RICE UNIVERSITY

TCP Offload through Connection Handoff

by

Hyong-youb Kim

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

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

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

Alan L. Cox
Associate Professor of Computer Science

Edward W. Knightly
Professor of Electrical and Computer
Engineering

T. S. Eugene Ng
Assistant Professor of Computer Science and
Electrical and Computer Engineering

Houston, Texas

August, 2006

UMI Number: 3256706

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI®

UMI Microform 3256706

Copyright 2007 by ProQuest Information and Learning Company.

All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

TCP Offload through Connection Handoff

Hyong-youb Kim

Abstract

TCP offload is a technique to improve TCP/IP networking performance of a network computer system by moving (parts of) TCP processing from the host processor to the network interface. There are several ways to achieve offload. The typical full offload moves all TCP functionalities to the network interface, and TCP processing is performed exclusively on the network interface. However, when the network interface has limited processing power, full offload creates a bottleneck at the network interface and degrades system performance. In contrast, TCP offload based on connection handoff allows the operating system to move a subset of connections to the network interface. This way, both the host processor and the network interface perform TCP processing, and the operating system can control the amount of work performed on the host processor and the network interface. Thus, by using connection handoff, the system can fully utilize the processing power of the network interface without creating a bottleneck in the system.

This dissertation presents a design, implementation, and evaluations of handoff-based TCP offload. The design enables the application to transparently exploit offload-capable network interfaces. The prototype implementation shows that the operating system can support offload without complicating the existing software architecture. The evaluations show that significant performance gains are possible for various web workloads and across a range of processing capabilities of the network interface.

Acknowledgments

A big thank-you goes to my adviser, Prof. Scott Rixner. He has guided me through good and difficult times, and without him, this thesis would have been impossible. This thesis topic was born out of my somewhat cavalier attitude and then became something of a race against time and available resources. I thank Prof. Alan L. Cox for his interests in this work, helpful advice, and comments on various papers, not to mention serving on my thesis committee. Prof. Vijay Pai, who is now at Purdue University, helped me in many ways during my early years at Rice University. I thank Prof. Edward W. Knightly and Prof. T. S. Eugene Ng for serving on my thesis committee. They provided me with valuable comments to improve this work.

Advanced Micro Devices (AMD) donated processors and financially helped purchase hardware equipment. Virtutech generously donated a site license for the Simics simulator. The systems group in Computer Science allowed me to use a cluster of Opteron-based machines for running simulations.

I believe the most valuable opportunity that schools offer is a chance to meet and learn from people who are better than I am. Although I do not mention specific names, there are numerous faculty members and fellow students whom I have learned from directly and indirectly. I am grateful for the financial support from various institutions for the past five years. Finally, I thank my parents for their financial and emotional support during my undergraduate and graduate years.

Contents

Abstract	iii
Acknowledgments	iv
List of Illustrations	viii
List of Tables	xi
1 Introduction	1
1.1 TCP/IP Network Stack Performance Issues	1
1.2 TCP Offload	2
1.3 Offload through Connection Handoff	4
1.4 Contributions	5
1.5 Organization	7
2 Related Work	8
2.1 TCP/IP Network Stack Performance	8
2.2 TCP Offload	11
2.3 Commercial TCP Offload NIC and Operating System Support	15
2.4 Connection Handoff	18
2.5 Summary	20
3 Performance Issues of Modern TCP/IP Network Stack	21
3.1 Network Stack Operations	21
3.2 Experimental Setup	25
3.3 Performance Analysis	27
3.4 Software Prefetching	33

3.5	Summary	36
4	Offload through Connection Handoff	37
4.1	Rationale	37
4.2	Design of Connection Handoff	39
4.2.1	Architecture	39
4.2.2	Interface	42
4.2.3	Using the Handoff Interface	47
4.3	Connection Handoff Framework	48
4.3.1	Priority-based Packet Processing	50
4.3.2	Load Control	52
4.3.3	Connection Selection	55
4.4	Experimental Setup	59
4.4.1	Prototype	59
4.4.2	Simulation Setup	61
4.4.3	Web Workloads	62
4.5	Prototype Results	63
4.6	Simulation Results	68
4.6.1	Priority-based Packet Processing and Load Control	68
4.6.2	Connection Selection Policy	73
4.6.3	NIC Speed	75
4.7	Summary	76
5	Impact of Processor Speed, Cache Size, and Offload	78
5.1	Experimental Setup	78
5.2	Host Processor Speed	81
5.3	L2 Cache Size	84
5.4	Offloading	87
5.5	Discussion	94

5.6	Summary	96
6	Placement Options for the Offloading Processor	98
6.1	As Part of the Host Processor Die	99
6.2	As Part of the Chipset	101
6.3	As Dedicated Processor Node	102
6.4	As Part of the Traditional PCI Network Interface	102
6.5	Discussion	103
6.6	Summary	104
7	Implementing Connection Handoff in FreeBSD and Linux	105
7.1	Similarities in FreeBSD and Linux Network Stacks	105
7.2	Implementation Difficulties	108
7.3	Summary	110
8	Conclusions	111
	Bibliography	115

Illustrations

3.1	Operations within the TCP/IP network stack	22
3.2	Execution profiles of the network stack during the execution of TCP Send microbenchmark.	28
3.3	Execution profiles of the network stack while the server executes the World Cup trace.	30
3.4	Execution profiles of the network stack while the server executes the IBM trace.	31
3.5	Execution profiles of the network stack while the server executes SPECweb99.	32
4.1	TCP/IP network stack extended to support connection handoff.	40
4.2	Important data structures of a connection.	41
4.3	Handoff commands provided by the NIC.	43
4.4	Handoff commands provided by the operating system.	45
4.5	Example use of handoff commands	47
4.6	Framework for connection handoff.	49
4.7	State machine used by the NIC firmware to dynamically control the number of connections on the NIC.	54
4.8	Distribution of connection lifetimes from SPECweb99 and the IBM and World Cup traces.	58
4.9	Execution profiles of the network stack of the prototype.	64
4.10	Messages exchanged across the local I/O interconnect.	65

4.11	Profiles of the simulated web server during the execution of the World Cup and IBM traces.	68
4.12	Dynamic adjustment of the number of connections on the NIC by the load control mechanism for configuration <i>W-4096-P-L-FCFS-400</i>	71
4.13	Profiles of the simulated web server during the execution of SPECweb99 Static and regular SPECweb99.	72
4.14	Profiles of the simulated web server when the operating system hands off connections in a first-come, first-served order or by using thresholds.	73
4.15	Profiles of the simulated web server when the NIC has a small number of connections and threshold-based connection selection is used.	74
4.16	Profiles of simulated web server when the instruction rate of the NIC is increased.	75
5.1	Web server throughput for the World Cup trace as a function of the host processor speed.	82
5.2	Web server throughput for the IBM trace as a function of the host processor speed.	82
5.3	Web server throughput for SPECweb99 with only static content as a function of the host processor speed.	83
5.4	Web server throughput for SPECweb99 with both static and dynamic content as a function of the host processor speed.	83
5.5	Web server throughput for the World Cup trace as a function of the L2 cache size.	84
5.6	Web server throughput for the IBM trace as a function of the L2 cache size.	84
5.7	Web server throughput for SPECweb99 with only static content as a function of the L2 cache size.	86
5.8	Web server throughput for SPECweb99 with both static and dynamic content as a function of the L2 cache size.	86

5.9	Web server throughput for the World Cup trace achieved with and without offloading while the host processor speed and the L2 cache size vary. . . .	87
5.10	Web server throughput for the IBM trace achieved with and without offloading while the host processor speed and the L2 cache size vary. . . .	91
5.11	Web server throughput for SPECweb99 achieved with and without offloading while the host processor speed and the L2 cache size vary. In this experiment, SPECweb99 uses only static content.	92
5.12	Web server throughput for SPECweb99 achieved with and without offloading while the host processor speed and the L2 cache size vary. In this experiment, SPECweb99 uses both static and dynamic content. . . .	93
6.1	System architectures and placement options for the offloading processor . .	99
7.1	Simplified view of TCP/IP network stack layers of FreeBSD and Linux and main data structures used within the network stack.	106

Tables

3.1	Impact of software prefetching on web server throughput	34
4.1	Impact of a heavily loaded NIC on the networking performance of a simulated web server.	50
4.2	Number of packets sent by connections from the same workloads used in Figure 4.8.	58
4.3	Simulator configuration.	61
4.4	Profile of the firmware.	67
4.5	Configuration shorthands used in Section 4.6	69
5.1	Simulator configuration.	79

Chapter 1

Introduction

Network servers such as web servers have become increasingly important due to their ubiquitous use. These servers typically use the standard TCP/IP protocol suite in order to operate on the Internet. As the number of networking applications and clients, and network bandwidth continue to grow, TCP networking performance of servers must also improve accordingly. This dissertation aims to address performance issues of TCP processing within a modern network server.

1.1 TCP/IP Network Stack Performance Issues

In current systems, the operating system running on the CPU implements the TCP/IP network stack, and TCP processing is performed on the CPU. Network servers may spend a significant fraction of CPU cycles processing TCP packets within the network stack. For instance, web servers can spend more than 60% of CPU cycles executing the network stack. Ideally, the system should spend as little time as possible processing TCP packets and spend more time running the application.

As past research has identified, packet processing time is dominated by CPU stalls on main memory accesses, not by the execution of the instructions involved in packet processing [24, 25]. The fact that the CPU speed has been increasing faster than main memory speed makes the problem worse, since the CPU must stall longer on main memory access. It can easily take over 200 cycles for a modern CPU to access main memory. In a test machine, processing a packet within the network stack requires only around 3000 instructions. So, just a few main memory accesses can increase packet processing time significantly.

Previously, data touching operations (checksum computation and data copy) were the

main cause for main memory accesses. The network stack computes TCP and IP checksums for each packet received and sent. It also copies user data between the kernel and user memory spaces. Both operations involve touching every byte of the data, and can incur a large number of main memory accesses when the data is not in the CPU caches. In order to eliminate these operations, checksum offload and zero-copy I/O techniques have been proposed [23, 28, 48, 70]. Using checksum offload, the network interface card (NIC), instead of the CPU, computes checksums while it transfers packets. Zero-copy I/O avoids data copies between the user and kernel memory spaces. So, these techniques can completely eliminate data touching operations. Almost all NICs now support checksum offload, and many operating systems implement some form of zero-copy I/O.

While modern network stack implementations largely avoid having to access packet data, there is another type of main memory accesses within the network stack that has not been studied previously. These are due to connection data structures such as protocol control blocks that store TCP connection states. Because TCP is a stateful connection-based protocol, the operating system must store connection states in main memory, and accesses to connection states are an integral part of TCP processing. Modern network servers need to be able to handle a large number of simultaneous connections. However, just several thousand connections easily overwhelm today's CPU caches, and accesses to connection states lead to main memory accesses. Experimental results show that these main memory accesses can now be as bad as data touching operations in terms of CPU cycles wasted on memory stalls. Unlike data touching operations, main memory accesses due to connection states cannot be eliminated because they are part of TCP processing. The gap between the CPU speed and main memory speed continues to grow, and the main memory latency problem within the network stack will become worse in the future.

1.2 TCP Offload

TCP offload is a technique that improves the TCP networking performance of a server by moving TCP processing from the host processor to the offloading processor on the

NIC. Colocating the offloading processor and a local memory close to each other on the NIC enables fast memory access for the offloading processor. The offloading processor would then exploit such fast local memory to store connections and process packets more efficiently than the host processor, which suffers from large main memory access times. In addition, hardware acceleration features such as hardware-based de-multiplexing can help the offload processor to further improve its efficiency. Since TCP processing is moved from the host processor, offloading reduces the instruction and memory bandwidth of the host processor, which can then spend more time executing applications and other parts of the operating system.

Despite the performance benefits of offloading, it also has problems. Typical full offload moves all TCP functionalities to the NIC, and TCP processing occurs only on the NIC. However, this full offload approach has problems including creating a potential bottleneck at the NIC, difficulties in designing the offload interface between the operating system and the NIC, modifying the existing network stack implementations [63]. For example, because the NIC has limited area and power available to it, the offloading processor on the NIC also has inherently limited compute power. So, the offloading processor on the NIC may simply become a bottleneck in the system, which then degrades overall performance. Port numbers and IP routing also complicate the existing software architecture with full offload. Port numbers are a global, system-wide resource. When the NIC actively establishes a connection, it must coordinate allocation with the operating system. The IP routing is another problem. The NIC must also be aware of the system routing table so that it can transmit packets to correct destinations through a right interface. Thus, offloading must be achieved in a way that allows the system to control the amount of work performed on the offloading processor (to avoid creating a bottleneck) and that does not overly complicate the existing software architecture.

1.3 Offload through Connection Handoff

The system can employ connection handoff in order to achieve the performance benefits of offload while avoiding the problems associated with full offload. Connection handoff is a technique that moves the state of a TCP connection from one processing element to another, which then takes over TCP processing for that connection. Thus, in order to achieve offload using connection handoff, the operating system first establishes a connection and then attempts to hand it off from the host processor to the offloading processor. Once the connection is handed off, the offloading processor performs TCP for that connection, thereby realizing the performance benefits of offload. In this way, the operating system is still responsible for connection establishments and routing. Since established connections already have correct ports and routes, the offloading processor only needs to perform data transfers through TCP. Either the operating system or the NIC can also restore the connections that are previously handed off to the NIC. For instance, when the operating system detects a route change, it can reclaim the connections from the NIC. Likewise, the NIC may ask the operating system to handle **some** of its connections when it runs low on resources and experiences performance loss. More importantly, connection handoff provides the operating system with a mechanism to control the amount of work performed on the host processor and the NIC by controlling **the** number of connections handed off to the NIC. This ability to control the number of connections on the NIC enables the system to avoid creating a bottleneck at the NIC. The system can increase or decrease the number of connections handed off to the NIC depending on the current utilization level of the resources on the NIC.

This dissertation presents a handoff-based framework for utilizing the offloading processor in order to achieve the advantages of connection handoff outlined above. Moreover, the framework for connection handoff does not require substantial changes to the existing network stack architecture. At minimum, it consists of the bypass layer and the extended device driver interface. The bypass layer simply forwards user requests from the application to the offloading processor through the extended device driver interface. Prototypes have been built using the FreeBSD and Linux operating systems, showing that connection

handoff is applicable across different systems.

This dissertation also presents several policies for using connection handoff effectively. The system can use these policies in order to fully utilize the offloading processor without overloading it. Specifically, they involve (1) prioritizing packet processing at the NIC in order to ensure performance of the connections that are processed by the host processor, (2) monitoring the load on the NIC and dynamically controlling the number of connections to avoid overloading the NIC, and (3) selecting connections and handing off those that would better make use of the resources on the NIC. These policies are straightforward to implement. The experiments with web server show that the system successfully avoids performance degradation when the offloading processor lack compute power, and that as the compute power of the offloading processor increases, the system achieves substantial throughput improvements.

Finally, the framework for connection handoff presented in this dissertation is independent of the location of the offloading processor. The offloading processor is assumed to be located on the NIC, as it is the most logical placement in current system architectures. However, it may be implemented in other places such as the host processor die and the system chipset. Regardless of the location, connection handoff can be used to control the amount of work performed on the offloading processor.

1.4 Contributions

This dissertation builds upon several publications by Kim and Rixner [45, 46, 47]. Key contributions are summarized as follows.

Analyzing the performance of the TCP/IP network stack. Past research has established that data touching operations (data copy and checksum calculation) are a bottleneck in the network stack, because of growing main memory latencies. Solutions to eliminate data touching operations are also known. Checksum offload and zero-copy I/O techniques can eliminate main memory accesses due to data touching operations. This dissertation shows that a large number of connections cause cache misses, and the resulting main mem-

ory accesses to connection structures can be as bad as data touching operations. Such main memory accesses are becoming a bottleneck in the network stack.

Defining a connection handoff interface. Offloading TCP processing to the network interface is a solution to improve system performance by reducing memory and instruction bandwidth on the host processor. The offload processor on the NIC can employ fast local memory to access connections quickly and thereby process packets more efficiently than the host processor. Full offload has been studied previously, but it can complicate the software architecture of the existing network stack and has the potential that the offload NIC itself becomes a bottleneck in the system. This dissertation defines a software interface between the operating system and the offload NIC that is based on connection handoff. Using connection handoff, the operating system moves established connections to the NIC. The operating system can control the division of work by controlling the number of connections that are processed by the host processor and the offload NIC. Moreover, supporting handoff does not require significant architectural changes to the existing network stack. So, connection handoff enables the system to achieve benefits of offload while avoiding the problems of full offload.

Developing a framework and policies for effectively utilizing offload NIC. In addition to defining the handoff interface, this dissertation shows a set of policies for using the handoff interface. These policies help the system ensure the following. First, offloading never degrades performance of connections that are handled by the host processor. Second, the offload NIC never becomes a bottleneck in the system. Third, the operating system hands off connections selectively in order to maximize performance benefits from offloading. By ensuring these properties, the system can use the offload NIC as an acceleration coprocessor and hand off to the NIC as much work as its resources will allow.

Evaluating the performance impact of offload. This dissertation quantitatively evaluates the impact of offload on performance of web server. First, it evaluates how individual handoff policies mentioned above affect server performance. Then, it evaluates performance improvements from offload for a range of the NIC speeds, the host processor speeds,

and the cache sizes. Such evaluations help to see how offloading would benefit in the future and also to compare performance gains from offloading against those from faster host processors.

1.5 Organization

This dissertation is organized as follows. Chapter 2 discusses previous work related to this dissertation. Chapter 3 presents an overview of operations within the TCP/IP network stack and describes current performance issues. The detailed profile data presented in this chapter motivates the use of offloading. Chapter 4 presents a design and evaluation of TCP offload based on connection handoff. This chapter is the main focus of this dissertation, and the reader can skip the other chapters if he/she is only interested in the design, operations, and potential benefits of TCP offload through connection handoff. Chapter 5 then presents further evaluations of connection handoff using various host processor configurations and also compares performance gains from increased host processor speeds, larger processor cache sizes, and offloading through connection handoff. Chapter 6 discusses various placement options for the offloading processor and their potential advantages and disadvantages. Chapter 7 describes implementation of connection handoff in the FreeBSD and Linux operating systems. It also discusses the generality of the connection handoff interface and implementation issues that arise due to different network stack implementations of FreeBSD and Linux. Chapter 8 draws conclusions.

Chapter 2

Related Work

Performance of TCP/IP network stack implementations has been a prolific research topic due to the widespread use of the protocol suite, and there is a significant amount of previous work related to the materials covered in this dissertation. This chapter discusses previous work on the performance of the TCP/IP network stack and also recent studies on TCP offload.

2.1 TCP/IP Network Stack Performance

The performance bottleneck in the network stack has shifted from instructions to main memory accesses as the CPU speed increases and the gap between the CPU speed and main memory speed widens. Early on, techniques like hash-based de-multiplexing of received packets [58] and header prediction [89], which optimizes the most common case processing for received packets, reduced the number of instructions required to process a packet. Then, a study by Clark *et al.* showed that protocol processing (instructions) was not the most time consuming, but rather operations that touch packet data consumed most CPU cycles [24]. These data touching operations consist of data copies and checksum calculations. Data copies take place when the application either sends or receives data through a socket. The packet data is copied from the user space to the kernel space when sending the data, and the packet data is copied from the kernel space to the user space when the application receives the data. Checksum calculation touches every byte of packet data. These data touching operations are costly not only because of the instructions involved in the operations, but more importantly because of large main memory access times. In a modern system, main memory access can take over 200 CPU cycles. So, the amount of

time spent by the CPU to perform the data touching operations could easily dwarf the rest of the operations performed in the TCP/IP stack. The results in Chapter 3 confirm this observation.

Numerous techniques have been proposed and implemented in order to reduce or eliminate data touching operations [23, 25, 48, 28, 70]. First, combining copy and checksum routines into one routine avoids having to access the packet data in main memory multiple times, since the CPU can perform both copy and checksum calculation when the packet data is brought into processor registers or caches. Some argued that protocols should be designed such that all data touching operations can be performed together in order to minimize main memory accesses [25]. The use of DMA controllers on the NIC saves the CPU from having to copy packets to the NIC memory through programmed I/O. Checksum offload techniques eliminate the need for the CPU to compute checksums altogether [48]. With checksum offload, the NIC calculates checksums while the NIC fetches packets from main memory or stores them into main memory. This enables very efficient calculations and requires almost no software overhead. Finally, zero-copy I/O techniques eliminate data copies between the kernel and user memory spaces [23, 28, 70]. These techniques combined can completely eliminate the need for the CPU to access packet data. Currently, all except zero-copy techniques are implemented by almost all operating systems and NICs. Most network stack implementations now perform a single data copy between the user and kernel spaces. These network stacks are commonly known as single-copy network stack. Although many operating systems do not fully support zero-copy I/O, most of them implement a zero-copy version of the `sendfile` system call. Using `sendfile`, the application gives the operating system a reference to the file to be sent. Since the file data is already in the kernel memory space (file cache), it is easy to avoid data copies.

