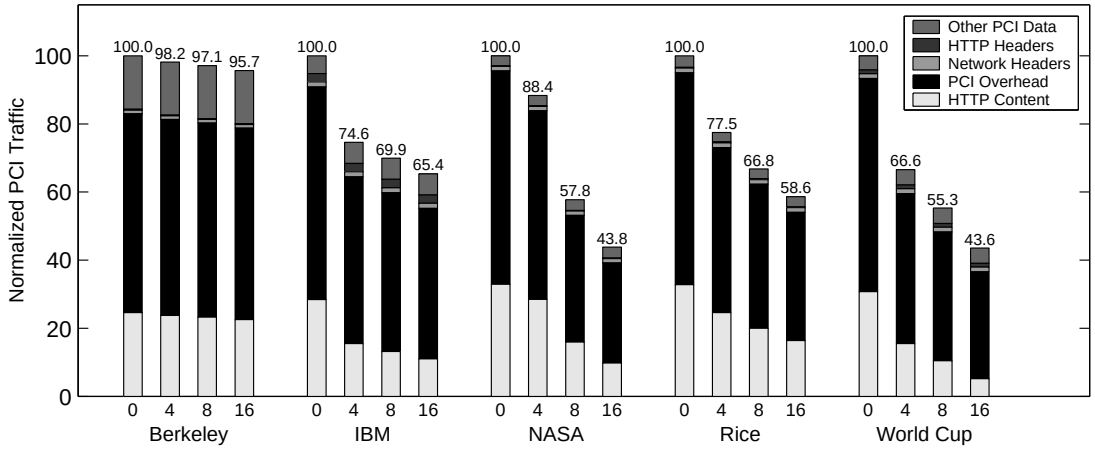

Figure 4.15 : Comparison of PCI traffic with and without network interface data caching. The server configuration is 2200/66 in Table 4.2. The cache sizes are same as those shown in Figure 4.13.

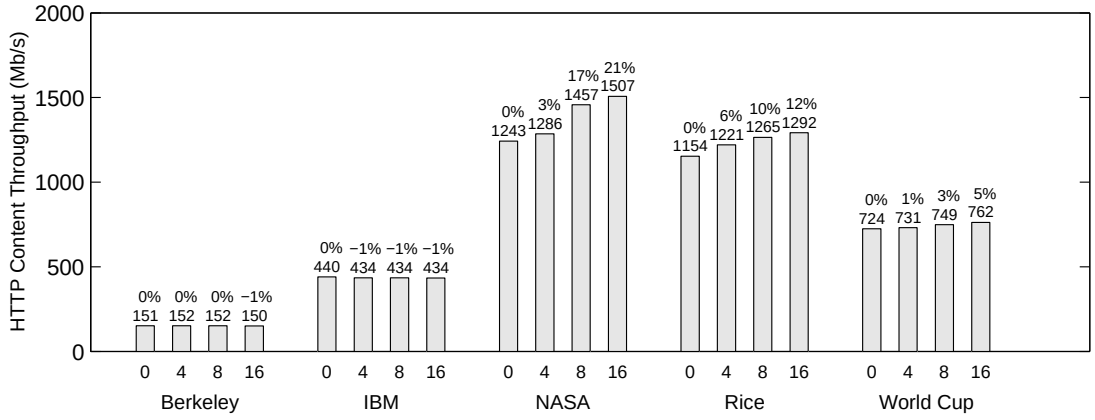

Figure 4.16 : Improvements in HTTP content thoughput from network interface data caching using the same server configuration as in Figure 4.15. Server throughputs are shown on top of the bars. The percent numbers show the throughput improvement over the 0 MB case (no caching).

the server configuration 2200/66 in Table 4.2. This configuration is the same as the one discussed in as in the previous section except that the server has a faster 64-bit/66 MHz PCI bus. Reductions in the HTTP content traffic are same as those shown in Figure 4.13 because PCI clock speeds do not affect cache hit rates, and network interface data caching

targets the HTTP content. The overall PCI traffic reductions are also roughly same as those shown in Figure 4.13 because the PCI overhead is caused by data transfers. Recall that with the 64-bit/66 MHz PCI bus, the PCI overhead accounts for about 60% as opposed to about 30% with the 64-bit/33 MHz PCI bus (See Figures 4.6 and 4.7). Since reductions in the HTTP content traffic do not change, reductions in the PCI overhead increase proportionally to the bus clock speed, resulting in the roughly same reductions in the overall PCI traffic. The overall reductions are now 34%–57% for all traces except the Berkeley trace (5%). Using the slower PCI bus, the reductions are 33%–54% for the same four traces and 4% for the Berkeley trace.

Figure 4.16 shows the throughput improvements from network interface data caching for the server configuration 2200/66 in Table 4.2. Throughput improvements are similar to those achieved in the server with the 64-bit/33 MHz PCI bus. With 16 MB caches, throughputs for the NASA, Rice, and World Cup workloads improve by 21%, 12%, and 5% (25%, 14%, and 6% with the 64-bit/33 MHz PCI bus). Remember that the server with the faster 64-bit/66 MHz PCI bus achieves slightly lower bus utilization than the server with the slower 64-bit/33 MHz PCI bus, as discussed in Section 4.2. Thus, as discussed in the previous section, these lower throughput improvements from caching with the 64-bit/66 MHz again indicate that throughput improvements from network interface data caching is proportional to the PCI bus utilization. The Berkeley and IBM traces again do not benefit from caching. The faster 64-bit/66 MHz PCI bus does improve peak server throughputs, albeit by less than 50 Mb/s. Actual reductions in the HTTP content traffic and PCI overhead as well as throughput improvements are shown in Table 4.5.

These reductions in PCI traffic and throughput improvements have several consequences.