The performance characterization presented in Chapter 3 shows that main memory accesses due to connection data structures can be as time consuming as data touching operations and are now becoming a new bottleneck. A key observation is that a large number of connections can easily overwhelm the CPU caches, causing cache misses and main mem-

ory accesses when the CPU tries to access connection states. A number of previous studies showed that cache misses within the network stack are a problem [51, 54, 65]. However, they do not investigate the cause of cache misses. Nahum *et al.* reported that caches have a big impact on packet processing time within the network stack [65]. Specifically, they show that larger, higher associative caches reduce packet processing time, and that the protocol processing time would scale with the processor performance as long as the caches provide the data necessary for the protocol processing. However, as the number of connections increases, the caches would have to become prohibitively large in order to capture the working set size of all connections. Others recently used a real machine and reported that cache misses can significantly affect the network stack performance [54]. However, they still ignore the impact of a large number of connections and long latencies that servers experience and consequently do not investigate their negative impact on performance that is shown in Chapter 3. Luo *et al.* evaluated a number of server workloads, including SPECweb99, using three different real systems [51]. They show that web servers spend a large fraction of time executing the kernel and in general achieve lower instructions per cycle (IPC) than the SPEC CPU2000 integer benchmarks. They also report frequent L2 cache misses on web servers and their noticeable impact on IPC. Finally, they compare different L2 cache sizes (512KB, 4MB, and 8MB) and show that even large L2 caches are not able to capture working sets. These findings generally agree with the results presented in Chapter 3. Some researchers used a full system simulator and also found that commercial workloads like web servers cause frequent L2 cache misses [3].

Several studies examined parallelization of the network stack to improve networking performance [14, 67, 79, 90, 88]. They show significant performance gains are realizable by exploiting packet-level and connection-level parallelism found in packet processing. Recent operating systems such as Solaris 10 have also begun to fully parallelize the network stack. TCP offloading is a form of parallel execution. Offloading packet processing to either dedicated processors or network interfaces enables the system to process packets on one or more processing elements while the user application or the operating system performs

other tasks. Connection handoff presented in Chapter 4 requires the bypass layer in the network stack. Since connection-level parallelism still exists within the bypass layer, the bypass layer may be parallelized along with the rest of the network stack. This way, the parallelized network stack should be able to operate concurrently with connection handoff.

Finally, a few studies tried to reduce communication overhead between the device driver and the NIC [12, 61]. Recent work by Binkert *et al.* notes that I/O accesses to the network interface consume a large number of cycles and that the device driver can consume a significant fraction of processor cycles [12]. They show that by placing the network interface on the processor die and connecting it directly to the processor caches, the I/O access cost drops dramatically and thus improve overall networking throughput. Others also tried to reduce I/O cost by moving the network interface closer to the processor. For instance, Minnich *et al.* showed dramatic reductions in I/O cost, therefore in network latency, by attaching the network interface to main memory (memory controller) [61]. While these techniques are more focused on improving low level communication between the processor and the network interface, offload based on connection handoff described in Chapter 4 also requires communication between the CPU and the network interface and may benefit from these techniques.

2.2 TCP Offload

There are a number of studies that present various forms of full TCP offload and evaluate their performance using real prototypes or emulations [17, 33, 73, 75, 76, 87]. They all show potential benefits of offload. Some of them mention the issue of load imbalance between the host processor and the offloading processor, in which the offloading processor becomes a bottleneck in the system and can degrade system performance. However, none of them actually address the problem. In contrast, offload based on connection handoff described in Chapter 4 enables the system to dynamically control the load on the offloading processor so that it does not become a bottleneck in the system.

First, TCP servers, based on Split-OS concept [9], splits TCP and the rest of the op-

erating system [73]. Theoretically, the TCP part can run on any piece of hardware such as dedicated processors, dedicated systems, or network interfaces. They implemented two types of TCP servers. One is built using a symmetric multiprocessor system, and the other uses a two-node cluster. In the multiprocessor system, one or more processors are dedicated to TCP processing and communicate with the processor running the user application through shared memory regions. In the two-node cluster, one node runs TCP and communicates with the other node running the user application through the MemNet API [72]. MemNet API provides the user application with a simple send and receive interface. This API is actually implemented on top of the Virtual Interface Architecture (VIA) [29] and exploits zero-copy data transfers and user-accessible interfaces of the VIA. The authors report that the prototype systems (whose processors run at 300 or 500MHz) running Linux 2.4 improve web server throughput by up to 30%, but that load-balancing becomes an issue. Either the processor or the node running the web server or the one running TCP becomes a bottleneck. In order to balance the load, the system would either have to re-partition the operating system or vary the number of dedicated processors or nodes.

Some argued that the system should dedicate one or more general-purpose processors in a multiprocessor system to TCP processing [76]. They call this approach embedded transport acceleration (ETA) and motivate it by noting the fact that systems will inevitably have multiple processors in the future, and dedicating one or more processors for packet processing can help improve networking performance. This approach is essentially same as TCP servers discussed above. Similar to TCP servers, one or more dedicated processors execute the TCP/IP network stack code, while other processors run the user application. Also, the stack running on the dedicated processor is modified to exploit the fact that the processor is dedicated to the stack alone, as in TCP servers. For example, they both poll the network interface card to detect received packets, instead of waiting for interrupts. Brecht *et al.* later extended the ETA work and adds an asynchronous I/O interface between the application and the dedicated packet processing processor [17]. They exploit the asynchrony to provide zero-copy send. They use a dual-CPU machine and compare web server

throughput achieved using a well-tuned regular Linux and the modified Linux that supports ETA. The results show that the ETA with the asynchronous I/O interface can achieve performance comparable to the regular Linux, but with reduced CPU utilization. The authors argue that this apparently improved efficiency stems from the fact that the dedicated processor does not have to perform synchronization (locking) operations of the regular Linux. However, the ETA achieves lower maximum throughput than the regular Linux because the processor dedicated for packet processing becomes a bottleneck in the system. The authors suggest the use of multiple dedicated packet processing processors. As mentioned above, this approach is essentially full offload to a dedicated processor. Thus, as their results show, there is a risk that the dedicated processor becomes a bottleneck in the system. Also, the dedicated processor still suffers from expensive main memory accesses. So, moving TCP processing to a dedicated processor does not necessarily make TCP processing more efficient, either.

Some of the authors at Intel that published the ETA work later proposed, but did not implement, additional techniques that can help improve packet processing [75]. While these additional techniques can help the dedicated processor to process packets more efficiently, they still do not address the potential risk that it may become a bottleneck in the system. The additional techniques include hardware supports for fine-grained threading, cache updates from the network interface card, and an asynchronous copy engine. The packet processor may switch threads on cache misses in order to overlap memory accesses and computation. The network interface card may transfer the network headers of received packets directly into the processor caches so that the first access to the headers does not incur compulsory cache misses and trigger main memory accesses. Others at Intel later showed that allowing the network interface to directly access the host processor caches can significantly reduce packet processing time on the host processor [38]. With the asynchronous copy engine, the processor initiates packet data transfers between the kernel and user spaces, and then continues processing other packets. So, the copy engine enables the processor to avoid having to copy data and save cycles, but it does not save main memory bandwidth. These

proposed techniques are now a part of the Intel I/O Acceleration Technology initiative. The copy engine can now be found on several Intel chipsets [41]. The other features may also materialize in future Intel systems.

Finally, Freimuth *et al.* also evaluated the impact of TCP offload on the local I/O interconnect traffic [33]. They motivate a full offload approach by arguing that network server performance does not scale with the increasing processor performance because of the growing processor and memory speed gap, expensive I/O reads and writes to device registers across the local I/O interconnect, poor local I/O interconnect utilization, and time-consuming interrupt handling. A central insight is that with offload, the network interface card and the operating system communicate at a higher level than the conventional network interface card, which exposes opportunities for optimizations. For instance, the offload NIC can handle user data that is much larger than the network MTU, thereby reducing the number of DMA transfers across the local I/O interconnect. Moreover, the conventional NIC transfers complete packets including network protocol headers, but the offload NIC only needs to transfer payloads not headers across the local I/O interconnect. So, offloading can reduce the local I/O interconnect traffic by transferring large data with fewer DMA transactions and by not transferring protocol headers. The authors implement a prototype system consisting of two machines. One machine runs the host operating system and applications, while the other acts as the offload NIC and performs TCP packet processing. However, the evaluation only shows reductions in the number of DMA transactions, and bytes transferred on the local interconnect, not how offload affects main memory accesses, I/O reads and writes, and interrupts. These reductions are not exclusive to a full offload approach, as shown in Chapter 4. Offload based on connection handoff can deliver the same level of reductions as well. Westrelin *et al.* also evaluated the impact of TCP offload [87]. They used a multiprocessor system in which one processor is dedicated to executing TCP and show a significant improvement in microbenchmark performance.

There is also a study that presents an analytical model for predicting the performance benefits of full TCP offload [83]. They model the performance gains from offload using

various parameters including the speed of the NIC and the host processor, and workload characteristics. They show that offloading can be beneficial but its benefits can vary widely depending on application and hardware characteristics. The results in Chapter 4 also show that offloading too much work to the NIC can degrade overall system performance when the NIC has insufficient processing capacity. However, unlike full TCP offload, connection handoff allows the system to dynamically control the amount of work performed on the NIC. So, the system can fully utilize the processing capacity of the NIC while avoiding performance degradations.

Recently, Mogul compiled a number of problems associated with full TCP offload [63]. The problems include creating a potential bottleneck at the NIC, difficulties in designing software interfaces between the operating system and the NIC, modifying the existing network stack implementations, and introducing a new source of software bugs at the NIC. TCP offload based on connection handoff overcomes most of these problems as discussed in Chapter 4. Specifically, connection handoff can avoid creating a bottleneck at the NIC and requires little modifications to the existing network stack architecture. However, it still suffers from the fact that the firmware running on the NIC may have software bugs.

2.3 Commercial TCP Offload NIC and Operating System Support

There are several commercial NICs that support TCP offload. Unfortunately, none of them have publicly available specifications, and many details are unknown. These commercial NICs include one or more specialized programmable processors that execute protocols and on-board memory that stores data including connections and packets. They typically implement full TCP offload, but they should also be able to implement connection handoff described in Chapter 4 as well, since it does not assume specific controller architectures. Some may also support other protocols like iSCSI [81] and remote direct memory access (RDMA) [74, 82] that run on top of TCP. Some of these commercial NICs are claimed to exploit packet-level and connection-level parallelism using special features of the protocol processor. For instance, pipelining the processing tasks allows multiple packets to be

processed simultaneously. Likewise, running multiple connections on multiple cores exploits connection-level parallelism. Some claim to use very large instruction word (VLIW) processors to exploit connection-level parallelism. The NICs support a fixed number of connections, which varies widely from at least one thousand to 64 thousands. So, despite the use of specialized architectures, they may become a bottleneck in the system, which necessitates some form of load-balancing (through connection handoff) between the host processor and the NIC.

There are very few publications on performance characteristics of commercial NICs that support TCP offload [8, 30]. None of them presents the details of the software and hardware architectures of the NICs or examines the performance limitations of the NICs and how they affect system performance. Feng *et al.* published a short evaluation of the 10 Gigabit Ethernet NIC from Chelsio that supports TCP offload and up to 64 thousand connections [30]. They measure application latency and throughput using a microbenchmark program and the Apache web server (with few connections) running on Linux, and show that offloading achieves lower host CPU utilization, lower latency, and higher throughput than the regular network stack without offloading. Overall, their Opteron-based system achieves close to 8Gb/s of unidirectional TCP throughput using the offloading NIC and the standard TCP maximum transmission unit (1500 bytes) for Ethernet. Some of the same authors later compared performance of Infiniband, Myrinet, and 10 Gigabit Ethernet for several scientific applications that run on a four-machine cluster [8]. They again use the offloading NIC from Chelsio. The key finding is that the cluster with 10 Gigabit Ethernet and the offloading NICs achieves comparable or better performance for these applications than the cluster with either Infiniband or Myrinet. The fact that this NIC achieves near-10Gb/s is encouraging and shows that the offloading NIC can handle significant packet rates.

There is also a research prototype programmable controller that is specifically designed for TCP processing at 10Gb/s rates [36]. This controller highlights both the importance of quick access to connection states and also the limitation on the amount of compute power available on the NIC. A most notable feature of this controller is its handling of

connection states (TCP control blocks). The control blocks are stored in an on-chip storage. Upon receiving a packet, the processor on the controller queries a cache-like structure to look up the corresponding control block. Then, the processor gets an index number of the control block. Using this index number, it issues an instruction that loads the entire control block into a dedicated register file. When packet processing is over, the control block is written back to the on-chip storage via another instruction. Thus, accesses to the connection states are fast register loads and stores. The authors report that the processor can receive and process 64B minimum-sized TCP packets at 10Gb/s. It runs at 4.82GHz and consumes 6.39W, which is significant considering that even though PCI devices can technically draw maximum 25W, most devices are designed to consume less than 10W. For these experiments, the processor runs TCP code that is written from scratch and only requires about 116 instructions to process one received TCP packet. Also, the control block is 33B and includes only those fields necessary for receive processing. There are no interactions with the host. Even though the prototype controller can achieve 10Gb/s rates, more complete and realistic TCP implementations such as those found in BSD and Linux operating systems require larger control blocks and more instructions and would force the processor to run faster and consume more power. Fully-functional offload firmware needs host interactions and would require even faster clock rates and more power, which may exceed the maximum available power. The prototype controller has a small on-chip storage, which can store only up to 64 connections. The authors note that it can be expanded to store several thousands of connections, but acknowledge that a greater number have to be stored off-chip and necessitate caches in order to reduce the time spent transferring control blocks to and from the dedicated register file. However, with the introduction of caches, the architecture would run into the exactly same problem that general-purpose host processors have; accesses to connection states may significantly increase packet processing time. Thus, even though special-purpose architectures like this prototype controller can employ fast memory and process packets far more efficiently than host processors, they can only do so to a certain extent. The system should not overwhelm them. The handoff

interface described in Chapter 4 helps the operating system to appropriately partition the work between the host processor and the NIC.

Currently, only the latest Microsoft Windows operating system supports a driver API for TCP offload NICs. To the author's best knowledge, only the NICs from Alacritech support Windows through this API at this point. Other hardware vendors are likely to support Windows in the future. This driver API is a part of the Chimney offload architecture [60]. The Chimney offload architecture is an extension to the network stack of the Microsoft Windows operating system. It is an interface between the network stack and the device driver based on connection handoff. Functions provided by the network stack and the device driver are similar to those described in Chapter 4. For instance, it uses a layer similar to the bypass layer, can offload established connections to the network interface, and restore them back from the network interface to the operating system. The design presented in Chapter 4 was developed independently by the author, roughly at the same time that an initial document on the Chimney offload architecture was published. Many details about this architecture are unclear: why the operating system chose to use connection handoff, how the operating system uses connection handoff, whether the operating systems implement a framework for implementing policies for using connection handoff, or whether the operating system and the NIC implement policies like those presented in Chapter 4.

2.4 Connection Handoff

TCP connection handoff techniques have been used in various systems [49, 68, 71, 84]. Pai *et al.* used a connection handoff mechanism in their request distribution system for a cluster of web servers [68]. In this system, once a client establishes a TCP connection to the front-end request distributor and sends a request, the distributor first examines the request and locates an appropriate back-end server for that request. Since the connection is already established, the connection is first handed off to the back-end server, and then the request is forwarded to the back-end as well. Most operating systems do not implement handoff techniques, so the authors modified the operating system used in their experiments.

However, handoff is transparent to the application running on both the client and the back-end server, the application requires no modifications.

Another technique uses back-end servers to locate the appropriate server for a request and to initiate connection handoff, rather than the front-end switch [71]. The authors note that using their technique, the back-end operating system requires no modifications. Persistent HTTP connections and pipelined HTTP requests complicate the request distribution process. Because multiple requests arrive on the same connection, and the most suitable server for each request might differ, either the front-end switch or the back-end server potentially needs to initiate handoff of the same connection multiple times. Kokku *et al.* proposed a mechanism for handling multiple handoffs [49]. Others have shown that connection handoff concept can be used to provide fault tolerance to long-lived connections by replicating connection states among redundant servers [84].

The idea of using connection handoff for TCP offloading has been suggested by Mogul *et al.* They recently argued that exposing transport (connection) states to the application creates opportunities for enhanced application features and performance optimizations [62]. For instance, web server can examine round trip times of the clients (connections) and provide different content to different clients. Another example they describe is moving connection states between the operating system and offload NICs. They also mention the possibility of restoring connections from the NIC when the NIC has insufficient processing capability. This idea is essentially offloading TCP processing through connection handoff.

A threshold-based connection selection and the load control mechanism presented in Chapter 4 borrow ideas from previous work. The idea of using a threshold for selecting connections to hand off to the NIC is borrowed from a previous study on load-balancing by Harchol-Balter and Downey. They examined a distribution of lifetimes of UNIX processes and developed a load-balancing scheme based on a statistical analysis of that distribution [34]. Both UNIX processes and TCP connections from web workloads exhibit similar lifetime distributions. A majority of processes (connections) are short-lived, and a

small number of long-lived processes (connections) account for a significant portion of the total lifetime. Their study shows that migrating only long-lived connections to remote idle machines significantly reduces average process completion times. The load control mechanism presented in Chapter 4 borrows ideas from random early detection. Random early detection uses the packet queue length to detect worsening congestion at gateways [32]. The load control mechanism presented in Chapter 4 also uses the length of the receive packet queue in order to detect overload and underload conditions on the NIC. The actual mechanism is considerably simpler than the early random detection, but the experimental results indicate that it is sufficient.

2.5 Summary

Past research on the performance of TCP/IP network stack implementation has identified that main memory accesses, not instructions, due to data touching operations are the bottleneck in TCP processing. Checksum offloading and zero-copy I/O techniques were proposed to reduce such accesses. The results in this dissertation confirm the impact of data touching operations and also the effectiveness of checksum offloading and zero-copy I/O. Further analysis then shows that main memory accesses due to connection data structures are becoming a bottleneck, which motivates the use of offloading to accelerate TCP processing. Previous work on TCP offloading has shown potential benefits as well as problems associated with offloading. This dissertation leverages these previous findings and presents a new framework for offloading that is based on connection handoff, which achieves performance benefits of offloading while avoiding the previously-identified problems.

Chapter 3

Performance Issues of Modern TCP/IP Network Stack

The performance of the TCP/IP network stack plays a crucial role in network servers. In order to understand performance issues, this chapter first describes important operations that take place within the TCP/IP network stack. Then, a performance characterization of the TCP/IP network stack is presented in order to identify the performance problems in the network stack. The performance problems identified in this chapter serve as a motivation for using offload in Chapter 4.

3.1 Network Stack Operations

The network subsystem of the operating system enables the application to exchange data with other applications running on a local or remote machine through various communication protocols. TCP/IP is one of the most commonly implemented protocol suites. It is implemented through a number of layers (hence the network *stack*), in which each layer implements one or more protocols. Throughout this dissertation, the network stack and the TCP/IP network stack are used interchangeably.

Figure 3.1 shows important operations within the TCP/IP network stack of the operating system. The descriptions below are based on the FreeBSD 4 operating system, but should apply to other implementations that are derived from the BSD network stack implementation. Refer to the book by Wright and Stevens [89] for detailed explanations of the BSD implementation of the TCP/IP network stack. Figure 3.1 shows the five layers of the network stack. The socket layer is the interface between the network stack and the user application. A socket includes a buffer (send socket buffer) to hold data to be sent out to the network and another buffer (receive socket buffer) to store data that has been received from

Figure 3.1 : Operations performed within the TCP/IP network stack of many operating systems. The arrows show the flow of data within the operating system. Each step is indicated by a number with a prefix. The prefix represents a distinct flow according to its initiator. H, W, R, and N show the flows initiated by hardware timer interrupts, synchronous system calls (write and read), and hardware interrupts from the NIC, respectively.

the network. The TCP and IP layers implement functionalities required for the TCP and IP protocols, which include reliability, flow control, and routing protocols. The Ethernet layer implements the protocols of the Ethernet medium. Finally, the device driver communicates with the NIC in order to send and receive packets.

The network stack executes in response to three types of events: system calls, NIC interrupts, and timer interrupts. In Figure 3.1, the arrows show the sequence of operations performed by the network stack in response to these events. First, in response to the write and read system calls, the network stack performs the operations indicated by W (write) and R (read), respectively. Only these system calls are shown as they are two most frequently-used system calls. When the user application calls write in order to send data (step W1), the user data is enqueued into the send socket buffer (step W2). The data is later removed when it is acknowledged by the receiver (this step is not shown in the figure but would be performed as part of step N5). Unless zero-copy I/O is used, the data is copied from the user memory space to the kernel memory space so that the user application cannot modify the original data until it is acknowledged. The TCP layer

then creates one or more packets that include the data in the send socket buffer (step W3). The packets are sized according to the maximum transmission unit (MTU). The IP layer appends IP headers to the packets and also determines the route (step W4). The Ethernet layer appends Ethernet headers to the packets (step W5). If the MAC address of the packet's destination is not known, it also executes the address resolution protocol (ARP) to determine the destination's MAC address and delays the packet transmission until the MAC address is resolved. Finally, the device driver communicates with the NIC in order to transmit the packets (step W6). Note that if the NIC does not support checksum offloading, TCP and IP checksums are computed in the TCP and IP layers. However, most modern NICs support checksum offloading, and the actual checksum calculations are performed by the NIC. When the application calls `read` in order to access the received data (step R1), the network stack dequeues data in the receive socket buffer (step R2) and returns it to the application. This data is copied from the kernel memory space to the user memory space unless zero-copy I/O is used. The dequeue operation may trigger TCP to generate new packets (step R3). The rest of the steps R4–6 in packet transmission are same as W4–6.

When a packet is received from the network, the NIC normally interrupts the CPU in order to notify the device driver that a new packet is available. In Figure 3.1, the operations indicated by N (NIC interrupt) are performed by the network stack in response to an interrupt from the NIC. Upon receiving an interrupt from the NIC (step N1), the device driver checks for errors that may have occurred during the reception of the packet (step N2). At this stage, the information regarding the packet such as the packet length is usually stored in the data structures specific to the NIC (driver). So, the device driver translates these data structures into those that are suitable for the rest of the network stack. The device driver then passes the packet to the Ethernet layer. The Ethernet layer de-multiplexes the packet based on the type information stored in the Ethernet header and strips off the headers (step N3). If the packet is an IP packet, then the packet is passed onto the IP layer. The IP layer similarly strips off the IP header and de-multiplexes the packet based on the type of the transport protocol stored in the IP header (step N4). If the packet is a TCP packet, then

it is passed onto the TCP layer. The TCP layer also de-multiplexes based on the source and destination IP addresses and port numbers stored in the TCP header in order to locate the corresponding connection, and then processes the packet (step N5). If the packet contains valid data, then it is enqueued into the receive socket buffer (step N6). In response to receiving the new data, the TCP layer generates an acknowledgment (ACK) packet (step N7). The acknowledgment packet is then sent to the IP layer, just like any other outgoing TCP packet. The rest of the steps N8–10 are same as W4–6. As mentioned above, if the NIC does not support checksum offloading, then the TCP and IP layer must compute checksums of the received packet and validate them against the values stored in the packet. When checksum offloading is used, the NIC either validates the checksums and notifies the network stack whether the packet has invalid checksums, or the NIC simply computes the checksums, which are then passed onto the network stack.

Finally, TCP also runs in response to hardware timer interrupts. In Figure 3.1, the operations indicated by H (hardware timer) are performed by the network stack when a retransmit timer expires. TCP is a reliable protocol, so if sent data is not acknowledged within a specified time period, the sender must retransmit the packet. The operating system uses hardware timer interrupts to determine if there are unacknowledged packets that need to be retransmitted. TCP employs several other timers including persist timer, keep-alive timer, delayed ACK timer, and `TIME_WAIT` timer. The steps taken to process these timers are similar to step H1–5. Upon receiving the retransmit timer (step H1), the TCP layer determines what portion of the send socket buffer must be retransmitted and then creates one or more packets to send (step H2). These packets are passed down to the IP layer just like any other outgoing packet.

Overall, most layers of the network stack shown in Figure 3.1 perform simple tasks. The TCP layer is more complex because it maintains connection states in memory and provides reliability. Despite its complexity, the number of instructions required to process a TCP packet can be fairly low, but expensive main memory accesses can dominate packet processing time [24]. The rest of this chapter examines the execution profiles of the network

stack running on a real machine in order to illustrate the impact of such main memory accesses.

3.2 Experimental Setup

A low-overhead, non-statistical profiler is used in order to collect execution profiles of a real system. The system runs web server software to exercise the network stack using realistic workloads, as well as a set of microbenchmarks that are used to isolate different aspects of performance of the network stack. All measurements are taken on a testbed that consists of a uniprocessor server and two client machines. The server includes a single AMD Athlon XP 2600+ CPU, 4GB of DDR SDRAM, one 40GB IDE disk that stores programs, one 32GB SCSI disk that stores web files, and two Intel PRO/1000 MT Server Adapters (Gigabit Ethernet/PCI interface) on a 64-bit/66MHz PCI bus. The Intel NIC implements TCP/UDP checksum offload for both outgoing and incoming packets. Each client machine has a single AMD Athlon XP 2800+ CPU, 1GB RAM, a single 40GB IDE disk, and two Intel PRO/1000 MT Desktop Adapters. The machines are connected through two isolated Gigabit Ethernet switches such that each machine is on two subnets using two interfaces. The AMD Athlon XP 2600+ CPU used in the server has a unified L2 cache and separate L1 instruction and data caches. Each L1 cache is a two-way set associative 64 KB cache with 64 byte lines. The unified L2 cache is a 16-way set associative 256 KB cache with 64 byte lines.

All machines in the testbed run the FreeBSD 4.7 operating system. The server machine runs the Flash web server [69]. Flash is a multi-threaded, event-driven web server. It uses the `sendfile` system call to send files through zero-copy I/O, the `kqueue` mechanism [50] to efficiently process events, and helper threads to handle disk I/O without blocking the main thread. `Kqueue` serves the same purpose as the traditional `select` system call, which informs the application of the status of sockets. For example, the application can determine when send socket buffers have free space for writing or when receive socket buffers have data to be read. The `kqueue` mechanism is more efficient and scalable than

the traditional `select` system call, enabling the application to handle a larger number of simultaneous connections (sockets).

Each client machine runs two instances of a synthetic web client program which replays a given web access log [10]. Each instance connects to the server through a different subnet. The client program opens multiple connections to the server to simulate multiple web clients and then generates web requests as fast as the server can handle them in order to determine the maximum sustainable server throughput. The traces are split equally among all four replayers. The web logs used for the experiments are from an IBM web site (IBM), and the 1998 World Cup web site (WC). The World Cup trace is available from the Internet Traffic Archive. Requests from the same client (identified by IP address) that arrive within a fifteen-second period are issued using a single persistent connection. In addition to the four web traces, SPECweb99 (SPECWEB) is also used. Unlike the synthetic clients described above, SPECweb99 clients try to enforce a fixed bandwidth per connection (400Kb/s) by scheduling requests appropriately, and the working set size of files requested by the clients grows as the number of connections increases. SPECweb99 results shown in this chapter are always based on runs that include the default mix of requests (70% static content and 30% dynamic content) and result in every connection achieving at least the default minimum required bandwidth (320Kb/s), unless otherwise noted.

Dummynet is used to increase the latency of communication between the clients and the server [78]. Since the Internet has orders of magnitude larger communication delays than systems sharing a single Gigabit Ethernet switch, it is important to simulate those delays using a mechanism like dummynet [66]. Finally, TCP's delayed ACK feature is disabled so that every sent segment is immediately acknowledged by the receiver; there are equal number of sent TCP segments and received ACK packets. With delayed ACKs enabled, the receiver may delay the generation of an ACK for a certain period in the hope that the ACK can be piggybacked onto a data segment. So, the ratio between sent segments and received ACK packets can vary depending on conditions.

The profiler used throughout this dissertation is a custom profiler developed by the au-

thor. Its implementation is significantly modified from the default kernel profiler of the FreeBSD 4.7 operating system. The default profiler is a statistical profiler in which it periodically polls program counters, and incurs high overhead. In contrast, the custom profiler is non-statistical. It measures the execution time and processor performance statistics of individual kernel functions or groups of kernel functions. It provides a cycle-accurate measure of function execution time using the processor timestamp register that is incremented each clock cycle. In a manner similar to the Digital Continuous Profiling Infrastructure (DCPI) [6], processor performance statistics are measured using hardware performance counters [1, 40]. These performance counters count the occurrence of events such as retired instructions and data cache misses. The profiler aims to accurately measure these statistics while minimally perturbing the overall system behavior. The profiler achieves the former by computing its measures online, taking the dynamic control flow into account. It achieves the latter by profiling only those functions that are specified by the user. For the workloads used in this dissertation, the impact of profiling on system behavior is very small. The HTTP content throughputs achieved with profiling enabled are at most 5% lower than those achieved with profiling disabled. Profiling increases or decreases the number of various performance counter events such as L2 cache misses and TLB misses per packet by at most 6%. Profiling requires about 900 instructions per packet regardless of the workloads used in this chapter.

3.3 Performance Analysis

Figure 3.2 shows the impact of main memory accesses within the network stack using a microbenchmark program (TCP Send). The microbenchmark program simply sends maximum-sized (1460 byte) TCP segments at 100 Mb/s to another machine evenly across a number of connections. The profiles in the figure show the number of processor cycles, instructions, micro-operations, and data memory accesses per TCP packet in each layer of the network stack. The layers are indicated by *System Call*, *TCP*, *IP*, *Ethernet*, and *Driver* on the x-axis. *Total* shows the sum of all the layers. There are several experimental con-

Figure 3.2 : Execution profiles of the network stack during the execution of TCP Send microbenchmark.

configurations denoted by A–E. *Zero-copy* and *Checksum offload* indicate whether the system call layer performs data copy and whether the stack calculates checksums, respectively. *Dummynet latency* indicates packet delays introduced by dummynet. 0ms latency means that dummynet is not used. For each configuration, the left stacked bar shows the number of processor cycles, while the right stacked bar shows the number of micro-operations. The processor cycles are further divided into two bars. The lower bar shows the number of cycles required to execute the instructions assuming that each instruction takes exactly one cycle to execute, thus it equals the number of instructions. The upper bar shows the rest of the cycles. The micro-operations are also divided into two bars. The lower bar shows the number of data memory accesses, and the upper bar shows the rest of the operations.

Zero-copy and checksum offload combined eliminate memory access to packet data (commonly referred to as data touching operations) and dramatically decrease packet processing time, from 11038 cycles to 5228 (see A, B, and C). Zero-copy alone is not as effective as one might expect because the IP layer still computes checksums and accesses main memory to fetch packet data. B shows a spike in the IP layer as a result. This im-

fact of data touching operations is well-known [24]. Zero-copy and checksum offload reduce the number of instructions (and micro-operations), but the reductions are insignificant (see A, B, and C). Figure 3.2 also shows that main memory accesses to connection state, such as TCP control blocks, have significant impacts on packet processing time, similar to zero-copy and checksum offload. With zero-copy and checksum offload, the remaining memory references are mainly to packet headers and connection data structures. As the number of connections increases, the connection data structures quickly grow larger than the processor cache size. Consequently, accesses to such structures suffer from large main memory latencies, which are easily over 200 cycles on modern processors. C and D show that 1024 connections require 35% more cycles to process each packet than a single connection, even though the number of instructions and memory references remain almost constant. Furthermore, E shows that as the network latency increases, the packet processing time increases by another 29% because of the increased number of cycles spent in the TCP layer. With longer round-trip times, acknowledgment packets return from the receiver much later, which increases the reuse distance of connection data structures. The increased reuse distance in turn reduces the likelihood that they will be found in the cache.

As configuration C in Figure 3.2 shows, by using zero-copy I/O and checksum offload, the network stack only executes about 3000 instructions (4600 micro-operations) in order to process a packet. Theoretically, this operation rate is not a serious constraint for the host CPU, as 10Gb/s of bidirectional TCP throughput using maximum-sized packets could be achieved with a processor capable of about 8 billion operations per second. However, as D and E show, because of main memory accesses, the actual number of processor cycles required to process a packet can be almost double the number of operations. As the figure shows, almost 50% of the micro-operations are data memory accesses. Typical integer applications have much less frequent memory accesses. For example, only 33% of the micro-operations are data memory accesses in the SPEC CPU2000 integer benchmark suite on the same machine. Thus, memory performance can significantly affect packet processing time.

Figure 3.3 : Execution profiles of the network stack while the server executes the World Cup trace.

Figure 3.2 uses a microbenchmark program to illustrate the impact of increasing connections and network latencies on network stack performance. The following three figures (Figure 3.3, Figure 3.4, Figure 3.5) show the impact of increasing connections and network latencies on web server for three different web workloads. First, Figure 3.3 shows network stack profiles during the execution of the World Cup trace. The web server, Flash, uses zero-copy `sendfile` to send static HTTP content. The World Cup trace used in the experiment consists of only static content, so all files are sent via zero-copy. However, HTTP headers and requests are still copied between the user and kernel memory spaces. Because of these data copies, the system call layer in Figure 3.3 shows slightly greater number of processor cycles than in Figure 3.2. As the number of connections increases from 1 to 1024, the number of cycles in the TCP layer increases from 1972 to 3008, by 53%, while the number of instructions (and micro-operations) remains near-constant (see A, B, and C). Increasing network latency to 20ms further increases the number of cycles spent in the TCP layer from 3008 to 3694, by 23% (see C and D). This behavior is same as the one shown by the TCP Send microbenchmark in Figure 3.2. As mentioned before, the processor cache is

Figure 3.4 : Execution profiles of the network stack while the server executes the IBM trace.

overwhelmed by a large number of connections, and the processor is forced to access main memory in order to fetch connection data structures, which in turn increases the number of cycles spent in the network stack. Also note that the number of instructions and the mix of micro-operations in the network stack are very close to those shown in Figure 3.2.

Figure 3.4 shows network stack profiles during the execution of the IBM trace. The system call layer shows a greater number of processor cycles than in Figure 3.3. HTTP responses in the IBM trace are smaller than those of the World Cup trace. So, there are more data copies (due to HTTP headers and requests) per packet in the IBM trace than in the World Cup trace. More frequent data copies account for the increased number of processor cycles in the system call layer. (from 1462 in Figure 3.3 to 2593, or a 77% increase). As the number of connections and network latencies increase, the TCP layer shows a trend very similar to that shown in Figure 3.3. As the number of connections increases from 1 to 1024, the number of cycles spent in the TCP layer increases from 2317 to 2819, by 22%. Imposing 20ms network latency increases the number of cycles to 3762, by 62%.

Finally, Figure 3.5 shows network stack profiles during the execution of SPECweb99.

Figure 3.5 : Execution profiles of the network stack while the server executes SPECweb99.

Unlike the World Cup and IBM traces, SPECweb99 uses both dynamic and static content. As mentioned above, static content is sent through zero-copy `sendfile`. However, dynamic content is copied from the user space to the kernel space. So, the system call layer now wastes even greater number of cycles copying data than it does for the IBM trace. For example, A in Figure 3.4 shows 2593 cycles in the system call layer. A in Figure 3.5 shows 3738 cycles in the system call layer (a 44% increase). As expected, the number of cycles spent in the TCP layer increases as the number of connections and network latencies increase. As the number of connections increases from 16 to 1024, the number of cycles increases from 2506 to 2994, by 19%. The number further increases to 3769, by 26%, when dummynet increases network latency to 20ms.

All three web workloads exhibit the similar behavior as the number of connections and network latencies increase. Furthermore, this behavior is as predicted by the microbenchmark in Figure 3.2. Thus, main memory access to connection data structures can be a performance issue for real servers that must handle a large number of connections.

The results presented in this section show that it is crucial to reduce the frequent, expensive main memory accesses within the network stack. These memory accesses can be

divided into accesses to packet data and connection data structures. Packet data accesses can be eliminated by using zero-copy I/O and checksum offload techniques, enabling a theoretical processor that could sustain 8 billion operations per second to saturate a full-duplex 10Gb/s link (assuming that the per-packet processing requirements remain as they are for configuration C in Figure 3.2). However, as network latency and the number of connections increase, connection data structures can no longer be cached. Therefore, memory speed, as well as processor performance, must improve significantly in order to saturate a 10Gb/s link with a large number of connections. Connection data structure accesses cannot be eliminated and cannot easily be cached, as a modest number of connections can easily require more storage than a cache can provide. The size of a connection in the FreeBSD operating system used for the experiments is 720B, including a 192B socket (`struct socket`), a 176B protocol-independent control block (`struct inpcb`), a 232B TCP control block (`struct tcpcb`), and 120B for five timers (`struct callout`). Even a 1MB L2 cache that is entirely filled with connection data structures could only store a maximum of 1456 connections. In practice, cache conflicts and the need to store other data in the cache will dramatically reduce the number of connection data structures that can be stored.

3.4 Software Prefetching

When caching is ineffective, software prefetching is often used to reduce main memory access latencies [20]. In order to determine whether prefetching can reduce the impact of L2 cache misses in the network stack, software prefetch instructions were manually inserted into the network stack code. In the device driver, a single prefetch instruction is used to fetch the headers of a received packet when it processes the received packet. In the TCP layer, six prefetch instructions are used to fetch the protocol control block and the socket when it processes received packets. In the system call layer, six prefetch instructions are used to fetch the socket and the protocol control block when it sends data to the TCP layer. All of the above 13 instructions are `prefetcht0`, which is available on the AMD

		User	Sys. Call	TCP	IP	Ethernet	Driver	Other	Overall
No Prefetching 564Mb/s	Cycle	1969	2095	4529	647	1048	2475	533	10794
	Instr.	502	29	1360	375	399	590	53	3024
	L2 miss	2.38	2.97	5.40	0.15	0.57	1.71	0.06	10.79
Prefetching 568Mb/s	Cycle	1930	2307	4140	683	681	2546	537	10355
	Instr.	501	309	1376	375	399	588	52	3047
	L2 miss	2.37	3.78	4.88	0.21	0.07	2.28	0.06	11.21
				TCP		Ethernet			Fetch
Explicit Fetch	Cycle			3952		665			679
	Instr.			1405		399			109
	L2 miss			3.61		0.03			2.22

Table 3.1 : Profile of web server during the execution of the World Cup trace with and without software prefetching.

Athlon XP 2600+ CPU [1]. These instructions are inserted as far in advance of the actual use of their target data as possible, without modifying the overall code structure. Several are inserted a function call ahead of the actual use of their target data.

Table 3.1 shows the effects of prefetching on web server performance for the World Cup trace. *Cycles*, *Instr.*, and *L2 cache miss* show processor cycles, instructions, and L2 cache misses per packet, respectively. *Overall* accounts for all regions except *User* and *Other*. *No Prefetching*, *Prefetching*, and *Explicit Fetch* indicate different kernel configurations. Mb/s numbers show the HTTP content throughput. For all kernel configurations, the server handles 5120 simultaneous connections, and the client machines impose a 10 millisecond latency in each way.

No Prefetching shows the server profile when software prefetching is not used. *Prefetching* shows the server profile when software prefetching is used. As expected, software prefetching reduces the number of cycles in *Ethernet* and *TCP* by 367 and 389 cycles, respectively. These regions also show reductions in L2 cache misses. While the number of L2 caches misses in *Ethernet* drops close to zero, the reduction in *TCP* is much smaller. This indicates that either the prefetches are useless or they are not executed early enough. Moreover, software prefetching actually increases the number of cycles in *System Call* and *Driver* by 212 and 71 cycles, respectively. These increases are likely due to resource

contention since the CPU is fully utilized (0% idle time), and the network interface cards and the CPU compete for memory bandwidth. Due to these increases in cycles, software prefetching reduces the number of cycles spent in the network stack only by 439 cycles, and improves HTTP content throughput only by 4%.

In order to determine whether the prefetches are executed early enough or are useless, the prefetch targets in *TCP* and *Ethernet* are fetched explicitly using regular loads. These load instructions are then moved into a separate function named *Fetch*. *Explicit Fetch* in Table 3.1 shows the resulting server profile. Only those regions that are immediately affected by *Fetch* (*TCP*, *Ethernet*, and *Fetch*) are shown in the table. Since the data is explicitly fetched ahead of its use, the number of cycles and L2 cache misses in *TCP* and *Ethernet* represent the minimum number that can be achieved using software prefetching. The number of cycles and L2 cache misses in *Ethernet* when software prefetching is used (see *Prefetching*) are already close to those shown in *Explicit Fetch*, indicating that the prefetch in the device driver is executed early enough. However, *Explicit Fetch* results in much fewer cycles and L2 cache misses in *TCP* than *Prefetching*. So, software prefetching can theoretically eliminate about two more L2 cache misses per packet if it were executed earlier. Also, the remaining 3.61 L2 misses in *TCP* indicate that prefetching does not target all the L2 misses that occur in *TCP*. Finally, *Fetch* incurs 2.22 L2 cache misses, while *TCP* and *Ethernet* combined show a reduction of 2.33 L2 caches misses. These numbers are very close, indicating that the prefetch instructions are not useless.

Software prefetching can potentially eliminate the remaining 3.61 L2 cache misses in *TCP* and further improve server throughput. However, it would be difficult to execute prefetches early enough to completely eliminate misses. In addition to prefetching, one may attempt to increase spatial locality of data structures that span multiple cache lines by re-arranging their fields. However, re-arranging the fields of the connection control block structures (TCP control block and protocol independent control block) in the most to least frequently accessed order results in no measurable impact on system performance.

Finally, some recent work studied the use of helper threads to prefetch data in antici-

pation of its use [44, 85]. Such studies so far have targeted few application loops that are known to cause a majority of cache misses. It is unclear how well the helper thread would perform for the network stack of the operating system.

3.5 Summary

While the number of instructions was an issue in the past, main memory accesses are now the performance bottleneck in the network stack. These accesses are mainly to packet data and connection data structures. The modern TCP/IP network stack supports zero-copy I/O, and most NICs now implement checksum offload. These techniques eliminate memory accesses to packet data. However, main memory accesses to connections are now becoming a performance bottleneck in the network stack. A large number of connections can easily saturate the processor caches, which forces the processor to access main memory. Conventional latency hiding techniques, such as software prefetching, do not work well. First, the small number of instructions makes it difficult to initiate prefetches early enough to hide the memory latency. Moreover, the high fraction of memory accesses stresses the memory subsystem and would increase prefetching latency, which in turn requires the processor to issue prefetches even earlier.