Trace	Cache Size per NIC	Reduction in HTTP Content	Reduction in PCI Overhead	Throughput (Mb/s)	Request Rate (req/s)	Cache Benefit
Berkeley	none			151	778	
	4 MB	4%	2%	152	779	0%
	8 MB	6%	2%	152	778	0%
	16 MB	9%	4%	150	772	-1%
IBM	none			441	19990	
	4 MB	44%	22%	434	19712	-1%
	8 MB	52%	25%	434	19712	-1%
	16 MB	60%	29%	434	19692	-1%
NASA	none			1243	2859	
	4 MB	13%	12%	1286	2958	3%
	8 MB	51%	41%	1457	3352	17%
	16 MB	70%	53%	1507	3467	21%
Rice	none			1154	4499	
	4 MB	27%	22%	1221	4760	6%
	8 MB	41%	32%	1265	4933	10%
	16 MB	51%	39%	1292	5039	12%
World Cup	none			724	13857	
	4 MB	49%	30%	731	13987	1%
	8 MB	66%	40%	749	14322	3%
	16 MB	83%	50%	762	14577	5%

Table 4.5 : Improvement in server performance from caching. The server configuration is 2200/66 in Table 4.2.

First, systems that are limited by the achievable PCI bandwidth will see an improvement in web server performance commensurate with this reduction in traffic. Furthermore, simply employing a PCI bus with a faster clock speed is ineffective in removing contention on the bus as can be seen in Figure 4.7. Network interface data caching is, however, equally effective for both 33 MHz and 66 MHz PCI buses. Second, systems that are not limited by the achievable PCI bandwidth will be able to scale other resources in the system beyond the limits currently imposed by the local interconnect. Scaling other resources such as memory and CPU would increase contention on the PCI bus, and then the system can employ

network interface data caching to further improve performance. In both cases, the potential to extract greater performance from existing shared I/O interconnects makes more radical changes to local I/O interconnect designs less attractive because of the additional engineering costs they impose in redesigning motherboards, peripheral interfaces, interconnection components, and operating systems. Furthermore, the experimental results seem to suggest that the memory bandwidth limits server performance, rather than the local I/O interconnect bandwidth. Thus, scaling local I/O interconnects alone would be unproductive.

4.5.3 Host Processor Speed

For workloads like the IBM and World Cup traces, the server CPU seems to limit the throughput. The throughput improvements from the network interface data caching are minimal for these workloads. As CPU speeds increase, servers will be able to utilize a greater fraction of the PCI bus, and then the high PCI traffic reduction from network interface data caching should translate into greater throughput improvements. In this section, the network interface data caching is evaluated using the prototype server with a single Athlon 2600+ CPU that runs at 2.1 GHz, 300 MHz or 17% faster than Athlon 2200+ CPU (1.8 GHz).

Figure 4.17 shows PCI bus utilization for the server configuration 2600/33 in Table 4.2. Compared to Figure 4.6, the overall PCI bus utilization factors are same or a little higher despite the increased CPU speed. For the NASA and Rice traces, the bus utilization factors remain same at 94% and 88%. The server utilizes 46% and 64% of the PCI bus during the execution of the IBM and World Cup traces, only up by 3% and 1% compared to Figure 4.6. These disproportionate increases in the PCI bus utilization imply that increasing

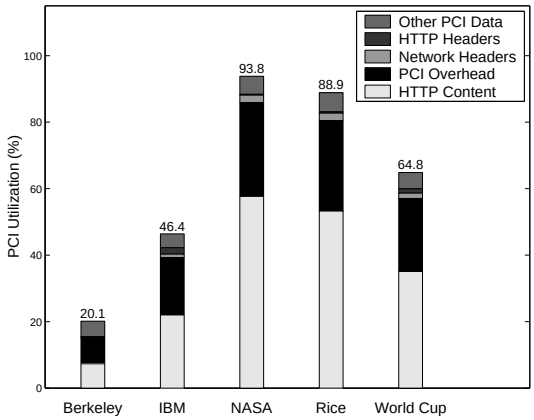

Figure 4.17 : Measured PCI bus utilization in a web server during the execution of the five web traces. The server configuration is 2600/33 in Table 4.2. The figure is in the same format as Figure 4.6.

CPU speeds is ineffective in improving server throughputs. Since the disk configuration is not changed, the use of the faster CPU does not affect the bus utilization of the Berkeley trace. As in Figure 4.6, the HTTP content and the PCI overhead account for roughly 60% and 30% of all PCI traffic, indicating that the CPU speed does not affect the composition of the PCI traffic.

Figures 4.18 and 4.19 show the PCI traffic reductions and throughput improvements from the network interface data caching for the server configuration 2600/33 in Table 4.2. Since the CPU speed does not affect network interface data cache hit rates, the reductions in the overall PCI traffic for all five traces are virtually same as the overall reductions for the server configuration 2200/33 shown in Figure 4.13 (they are off by at most 1%). Despite the increased CPU speed, the server achieves only about 4% greater improvements in throughput from caching than the server with the slower Athlon 2200+ CPU. With 16 MB caches, the server throughput improvements for the NASA, Rice, and World Cup traces are

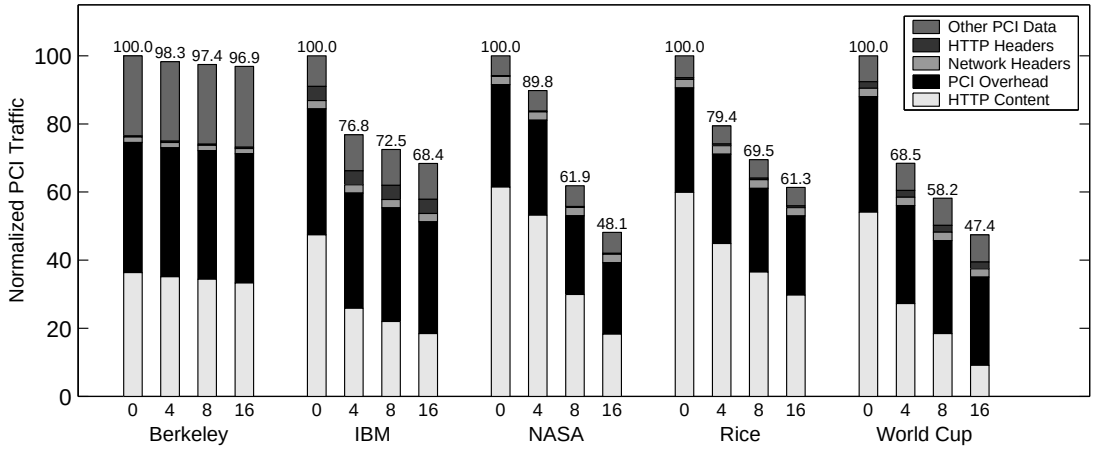

Figure 4.18 : Comparison of PCI traffic with and without network interface data caching. The server configuration is 2600/33 in Table 4.2. The cache sizes are same as those shown in Figure 4.13.