Chapter 4

Offload through Connection Handoff

As the gap between processor and memory speeds continues to grow, the memory latency bottleneck in TCP processing (discussed in Chapter 3) becomes increasingly worse. Offloading TCP processing onto the NIC can alleviate the memory latency problem by allowing connection data structures to be stored in a fast dedicated memory on the NIC. By exploiting fast memory, the NIC can process TCP packets more efficiently than the host processor. There are different ways to achieve offload. The most straightforward approach (full offload) performs all of TCP functionalities on the NIC including connection establishments. A great disadvantage of this approach is the possibility that the NIC may become a bottleneck in the system. In contrast, connection handoff enables offload by allowing the host operating system to move established connections to the NIC, which then processes TCP packets. The handoff approach has a distinct advantage in which the operating system retains complete control over the distribution of work between the host processor and the NIC. This chapter first describes the rationale behind the use of connection handoff as a way to accomplish offload, and then presents the design of connection handoff and performance evaluations.

4.1 Rationale

A typical offload approach partitions the network stack into two components, one consisting of the layers above TCP and the other consisting of TCP and the layers below it. So, all TCP processing occurs on the network interface (full TCP offload). However, there are several problems with full offload as described below.

Computation resources. As discussed in Chapter 3, it can take upwards of 8 billion operations per second to saturate a full-duplex 10Gb/s link. It is unreasonable to expect to be able to place a multi-gigahertz processor on a space and power constrained NIC. There is no room for the required cooling and insufficient power delivery to peripherals to achieve such computation rates. The only way to efficiently achieve such computation rates is likely to be through custom hardware. This is not only more costly, but also restricts the flexibility of the system.

Memory capacity limits. Ideally, the memory on the NIC would be large enough to store connection state for all of the connections in the system. However, a memory of that size is likely to be just as slow as the host's main memory, negating the advantages of full TCP offload, and would likely consume more area than is available on a network interface. So, the memory on the NIC must be smaller than the memory on the host, in order to allow fast access, but significantly larger than the host processor's caches, in order to store more connections. These capacity constraints will place a hard limit on the number of connections that can be processed by the NIC, which is likely to degrade overall system performance.

Software architecture complexity. Offloading the network stack onto the network interface can greatly complicate the software architecture of the network subsystem [63]. Two important issues are port number assignment and IP routing. Both must be handled globally, rather than by a single network interface. Allowing each network interface to establish its own connections could potentially lead to system wide conflicts. Furthermore, IP route changes can affect multiple network interfaces, so would require coordination among offloading network interfaces. These and many other global decisions are more efficiently made by the operating system than a peripheral. Allowing distributed control greatly complicates the system design.

Connection handoff can overcome these problems by allowing the operating system to have ultimate control over the network subsystem. The operating system can choose to handoff a connection to the appropriate network interface if and only if the network inter-

face has enough computation and memory resources and implements the appropriate protocols correctly. This allows collaboration between the operating system and the network interface, enabling many of the benefits of full TCP offload while limiting the drawbacks. Connection handoff to the NIC is conceptually an easy process. The operating system first establishes a connection. Then, if it wishes, the operating system transfers the state of the connection from the main memory to the NIC and then suspends TCP processing within the operating system for that connection. Once a connection is offloaded to the NIC, the operating system must relay requests from the application to the NIC and changes in the connection state from the NIC to the application. Using handoff to offload TCP processing has a number of advantages from the viewpoint of software engineering, because the operating system establishes connections. First, the operating system can still exercise admission policies. Second, the operating system also retains control over the allocation of port numbers. Third, the operating system continues to make routing decisions. When a connection is offloaded, the current route is already known. The operating system then only needs to take proper actions, such as informing the NIC of the new route, when the route changes.

4.2 Design of Connection Handoff

In order to support connection handoff, the operating system needs modifications to the existing TCP/IP network stack. They involve creating a new path within the network stack to the offloading NIC and defining a software interface between the operating system and the offloading NIC.

4.2.1 Architecture

Figure 4.1 shows the network stack modified to support connection handoff. The left half of the figure represents the traditional network stack, while the right half (shaded) represents the new stack used by offloaded connections. The NIC now includes the socket, TCP/IP, Ethernet, and lookup layers in order to process TCP packets, in addition to the traditional

Figure 4.1 : TCP/IP network stack extended to support connection handoff.

functionalities that enable transmission and reception of Ethernet frames. The lookup layer hashes incoming TCP packets to quickly determine whether they belong to the connections on the NIC, which may slightly increase receive packet latency.

TCP packets can take two different paths within the modified stack. The dotted arrows in Figure 4.1 show the traditional data flow for the connections residing in the main memory. TCP packets are sent through the layers of the unmodified network stack. Received packets, however, must first be checked by the NIC to determine if they belong to an offloaded connection in the lookup layer. If they do not, they can be sent directly to the operating system at that point. Offloaded connections are processed entirely on the NIC, as shown by the solid arrows in the figure. These connections use the bypass layer within the operating system, as shown by the dashed arrows in the figure, which forwards data and changes in connection states between the operating system's socket and the NIC's socket in order to keep them synchronized.

Figure 4.2 shows major data structures that exist for an offloaded connection on the NIC. A connection residing in the main memory only requires those structures that belong to the host operating system. The data structures and their organization are based on the FreeBSD 4 operating system, but they should be similar in most operating systems that im-

Figure 4.2 : Important data structures of a connection.

plement the sockets API. The file, socket, and control block structures are all linked through pointers but otherwise are unaware of each other's internal implementation so that the application can access any protocol through the same sockets API. An offloaded connection has the socket, socket buffer, and protocol control block structures both in the main memory and the NIC memory. However, the operating system accesses only socket and socket buffers since the NIC processes TCP packets. The other structures such as the protocol control block are not de-allocated in case the TCP connection state needs to be restored from the NIC. The socket now contains two additional fields: a connection identifier and a pointer to the data structure representing the device driver for the NIC. These two fields are used to communicate with the NIC. The bypass layer does not introduce any additional data structures.

Socket buffers exist in both the main memory and the NIC memory. The actual user data has to exist only in the main memory. However, storing the data in the NIC memory can facilitate zero-copy I/O. When the data only exists in the main memory, the send socket buffer in the NIC simply stores a list of pairs of the physical address and length of the user data in the main memory. The user data is fetched to the NIC memory only when transmission of the data takes place. The receive socket buffer on the NIC temporarily stores newly received data before it is transferred to the receive buffer in the main memory. Once the data is transferred to main memory, it is de-allocated in NIC memory, and the

receive socket buffer on the NIC only keeps track of the number of bytes in the socket buffer. Since the actual socket buffer data resides in the main memory, the NIC only needs to store meta data such as protocol control blocks and sockets.

The above scheme minimizes the memory requirement on the NIC. In order to support zero-copy receive, the NIC may buffer the received user data for a longer period. Normally, the operating system informs the NIC of the physical addresses and lengths of pre-posted user buffers so that the NIC can directly transfer the data to the user buffer. However, if the NIC runs low of memory while buffering the user data, it can simply transfer the data to the receive buffer in the main memory.

Note that the existing event notification mechanisms, such as the traditional `select` system call, are unaffected because they are implemented in the socket layer. As shown in Figure 4.2, events are attached to the sockets and are independent of the layers below the socket layer.

4.2.2 Interface

The handoff interface resides between the bypass layer and the device driver shown in Figure 4.1. In most operating systems, it should be possible to extend the existing device driver interface to incorporate the handoff interface. The handoff interface is used to synchronize the socket in the host operating system and the corresponding socket on the NIC. The interface consists of several types of command messages provided by the NIC and by the operating system. Upon receiving a command message, the NIC or the operating system performs the tasks specified within that message. The messages are transferred between the operating system's main memory and the NIC memory through direct memory access (DMA). When sending a message to the NIC, the device driver creates a message buffer in main memory and notifies the NIC of the location (address and length) of the message, typically by accessing the NIC's registers through programmed I/O. The NIC then fetches the message through DMA and processes it. When sending a message to the operating system, the NIC creates a message buffer in the NIC memory, transfers it through DMA to

- `ch_nic_handoff(connection)`
Allocate a connection on the NIC and transfer the host connection state to the NIC. At minimum, the device driver needs to examine the socket and TCP control block and then transfer the necessary information to the NIC. The device driver gives the operating system a unique connection identifier `cid`. All subsequent commands for the connection carry the identifier.
- `ch_nic_restore(cid)`
Restore the state of the connection offloaded to the NIC. Upon receiving this command, the NIC transfers the state to the OS and de-allocates its connection.
- `ch_nic_send(cid, paddr, len, flags)`
Enqueue the pair of the physical address `paddr` and length `len` onto the socket on the NIC. The address and length specify the new user data enqueued onto the send socket buffer of the operating system. `flags` can be used to specify the type of data, such as out-of-band or in-band.
- `ch_nic_recvd(cid, len)`
Remove the first `len` bytes from the receive socket buffer.
- `ch_nic_ctrl(cid, cmd)`
Perform control operations on the connection. `cmd` stores information about the operation.
- `ch_nic_forward(cid, paddr, len)`
Forward the IP packet of length `len` bytes located at the physical address `paddr` to the NIC. Upon receiving this command, the NIC fetches the packet through DMA and processes the packet as if it had been received to the connection `cid` from the network.
- `ch_nic_post(cid, paddr, len)`
The main memory buffer of `len` bytes located at the physical address `paddr` is available for receive. This buffer is either allocated by the device driver or pre-posted by the user for zero-copy receive.

Figure 4.3 : Handoff commands provided by the NIC.

a main memory buffer that is pre-allocated for messages, and interrupts the host CPU. The operating system then processes the message.

Figure 4.3 shows the six commands exported by the NIC. The operating system uses these commands in order to offload connections and alert the NIC of new requests from the application. The operating system may attempt to initiate a connection handoff anytime it wishes through `ch_nic_handoff`. For instance, it might be appropriate to offload a connection to the NIC when the application accepts the connection through the `accept`

system call. The device driver then packages the necessary TCP and socket buffer states and the route information into a message to the NIC. Once the handoff succeeds, the operating system switches the socket's transport protocol from TCP to bypass and stores the connection identifier and pointer to the device driver in socket. At this point, the IP receive queue may still have several received packets that belong to this connection. When the TCP layer encounters those packets, it notices that the connection has been offloaded and forwards the packets to the NIC through `ch_nic_forward`. Alternatively, it may simply drop the packets and let the TCP retransmission mechanism force the client to **send** the packets again, which will then be processed by the NIC. `ch_nic_restore` is the opposite of `ch_nic_handoff` and moves the offloaded connection from the NIC to the operating system. The NIC transfers the necessary state information to the device driver, which then can set appropriate fields in socket and TCP control block in the main memory and switch the protocol from bypass back to TCP.

Any changes in the socket or TCP control block states requested by the application are forwarded to the NIC through `ch_nic_ctrl`. For example, when the application closes a socket through the `close` system call, the operating system calls a protocol specific function in charge of disconnecting the connection and de-allocating data structures. The bypass layer notifies the NIC that the user wishes to close the connection through `ch_nic_ctrl`. The actual de-allocation of data structures in both main memory and NIC memory may occur later when the TCP layer running on the NIC decides to de-allocate the structures. Socket options are also transferred through `ch_nic_ctrl`.

When the application sends data using the `write` system call, the operating system checks whether the send socket buffer has enough space and calls a TCP specific function. That function may choose to enqueue the data into the send socket buffer and take further actions. For bypass, it passes the physical address and length of the new data along with the connection identifier to the NIC using `ch_nic_send`. The data is then enqueued onto the send buffer of the operating system, and the NIC enqueues the address and length pair into its send buffer. The TCP layer on the NIC may then transmit packets containing the

- `ch_os_recv(cid, addr, len, flags)`
Enqueue the data located at the address `addr` of length `len` bytes onto the receive socket buffer of the operating system. Prior to using this command, the NIC transfers the data to the main memory through DMA. `flags` specify the type of data.
- `ch_os_ack(cid, len)`
Drop `len` bytes from the head of the send socket buffer of the operating system.
- `ch_os_ctrl(cid, cmd)`
Change the connection state (such as the socket state). `cmd` specifies the change.
- `ch_os_resource(cid, type, num)`
Advertise that `num` items of resource type `type` on the NIC are available for the operating system to use.
- `ch_os_restore(cid)`
Synchronize the connection state of the operating system and the NIC, after which the NIC de-allocates its connection. Following this command, the operating system and the NIC may exchange further messages in order to transfer the connection state.

Figure 4.4 : Handoff commands provided by the operating system.

new data.

When the application reads data from the socket using the `read` system call, the operating system removes the data from the receive socket buffer and informs the protocol of the removal. The bypass layer tells the NIC to drop the same number of bytes from its receive socket buffer through `ch_nic_recvd`. Finally, `ch_nic_post` can be used to support zero-copy receive, by informing the NIC of available user buffers.

Figure 4.4 shows the five commands provided by the operating system. The NIC uses these commands to inform the operating system of any changes in the connection state, such as the arrival of new data or connection teardown. When the NIC receives new data, it increments the byte count of the receive socket buffer, transfers the data to the main memory, and informs the operating system of the data through `ch_os_recv`. The operating system then enqueues the data onto its receive buffer. As discussed in Section 4.2.1, the NIC may actually store the data until it is requested by the user application in order to implement zero-copy receive. The NIC uses `ch_os_ack` to tell the operating system that a number of bytes have been acknowledged by the receiver. The NIC removes the address

and length pairs that correspond to the acknowledged bytes from its send socket buffer. The operating system removes the actual user data from its send socket buffer.

`ch_os_ctrl` behaves just like `ch_nic_ctrl` but in the other direction. The NIC alerts the operating system of changes in socket state such as the connection teardown initiated by the client or the de-allocation of the offloaded connection through `ch_os_ctrl`. The NIC has a finite amount of memory for storing connections and other structures such as the address and length pairs. It sends `ch_os_resource` periodically to inform the operating system of the types and numbers of currently available resources. The operating system honors the availability of resources advertised by the NIC. So, `ch_os_resource` serves as a flow control mechanism for the NIC. Finally, the NIC may ask the operating system to move the offloaded connection back to the main memory through `ch_os_restore`. For instance, it can be used to move the connections in `TIME_WAIT` state back to the operating system since they simply wait for timers to expire and consume precious NIC memory.

`ch_nic_restore` and `ch_os_restore` are useful when the system needs to move offloaded connections from the NIC back to the operating system during their lifetime, due to one of the following four events: NIC hardware failure, network link failure, route change, and exhaustion of NIC resources. First, when the NIC fails without notice, which is usually detected by the device driver that checks the hardware status periodically, the operating system has no choice but to drop the offloaded connections since it cannot retrieve the connection states from the NIC. However, in case of the other three events, the system can adapt to the events by restoring the offloaded connections. When the route changes such that the NIC is no longer the correct interface, the operating system needs to either move the offloaded connections back to the main memory through `ch_nic_restore` or drop them. The system may choose to implement either option. Some systems may be able to cope with route changes transparently. For such systems, dropping the connections weakens the tolerance to dynamic route changes while the connections are still alive. When the network link fails, the NIC can no longer send or receive packets. In most operating systems, this event amounts to a route change, so it can take the actions mentioned above.

Figure 4.5 : Example sequence of events and commands exchanged between the operating system and the NIC.

Finally, the NIC may run out of resources during the normal operation. It may drop some of the offloaded connections and ask the operating system to not offload any more connections using `ch_os_resource`. Alternatively, it can force the operating system to restore offloaded connections using `ch_os_restore`.

4.2.3 Using the Handoff Interface

Figure 4.5 illustrates the use of the handoff interface. The sequence shown in the figure roughly follows the lifetime of a TCP connection through which the server accepts a connection from a client, receives a user request, sends a response, and then closes the connection following the client's close.

In (1), the operating system offloads the established connection using `ch_nic_handoff`. In (2), the NIC receives data (user request) from the client, transfers the data to the main memory, and enqueues it into the receive socket buffer in the operating system using `ch_os_rcv`. Suppose that the NIC chooses not to store the actual data in the NIC memory. Then, the NIC's receive socket buffer has no data, but the NIC increments the buffer's byte count in order to account for the received data that has not been read by the application. In (3), the application reads the data from the receive socket buffer using the `read` system call. The operating system tells the NIC the number of bytes read by the application using `ch_nic_rcvd`. The NIC decrements the byte count of its receive socket buffer by the same number.

In (4), the application sends data (user response) using the `write` system call. The

operating system enqueues the data into the send socket buffer and informs the NIC of the data's physical address and length using `ch_nic_send`. If the data consists of multiple physically non-contiguous buffers, `ch_nic_send` is used once for each contiguous buffer. The NIC enqueues the physical address and length pair into its send socket buffer. It computes the number of bytes to be sent, transfers the data from the main memory to form complete packets, and transmits them. Until acknowledged by the client, the operating system must not move the data in the main memory to a different location since the NIC may need to retransmit the data. In (5), the data is acknowledged by the client. The NIC first drops the address and length pairs corresponding to the acknowledged bytes and uses `ch_os_ack` to tell the operating system to drop the same number of bytes from its send socket buffer.

In (6), the client closes the connection. In the BSD TCP/IP network stack, this changes the socket state to signify that the socket cannot receive any more data. The NIC tells the operating system of the state change using `ch_os_ctrl`. In practice, at this point, the NIC may want to transfer the connection back to the operating system, using `ch_os_restore`, in order to free up resources for active connections on the NIC. However, this is not required, and the following steps would occur if the NIC maintains control of the connection. In (7), the application closes the connection through the `close` system call. The operating system alerts the NIC of the user close using `ch_nic_ctrl`. The NIC changes its socket state accordingly and initiates the connection teardown. In (8), when the connection is finally de-allocated, the NIC tells the operating system to de-allocate its connection using `ch_os_ctrl`. At this point, the connection ceases to exist.

4.3 Connection Handoff Framework

The connection handoff interface described in Section 4.2 allows the system to control the division of work (connections) between the host processor and the NIC. The system should use this ability to achieve maximum performance improvements. Conceptually, this can be accomplished through the following three objectives: (1) ensure that connection

Figure 4.6 : Framework for connection handoff.

handoff does not hurt the performance of connections that are handled by the host operating system, (2) hand off as many connections to the NIC as the NIC resources will allow while avoiding overloading the NIC, and (3) select and hand off those connections that can best utilize the available NIC resources. (1) and (2) ensure that the use of connection handoff never degrades overall system performance, even if the NIC has limited compute power and memory. (2) and (3) ensure that the NIC resources are utilized as much as possible for the given workload. So, the performance improvement from connection handoff will be proportional to the processing power of the NIC. Under no circumstances, the system will see performance degradation as a result of connection handoff.

Figure 4.6 illustrates the framework for implementing policies that aim to achieve the objectives outlined above. *Priority-based Packet Processing* dictates the order of packet processing performed on the NIC and aims to ensure that the use of handoff does not hurt the performance of connections on the host operating system. *Load Control* regulates the load on the NIC and aims to keep the NIC fully utilized while avoiding overloading the NIC. *Connection Selection* decides which connections are handed off to the NIC and aims to identify connections that can best utilize the NIC resources. The following sections describe these policies in detail.

Offloaded Connections	Packet Priority	Host		NIC		Requests/s
		Packet Delay (usec) Send	Receive	Idle (%)	Idle (%)	
0	<i>no handoff</i>	2	2	0	62	23031
256	FCFS	3	3	0	49	23935
1024	FCFS	679	301	62	3	16121
1024	Host first	10	6	0	5	26663

Table 4.1 : Impact of a heavily loaded NIC on the networking performance of a simulated web server. Packet delays are median values, not averages. Idle represents the fraction of total cycles that are idle.

4.3.1 Priority-based Packet Processing

With connection handoff, the NIC must process two types of packets, and the order of processing them can negatively affect the performance of connections on the host. First, *host packets* belong to those connections handled by the host operating system. The NIC simply needs to transfer these packets between the network and main memory. Since very little processing is involved with data transfers, the NIC can process host packets fairly quickly. Traditional NICs that do not support connection handoff only process this type of packets. Second, *NIC packets* belong to those connections handed off to the NIC. Since the NIC must perform TCP processing, NIC packets require significantly more processing than host packets. So, NIC packet processing can delay the processing of host packets, which in turn may reduce the throughput of connections that remain within the host operating system. Worse, if the NIC becomes loaded with TCP processing and runs out of packet buffers, then it must drop host packets. Packet drops further reduce the throughput of connections on the host operating system.

Table 4.1 illustrates the effect of connection handoff on host packet delays and overall performance of a web workload that uses 2048 simultaneous connections. See Section 4.4 for detailed experimental **setup**. The first row of the table shows the performance of the system when no connections **are** handed off to the NIC. In this case, the web server is able to satisfy 23031 requests **per second** with a median packet processing time on the

NIC of only 2 μ s for host packets. The second row shows that when 256 of the 2048 connections (12.5%) are handed off to the NIC, the request rate increases by 4% with only slight increases in host packet processing time. However, as shown in the third row, when 1024 connections are handed off to the NIC, the NIC is nearly saturated and becomes the bottleneck in the system. The median packet delay on the NIC for host packets increases dramatically, and the NIC also drops received packets as the receive queue fills up. As a result, the server's request rate drops by 33%.

For both the second and third rows of Table 4.1, the NIC processes all packets on a first-come, first-served (FCFS) basis. As explained above, the NIC delays host packets too much and may also drop them when it becomes heavily loaded. Since the NIC must perform TCP processing for NIC packets, additional delays to host packets are inevitable. However, such delays need to be minimized in order to maintain the performance of connections on the host operating system. Since host packets require very little processing on the NIC, they can be given a priority over NIC packets without significantly reducing the performance of connections that have been handed off to the NIC. The priority queues, shown in Figure 4.6, enable such a prioritization. As discussed in Section 4.2, all received packets must first go through the lookup layer, which determines whether a packet belongs to a connection on the NIC. Once the lookup task completes, the NIC now forms two queues. One queue includes only host packets, and the other queue stores only NIC packets. The NIC also maintains a queue of host packets to be sent and another queue of handoff command messages from the host operating system. In order to give priority to host packets, the NIC always processes the queue of received host packets and the queue of host packets to be sent before NIC packets and handoff command messages. The fourth row of Table 4.1 shows the impact of this *host first* packet processing policy in which the NIC always processes host packets before it processes NIC packets. With the *host first* policy on the NIC, the median packet delay of host packets is about 6–10 μ s even though 1024 connections are handed off to the NIC. Handoff now results in a 16% improvement in request rate. Thus, this simple prioritization scheme can be an effective mechanism to ensure that TCP processing on the

NIC does not hurt the performance of connections that are handled by the host operating system. Further evaluation is presented in Section 4.6.

The above scheme gives a strict priority to host packets over NIC packets. Doing so does create a possibility of a constant stream of host packets starving NIC packets. It is easy to extend the basic scheme and to guarantee forward progress of NIC packets. For instance, the NIC can periodically invert the priority and process NIC packets before it processes host packets. However, because host packets take little time to process, it is unlikely that host packets can starve NIC packets in practice, and such situation never occurs for the workloads studied in this dissertation.

4.3.2 Load Control

In order to fully utilize the NIC resources without overloading the NIC, the NIC must determine an optimal number of connections that it can handle. Due to the finite amount of memory on the NIC, there is a hard limit on the total number of connections on the NIC. However, depending on the workload and the available processing resources on the NIC, the NIC may become saturated well before the number of connections reaches the hard limit. Thus, the optimal number of connections depends on workloads, and the NIC must dynamically figure out the optimal number. Ideally, the number of connections on the NIC should always be kept at optimal; the NIC resources are utilized as much as possible, but not to the point where the performance of connections on the NIC degrades. However, doing so is very difficult, if not impossible. This section describes a mechanism that measures the load on the NIC and then dynamically adapts the number of connections to the current load.

As discussed in Section 4.3.1, the NIC maintains a queue of received NIC packets. As the load on the NIC increases, the NIC cannot service the receive queue as promptly. Therefore, the number of packets in the receive queue (queue length) is a good indicator of the current load on the NIC. This holds for send-dominated, receive-dominated, and balanced workloads. For send-dominated workloads, the receive queue mainly stores acknowledg-

ment (ACK) packets. A large number of ACKs on the receive queue indicate that the load is too high because the host operating system is sending data much faster than the NIC can process ACKs returning from a remote machine. For receive-dominated workloads, the receive queue mainly stores data packets from remote machines. A large number of data packets on the receive queue indicates that data packets are being received much faster than the NIC can process and acknowledge them. In balanced workloads, a combination of the above factors will apply. Therefore, a large number of packets in the receive queue indicates that the NIC is not processing received packets in a timely manner which will increase packet delays for all connections.

The NIC uses six parameters to control the load on the network interface: *Hard_limit*, *Soft_limit*, *Qlen*, *Hiwat*, *Lowat*, and *Cnum*. *Hard_limit* is the maximum possible number of connections that can be handed off to the network interface and is determined based on the amount of physical memory available on the network interface. *Hard_limit* is set when the network interface firmware is initialized and remains fixed. *Soft_limit* is the current maximum number of connections that may be handed off to the NIC. This parameter is initially set to *Hard_limit*, and is always less than or equal to *Hard_limit*. *Qlen* is the number of NIC packets currently on the receive packet queue. *Hiwat* is the high watermark for the receive packet queue. When *Qlen* exceeds *Hiwat*, the network interface is overloaded and must begin to reduce its load. A high watermark is needed because once the receive packet queue becomes full, packets will start to be dropped, so the load must be reduced before that point. Similarly, *Lowat* is the low watermark for the receive packet queue, indicating that the network interface is underloaded and should allow more connections to be handed off, if they are available. As with *Hard_limit*, *Hiwat* and *Lowat* are constants that are set upon initialization based upon the processing capabilities and memory capacity of the network interface. For example, a faster NIC with larger memory can absorb bursty traffic better than a slower NIC with smaller memory, so it should increase these values. Currently, *Hiwat* and *Lowat* need to be set empirically. However, since these values only depend on the hardware capabilities of the network interface, only the network interface manufacturer

Figure 4.7 : State machine used by the NIC firmware to dynamically control the number of connections on the NIC.

would need to tune the values, not the operating system. Finally, *Cnum* is the number of currently active connections on the NIC.

Figure 4.7 shows the state machine employed by the NIC in order to dynamically adjust the number of connections. The objective of the state machine is to maintain *Qlen* between *Lowat* and *Hiwat*. When *Qlen* grows above *Hiwat*, the NIC is assumed to be overloaded and should attempt to reduce the load by reducing *Soft_limit*. When *Qlen* drops below *Lowat*, the NIC is assumed to be underloaded and should attempt to increase the load by increasing *Soft_limit*.

The state machine starts in the MONITOR state. When *Qlen* becomes greater than *Hiwat*, the NIC reduces *Soft_limit*, sends a message to the device driver to advertise the new value, and transitions to the DECREASE state. While in the DECREASE state, the NIC waits for connections to terminate. Once *Cnum* drops below *Soft_limit*, the state machine transitions back to the MONITOR state to assess the current load. If *Qlen* decreases below *Lowat*, then the NIC increases *Soft_limit*, sends a message to the device driver to notify it of the new *Soft_limit*, and transitions to the INCREASE state. In the INCREASE state, the NIC waits for new connections to arrive. If *Cnum* increases to *Soft_limit*, then the NIC transitions to the MONITOR state. However, while in the INCREASE state, if *Qlen* increases above *Hiwat*, then the NIC reduces *Soft_limit*, sends a message to the device driver to alert it of the new value, and transitions to the DECREASE state. The state machine is simple and runs only when a packet arrives, the host hands off a new connection to the NIC, or an existing

connection terminates. Thus, the run-time overhead of the state machine is insignificant.

As mentioned above, the NIC passively waits for connections to terminate while in the DECREASE state. Instead, the NIC may also actively restore the connections back to the host operating system and recover from an overload condition more quickly. The state machine described above easily supports such active restoration to the operating system. However, for this to be effective, the NIC would also need a mechanism to determine which connections are generating the most load, so should be restored first. Active restoration can also be used when the NIC tries to transition to the INCREASE state but cannot do so because the number of connections has already reached the hard limit. In this case, the NIC can actively restore connections that are either idle or transferring few packets and reclaim memory. The host operating system will then be able to hand off new connections. Actively restoring connections in this manner was not necessary for the workloads evaluated in this dissertation, but it may help improve performance for other types of workloads.

Finally, the length of the receive packet queue is easy to exploit and requires no extra accounting because the NIC must keep track of the queue length as a part of its normal operations without dynamic load control. It is possible to use other measures of load. For instance, instead of using the queue length, the NIC can track the changes (gradient) in the queue length. A continuous increase is likely to indicate that an overload is imminent. The NIC may also compute the amount of idle time or the packet rate and use it to control the load. However, these measures are not as straightforward to use as the queue length.

4.3.3 Connection Selection

When connections have different characteristics such as packet rate and lifetime, selecting which connections to hand off to the NIC influences the utilization of NIC resources, which in turn affects performance improvements from handoff. The operating system should select those connections that utilize the NIC most, in order to maximize performance improvements. For instance, suppose the operating system needs to pick one of the following two connections. One connection is long-lived and achieves high packet rates. The other

connection is short-lived and achieves low packet rates. The operating system should hand off the long-lived connection with high packet rates since it utilizes the NIC much more and leads to a greater performance improvement. This example only involves two connections. Ideally, connection selection should examine all the established connections in the system, compute their resource usage, and then determine a set of connections that lead to a maximum utilization of the available NIC resources. However, this approach is not practical since examining all connections may require significant processing power and determining a connection's resource usage would require future knowledge. Instead, this section presents a restricted form of connection selection and discusses two specific selection policies.

The connection selection policy component of the framework depicted in Figure 4.6 decides whether the operating system should attempt to hand off a given connection. So, the policy only needs to make a binary decision for a single connection. It may exploit such information as the application type (based on port number) and the connection's past resource usage when making a decision. As described previously, the device driver performs the actual handoff. The operating system may attempt handoff at any time. For instance, the operating system may hand off a connection right after it is established, or when a packet is sent or received. When the handoff attempt fails, the operating system can try to handoff the connection again in the future. For simplicity, the current framework invokes the selection policy upon either connection establishments or send requests by the application and does not connections for handoff if the first handoff attempt for that connection fails.

The simplest connection selection policy is first-come, first-served. If all connections in the system have similar packet rates and lifetimes, then this is a reasonable choice, as all connections will benefit equally from offload. However, as mentioned above, if connections in the system exhibit widely varying packet rates and lifetimes, then it is advantageous to consider the expected benefit of offloading a particular connection. These properties are highly dependent on applications, so a single selection policy would not be able to perform

well for all applications. Since applications typically use specific ports, the operating system should be able to employ multiple application-specific (per-port) connection selection policies.

Furthermore, the characteristics of the NIC can influence the types of connections that should be offloaded. Some offload processors may only be able to handle a small number of connections, but very quickly. For such offload processors, it is advantageous to hand off connections with high packet rates in order to fully utilize the processor. Other offload processors may have larger memory capacities, allowing them to handle a larger number of connections, but not as quickly. For these processors, it is more important to hand off as many connections as possible.

The expected benefit of handing off a connection is the packet processing savings over the lifetime of the connection minus the cost of the handoff. Here, the lifetime of a connection refers to the total number of packets sent and/or received through the connection. Therefore, it is clear that offloading a long-lived connection is more beneficial than a short-lived connection. The long-lived connection would accumulate enough per-packet savings to compensate for the handoff cost and also produce greater total saving than the short-lived connection during its lifetime.

In order for the operating system to compute the expected benefit of handing off a connection, it must be able to predict the connection's lifetime. Fortunately, certain workloads, such as web requests, show characteristic connection lifetime distributions, which can be used to predict a connection's lifetime. Figure 4.8 shows the distribution of connection lifetimes from several web workloads. The figure plots the cumulative fraction of sent packets and sent bytes of all connections over the length of the run, sorted by lifetime. As shown in the figure, there are many short-lived connections, but the number of packets and bytes due to these connections account for a small fraction of total packets and bytes. For example, half of the connections are responsible for sending less than 10% of all packets (and bytes) for all of the workloads. The other half of the connections send the remaining 90% of the packets (and bytes). In fact, more than 45% of the total traffic is handled by less than 10%

Figure 4.8 : Distribution of connection lifetimes from SPECweb99 and the IBM and World Cup traces. Lifetimes are expressed in the number of sent packets and bytes. Connection rank is based on the number of sent packets.

Percentile	IBM	World Cup	SPECweb99
10	2 (99)	3 (99)	2 (99)
20	6 (98)	5 (98)	2 (98)
30	8 (97)	7 (97)	3 (98)
40	12 (95)	10 (96)	4 (97)
50	18 (92)	14 (93)	5 (96)
60	31 (87)	20 (90)	6 (94)
70	54 (78)	30 (85)	15 (92)
80	72 (64)	46 (77)	37 (83)
90	103 (45)	79 (64)	87 (65)

Table 4.2 : Number of packets sent by connections from the same workloads used in Figure 4.8. Percentile is based on the number of sent packets. The numbers in parenthesis show the percentage of all packets that are sent by the connections whose percentile equals or exceeds the specified percentile.

of the connections.

Table 4.2 shows the actual number of packets sent over connections at several percentiles. Only packets with non-zero payloads are counted. For instance, an 80th percentile connection in SPECweb99 sends 37 packets, and connections greater than the 80th percentile account for 83% of all sent packets. In other words, all connections less than the

80th percentile account for only $100 - 83 = 17\%$ of all sent packets. The data shown in Figure 4.8 and Table 4.2 assumes that persistent connections are used. A persistent connection allows the client to reuse the connection for multiple requests. Persistent connections increase the average lifetime, but not the shape of distribution of lifetimes. Previous studies have shown that web workloads exhibit this kind of distribution [7, 18, 21]. The operating system may exploit such distribution in order to identify and hand off long-lived connections. For instance, since the number of packets transferred over a long-lived connection far exceeds that of a short connection, the system can use a threshold to differentiate long and short-lived connections. The operating system can simply keep track of the number of packets sent over a connection and hand it off to the NIC only when the number reaches a certain threshold.

4.4 Experimental Setup

Performance of connection handoff is evaluated using both a real prototype and simulations. The real prototype is built on the FreeBSD 4.7 operating system and shows reductions in instruction and memory bandwidth on the host processor. However, it does not show performance improvements due to the limited performance of available network interfaces. There are no offload controllers with open specifications, to the author's best knowledge. So, an extended version of the full-system simulator Simics [53] is used to evaluate the performance gains from handoff. Simics models the system hardware with enough detail that it can run complete and unmodified operating systems.

4.4.1 Prototype

A prototype was built using a uniprocessor system and a programmable Gigabit Ethernet NIC (3Com 710024). The system has a single AMD Athlon XP 2600+ CPU (2.1GHz clock rate and 256KB of L2 cache) and runs a modified FreeBSD 4.7 that supports connection handoff. The NIC is based on the Alteon Tigon controller, which was designed in 1997. The Tigon includes two simplified 88MHz MIPS cores and hardware that interfaces with

the Ethernet wire and the PCI bus. The NIC also has 1MB SRAM. This NIC is used for the prototype because it is the only programmable Ethernet NIC for which full documentation on the hardware architecture, software architecture, and firmware download mechanisms is publicly available. The NIC firmware implements the TCP/IP stack and all the features of the handoff interface discussed in Section 4.2 except for zero-copy receive and `ch_nic_restore`. Also, `ch_os_restore` can only restore connections that enter the `TIME_WAIT` state, and `ch_nic_handoff` does not handle connections with non-empty socket buffers.

All modifications to the operating system are isolated within the network stack and the device driver. The handoff interface functions are implemented in the driver, and the bypass layer is added to the network stack. The TCP layer now tries to handoff connections both when they are established (detected in the `tcp_input` function) and when the application accepts established connections (detected in the `tcp_usr_accept` function). The device driver keeps track of the available resources on the NIC and may reject handoff or other types of requests when resources become scarce. Once a connection is offloaded to the NIC, the operating system switches the connection's protocol to bypass by modifying a field in the socket structure that points to the table of functions exported by the protocol (`struct protosw`). Requests from the socket layer are then forwarded to the bypass layer instead of the TCP layer. As discussed previously, the IP queue may have packets destined to connections that have just been offloaded. Currently, they are forwarded to the NIC.

The NIC's TCP/IP stack is based on the stack implementation of FreeBSD 4.7. The TCP layer itself has trivial modifications that enable interactions with the handoff interface. All the functions run on one MIPS core. The other core is used for profiling purposes. The instructions require about 160KB of storage. The rest of the 1MB are used for dynamic memory allocations and Ethernet transmit and receive buffers. The firmware currently allows for a maximum of 256 TCP connections. Currently, the maximum TCP throughput over an offloaded connection is only about 110Mb/s (using 1460B TCP segments) on the

	Configuration
CPU	Functional, single-issue, 2GHz x86 processor Instantaneous instruction fetch
L1 cache	64KB data cache Line size: 64B, associativity: 2-way Hit latency: 1 cycle
L2 cache	1MB data cache Line size: 64B, associativity: 16-way Hit latency: 15 cycles Prefetch: next-line on a miss
DRAM	DDR333 SDRAM of size 2GB Access latency: 195 cycles
NIC	Functional, single-issue processor Varied instruction rates for experiments Varied maximum number of connections for experiments 10Gb/s wire

Table 4.3 : Simulator configuration.

prototype system. The MIPS core is fully saturated and is the bottleneck. Even if the firmware is parallelized across the two cores on the controller, the maximum throughput would increase to around 200Mb/s.

4.4.2 Simulation Setup

Simics is a functional full-system simulator that allows the use of external modules to enforce timing. For the experiments, Simics has been extended with a memory system timing module and a network interface card module. The processor core is configured to execute one x86 instruction per cycle unless there are memory stalls. The memory system timing module includes a cycle accurate cache, memory controller, and DRAM simulator. All resource contention, latencies, and bandwidths within the memory controller and DRAM are accurately modeled [77]. Table 4.3 summarizes the simulator configuration.

The network interface simulator models a MIPS processor, 32 MB of memory, and several hardware components: PCI and Ethernet interfaces and a timer. The PCI and Ethernet interfaces provide direct memory access (DMA) and medium access control (MAC)

capabilities, respectively. These are similar to those found on the Tigon programmable Gigabit Ethernet controller from Alteon [5]. Additionally, checksums are computed in hardware on the network interface. The firmware of the NIC uses these checksum values to support checksum offload for host packets and to avoid computing the checksums of NIC packets in software. The NIC does not employ any other hardware acceleration features such as hardware connection lookup tables [36]. The processor on the NIC runs the firmware and executes one instruction per cycle at a rate of 400, 600, or 800 million instructions per second (MIPS). The instruction rate is varied to evaluate the impact of NIC performance. Modern embedded processors are capable of such instruction rates with low power consumption [26]. At 400MIPS, the NIC can achieve 1Gb/s of TCP throughput for one offloaded connection and another 1Gb/s for a host connection simultaneously, using maximum-sized 1518B Ethernet frames. The maximum number of connections that can be stored on the NIC is also varied in order to evaluate the impact of the amount of memory dedicated for storing connections. The network wire is set to run at 10Gb/s in order to eliminate the possibility of the wire being the bottleneck. The local I/O interconnect is not modeled due to its complexity. However, DMA transfers still correctly invalidate processor cache lines, as others have shown the importance of invalidations due to DMA [11].

The NIC firmware implements the TCP/IP stack and the handoff interface, which allows the operating system to offload connections. The firmware's TCP/IP stack is a modified version of the FreeBSD 4.7 stack implementation. The firmware also implements priority-based packet processing and the load control mechanism described in Section 4.3. These features can be disabled in order to evaluate their impact.

4.4.3 Web Workloads

The experiments use SPECweb99 and two real web traces to drive the Flash web server [69]. These are the same web workloads used in Chapter 3. They are briefly described below. SPECweb99 emulates multiple simultaneous clients. Each client issues requests for both static content (70%) and dynamic content (30%) and tries to maintain its bandwidth be-

tween 320Kb/s and 400Kb/s. The request sizes are statistically generated using a Zipf-like distribution in which a small number of files receive most requests. For static content, Flash sends HTTP response data through zero-copy I/O (the `sendfile` system call). All other types of data including HTTP headers and dynamically generated responses are copied between the user and kernel memory spaces.

The two web traces are from an IBM web site and the web site for the 1998 Soccer World Cup. A simple trace replayer program reads requests contained in the traces and sends them to the web server [10]. Like SPECweb99, the client program emulates multiple simultaneous clients. Unlike SPECweb99, it generates requests for static content only and sends new requests as fast as the server can handle. Both the replayer and SPECweb99 use persistent connections by default. The replayer uses a persistent connection for the requests from the same client that arrive within a fifteen second period in the given trace. SPECweb99 statistically chooses to use persistent connections for a fraction of all requests. To compare SPECweb99 against the two traces, the experiments also evaluate SPECweb99 that uses a non-default configuration and sends only static content requests.

For all simulation experiments, the first 400000 packets are used for warming up the simulators, and measurements are taken during the next 600000 packets. Many recent studies based on simulations use purely functional simulators during the warmup phase to reduce simulation time. However, one recent study shows that such method can produce misleading results for TCP workloads and that the measurement phase needs to be long enough to cover several round trip times [37]. For the experimental results presented in this chapter, the warmup phase simulates timing, and 600000 packets lead to at least one second of simulated time for the experiments presented in this chapter.

4.5 Prototype Results

Experiments on the real prototype focus on reductions in instructions and memory accesses due to the use of handoff. Performance improvements are simulated as discussed in the previous section, and are presented in Section 4.6.

Figure 4.9 : Execution profiles of the network stack of the prototype.