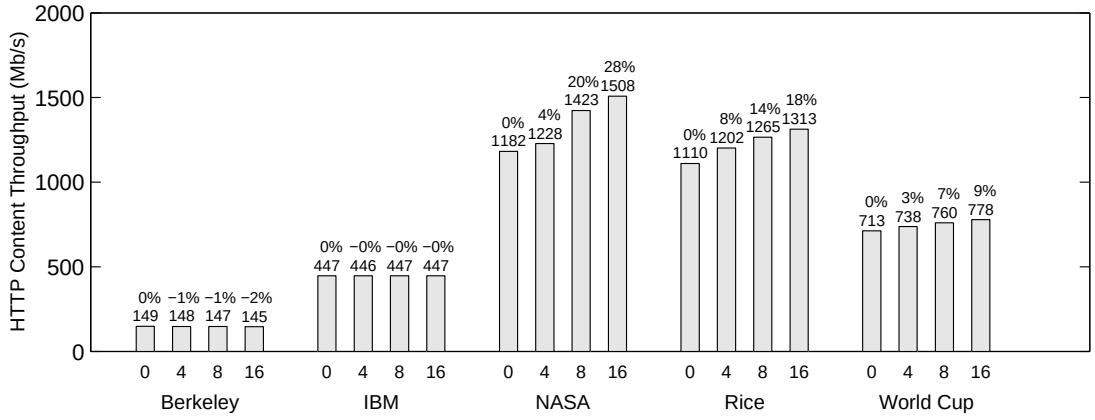

Figure 4.19 : Improvements in HTTP content thoughput from network interface data caching using the same server configuration as in Figure 4.18. Server throughputs are shown on top of the bars. The percent numbers show the throughput improvement over the 0 MB case (no caching).

28%, 18%, and 9%, as opposed to 25%, 14%, and 6% throughput improvements for the server configuration 2200/33 (See Table 4.4). The server throughputs also increase only marginally, about 30 Mb/s additional throughput over the server configuration 2200/33. The server throughput for the IBM trace increases by about 20 Mb/s but does not benefit

Trace	Cache Size per NIC	Reduction in HTTP Content	Reduction in PCI Overhead	Throughput (Mb/s)	Request Rate (req/s)	Cache Benefit
Berkeley	none			149	763	
	4 MB	4%	1%	148	758	0%
	8 MB	6%	1%	147	757	-1%
	16 MB	9%	1%	145	747	-2%
IBM	none			447	20303	
	4 MB	44%	9%	447	20263	0%
	8 MB	52%	10%	447	20301	0%
	16 MB	60%	11%	447	20274	0%
NASA	none			1182	2718	
	4 MB	13%	7%	1228	2825	4%
	8 MB	51%	23%	1423	3274	20%
	16 MB	70%	30%	1508	3470	28%
Rice	none			1110	4265	
	4 MB	27%	14%	1203	4690	8%
	8 MB	40%	20%	1265	4934	14%
	16 MB	51%	24%	1313	5121	18%
World Cup	none			713	13636	
	4 MB	49%	15%	738	14110	3%
	8 MB	66%	19%	760	14539	7%
	16 MB	83%	24%	778	14891	9%

Table 4.6 : Improvement in server performance from caching. The server configuration is 2600/33 in Table 4.2.

from network interface data caching. These throughputs again suggest that increasing CPU speed is not effective in improving server throughputs. Table 4.6 shows the actual throughput improvements as well as reductions in the HTTP content traffic and PCI overhead for the server configuration 2600/33.

The PCI bus utilization in the server with a faster 64-bit/66 MHz PCI bus (server configuration 2600/66) is shown in Figure 4.20. Again, despite the increased CPU speed, the overall PCI bus utilization during the execution of each trace is same or only slightly higher than that shown in Figure 4.7, which shows PCI bus utilization for the server configuration

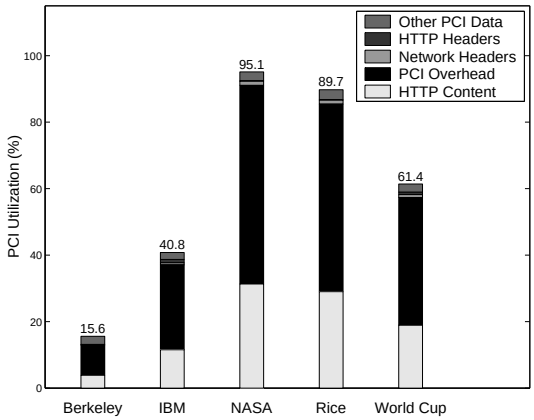

Figure 4.20 : Measured PCI bus utilization in a web server during the execution of the five web traces. The server configuration is 2600/66 in Table 4.2. The figure is in the same format as Figure 4.6.

2200/66. The overall bus utilization increases by at most 2%–3%. As discussed in the previous section, because of vastly increased PCI overhead, the use of the 64-bit/66 MHz PCI bus instead of the 64-bit/33 MHz PCI bus results in a very little decrease in the overall bus utilization (3%–5% for the Berkeley, IBM, and World Cup traces).