Figure 4.9 shows processor cycles, instructions, and L2 cache misses in the host network stack with and without handoff. The statistics are per TCP packet sent and received by the system. The first six groups show the numbers in each layer of the stack, and the last group shows the total numbers. Each group has eleven bars representing different experiments. The profile data for the first nine bars are collected while a microbenchmark program sends 1460B messages across varying number of connections. The last two bars are generated by SPECweb99 with 64 clients. The legends show the number of connections handled by the operating system and the NIC. For instance, OS 0//NIC 256 indicates that all 256 connections are handled by the NIC. When all connections are offloaded onto

Figure 4.10 : Messages exchanged across the local I/O interconnect.

the NIC, as shown by OS 0//NIC 1 and OS 0//NIC 256, handoff reduces the number of instructions executed in the host stack and the number of cache misses, thereby reducing the cycles spent in the stack. The reductions are more significant with 256 connections (84% reduction in cycles) than with the single connection (76% reduction in cycles) because the connections start to cause noticeable L2 cache misses in the host network stack without handoff. With handoff, the number of L2 cache misses drops down to near zero since all 256 connections reside on the NIC, and the memory footprint size becomes smaller than the L2 cache. When 512 connections are used, half the connections are offloaded onto the NIC, and handoff now reduces cycles by 36%, not as much as when all connections are offloaded. As the number of connections increases to 1024, and only 256 are offloaded, the benefits from handoff further decrease, as expected, to a 16% reduction in processor cycles. Overall, the bypass layer introduces less than 140 instructions and has negligible impact on the memory footprint size, as shown by the number of L2 cache misses. With handoff, even the device driver requires fewer instructions and cycles, showing that the handoff interface can be implemented efficiently.

Using multiple NICs can help increase the total number of offloaded connections. OS 0//NIC1 256//NIC2 256 shows the profile when 512 connections are offloaded onto two NICs, 256 on each. When compared against OS 256//NIC 256, two NICs can further re-

duce cache misses and cycles than one NIC. The use of multiple NICs has little impact on the number of instructions executed or cycles spent in the device driver and the bypass layer because their memory footprint size is largely fixed. This shows that the handoff interface works transparently across multiple NICs, and that the system may gain performance by adding additional NICs to the system. Finally, the profiles for SPECweb99 show similar trends as the microbenchmark. With handoff, instructions and L2 cache misses both decrease, leading to about 54% drop in the number of processor cycles spent to process a packet.

Without handoff, packet headers as well as payloads must be transferred across the local I/O interconnect, such as the PCI bus. With handoff, only payloads are transferred, thereby reducing bus traffic. The device driver and the NIC must also transfer command messages in order to inform each other of new events (for example, there are packets to be sent or received packets). One may believe that handoff increases the volume of message traffic since it employs many types of command messages. However, it can actually decrease the volume of message traffic. Figure 4.10 shows the number of messages per TCP packet transferred across the PCI bus. `ch_nic_post` is not implemented as a message and is not shown. The device driver maintains a ring of buffer descriptors for free buffers using a consumer index and a producer index. A buffer descriptor contains the physical address and length of a buffer. The device driver allocates main memory buffers, sets up the descriptors, and simply writes the producer index into a NIC register through programmed I/O. The NIC then fetches the descriptors through DMA. The same mechanism is used with and without handoff, so comparison is unnecessary. Receive and transmit descriptors are 32B and 16B, respectively. All of the handoff messages are 16B, except for `ch_nic_handoff`, `ch_nic_restore`, and `ch_os_restore`. These messages need a minimum of 16B, followed by a variable number of bytes depending on the socket state. For instance, `ch_nic_handoff` currently requires at least 96B of data if the send socket buffer is empty at the time of handoff. If not, it would need to transfer more information about the socket buffer.

	Handoff	TCP	IP/Eth.	Other	Total
Cycles	794	4033	529	1417	6773
Instr.	413	1402	169	1130	3144

Table 4.4 : Profile of the firmware.

In Figure 4.10, the bars labeled TCP send show the number of messages while the microbenchmark program sends 1460B TCP segments across 256 connections. Without handoff, the host TCP creates a buffer for TCP/IP/Ethernet headers, separate from the packet data. Since they are non-contiguous in physical memory, each sent packet needs two transmit descriptors, one for the header buffer and another for the packet data. The headers and packet data of a received packet are stored contiguously on the NIC, so transferring it requires only one receive descriptor. There are almost equal numbers of sent and received packets in this experiment, explaining about 1 transmit descriptor per packet and 0.5 receive descriptors per packet. With handoff, 256 connections are first offloaded onto the NIC using 256 `ch_nic_handoff` messages. Since the host operating system no longer creates headers, each 1460B message sent just needs one `ch_nic_send`. The socket layer is unaware of the maximum transmission unit. When sending a large amount of data, the host only needs `ch_nic_send` per 4KB page. In contrast, TCP requires at least one transmit descriptor for each 1460B packet. The NIC coalesces multiple ACKs, which explains why there are far fewer `ch_os_ack` messages than there are ACK packets received. The bars labeled SPECweb show the number of messages while SPECweb99 emulates 64 clients. There are both sends and receives, as well as connection establishments and teardowns. Handoff still exchanges far fewer messages across the bus. As mentioned above, handoff command messages are smaller or have the same size as transmit and receive, so handoff actually reduces the volume of message traffic as well as the number of messages. Freimuth *et al.* recently reported similar findings that offload can reduce traffic on the local I/O interconnect [33]. Reduced message traffic can lead to reduced interrupt rates as well. However, interrupt rates can be misleading because interrupt coalescing factors arbitrarily affect the interrupt rates.

Figure 4.11 : Profiles of the simulated web server during the execution of the World Cup and IBM traces.

Finally, The NIC executes about 3100 instructions to process one packet. This number is close to the number of instructions executed on the host CPU without handoff. Table 4.4 shows the profile of the firmware. TCP and IP/Eth. account for the TCP, IP, and Ethernet layers. Handoff is the firmware's handoff interface that communicates with the device driver. Other includes all the other tasks performed by the firmware such as transmit or receive of Ethernet frames. The NIC currently stalls a large number of cycles due to frequent memory accesses to the off-chip SRAM on the NIC.

4.6 Simulation Results

This section presents performance improvements from handoff and the impact of handoff policies described in Section 4.3.

4.6.1 Priority-based Packet Processing and Load Control

Figure 4.11 shows the execution profiles of the simulated web server using various configurations. The Y axis shows abbreviated system configurations (see Table 4.5 for an explanation of the abbreviations). The first graph shows the fraction of host processor cycles spent in the user application, the operating system, and idle loop. The second graph shows the amount of idle time on the NIC. The third and fourth graphs show connection and packet

Configuration shorthands have the form
Workload–NIC Connections–Packet Priority–Load Control–Selection Policy–NIC MIPS.

Workload	Web server workload W: World Cup, 2048 clients I: IBM, 2048 clients D: SPECweb99, 2048 clients S: SPECweb99-static, 4096 clients
NIC Connections	Maximum number of connections on the NIC 0 means handoff is not used.
Packet Priority	<i>host first</i> priority-based packet processing on the NIC P: Used N: Not used
Load Control	Load control mechanism on the NIC L: Used N: Not used
Selection Policy	Connection selection policy used by the operating system FCFS: First-come, first-served Tn: Threshold with value n
NIC MIPS	Instruction rate of the NIC in million instructions per second

Table 4.5 : Configuration shorthands used in Section 4.6

rates. These graphs also show the fraction of connections that are handed off to the NIC, and the fraction of packets that are consumed and generated by the NIC while processing the connections on the NIC. The last two graphs show server throughput in requests/s, and HTTP content megabits/s. HTTP content throughput only includes HTTP response bytes that are received by the client. Requests/s shows the request completion rates seen by the client.

W-0-N-N-FCFS-400 in Figure 4.11 shows the baseline performance of the simulated web server for the World Cup trace. No connections are handed off to the NIC. The host processor has zero idle time, and 57% of host processor cycles (not shown in the figure) are spent executing the network stack below the system call layer. Since the NIC has 62% idle time, handoff should be able to improve server performance. In *W-256-N-N-FCFS-400*, the NIC can handle a maximum of 256 connections at a time, and the host operating system

hands off connections in a FCFS order as soon as they are established. 13% of connections are handed off to the NIC as expected, since there are 256 connections on the NIC and 2048 simultaneous connections in the system ($\frac{256}{2048} = 12.5\%$). The request rate improves by 4%.

In *W-1024-N-N-FCFS-400*, the NIC can handle a maximum of 1024 connections at a time. At first, 2048 connections are established, and 1024 of them are handed off to the NIC. As the NIC becomes nearly saturated with TCP processing (only 3% idle time), it takes too long to deliver host packets to the operating system. On average, it now takes more than 1 millisecond for a host packet to cross the NIC. Without handoff, it takes less than 10 microseconds. The 62% idle time on the host processor also shows that host packets are delivered too slowly. So, the connections on the NIC progress and terminate much faster than the connections on the host. When the client establishes new connections, they are most likely to replace terminated connections on the NIC, not the host. Consequently, the NIC processes a far greater share of new connections than the host. Overall, 88% of all connections during the experiment are handed off to the NIC. Note that at any given time, roughly half the active connections are being handled by the NIC and the other half are being handled by the host. Since the NIC becomes a bottleneck in the system and severely degrades the performance of connections handled by the host, the request rate drops by 33%. This configuration clearly shows that naive offloading can degrade system performance. In *W-1024-P-N-FCFS-400*, the NIC still has a maximum of 1024 connections but employs *host first* packet processing to minimize delays to host packets. The mean time for a host packet to cross the NIC drops to less than 13 microseconds even though the NIC is still busy with TCP processing (only 5% idle time). The fraction of connections handed off to the NIC is now 48%, close to a half as expected. The host processor shows no idle time, and server throughput continues to improve.

In *W-4096-P-N-FCFS-400*, the NIC can handle a maximum of 4096 connections at a time. 100% of connections are handed off to the NIC since there are only 2048 concurrent connections in the system. The NIC is fully saturated and again becomes a bottleneck in the system. Processing each packet takes much longer, and there are also dropped packets.

Figure 4.12 : Dynamic adjustment of the number of connections on the NIC by the load control mechanism for configuration *W-4096-P-L-FCFS-400*.

As a result, the host processor shows 64% idle time, and the request rate drops by 52% from 26663/s to 12917/s. Thus, giving priority to host packets cannot prevent the NIC from becoming the bottleneck in the system. Note that *host first* packet processing still does its job, and host packets take only several microseconds to cross the NIC.

In *W-4096-P-L-FCFS-400*, the NIC can handle a maximum of 4096 connections at a time, just like *W-4096-P-L-FCFS-400*, but uses the load control mechanism discussed in Section 4.3.2. Figure 4.12 shows how the NIC dynamically adjusts the number of connections during the experiment. Initially 2048 connections are handed off to the NIC, but received packets start piling up on the receive packet queue. As time progresses, the NIC reduces connections in order to keep the length of the receive packet queue under the threshold 1024. The number of connections on the NIC stabilizes around 1000 connections. The resulting server throughput is very close to that of *W-1024-P-N-FCFS-400* in which the NIC is manually set to handle up to 1024 concurrent connections. Thus, the load control mechanism is able to adjust the number of connections on the NIC in order to avoid overload conditions. The NIC now has 9% idle time, slightly greater than 5% shown in *W-1024-P-N-FCFS-400*, which indicates that the watermark values used in the load control mechanism are not optimal. Overall, handoff improves server throughput by

Figure 4.13 : Profiles of the simulated web server during the execution of SPECweb99 Static and regular SPECweb99.

12% in packet rate, 12% in request rate, and 10% in HTTP content throughput (compare *W-0-N-N-FCFS-400* and *W-4096-P-L-FCFS-400*). The server profiles during the execution of the IBM trace also show that both *host first* packet processing and the load control on the NIC are necessary, and that by using both techniques, handoff improves server throughput for the IBM trace by 19% in packet rate, 23% in request rate, and 18% in HTTP content throughput (compare *I-0-N-N-FCFS-400* and *I-4096-P-L-FCFS-400*).

Unlike the trace replayer, SPECweb99 tries to maintain a fixed throughput for each client. Figure 4.13 shows server performance for SPECweb99 Static and SPECweb99. The static version is same as SPECweb99 except that the client generates only static content requests, so it is used to compare against the results produced by the trace replayer. *S-0-N-N-FCFS-400* shows the baseline performance for SPECweb99 Static. Since each client of SPECweb99 is throttled to a maximum of 400Kb/s, 4096 connections (twice the number used for the trace replayer) are used to saturate the server. Like *W-0-N-N-FCFS-400*, the host processor has no idle cycles and spends more than 70% of cycles in the kernel. Handing off 1024 connections improves the request rate by 14%. However, when 2048 connections are handed off, the request rate drops by 24%. As in *W-1024-N-N-FCFS-400*, host packets are delivered to the operating system too slowly, and the host processor shows 50% idle time. The use of *host first* packet processing on the NIC overcomes this problem,

Figure 4.14 : Profiles of the simulated web server when the operating system hands off connections in a first-come, first-served order or by using thresholds.

and server throughput continues to increase. Increasing the number of connections further will simply overload the NIC as there is only 8% idle time. *S-4096-P-L-FCFS-400* uses both *host first* packet processing and the load control mechanism on the NIC. Although the NIC can store all 4096 connections, the load control mechanism reduces the number of connections to around 2000 in order to avoid overload conditions. Overall, by using *host first* packet processing and the load control mechanism on the NIC, handoff improves the request rate for SPECweb99 Static by 31%. These techniques help improve server performance for regular SPECweb99 as well. Handoff improves the request rate by 28%.

4.6.2 Connection Selection Policy

Figure 4.14 compares FCFS and threshold-based connection selection policies. For threshold-based policies denoted by T_n , the trailing number indicates the minimum number of enqueue operations to the send socket buffer of a connection that must occur before the operating system attempts to hand off the connection. The number of enqueue operations is proportional to the number of sent packets. For instance, using T_4 , the operating system attempts a handoff when the fourth enqueue operation to the connection's send socket buffer occurs. Comparison between *W-4096-P-L-FCFS-400*, *W-4096-P-L-T10-400*, and *W-4096-P-L-T20-400* shows that fewer connections are handed off to the NIC as the threshold value

Figure 4.15 : Profiles of the simulated web server when the NIC has a small number of connections and threshold-based connection selection is used.

increases. When the FCFS policy is used, 43% of connections are handed off to the NIC. When threshold 20 is used, only 13% of connections are handed off to the NIC. At the same time, the NIC processes an increasing number of packets per connection on the NIC, which explains why the fraction of packets processed by the NIC drops more slowly than the fraction of connections handed off to the NIC. With the FCFS policy, the NIC processes 109 packets per connection on the NIC. When threshold 20 is used, it processes 222 packets per connection on the NIC. The same observation holds for the IBM trace and SPECweb99. For the IBM trace, the NIC processes 117 packets per connection on the NIC when FCFS policy is used. The number increases to 170 when threshold 20 is used. For SPECweb99 Static, the number increases from 66 to 150. For regular SPECweb99, it increases from 78 to 173.

Even though the operating system hands off fewer but longer connections by using thresholds, the use of thresholds has either negligible effect on server throughput or degrades it because fewer packets are processed by the NIC. For example, when FCFS is used for the World Cup trace (*W-4096-P-L-FCFS-400*), 44% of packets are processed by the NIC. In contrast, when threshold 20 is used, the NIC processes only 27% of packets (see *W-4096-P-L-T20-400*).

Handing off long-lived connections has a greater potential to improve server performance when the NIC has fewer connections. Offload processors may use small on-chip

Figure 4.16 : Profiles of simulated web server when the instruction rate of the NIC is increased.

memory to store connections for fast access. These processors would be able to support only a small number of connections but process them very quickly. *W-256-P-L-FCFS-400* in Figure 4.15 shows a case in which the NIC can handle up to 256 connections at a time. The operating system hands off connections on a FCFS basis. 13% of connections are handed off to the NIC as expected, and 12% of packets are processed by the NIC. The NIC shows 47% idle time and is evidently under-used. In *W-256-P-L-T20-400*, a threshold policy hands off longer connections. The NIC now processes 24% of packets, and the request rate improves by 6%. However, the NIC still has 34% idle time. This suggests that the operating system should also try to hand off short-lived connections when the NIC has ample idle time. Such a policy would be able to fully utilize the NIC and improve server performance further. The same observation holds for the other workloads shown in Figure 4.15.

4.6.3 NIC Speed

The results so far have shown that the NIC must employ *host first* packet processing and dynamically control the number of connections. As the instruction rate of the NIC increases, the NIC processes packets more quickly. The load control mechanism on the NIC should be able to increase the number of connections handed off to the NIC. Figure 4.16 shows

the impact of increasing the instruction rate of the NIC. *W-4096-P-L-FCFS-400* in the figure is same as the one in Figure 4.11 and is used as the baseline case. As the instruction rate increases from 400 to 600 and 800MIPS, the fraction of connections handed off to the NIC increases from 45% to 70% and 85%. Accordingly, the request rate of the server increases from 25830/s to 29398/s and 36532/s (14% and 41% increases). For the IBM trace, increasing the instruction rate from 400 to 600MIPS results in a 21% increase in request rate. At 600MIPS, nearly all connections (95%) are handed off to the NIC. So, the faster 800MIPS NIC improves the request rate by only 3%.

Faster NICs improve server throughput for SPECweb99 Static as well. As the instruction rate increases from 400 to 600MIPS, the request rate improves by 16%. The 800MIPS NIC further improves the request rate by 13%. Faster NICs do not benefit SPECweb99 because the 400MIPS NIC already achieves more than the specified throughput. With 2048 connections, SPECweb99 aims to achieve a maximum HTTP throughput of about 819Mb/s = $2048 \times 400\text{Kb/s}$. In reality, throughput can become greater than the specified rate as it is difficult to maintain throughput strictly under the specified rate. With the 400MIPS NIC, HTTP content throughput is near 1Gb/s. So, faster NICs simply have greater idle time.

These results show that the system can transparently exploit increased processing power on the NIC by using the load control mechanism and *host first* packet processing on the NIC. Thus, hardware developers can improve NIC capabilities without worrying about software changes as the firmware will adapt the number of connections and be able to use the increased processing power.

4.7 Summary

Offloading TCP processing to the NIC can improve system throughput by reducing computation and memory bandwidth requirements on the host processor. However, full offload complicates software architecture and runs the risk that the NIC becomes a bottleneck in the system since it has limited resources. So, the system should support offloading in a way that avoids creating a bottleneck at the NIC. Offload based on connection handoff en-

ables the operating system to control the division of work between the host processor and the NIC by controlling the number of connections handled by each. Thus, using connection handoff, the system should be able to treat the NIC as an acceleration coprocessor by handing off as many connections as the resources on the NIC will allow.

Supporting connection handoff requires the addition of the bypass layer and modifications to the device driver and the firmware of the offload NIC. In order to fully utilize the NIC without creating a bottleneck in the system, both the NIC and the operating system need to implement the policies described in Section 4.3. First, the NIC gives priority to those packets that belong to the connections processed by the host processor. This ensures that packets are delivered to the operating system in timely manner and that TCP processing on the NIC does not degrade the performance of host connections. Second, the NIC dynamically controls the number of connections to avoid overloading itself and creating a bottleneck in the system. Third, the operating system can differentiate connections and hand off only long-lived connections to the NIC in order to better utilize the offload NIC which lacks memory capacity for a large number of connections. Full-system simulations of web workloads show that without any of the policies, handoff reduces the server request rate by up to 44%. In contrast, connection handoff augmented with these policies successfully improves server request rate by 12–31%. When a faster offload processor is used, the system transparently exploits the increased processing capacity of the NIC, and connection handoff achieves request rates that are 33–72% higher than a system without handoff.

Chapter 5

Impact of Processor Speed, Cache Size, and Offload

Offloading can substantially improve server throughput by reducing instruction and memory bandwidth on the host processor, as shown in Chapter 4. However, Chapter 4 uses only one host configuration (2GHz processor with 1MB L2 cache) to evaluate the performance benefits of offload. While this configuration is a representative of modern systems, the performance of the host processor is certainly going to increase in the future. Increasing the clock rate of the host processor and the L2 cache size (and/or L3 cache if available) are two common techniques to improve performance of the host processor. A faster clock rate of the host processor increases the processor's instruction rate by reducing the amount of time it takes to execute an instruction. A larger L2 cache indirectly increases the processor's instruction rate by reducing expensive main memory accesses, which in turn reduces processor stalls. This chapter first examines the impact of increasing the host processor speed and the L2 cache size on server performance, and then their impact on offload. It also compares the performance gains from increasing the host processor speed and the L2 cache size against the performance gains from offloading. This comparison helps to determine which of the three techniques (faster clock rate of the host processor, larger L2 cache, and offload) is more effective in improving server performance.

5.1 Experimental Setup

This chapter uses the full-system simulation setup similar to that used in Chapter 4 in order to vary the host processor speed, the L2 cache size, and the NIC configurations. Briefly, a full-system simulator, Simics [53], is extended with the memory subsystem module that models cycle-accurate caches and main memory (DRAM) and the NIC module that models

	Configuration
CPU	Functional, single-issue, x86 processor Instantaneous instruction fetch Varied clock rates for experiments
L1 cache	64KB data cache Line size: 64B, associativity: 2-way Hit latency: 1 cycle
L2 cache	Data cache, size is varied for experiments Line size: 64B, associativity: 16-way Hit latency: 15 cycles Prefetch: next-line on a miss
DRAM	DDR333 SDRAM of size 2GB Access latency: 195 cycles when 2GHz processor is used
NIC	Functional, single-issue processor Varied instruction rate for experiments 10Gb/s wire

Table 5.1 : Simulator configuration.

a functional MIPS processor and a number of hardware components on the NIC. Table 5.1 summarizes the simulator configuration. Much of the configuration is same as that used in Chapter 4. The main difference is that the host processor's clock rate and the size of the L2 cache are now variable. While the experiments vary the clock rate of the host processor from 2 to 4GHz and the size of the L2 cache from 1 to 16MB, the L2 cache access latency is fixed at 15 cycles. This fixed latency is biased in favor of the host processor performance. As the processor's clock rate and the cache size increase, the cache access latency tends to increase as well, because it is difficult to access large areas of the processor die quickly. For example, the 12MB (per-core) L3 cache on the latest dual-core Intel Itanium processor, whose clock rate is less than 2GHz, has a 15-cycle access latency [59]. In fact, significant engineering efforts were necessary in order to maintain the 15-cycle access latency as the L3 cache size increased from 3MB [52] to 6MB [80], 9MB [22], and then to 12MB [59]. As mentioned in Chapter 4, the host processor is functional and executes one x86 instruction per cycle unless there are memory stalls. For example, a 2GHz processor executes 2 billion instructions per second, assuming no memory stalls. For all experiments, main memory is

DDR333. So, as the clock rate of the host processor increases, main memory access time (in host processor cycles) increases as well. As in Chapter 4, the instruction rate of the NIC is variable and is expressed in millions instructions per second (MIPS).