Figures 4.21 and 4.22 show the PCI traffic reductions and throughput improvements from network interface data caching for the server configuration 2600/66. As discussed previously, the CPU speed or PCI clock speed does should not affect the overall PCI traffic reductions. The reductions in the overall PCI traffic are almost identical to the reductions for the server configuration 2200/66 (See Figure 4.15). Finally, when compared to the server configuration 2200/66, the throughput improvements for the NASA, Rice, and World Cup traces increase only by 2%–3% resulting in 24%, 15%, and 8% improvements with 16 MB caches. These minimal increases in throughput improvements again indicate that increasing CPU speed does not effectively improve server throughputs. Since this server

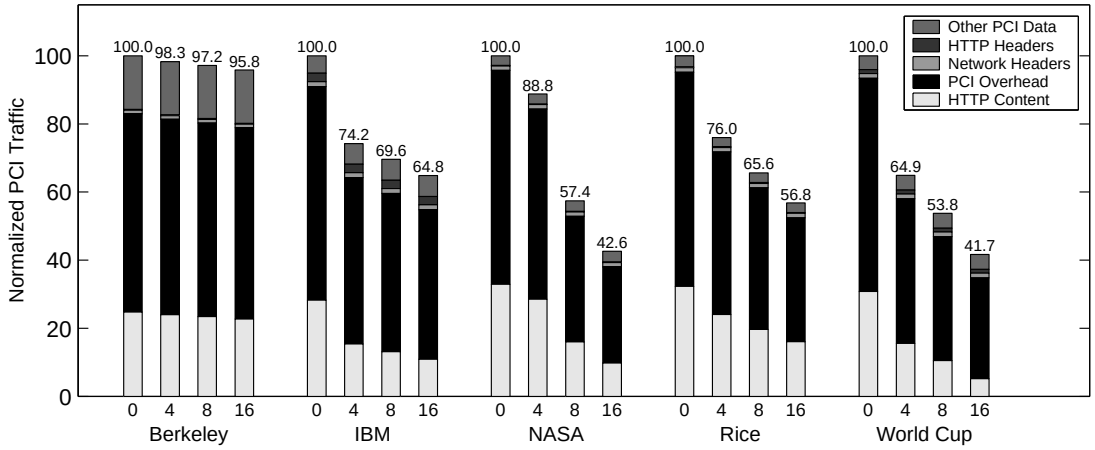

Figure 4.21 : Comparison of PCI traffic with and without network interface data caching. The server configuration is 2600/66 in Table 4.2. The cache sizes are same as those shown in Figure 4.13.

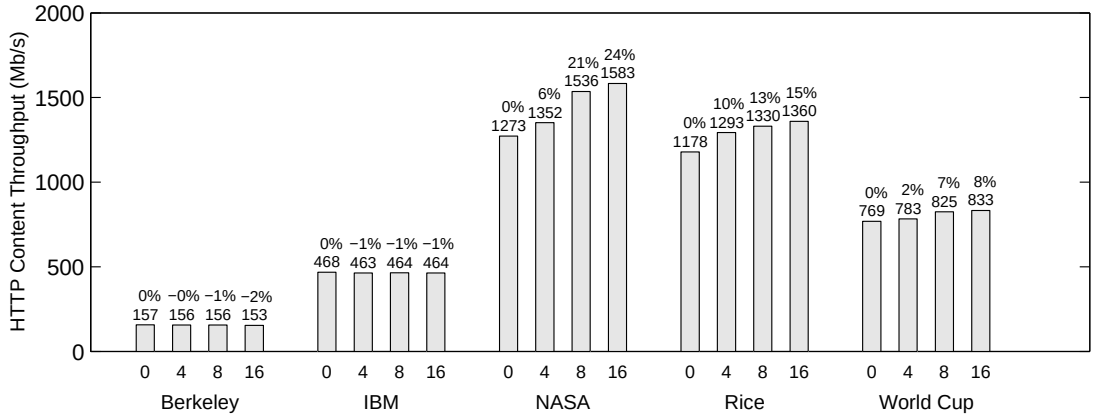

Figure 4.22 : Improvements in HTTP content thoughput from network interface data caching using the same server configuration as in Figure 4.21. Server throughputs are shown on top of the bars. The percent numbers show the throughput improvement over the 0 MB case (no caching).

has the fastest hardware, it does achieve the highest peak throughputs for all five traces. With 16 MB caches, the server achieves 1583 Mb/s and 1360 Mb/s for the NASA and Rice traces.

The results presented in this section show that the use of a faster CPU increases through-

Trace	Cache Size per NIC	Reduction in HTTP Content	Reduction in PCI Overhead	Throughput (Mb/s)	Request Rate (req/s)	Cache Benefit
Berkeley	none			157	805	
	4 MB	4%	1%	156	802	0%
	8 MB	6%	2%	156	801	-1%
	16 MB	9%	3%	153	786	-2%
IBM	none			468	21248	
	4 MB	44%	22%	463	21024	-1%
	8 MB	52%	26%	465	21079	-1%
	16 MB	60%	30%	464	21042	-1%
NASA	none			1273	2927	
	4 MB	13%	11%	1351	3110	6%
	8 MB	51%	41%	1536	3533	21%
	16 MB	70%	55%	1583	3641	24%
Rice	none			1179	4575	
	4 MB	27%	24%	1293	5040	10%
	8 MB	40%	34%	1330	5187	13%
	16 MB	51%	42%	1360	5288	15%
World Cup	none			769	14708	
	4 MB	49%	32%	783	14972	2%
	8 MB	66%	42%	825	15784	7%
	16 MB	83%	53%	833	15927	8%

Table 4.7 : Improvement in server performance from caching. The server configuration is 2600/66 in Table 4.2.

put improvements from network interface data caching, but only marginally. Consequently, increasing CPU speed seems ineffective in improving server throughputs. The evaluation of the network interface data caching using 64-bit/33 MHz and 64-bit/66 MHz PCI buses shows that the use of a faster I/O bus increases the transfer overhead rather than effectively reducing contention on the bus. The network interface data caching is shown to be equally effective for both types of PCI buses even though the 66 MHz PCI bus can theoretically provide twice the throughput of the 33 MHz PCI bus. As mentioned in Section 4.5.2, this result implies that scaling the PCI bus speed would be ineffective in improving server

performance.