The host operating system and the NIC firmware are exactly same as those used in Chapter 4. The NIC implements the load control mechanism and *host first* packet processing described in Section 4.3, and the host operating system hands off connections in a first-come-first-served order. The threshold values for the load control mechanism are set conservatively so that the NIC always has idle time in all experiments. This chapter also uses the same set of web workloads and web server software used in Chapter 4. The server runs the Flash web server software [69]. The World Cup and IBM traces are replayed using a trace replayer program [10], and two versions of SPECweb99 are used. The first version uses only static content responses, whereas the second version uses the default mix of static content (70%) and dynamic content (30%) responses. The default maximum bandwidth for each SPECweb99 client is 400Kb/s (Chapter 4 uses this default bandwidth). In this chapter, the maximum bandwidth is raised to 1Mb/s. As the host processor's clock rate and the L2 cache size increase, SPECweb99 needs an increasing number of clients in order to saturate the server. The maximum bandwidth for each client is raised in order to avoid having to simulate an excessively large number of clients. For all experiments, there are 4096 simultaneous clients (connections). The simulator is warmed up during the first one million packets, and all statistics are computed during the next one million packets. Note that these numbers are increased from those used in Chapter 4 since faster systems are simulated in this chapter.

The web workloads are simulated for each combination of the following host processor clock rates, L2 cache sizes, and instruction rates of the NIC. Host processor clock rates: 2, 2.5, 3, 3.5, and 4GHz. L2 cache sizes: 1, 2, 4, 8, and 16MB. Instruction rates of the NIC: 400, 600, 800, and 1000MIPS. The exception is the regular SPECweb99 that uses both static and dynamic content requests. For the regular SPECweb99, 2 and 2.5GHz host processors are not simulated. When either 2 or 2.5GHz host processor is used, the

simulated server stops making forward progress during the setup phase of SPECweb99.

5.2 Host Processor Speed

Figure 5.1 and Figure 5.2 show throughput of the simulated web server for the World Cup and IBM traces, respectively. Offloading is not used in this experiment. In these figures, the x-axis shows the host processor speed, and the y-axis shows server throughput expressed in 10 thousand requests per second. There are multiple lines in each figure, representing different L2 cache sizes. As the figures show, server throughput increases almost linearly as the host processor speed increases. This behavior is expected since the host processor is always saturated in this experiment. For instance, the 2GHz host processor with 1MB L2 cache achieves 24119 requests per second for the World Cup traces. The 3GHz host processor (50% increase in clock rate) with 1MB L2 cache improves throughput to 30608 requests per second, or a 27% increase. The 4GHz host processor (100% increase in clock rate) with 1MB L2 cache results in 36136 requests per second, a 50% improvement over the 2GHz processor. Ideally, the throughput improvement would be close 100% since the host processor speed is doubled. However, main memory accesses due to L2 cache misses prevent the host processor from achieving the ideal speedup. Since main memory remains same (DDR333), increasing the processor speed also increases the number of processor cycles spent waiting for main memory access to complete. For the IBM trace, the server with the 2GHz host processor and 1MB of L2 cache achieves 28233 requests per second. The 4GHz host processor with 1MB of L2 cache results in 43351 requests per second, a 54% improvement. Again, main memory accesses prevent the system from achieving the ideal 100% speedup.

With larger L2 caches, the server achieves greater throughput improvements as the host processor speed increases. In Figure 5.1 and Figure 5.2, the lines associated with 2, 4, 8, and 16MB L2 caches show steeper slopes than the 1MB L2 cache. For example, the server throughput for the World Cup trace is 28577 requests per second with the 2GHz host processor and 16MB L2 cache. The 4GHz host processor with 16MB L2 cache im-

Figure 5.1 : Web server throughput for the World Cup trace as a function of the host processor speed.

Figure 5.2 : Web server throughput for the IBM trace as a function of the host processor speed.

proves throughput to 52621 requests per second, a 84% increase. As mentioned above, the improvement is only 50% with 1MB L2 cache. Similarly, the server throughput for the IBM trace increases from 46426 requests per second when the 2GHz host processor with 16MB L2 cache is used to 87520 requests per second (89% increase) when the 4GHz host processor with 16MB L2 cache is used. This behavior is expected since larger L2 caches reduce main memory accesses, which in turn reduces processor stalls due to main memory access.

Figure 5.3 shows server throughput for SPECweb99 that uses only static content. Figure 5.4 shows server throughput for regular SPECweb99 that uses both static and dynamic content. As the clock rate of the host processor increases, server throughput increases almost linearly, just like it does for the World Cup and IBM traces. For SPECweb99 that uses only static content, server throughput increases from 14232 requests per second when the 2GHz processor with 1MB L2 cache is used to 17346 requests per second (22% increase) when the 3GHz processor with 1MB L2 cache is used. Using the 4GHz processor with 1MB L2 cache, server throughput further increases to 21043 requests per second, a 48% improvement over the throughput achieved using the 2GHz processor. These throughput improvements are similar to those for the World Cup and IBM traces, as discussed above.

Figure 5.3 : Web server throughput for SPECweb99 with only static content as a function of the host processor speed.

Figure 5.4 : Web server throughput for SPECweb99 with both static and dynamic content as a function of the host processor speed.

Impacts of larger L2 caches on SPECweb99 throughput are also similar to those for the World Cup and IBM traces. Using the 16MB L2 cache, server throughput for SPECweb99 that uses only static content increases from 18600 requests per second when the 2GHz processor is used to 35339 requests per second (90% increase) when the 4GHz processor is used. This speedup (90%) is close to the ideal 100% improvement in throughput expected from doubling the processor's clock rate. As shown in Figure 5.4, throughput of regular SPECweb99 increases much more slowly than SPECweb99 that uses only static content because even 16MB of L2 cache is not able to reduce L2 cache misses significantly. Overall, the results shown in this section confirm the conventional wisdom that increasing the processor's clock rate often has limited impact on system performance because a large portion of the processor cycles are simply wasted by main memory access time. It is necessary to increase the cache size as well, in order to better utilize the increased clock rate of the processor.

Figure 5.5 : Web server throughput for the World Cup trace as a function of the L2 cache size.

Figure 5.6 : Web server throughput for the IBM trace as a function of the L2 cache size.

5.3 L2 Cache Size

Figure 5.5 and Figure 5.6 show throughput of the simulated web server for the World Cup and IBM traces, respectively. The data points shown in these figures are identical to those shown in Figure 5.1 and Figure 5.2. However, the x-axis now shows the L2 cache size so that it is easier to see the impact of the L2 cache sizes on server throughput. Each line in the figures represents a distinct processor speed. For the World Cup trace, increasing the L2 cache size from 1MB to 4MB noticeably improves server throughput, and then at 4MB, server throughput starts leveling. When the 2GHz processor is used, server throughput improves from 24119 requests per second with 1MB L2 cache to 27545 requests per second (14% increase) with 4MB L2 cache. The use of the 16MB L2 cache increases throughput to 28577 requests per second, by only about 1000. Evidently, the 8MB L2 cache is large enough to capture the working set of the web server for the World Cup trace. With the 8MB L2 cache, there is only about 1 L2 cache miss per packet. This number is close to the minimum possible since processing a received packet always incurs compulsory cache misses due to DMA cache invalidations. As mentioned in Chapter 3, the NIC stores received packets in main memory through DMA transfers, during which the cache lines corresponding to the packets are invalidated. For the IBM trace, server throughput continues to increase until

the L2 cache size reaches 8MB, and then it starts leveling. Using the 2GHz processor with 1MB L2 cache, server achieves 28233 requests per second. Throughput increases to 45817 requests per second (62% increase) when the same processor with 8MB L2 cache is used. The use of 16MB L2 cache only improves throughput by about 600 requests per second, indicating that 8MB L2 cache is large enough to eliminate most main memory accesses for the IBM trace. Incidentally, server throughput achieved using the 2GHz processor with 8MB L2 cache (45817 requests per second) is greater than throughput achieved using the 4GHz processor with 1MB L2 cache (43351 requests per second). This result again shows the importance of reducing main memory accesses.

Figure 5.5 and Figure 5.6 also show that faster processors see greater performance gains from large caches than slower processors. For instance, for the World Cup trace, the 2GHz processor with 16MB L2 cache achieves 18% greater throughput than the 2GHz processor with 1MB L2 cache (24119 vs. 28577 requests per second). In contrast, the 4GHz processor with 16MB L2 cache achieves 46% greater throughput than the 4GHz processor with 1MB L2 cache (36136 vs. 52621 requests per second). The same observation holds for the IBM trace. The 2GHz processor with 16MB L2 cache achieves 64% greater throughput than the 2GHz processor with 1MB L2 cache (28233 vs. 46426 requests per second). The 4GHz processor with 16MB L2 cache achieves 102% greater throughput than the 4GHz processor with 1MB L2 cache (43351 vs. 87520 requests per second). As discussed above, since main memory speed remained fixed (DDR333), main memory accesses have greater impact on faster processors than slower processors.

Figure 5.7 and Figure 5.8 show server throughputs for the two versions of SPECweb99 used in this chapter. Increasing the L2 cache size improves server throughput for both versions of SPECweb99. For SPECweb99 that uses only static content, the 2GHz processor with 1MB L2 cache achieves 12633 requests per second. As the L2 cache size is increased to 8MB, server throughput also increases to 16549 requests per second (31% increase). Further increasing the L2 cache size to 16MB results in a little throughput improvement by about 200 requests per second. Server throughput starts leveling at 8MB for faster

Figure 5.7 : Web server throughput for SPECweb99 with only static content as a function of the L2 cache size.

Figure 5.8 : Web server throughput for SPECweb99 with both static and dynamic content as a function of the L2 cache size.

processors as well. Using 16MB L2 cache, the server incurs only 2 L2 misses per packet. SPECweb99 that uses both static and dynamic content achieves lower throughput than the other web workloads because the host processor spends a significant number of cycles generating dynamic content. Server throughput increases noticeably until the L2 cache size reaches 8MB, and then it starts leveling. Unlike the other workloads, the server still incurs 4.2 L2 misses per packet even when 16MB L2 cache is used. This indicates that regular SPECweb99 requires a much larger cache in order to eliminate main memory accesses (at least for the particular SPECweb99 configuration used in this chapter).

Just as Figure 5.5 and Figure 5.6 show, Figure 5.7 and Figure 5.8 also show that faster processors see greater performance gains from increased cache sizes. For instance, for SPECweb99 that uses only static content, the 2GHz processor with 16MB L2 cache achieves 32% greater throughput than the same processor with 1MB L2 cache. The 4GHz processor with 16MB L2 cache achieves 92% greater throughput than the same processor with 1MB L2 cache. As mentioned above, faster processors waste a greater number of cycles due to main memory access than slower processors. So, eliminating main memory accesses by increasing the cache size has greater impact on the performance of faster processors.

Figure 5.9 : Web server throughput for the World Cup trace achieved with and without offloading while the host processor speed and the L2 cache size vary.

The results shown so far in this chapter show that increasing the clock rate of the host processor and/or the L2 cache size improve server performance, as expected. However, increasing either one alone is insufficient. Increasing the clock rate alone can simply lead to a greater number of cycles wasted due to main memory accesses. Likewise, increasing the L2 cache size alone can reduce memory stalls but may not increase the instruction rate of the processor. So, the host processor should increase both the clock rate and the cache size in order to achieve substantial performance improvements.

5.4 Offloading

The previous two sections examined the impact of increasing the host processor's clock rate and the L2 cache size on web server throughput. This section examines how faster host processor and larger L2 cache sizes affect performance improvements from offloading. Figure 5.9 shows the impact of various system configurations on performance of the simulated web server for the World Cup trace. Each bar shows server throughput in HTTP requests per second. The bars are grouped according to the host processor configuration. Each group represents a unique combination of the host processor speed and the L2 cache size. The x-axis shows the host processor speeds (2, 2.5, 3, 3.5, and 4GHz) and the pro-

cessor's L2 cache sizes (1, 2, 4, 8, 16MB). For example, the group indicated by 2GHz and 1MB shows web server throughput achieved using the 2GHz processor with 1MB L2 cache. Each group has two bars. The left bar shows web server throughput achieved without offload, and the right bar shows throughput achieved with offload. Throughput numbers for the left bars (non-offload throughput) are identical to those that appear in the previous two section (see Figure 5.1 and Figure 5.5). The right bar is further broken into four parts according to the speed of the offload processor on the NIC. The bottom of the bar indicated by 400MIPS shows server throughput achieved with the 400MIPS offload processor. The region indicated by 600MIPS shows additional server throughput gained by using the faster 600MIPS offload processor. Likewise, the regions indicated by 800MIPS and 1000MIPS show additional server throughput achieved by using the 800MIPS and 1000MIPS offload processors, respectively. The rest of the figures that appear in this section all have the same format as Figure 5.9.

As shown in Figure 5.9, offloading improves server throughput for all host processor configurations. The 400MIPS offload processor improves throughput by 5–26% over the baseline (non-offload) throughput. The least throughput improvement (5%) occurs when the 2.5GHz host processor with the 2MB L2 cache is used. However, note that offload never degrades throughput, showing that the load control mechanism on the NIC avoids performance degradations, as expected. The 600MIPS offload processor improves server throughput by 14–36% over the baseline. The faster 800MIPS and 1000MIPS offload processor improve throughput by 20–49% and 28–64% over the baseline, respectively.

When the instruction rate of the offload processor (MIPS) is fixed, the host processor becomes faster with respect to the offload processor as the host processor speed and/or L2 cache size increase. So, the fraction of connections (or packets) that are processed by the NIC decreases as the host processor becomes faster. For example, suppose the system uses the 800MIPS offload processor. 60% of connections (56% of packets) are processed by the NIC when the 2GHz host processor with 1MB L2 cache is used. As the L2 cache size is increased to 16MB, the NIC processes 49% of connections (46% of packets). Likewise, as

the host processor speed is increased to 4GHz while the L2 cache size is fixed at 1MB, 44% of connections (45% of packets) are processed by the NIC. When the 4GHz host processor with 16MB L2 cache is used, the fraction of connections handled by the NIC drops to 24%. Consequently, throughput improvements from offload over the baseline (in percent) also decrease. Offload improves throughput over the baseline by up to 49% when the 2GHz host processor is used. As the host processor speed increases to 3GHz, offload improves throughput over the baseline by up to 35%. As the host processor speed further increases to 4GHz, offload now improves throughput over the baseline by up to 32%. Note that these improvements are relative to the baseline and that the decreases shown above do not imply decreases in the absolute amount of throughput gained from offload.

When both the instruction rate of the offload processor (MIPS) and the size of the L2 cache (MB) are fixed, the amount of throughput increases from offload do vary as the host processor speed increases. However, there is no clear correlation between the host processor speed and the amount of throughput increases from offload. For example, suppose the L2 cache size is fixed at 1MB. The 600MIPS offload processor improves throughput by 6336 requests per second (with 2GHz host processor), 5058 (with 3GHz host processor), and 5158 (with 4GHz host processor). Likewise, the 800MIPS offload processor improves throughput by 8090 requests per second (with 2GHz host processor), 8115 (with 3GHz host processor), and 8338 (with 4GHz host processor).

Now, when the host processor speed (GHz) and the instruction rate of the NIC (MIPS) are fixed, the amount of additional throughput gained from offload tends to increase as the L2 cache size increases. This occurs when the NIC has sufficient compute power and has idle time, but the host processor with a small cache is unable to hand off more work (connections) to the NIC. With a larger cache, the host processor handles user requests more quickly and hands off more connections to the NIC, which then increases the throughput gained from offload. For example, when the 2GHz host processor and the 800MIPS offload processor are used, offload improves throughput over the baseline by 8090 requests per second (with 1MB L2 cache) and by 13883 (with 16MB L2 cache). Likewise, when the 4GHz

host processor and the 800MIPS offload processor are used, offload improves throughput over the baseline by 8338 requests per second (with 1MB L2 cache) and by 16912 (with 16MB L2 cache). When the 2GHz host processor and 1MB L2 cache are used, the NIC has 29% idle time. So, the NIC can evidently handle more connections, but the host processor is not processing requests quickly enough. With the larger 16MB L2 cache, the host processor processes requests more quickly and is able to hand off more connections to the NIC. The NIC now has 18% idle time and handles more connections, which explains the increase in the amount of throughput gained from offload. Likewise, when the 4GHz host processor with 1MB L2 cache is used, the NIC has 14% idle time. When 16MB L2 cache is used, the NIC's idle time drops to 5% because the host processor now hands off more connections. As mentioned in Section 5.1, the threshold values for the load control mechanism on the NIC are set conservatively so that the NIC has idle time for all configurations. Suppose the threshold values are set such that the NIC has as little idle time as possible. There are two possible outcomes. If the NIC has zero idle time, then increasing the L2 cache size will have little effect on the amount of throughput gained from offload. If the NIC has idle time with small caches, then the NIC is evidently able to handle nearly all connections in the system, and the host processor is simply not fast enough to hand off more connections to the NIC. In this case, larger caches will increase the amount of throughput gained from offload since the host processor will be able to process requests more quickly and hand off more connections to the NIC. This effect appears more prominently for the other web workloads, as will be discussed later.

Figure 5.10 shows server throughput achieved using with and without offloading for the IBM trace. The server throughputs achieved without offload (left bars) are same as those that appear in Figure 5.2 and Figure 5.6. The bars in Figure 5.10 show trends similar to the three identified above for the World Cup trace. First, increasing the instruction rate of the offload processor increases throughput improvements from offload. Using the 400MIPS offload processor, the system achieves 2–24% greater throughput than the baseline (non-offload) throughput. The 600, 800, and 1000MIPS offload processors im-

Figure 5.10 : Web server throughput for the IBM trace achieved with and without offloading while the host processor speed and the L2 cache size vary.

prove server throughput by 9–41%, 18–47%, and 23–51% over the baseline, respectively. Second, as the host processor becomes faster with respect to the offload processor (by increasing the speed of the host processor and/or the size of the L2 cache), the fraction of connections handed off to the NIC decreases. This in turn reduces throughput improvements from offload relative to the baseline throughput. For example, the 800MIPS offload processor improves throughput over the baseline by up to 47% (with the 2GHz host processor), by up to 28% (with the 3GHz host processor), and by up to 20% (with the 4GHz host processor). Third, as the L2 cache size increases, the amount of throughput gained from offload also increases. This trend is more clearly shown in Figure 5.10 than in Figure 5.9. For example, when the 2GHz host processor with 1MB L2 cache is used, the 1000MIPS offload processor improves throughput by 13079 requests per second over the baseline. As the L2 cache size increases to 16MB, the same offload processor now improves throughput by 23600 requests per second, 80% more than it does when 1MB L2 cache is used. As discussed previously, with a small cache, the host processor is not processing requests fast enough and thus not handing off connections to the NIC even though the NIC has idle time. The 1000MIPS offload processor has 45% idle time and processes 99% of connections when the 2GHz processor with 1MB L2 cache is used. As the L2 cache size increases,

Figure 5.11 : Web server throughput for SPECweb99 achieved with and without offloading while the host processor speed and the L2 cache size vary. In this experiment, SPECweb99 uses only static content.

the host processor is able to hand off connections more quickly to the NIC, which increases the amount of throughput gained from offload.

Figure 5.11 shows server throughput achieved using with and without offloading for SPECweb99 that uses only static content. The server throughputs achieved without offload are same as those shown in Figure 5.3 and Figure 5.7. The experimental results for SPECweb99 also show the three trends discussed above. First, increasing the instruction rate of the offload processor increases throughput improvements from offload. Using the 400, 600, 800, and 1000MIPS offload processors results in 3–22%, 10–47%, 23–59%, and 29–68% throughput improvements over the baseline throughput, respectively. Second, relative throughput improvements from offload decrease as the host processor becomes faster with respect to the offload processor. For instance, the 800MIPS offload processor improves throughput over the baseline by up to 59% when the 2GHz host processor is used, by up to 39% when the 3GHz host processor is used, and by up to 38% when the 4GHz host processor is used. As previously discussed, a faster host processor increases the baseline throughput, while the amount of throughput gained from offload remains roughly same. So, the improvement relative to the baseline decreases as a result. Third, as the L2 cache size increases, the amount of throughput gained from offload also increases. This behavior

Figure 5.12 : Web server throughput for SPECweb99 achieved with and without offloading while the host processor speed and the L2 cache size vary. In this experiment, SPECweb99 uses both static and dynamic content.

closely resembles that shown in Figure 5.10. For example, when the 2GHz host processor with 1MB L2 cache is used, the 800MIPS offload processor increases throughput by 4702 requests per second over the baseline. As the L2 cache size is increased to 16MB, the same 800MIPS offload processor improves throughput by 8167 requests per second over the baseline. With small caches, the host processor is not processing requests quickly enough to saturate the offload processor. With larger caches, the host processor processes requests more quickly and hands off more connections to the offload processor.