4.6 Discussion

4.6.1 Local I/O Bottleneck

The results presented in Section 4.5 imply that local I/O interconnect and main memory are two significant performance limiters in networking servers that service most user requests from the main memory. First, increasing the clock speed of a PCI bus does not necessarily translate into increased I/O performance. Instead, the PCI bus may spend a large fraction of cycles waiting for the main memory. As discussed in Section 4.1.2, modern main memories built using DRAM require several cycles before producing requested data, during which wait cycles are injected onto the PCI bus. The PCI traces collected in the prototype web server during the execution of various web workloads show that the percentage of the main memory stall cycles increases as the PCI clock speed increases from 33 MHz to 66 MHz. Unless the main memory speed increases proportionally to the PCI clock speed, the PCI overhead would increase proportionally to the PCI clock speed. Second, the PCI bus width is unlikely to grow beyond 64 bits because of space constraints in the system. The future generation PCI-X bus is 64-bit wide and employs faster clock speeds [31]. Thus, main memory latencies are likely to restrict the effectiveness of future I/O interconnects such as the PCI bus that rely on fast clock speeds. Techniques such as network interface data caching that reduce both interconnect traffic and memory bandwidth will become more important in the future to address local I/O bottleneck.

4.6.2 Operating System Design Alternatives

A more radical design of the operating system could better exploit knowledge of the network interface data cache to shape performance decisions differently. For example, systems with multiple NICs are often *multi-homed,* with addresses on separate networks and multiple routes available to them to reach certain destinations. Such systems can use routing protocol information to construct a table of routing metrics to different networks. In contrast, this study uses multiple NICs on separate private subnets, leaving the server with no choice as to which NIC to use when sending data. The design of Section 4.3 assumes that the network interface is selected in a higher level of the networking stack, and the NIC device driver is responsible for determining if the data is cached. A more integrated design could use information about the likelihood of holding data in one network interface data cache rather than another to bias the routing decision for a flow of packets. If this decision is made accurately, such a strategy may improve effective cache capacity by reducing the likelihood of replicating the same data in multiple network interface data caches.

The prototype maintains cache coherence by lazily invalidating cached blocks that do not match the current file revision identifier. Other alternatives for coherence include update-based protocols that copy modified blocks to the cache or eager invalidation protocols that remove a block from the cache directory as soon as a modification takes place. Update protocols can reduce transmit latencies. However, updates may require extra PCI traffic if multiple writes take place before the data is read again. If the PCI bus is saturated, then extra PCI traffic for updates wastes valuable bandwidth that would otherwise be used to transfer user requests. Eager invalidation may allow for more intelligent cache replace-

ment policies if modified blocks are freed more rapidly. Both variants require additional code complexity since the operating system must also observe actions that write to cached blocks. However, such variants may be required in applications or deployments without valid revision identifiers.

4.6.3 Deployment in Other Applications

Network interface data caching has potential value beyond improving the throughput of web servers. Any networking server that sends repeated responses to frequent requests could make use of network interface data caching. While network interface data caching presented in this chapter focuses on web servers using TCP/IP, other application servers and protocols could also benefit from the technique. To be useful, the system must exhibit locality in the responses to network requests and have high enough bandwidth demands that local interconnect traffic is a bottleneck. For example, NFS and streaming media servers could potentially benefit from caching data within the network interface and reducing traffic over the local interconnect.

An NFS server potentially sends the same file out over the network several times without the file being modified, allowing caching at the network interface. However, NFS servers introduce additional complexity into the coherence protocol, as files may be updated remotely. The design must then make choices about whether to invalidate cached data on a write, have the host update the cached copy, or have the network interface recognize that incoming packets are updating data within its cache.

Streaming media servers come in two basic flavors: live broadcasts and on-demand streaming. Although a live stream has no temporal reuse of data, multiple simultaneous

clients receive the same content at the same time. In this model, the host processor would send a live block to the network interface data cache once. The clients viewing the stream would then have this block served to them from the cache, with IP and either TCP or UDP headers generated by the host. The block could be replaced immediately after delivery, since a live stream has no temporal locality beyond the length of a single frame of data. The potential benefit of caching is substantial since such a server is likely to be bandwidth bound, increasing the likelihood of a PCI bus bottleneck. Media servers for previously-recorded on-demand content may see benefits akin to those seen in web servers from repeated access to the same streams, particularly for short files such as MP3 audio files and video advertisements. However, full-length videos may have impractically large file sizes for any substantial caching.

4.7 Summary

Repeatedly transferring frequently-requested data across an expensive local interconnect leads to an inefficient use of system resources. Furthermore, interconnects scale more slowly than processing power or network bandwidth because of the need for standardized interfaces across devices. Caching data directly on a programmable network interface reduces local interconnect traffic on networking servers by eliminating repeated transfers of frequently-requested content. A prototype implementation of network interface data caching reduces PCI bus traffic by 35–58% on four web workloads with only 16 MB caches on two network interfaces. This technique allows application-level performance to scale with more aggressive CPUs and network links beyond the point at which less efficiently

utilized local interconnects would become a bottleneck. Such reductions in interconnect traffic only require a modest amount of DRAM in the network interface and impose no constraints on the network interface's processor.

Network interface data caching only requires the addition of five fields to the `mbuf` structures that refer to kernel data buffers, about 150 modified lines in the `sendfile` system call and `mbuf` manipulation routines, and roughly 850 lines of new code in the device drivers for the network interface. These simple additions to the operating system and 16 MB caches on two network interfaces enable web server throughput improvements of 6–25% for three web workloads studied, directly resulting from the reduction of data transfers from main memory to the network interface. Therefore, network interface data caching shows that programmable network interfaces, with little effort, can improve server performance by exploiting their storage capacity as well as their computation power.

Reductions in local interconnect traffic lead to reductions in main memory bandwidth. The experimental results using the prototype server of various configurations indicate that main memory latencies, rather than the bandwidth of local I/O interconnects, will likely limit the achievable server throughputs in the future. Future local I/O interconnects with faster clock speeds will be ineffective in improving server throughputs unless main memory latencies improve proportionally to the interconnect speeds. Thus, network interface data caching will be able to improve the efficiency of future local I/O interconnects with faster clock speeds as well as existing interconnects.

The processor utilization during the experiments indicates that the network processing consumes a large fraction of processor cycles, which may limit the server throughput. While network interface data caching reduces interconnect traffic, server throughput does

not improve if the host processor is saturated executing the network protocols and the device driver. As programmable network interfaces can alleviate local I/O interconnect bottleneck through network interface data caching, future programmable network interfaces would be able to assist the host processor by providing extended network services or offloading network protocol processing in order to reduce processing requirement on the host processor. Of course, no extended services on programmable network interfaces including network interface data caching would be useful unless programmable network interfaces provide sufficient computation power. Thus, the viability of any future services on programmable network interfaces is predicated on the success of efficient architectures for the interfaces and programming techniques such as the parallelization strategy described in Chapter 3 that exploit such architectures.

Chapter 5

Related Work

Network interface data caching introduced in the previous chapter is an example of extended services that programmable network interfaces can provide. While this technique directly aims to improve server performance by reducing local interconnect traffic, previous research has used programmable network interfaces primarily to improve general networking performance by reducing load on the host processor. This chapter examines some of previous work on improving networking performance using programmable network interfaces.

Network interfaces based on the Tigon and the Myricom LANai adapters are two widely used programmable network interfaces. The Myricom LANai adpaters are for a system area network called Myrinet [6], whereas the Tigon is an Ethernet interface controller as described in Section 3.1. The projects that used these programmable network interfaces range from implementing various optimizations for basic send and receive processing to implementing all or parts of network protocol processing on the network interface.

Gallatin et al. developed the Trapeze/IP firmware for LANai adapters, implementing various techniques to reduce the overhead of sending and receiving packets [16]. Techniques include zero-copy I/O, checksum offloading, message pipelining, and interrupt coalescing. Zero-copy I/O and checksum offloading are discussed several times throughout this thesis. They aim to reduce expensive memory operations. To facilitate zero-copy

socket, the Trapeze/IP firmware separates headers from packet payloads so that the kernel can remap the page containing packet payload into a page in the user space without copying the payload. An interrupt coalesching mechanism implemented on the NIC allows the NIC to generate an interrupt to the host processor after a certain number of packets have been sent or received or a certain time interval has elapsed. The purpose of the interrupt coalescing is then to reduce the number of interrupts to the host processor by letting it process multiple packets per interrupt rather than one. The Trapeze/IP firmware implements message pipelining in order to achieve low latency on large packets. The message pipelining enables the NIC to transmit packet as soon as the start of a packet is fetched to the NIC so that the latency of fetching the entire packet is saved. This mechanism is not compatible with IP checksum offloading because the checksum field resides in the header of packet, and checksum cannot be computed until entire packet is fetched. The Trapeze/IP firmware inserts the checksum in the tail of packet violating the IP packet format. The firmware and device driver correct the IP checksum field. By using large messages (much greater than standard Ethernet frames), the authors were able to achieve near-Gigabit rates on relatively slow machines.

In an attempt to give user applications direct access to network interfaces, Pratt and Fraser developed Arsenic [34]. Arsenic is a Gigabit Ethernet network interface with extended software interface. Arsenic provides virtual interfaces to user applications through which each application is given a flow independent from other applications. The network interface is then responsible for multiplexing and demultiplexing packets to appropriate virtual interfaces and deliver them to applications. Virtual interfaces also allow user-level network protocol processing such as TCP/IP protocol processing that is normally done in

the kernel. Arsenic prototype is built using a NIC based on the Tigon. The authors show that handling network protocols in the user space improves TCP/UDP throughputs. They also show that when multiple applications compete for a single NIC, an application that requires low latency can avoid congestion through the use of virtual interfaces.

Buonadonna and Culler developed Queue Pair IP, which replaces the traditional socket abstraction in order to reduce networking overhead for system area networks [9]. Queue Pair IP uses the existing Internet protocols as its transport layer, and the prototype implements components of TCP/IP directly in the LANai adapter. The host processor no longer handles TCP/IP protocols, thereby reducing load on the processor. The authors show that the Queue Pair IP requires very low CPU utilization and provides lower latency and higher bandwidth than either Gigabit Ethernet or Myrinet with the traditional socket layer.

Shivam et al. developed a message layer called Ethernet Message Passing (EMP) [36]. EMP is a special-purpose protocol to provide message passing interface using Ethernet. EMP is implemented on a NIC based on the Tigon and provides reliable messaging interfaces to applications. Similar to Queue Pair IP and Arsenic, EMP handles messaging protocols directly in the network interface, and the kernel does no protocol processing. EMP also provides zero-copy I/O to user applications avoiding expensive data copying. Experimental results indicate that EMP provides much lower latency than TCP and similar latency to the GM messaging system for Myrinet. Like EMP, the GM also implements protocol processing in the network interface (LANai adapters).

Shivam et al. later implemented a parallelized version of the EMP protocol on the Tigon [37]. Their parallelization strategy is similar to that described in Section 3.2. Based on their experimental results, they argue that parallelizing the receive processing of the

EMP protocol is most effective. It is unclear how the functions of the receive processing are distributed across multiple processors. Furthermore, EMP is a specialized user-level protocol, whereas the parallelization described in Chapter 3 deals with parallelization of Ethernet firmware. Thus, performance of the parallelized Tigon firmware described in Section 3.2 cannot be compared to that of the parallelized EMP. Moreover, the parallelization strategy for Ethernet firmware has a broader applicability since it can improve the throughput of standard IP-based protocols and applications.

Except Arsenic, the projects described above and other similar projects focus on system area or cluster networks. In summary, they seek to reduce the amount of processing required on the host processor to send and receive packets. The techniques typically consist of a combination of zero-copy I/O, the use of user-level protocol processing (OS bypass), and protocol implementation offloaded onto the network interface.

Finally, the Quadrics network uses programmable network interfaces called Elan to support specialized services for high performance clusters [32]. Like Myrinet, the Quadrics network provides high performance messaging system using special-purpose network interfaces, switches, and links. However, the Quadrics network provides much greater transmission bandwidth of 400 MB/s as opposed to 250 Mb/s (2 Gb/s) of Myrinet. The Quadrics network aims to achieve high performance by creating a global distributed shared virtual memory space through hardware support. The Elan network interface includes two programmable processors (one as thread controller and the other as thread processor), 8 KB cache, 64 MB on-board SDRAM, and memory management unit. The programmable processors implement messaging libraries, and the memory management unit can translate virtual addresses into local or PCI addresses. This memory translation capability on Elan

along with the operating system support allow Elan and host processor to share a region of host processor's virtual memory space, providing a simple zero-copy I/O interface to user-level libraries. Thus, this memory translation capability on Elan and global shared memory allow user-level applications to initiate zero-copy DMA transfers to a remote host (remote DMA) using virtual memory addreses.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Networking servers, such as web servers, have become an integral part of the Internet. The current trends of computing seem to indicate that the global network will continue to grow and demand higher-performance networking. Performance of networking servers have improved substantially in recent years. However, as high-speed networks such as Gigabit Ethernet become common, networking servers require enormous system resources to fully utilize these high-speed network links. Adding programmability and storage capacity to network interfaces provides new opportunities to extend network services performed by the network interfaces. Programmable network interfaces can improve networking server performance by offloading network protocol processing or providing novel network services to address performance issues in networking servers.

However, programmable processors must be embedded on a peripheral device. The performance of programmable network interfaces may suffer from instruction processing due to technology constraints such as limited cooling area and power available on a peripheral device. Chapter 3 examines the Tigon programmable Gigabit Ethernet controller based on a multiprocessor architecture. It shows that the Tigon firmware can effectively exploit task-level concurrency found in network interface processing. The parallelized ver-

sion of the firmware improves throughput for real network services by 32–107%. This improved performace falls within 10–20% of a modern ASIC-based network interface. Considering that the Tigon was released in 1997, modern process technology would enable greater performance than ASIC-based network interfaces. Thus, with proper architectures and software implementation strategies, programmable network interfaces can overcome the performance disadavntages over ASIC-based network interfaces that result from programmability.

Chapter 4 then presents network interface data caching, a novel network service on programmable network interfaces that exploit their storage capacity as well as computation power in order to alleviate local interconnect bottleneck. This technique allows frequently requested content to be cached directly on the network interface. A modest amount of DRAM (16 MB or less) on the network interface can effectively reduce local interconnect traffic, alleviating contention on local I/O interconnects. A prototype implementation of network interface data caching shows that using 16 MB caches, a PC-based web server achieves 35–58% reductions in the overall PCI bus traffic for four out of five workloads used to evaluate network interface data caching. These reductions result in throughput improvements of 6–25% for three workloads and the peak throughput of 1583 Mb/s. These results show that programmable network interfaces with storage capacity can address performance issues in networking servers and improve server performance.

Thus, this thesis shows that programmable network interfaces can improve networking server performance by exploiting the flexibility of programmability and storage capacity without the performance disadvantage over ASIC-based network interfaces that results from programmability. Consequently, programmable network interfaces with storage ca-

pacity will be a viable approach to addressing performance issues in networking servers and warrant further research in order to explore and develop extended network services. Finally, as discussed in Section 3.5, future programmable network interfaces for network links running at 10 Gb/s or beyond would have to resort to multiprocessor architectures due to technology constraints on a peripheal device. Thus, futher research in computer architecture will also be required to develop efficient multiprocessor architectures for future programmable network interfaces and to ensure the success of these interfaces in the future.

6.2 Future Work

Existing programmable network interfaces perform a rather restricted set of functions. As discussed in Chapter 5, the current use of programmable network interfaces is largely to reduce load on the host processor by either exposing hardware to user-level applications or offloading parts of the network stack or messaging library. With proper hardware support, future programmable network interfaces may perform a broader range of functions. The use of programmable network interfaces can range from implementing intelligent packet filters to possibly providing application layer services to the host processor.

Modern network interfaces provide a filtering mechanism in which packets are filtered at the network interface based on simple rules. Thus, filtered packets are not transferred to main memory saving memory bandwidth and load on host processor. Implementing more intelligent filters that act as a firewall directly on the network interface would improve server performance by preventing unwanted packets from entering the system. For instance, the network interface could implement a mechanism to avoid receive livelock [24]

or detect attacks from malicious users. Since the network interface is the source of asynchronous interrupts, avoiding receive livelock through controlled interrupts would be more easily implemented on the network interface than on the host processor.

Researchers have speculated that network processing in the kernel such as executing the TCP/IP protocols is a potential bottleneck in networking servers and that copying operations are a primary cause [11, 13, 19]. The use of a zero-copy `sendfile` system call in fact improves the prototype server performance by over 20%, as shown in Section 4.4.4. However, the experimental results using a prototype server with network interface data caching shows that even with zero-copy I/O, network processing still requires enormous CPU power to utilize high-speed network links. Most current efforts to alleviate this problem seem to have a common approach; process network protocols anywhere but in the kernel. For instance, the Arsenic and GM provide user-level protocol processing while EMP and Queue Pair IP implement protocols on the network interface (refer to Chapter 5 for descriptions of these projects). Conceptually, transport layer protocols such as TCP handle end-to-end connections. It is logical for the network interface that enables physical network communication to expand its role and provide transport layer interfaces to the host processor. However, in order to implement efficient offloading of network processing, it will be necessary to determine the exact sources of the inefficiency of current network processing on the host processor.

As discussed in Section 3.5, programmable network interfaces would rely on multiprocessor architectures in the future. Offloading network protocol processing onto the network interface may require parallelizing network protocols. Others have studied increasing networking performance by parallelizing network protocols on general-purpose multipro-

cessor operating systems [5, 17, 27]. Such parallelization schemes exploit concurrency at various levels, such as across packets, protocol layers, and connections. With proper hardware support, offloading network protocols processing would be feasible without loss of performance by exploiting these types of concurrency.

Recently, several companies have started manufacturing TCP offload engines (TOE). Various types of TOEs offload parts or the entire TCP/IP protocol stack onto ASIC-based chips. Currently the primary purpose of TOEs seems to be improving performance of storage servers that use the iSCSI protocol. Since iSCSI depends on the traditional TCP/IP network stack, offloading it onto a dedicated chip on the network interface reduces load on the host processor. A main reason behind the use of ASIC-based chips seems that these chips can easily exploit packet-level parallelism. Programmable network interfaces with multiple processors would also allow firmware to realize the same benefits from exploiting packet-level parallelism.

Finally, the use of programmable network interfaces can extend beyond network protocol processing. They may implement simple applications to assist server applications running on the host processor. For instance, a simple web server on the network interface could handle user requests to a few popular web pages. Then, the host processor can handle user requests that require more processing. In fact, there are various types of embedded web servers built using simple microcontrollers [7] including the Stanford Matchbox web server [12]. More powerful programmable network interfaces would be able to assist busy web servers by offloading part or all of the HTTP processing.

Bibliography

[1] Alteon Networks. *Tigon/PCI Ethernet Controller*, August 1997. Revision 1.04.

[2] Alteon WebSystems. *Gigabit Ethernet/PCI Network Interface Card: Host/NIC Software Interface Definition*, July 1999. Revision 12.4.13.

[3] Martin F. Arlitt and Carey L. Williamson. Web Server Workload Characterization: The Search for Invariants. In *Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 126–137. ACM Press, 1996.

[4] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the USENIX 1999 Annual Technical Conference*, pages 253–265, June 1999.

[5] Mats Björkman and Per Gunningberg. Locking effects in multiprocessor implementations of protocols. In *Proceedings of the ACM SIGCOMM '93 Conference*, pages 74–83. ACM Press, 1993.

[6] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE MICRO*, 15(1):29–36, 1995.

[7] Gaetano Borriello and Roy Want. Embedded Computation Meets the World Wide Web. *Communications of the ACM*, 43(5):59–66, 2000.

[8] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Schenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of IEEE INFOCOM '99*, volume 1, pages 126–134, March 1999.

[9] Philip Buonadonna and David Culler. Queue Pair IP: A Hybrid Architecture for System Area Networks. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 247–256, May 2002.

[10] Pei Cao and Sandy Irani. Cost-Aware WWW Proxy Caching Algorithms. In *Proceedings of the 1997 USENIX Symposium on Internet Technology and Systems*, pages 193–206, December 1997.

[11] C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley. Afterburner. *IEEE Network*, pages 36–43, 1993.

[12] Greg DeFouw and Vaughan Pratt. The Matchbox PC: A Small Wearable Platform. In *Proceedings of the Third International Symposium on Wearable Computing*, pages 172–175, October 1999.

[13] Peter Druschel. Operating System Support for High-Speed Communication. *Communications of the ACM*, 39(9):41–51, 1996.

[14] Peter Druschel and Larry L. Peterson. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the 14th Symposium on Operating Systems Principles (SOSP-14)*, pages 189–202, December 1993.

[15] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol - HTTP 1.1. IETF RFC 2616, June 1999.

[16] Andrew Gallatin, Jeff Chase, and Ken Yocum. Trapeze/IP: TCP/IP at Near-Gigabit Speeds. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*, June 1999.

[17] Norman C. Hutchinson and Larry L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[18] Intel Corporation. *Intel Pentium 4 Processor with 512-KB L2 Cache on 0.13 Micron Process at 2 GHz-3.06 GHz, with Support for Hyper-Threading Technology at 3.06 GHz Datasheet*, January 2003.

[19] Jonathan Kay and Joseph Pasquale. Profiling and Reducing Processing Overheads in tcp/ip. *IEEE/ACM Transactions on Networking (TON)*, 4(6):817–828, 1996.

[20] Karl Kleinpaste, Peter Steenkiste, and Brian Zill. Software Support for Outboard Buffering and Checksumming. In *Proceedings of the ACM SIGCOMM '95 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 87–98, August 1995.

[21] Jonathan Lemon. Kqueue: A generic and scalable event notification facility. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, June 2001.

[22] A.J. McGregor, H-W Braun, and J.A. Brown. The NLANR Network Analysis Infrastructure. *IEEE Communications Magazine*, pages 122–128, May 2000.

[23] Larry McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 USENIX Technical Conference*, pages 279–295, January 1996.

[24] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-Driven Kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, 1997.

[25] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM SIGOPS Operating Systems Review*, 34(2):24–25, 2000.

[26] Erich M. Nahum, Tsipora Barzilai, and Dilip Kandlur. Performance Issues in WWW Servers. *IEEE/ACM Transactions on Networking*, 10(2):2–11, February 2002.

[27] Erich M. Nahum, David J. Yates, James F. Kurose, and Donald F. Towsley. Performance Issues in Parallelized Network Protocols. In *Proceedings of the Operating Systems Design and Implementation*, pages 125–137, 1994.

[28] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, pages 199–212, June 1999.

[29] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. I/O-Lite: A Unified I/O Buffering and Caching System. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation*, pages 15–28, February 1999.

[30] PCI Special Interest Group. *PCI Local Bus Specification*, December 1995. Revision 2.2.

[31] PCI Special Interest Group. *PCI-X Addendum to the PCI Local Bus Specification*, December 1999. Revision 1.0.

[32] Fabrizio Petrini, Wu-Chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The Quadrics Network: High-Performance Clustering Technology. *IEEE MICRO*, 22(1):46–57, January 2002.

[33] Jef Poskanzer. *thttpd - tiny/turbo/throttling HTTP server*. Acme Labs, February 2000. Unix manual page.

[34] Ian Pratt and Keir Fraser. Arsenic: A User-Accessible Gigabit Ethernet Interface. In *Proceedings of IEEE INFOCOM '01*, pages 67–76, 2001.

[35] Julian Satran, Kalman Meth, Costa Sapuntzakis, Mallikarjun Chadalapaka, and Efri Zeidner. iSCSI. IETF Internet draft `draft-ietf-ips-iscsi-14.txt`, work in progress, July 2002.

[36] Piyush Shivam, Pete Wyckoff, and Dhabaleswar Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (SC2001)*, November 2001.

[37] Piyush Shivam, Pete Wyckoff, and Dhabaleswar Panda. Can User-Level Protocols Take Advantage of Multi-CPU NICs? In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02)*, pages 64–69, April 2002.

[38] Gary Tyson, Matthew Farrens, John Matthews, and Andrew R. Pleszkun. A Modified Approach to Data Cache Management. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 93–103, December 1995.

[39] Ken Yocum and Jeff Chase. Payload Caching: High-Speed Data Forwarding for Network Intermediaries. In *Proceedings of the 2001 Annual USENIX Technical Conference*, pages 305–317, June 2001.