Figure 5.12 shows server throughput achieved using with and without offloading for the regular SPECweb99 that uses both static and dynamic content. The server throughputs achieved without offload are same as those that appear in the previous two sections (see Figure 5.4 and Figure 5.8). Like the other three workloads discussed so far, the regular SPECweb99 also shows the three trends identified previously. A notable difference is that the impact of the L2 cache size is more prominent in this workload than in the other workloads. First, throughput improvements from offload increase as the instruction rate of the offload processor increases. The 400MIPS offload processor improves throughput over the baseline throughput (throughput achieved without offload) by 3–32%. The 600MIPS offload processor improves throughput over the baseline throughput by 10–37%. The 800

and 1000MIPS offload processors improve throughput over the baseline throughput by 10–43% and 26–49%, respectively. Second, as the host processor becomes faster with respect to the offload processor, throughput improvements from offload, relative to the baseline throughput, decreases. For instance, the 800MIPS offload processor improves throughput over the baseline by up to 43% when the 3GHz host processor is used. As the host processor speed increases to 4GHz, the 800MIPS offload processor now improves throughput over the baseline throughput by up to 39%. Third, as the L2 cache size increases, the amount of throughput gained from offload also increases. For example, when the 3GHz host processor with 1MB L2 cache is used, the 800MIPS offload processor improves throughput over the baseline throughput by 11885 requests per second. However, when the L2 cache size is increased to 16MB, the same offload processor improves throughput over the baseline by 19320 requests per second (64% greater than 11885 requests per second). As previously discussed, with a small L2 cache, the host processor may not be able to hand off additional connections to the NIC even though the offload processor has idle time, as is the case for the regular SPECweb99 shown in Figure 5.12. With larger L2 caches, the host processor can process requests (connections) more quickly and hand off a larger number of connections to the NIC.

5.5 Discussion

The previous sections examined the impact of offload and increasing the host processor's compute power (through increased clock rate and larger L2 cache) on web server throughput. It is useful to compare cost-effectiveness of these two approaches even though such comparisons are necessarily ambiguous to some extent. As the results in Section 5.2 and Section 5.3 show, the system needs to increase both the host processor's clock rate and the cache size in order to achieve significant performance improvements. In fact, server processors such as AMD Opteron, Intel Itanium, and IBM Power have been following that approach. However, it becomes increasingly more difficult to do so as the clock rate and the cache size increase. First, increasing the clock rate of the host processor adds complexity

and leads to diminishing performance gains [2]. Second, it is difficult to increase the size of on-chip caches while maintaining low latency. 1MB L2 caches with access latency around 15 cycles are now common. Much larger on-chip caches are possible, as evidenced by Intel Itanium processors. Itanium's L3 cache size has grown from 3MB [52] to 6MB [80] to 9MB [22] and to 12MB [59] while the clock rate of the processor increased from 1GHz to 1.6GHz. Intel did manage to keep the L3 access latency at 15 cycles through significant engineering efforts [22, 59, 80]. However, such large caches are rare. Moreover, it is unclear whether (1) larger caches are even realistic since caches already account for well over half of the chip area and (2) their access latency can be kept constant with faster clock rates. Larger caches would likely have to go off-chip, which then significantly increases access latency, as in the IBM Power5 processor [43].

Offloading provides the system with an alternative way to improve performance. For instance, Figure 5.9 shows that the fastest system (4GHz host processor with 16MB L2 cache) achieves 52621 requests per second. A similar throughput, 52853 requests per second, can be achieved using the 800MIPS offload processor and the 3GHz host processor with 4MB L2 cache. Dedicating a specialized processor to specific tasks can generally improve overall system efficiency, thus cost-effectiveness, because it allows the processor design to focus on a specific set of tasks and thus perform those tasks more efficiently than the general-purpose (host) processor. For the offload processor, it can exploit fast local memory in order to quickly access connection states whereas the host processor may suffer from main memory accesses. Various hardware acceleration features such as hardware-based connection lookup [36] also help the offload processor to process packets more efficiently than the host processor. Moreover, there is evidence that specialized offload processors can achieve very high packet rates. A research prototype processor from Intel can process TCP packets at 10Gb/s [36], and it is reported that the offloading NIC from Chelsio can achieve over 7Gb/s of TCP throughput [30]. Commercial offloading NICs have up to 256MB of memory, which can theoretically store hundreds of thousands of connections and packets. Commercial offloading NICs are much more expensive than

regular Ethernet NICs, but their cost should decrease over time as manufacture volume increases. So, offloading has the potential to become a cost-effective way of improving networking performance in the future.

Finally, manufacturers recently began adding multiple cores on a chip, instead of solely focusing on increasing the clock rate and the cache size. Conceptually, this approach adds little complexity to the existing processor architecture, so some view that additional cores come *for free*. There are reports that dedicating a core to network processing [17] or parallelizing the network stack [88] can improve system performance by utilizing multiple cores. However, the use of multiple cores does not alleviate main memory bottleneck. In contrast, they exacerbates it since the additional cores increase memory bandwidth requirement, and scarce bandwidth leads to increased main memory access time [19].

5.6 Summary

This chapter evaluates how increasing the host processor clock rate and the L2 cache size, while main memory speed is fixed, affects performance of web server and offloading. First, increasing the clock rate of the host processor and the L2 cache size both improve server throughput, as expected. However, increasing the clock rate alone is not very effective. For instance, doubling the clock rate of the host processor from 2GHz to 4GHz when the L2 cache size is fixed at 1MB only results in a 25–54% increase in server throughput for three of the four web workloads used in this chapter. Because main memory speed is fixed, a faster (in clock rate) host processor simply wastes a large fraction of cycles due to main memory accesses. Increasing the L2 cache size also improves server throughput. For example, when the clock rate of the host processor is fixed at 2GHz, increasing the L2 cache size from 1MB to 16MB improves server throughput by 18–64%. While the 64% improvement is significant, one would hope to gain greater performance improvements from such a large cache. Increasing both the clock rate of the host processor and the size of the L2 cache leads to most significant performance improvements since doing so increases maximum instruction rate and reduces main memory accesses. For instance, the 4GHz

host processor with 16MB L2 cache achieves 118–210% greater throughput than the 2GHz host processor with 1MB L2 cache. However, such a processor would be prohibitively expensive to design and manufacture.

Offloading can also achieve significant performance improvements. For example, when the 2GHz host processor with 1MB L2 cache is used, the use of the 800MIPS offload processor improves server throughput by 34–45%, comparable to the performance improvements from doubling the clock rate of the host processor or increasing the size of the L2 cache from 1 to 16MB. Even when the host processor's compute power is artificially increased (4GHz clock rate with 16MB L2 cache), the use of the 800MIPS offload processor results in a 20–32% throughput increase. Because the offload processor can avoid large memory latencies by utilizing fast local memory, it can process network traffic more efficiently than the host processor. Thus, offloading can improve overall system efficiency while achieving the above performance improvements.

Chapter 6

Placement Options for the Offloading Processor

So far in this dissertation, the offloading processor is assumed to be located on the NIC. Since the NIC occupies the lowest levels of the network protocol stack and is responsible for actual transmission and reception of packets, it is the most intuitive place to insert the offloading processor. In the current system architectures, the NIC is a peripheral, typically a PCI device. However, the offloading processor is not required to be placed on the PCI NIC, nor the NIC is required to be on the PCI bus. This chapter discusses several possible locations for the offloading processor, implementation issues associated with each location, and the applicability of connection handoff. Also, potential performance advantages of various locations are discussed.

Figure 6.1 shows high-level diagrams of the system architectures used in most computers and possible locations to place the offloading processor. On the left of the figure, the system uses the front-side bus (FSB) that connects the host processor(s) and the bridge (commonly known as the north bridge). This bridge typically interfaces with main memory (DRAM) through the memory controller implemented as a part of the bridge. It also interfaces with one or more high-speed PCI buses which connect peripherals such as SCSI disk controllers and Gigabit Ethernet NICs. The second bridge (commonly known as the south bridge) interfaces with slower peripherals such as Fast Ethernet NICs and legacy ISA devices. These two bridges comprise a chipset of the system. The right side of the figure illustrates a more recent system that utilizes HyperTransport [39] and processors like AMD Opteron that has an on-chip integrated memory controller. Main memory is now directly connected to the processor since the processor has the memory controller. The system still requires a chipset in order to interface with PCI devices and other peripherals.

Figure 6.1 : Simplified system architectures. The numbers in squares indicate potential locations to insert the offloading processor.

In both system architectures, there are several places to insert the offloading processor. In Figure 6.1, these are indicated by the numbers in squares. In the following sections, each of the locations is discussed in detail.

6.1 As Part of the Host Processor Die

The offloading processor can be implemented as part of the host processor die. In Figure 6.1, this placement option is indicated by 1 for both system architectures. The primary reason for this placement is the possibility of reducing communication overhead between the host processor and the offloading processor. The device driver communicates with the offloading processor (or NIC in general) through programmed I/O (PIO) and main memory. The device driver typically performs one PIO write to a register on the offloading processor for each message sent by the driver. Also, the offloading processor sends messages to the device driver by storing them in main memory via DMA transfers. Since the offloading processor is now physically close to the host processor, PIO accesses should be able to complete quickly. However, the host processor does not have to stall on PIO writes. Since the device driver rarely performs PIO reads, which would force the host processor to wait for their results, the reduced PIO latency would have little impact on the amount of time spent in the device driver. The offloading processor also has the potential to directly access

the host processor caches to reduce main memory accesses within the device driver. For instance, instead of storing messages in main memory via DMA, the offloading processor can store the messages in the host processor caches, which would then avoid having to access main memory to fetch them.

Connection handoff interface is not affected by the integration of the offloading processor on the host processor die, as long as the offloading processor is treated as a device, and the operating system communicates with the offloading processor through the device driver. The offloading processor may be presented to the operating system as a logically PCI device. Many peripheral devices are now integrated into the chipset. From the viewpoint of the operating system, these devices appear as regular PCI devices so that the operating system can use the same set of device drivers for the devices regardless whether they are integrated or not. Even if the offloading processor appears does require a new access method, only the device driver will need to change.

There are a number of technical complications associated with integrating the offloading processor into the host processor die. First, the offloading processor needs memory to store connection states and packet data. It may be costly to build dedicated fast-access on-chip memory for the offloading processor. The offloading processor may share main memory with the host processor and store connections and packets in main memory. However, this approach essentially defeats the objective of offload in the first place, since the offloading processor will suffer from long-latency main memory accesses and will not be able to process packets quickly. Second, in order to transmit and receive packets, the offloading processor must interface with media access control (MAC) and direct memory access (DMA) hardware. It is certainly possible to integrate these into the host processor die as well, but it may be costly. Also, in case of MAC, it requires extra pins to transfer data to and from the transceiver that actually transmits and receives bits on the network wire. Alternatively, the offloading processor may utilize non-offloading PCI NICs for transmitting and receiving packets. However, the offloading processor now must communicate with the PCI NIC. This approach simply shifts the burden of communication with devices from

the host processor to the offloading processor. Third, in a system that uses FSB (see the left side of Figure 6.1), every packet would have to be transferred between the offloading processor and main memory across FSB. This packet traffic may reduce bandwidth available for the host processor and degrade its performance.

6.2 As Part of the Chipset

The offloading processor can also be implemented as part of the chipset. In Figure 6.1, this placement option is indicated by 2 for both system architectures. In many systems, a number of commonly-used peripherals are integrated into the chipset in order to reduce the number of components in the system and thus overall cost of the system. Integrating the offloading processor into the chipset may be able to reduce cost as well. It can also reduce communication overhead between the offloading processor and the host processor. Since the chipset is connected to the host processor through the front-side bus or HyperTransport, the offloading processor still has the potential to access the host processor caches. The front-side bus typically supports cache coherence in order to resolve coherence due to DMA. HyperTransport also provides cache coherence as an option. So, the offloading processor can potentially store messages in the host processor caches and reduce main memory accesses within the device driver.

Like other devices integrated into the chipset, the offloading processor implemented as a part of the chipset would appear as a PCI device. As discussed in Section 6.1, the operating system can use connection handoff without any changes as long as the offloading processor appears as a regular device, and the operating system communicates with it through a device driver.

Integrating the offloading processor into the chipset presents a set of technical complications similar to those associated with integrating it into the host processor die. As discussed in Section 6.1, the offloading processor requires several non-trivial components in order to function correctly. The offloading processor requires memory for storing connection states and packet data, MAC hardware for transmitting and receiving packets, and

DMA hardware for transferring data from and to main memory. Integrating them all into the chipset may be too costly in terms of the area required to implement them. Current Gigabit Ethernet controllers are not integrated into the chipset, either. They either reside on the system board as an on-board chip or on the PCI NIC. In both cases, the controller appears as a PCI device.

6.3 As Dedicated Processor Node

The offloading processor can also be connected directly to the front-side bus or HyperTransport, much like a general-purpose host processor in a multi-processor system. This placement is indicated by 4 for both system architectures shown in Figure 6.1. It should be possible to build a complete NIC and connect it to FSB or HyperTransport. Such a NIC is essentially same as the regular PCI NIC, except that it is connected to FSB or HyperTransport. The offloading processor can access the host processor caches through cache coherent FSB or HyperTransport if necessary. This placement option is more appropriate for a HyperTransport-based system than a FSB-based system. First, FSB is specific to the processor, whereas HyperTransport is a standardized interconnect. Second, in a system that uses FSB, packets have to be transferred across FSB between the NIC and main memory. This packet traffic may reduce FSB bandwidth available for the host processor. In a system that uses HyperTransport, packets always cross HyperTransport and are transferred go through the main memory controller on the host processor, regardless of the location of the NIC. Like the other placement options discussed so far, the operating system would communicate with the offloading processor through a device driver, and connection handoff interface can be used to control the offloading processor.

6.4 As Part of the Traditional PCI Network Interface

Finally, the offloading processor can be placed on the PCI NIC. This placement is indicated by 3 for both system architectures shown in Figure 6.1. Since the NIC already implements

MAC and DMA, the offloading processor can quickly access them. It can easily utilize on-board memory for storing connection states and packet data. It is also easy to upgrade the offloading processor since doing so only involves replacing the PCI board. Despite its location, the NIC can also access the host processor caches through hardware changes [38], so direct access to the host processor caches is not predicated on the proximity between the offloading processor and the host processor. A potential drawback of placing the offloading processor on the PCI NIC is that because the PCI board has limited area and strict power budget, the offloading processor on the NIC may have less processing capacity than the other placement options discussed so far.

6.5 Discussion

Recently, some argued that placing the NIC close to the host processor can improve performance of network-intensive workloads and that integrating the NIC directly into the host processor die yields significant performance gains [12, 13]. The intuitive argument is that the host processor frequently communicates with the NIC, and that the farther the NIC is from the host processor, the longer it takes for them to communicate with each other. As mentioned in Section 6.1, communication between the NIC (or the offloading processor) and the host processor takes place through PIO and main memory. Moving the NIC (or the offloading processor) closer to the host processor can reduce the PIO latency and can enable the NIC to access the host processor caches in order to reduce main memory accesses. PIO reads can be very expensive since the host processor must wait for them to complete. In some systems, such a PIO read can take over a thousand host processor cycles [33]. However, for many NICs (and the offloading processor), the device driver mostly perform PIO writes, and PIO reads are rare. The host processor does not need to wait for PIO writes to complete. Thus, the reduced PIO latency would have little impact on reducing communication overhead.

In contrast, communicating through the processor cache, instead of main memory, may be able to reduce communication overhead noticeably. Mukherjee *et al.* showed that cache-

coherent network interfaces can significantly reduce inter-processor message latencies by reducing the amount of host processor time spent communicating with the network interface [64]. A more recent study argues that allowing the NIC to store received packets directly into the host processor caches can significantly reduce packet processing time [38]. However, accessing the processor caches require additional hardware support. For instance, the NIC (or the offloading processor) may need to participate in cache coherence. Simply moving the NIC (or the offloading processor) close to the host processor does not automatically allow it to access the processor caches.

6.6 Summary

Offload through connection handoff described in Chapter 4 assumes that the offloading processor is a part of the NIC. Alternatively, the offloading processor may be placed on the host processor die, on the system chipset, on the system board, or on the FSB (or HyperTransport). While each placement option has its own advantages, the system still needs to control the division of work between the host processor and the offloading processor in order to avoid making the offloading processor a bottleneck in the system. Moreover, connection handoff interface is applicable regardless of the particular placement. The offloading processor is most likely to be treated as an acceleration device in the system. So, the operating system would still communicate with the offloading processor through a device driver. Since the device driver is responsible for directly communicating with the offloading processor, it may require substantial changes depending on the location of the offloading processor. Otherwise, the rest of the operating system is oblivious of the location of the offloading processor, and the system can still use connection handoff to achieve offload.

Chapter 7

Implementing Connection Handoff in FreeBSD and Linux

The original prototype of connection handoff was built using the FreeBSD 4.7 operating system, as mentioned in Chapter 4. This prototype was used for all experiments throughout this dissertation. Another prototype was built using the Linux 2.6.9 operating system in order to show that the handoff interface defined in Chapter 4 is applicable to operating systems other than FreeBSD. This chapter first describes similarities between the network stack implementations of the two operating systems and discusses how such similarities enable both operating systems to support connection handoff. Then, it discusses differences between the network stacks of FreeBSD and Linux and how they lead to implementation difficulties.

The implementation specifics discussed in this chapter are based on FreeBSD 4.7 and Linux 2.6.9. The interested reader can find more details in publicly available sources of both operating systems. There are also several references on network stack implementations of both operating systems [15, 27, 35, 86, 89]. The book by Wright and Stevens [89] does not directly apply to FreeBSD but explanations of many implementation details still hold true.

7.1 Similarities in FreeBSD and Linux Network Stacks

Although specifics of the network stack implementations of FreeBSD and Linux are very different, their overall architectures resemble each other. More importantly, the interface between the socket layer and the TCP layer in Linux is similar to that of FreeBSD. This enables the handoff interface to be applicable to both operating systems. This section discusses the network stack implementations of FreeBSD and Linux, their similarities, and

Figure 7.1 : Simplified view of TCP/IP network stack layers of FreeBSD and Linux and main data structures used within the network stack.

the reason why they are similar, to the extent that applies to connection handoff.

Both FreeBSD and Linux network stacks use similar organizations and data structures. Figure 7.1 illustrates a simplified version of the network stack and also shows main data structures. Many details such as IP and Ethernet layers, various minor data structures and synchronizations are omitted intentionally in order to discuss features that are most relevant to supporting offload through connection handoff. First, both FreeBSD and Linux network stacks utilize layers, and the layers communicate with each other through well-defined software interfaces. These interfaces are typically implemented using function pointers, so they can be stored as fields within data structures such as `struct pr_usrreqs`, `struct proto`, `struct ifnet`, and `struct net_device`. The user application performs operations on sockets such as send and receive through the sockets interface. Sockets then forward such user requests to TCP through the transport interface. Protocols communicate with device drivers through the driver interface. Second, both FreeBSD and Linux network stacks make use of similar data structures to represent each layer. For instance, the state of a TCP connection is represented by `struct tcpcb` in FreeBSD and `struct tcp_sock` in Linux. Socket buffers and packets are represented by `struct`

`mbuf` in FreeBSD and `struct sk_buff` in Linux.

As discussed in Chapter 4, the network stack of the host operating system needs to be extended in order to support connection handoff. The two main tasks involved with implementing connection handoff are adding the bypass layer and extending the driver interface. Since the FreeBSD and Linux network stacks have similar organizations, both operating systems can theoretically implement the bypass layer and the handoff interface in similar ways. However, the commands defined in the handoff interface may not be able to accommodate the types of operations performed on sockets for both operating systems. Remember that the bypass layer forwards user requests (from sockets) to the device driver through the handoff interface (the extended device driver interface), and that the handoff interface is primarily used to synchronize the sockets on the host operating system and the NIC. Thus, as long as the transport interfaces in Figure 7.1 are similar in both operating systems, they should be able to implement handoff without altering the handoff interface.

Fortunately, the transport interfaces are similar in both operating systems, by design, not by coincidence. The transport interface, the software interface between sockets and TCP, is dictated by the following two needs. First, it needs to be able to handle sockets API functions such as listen, accept, connect, close, send, and receive by making corresponding requests to TCP. There is typically one function defined in the transport interface for each of the sockets API functions. Since the sockets API is largely same in both operating systems, the part of the transport interface that handles sockets API also needs to be similar. Functions related to the sockets API in fact account for a bulk of the transport interface. Second, a small number of functions in the transport interface are used to control allocation of TCP control block and/or its association with socket. A socket (`struct socket`) is logically owned by the user process and operates synchronously with respect to user requests; its state changes as well as its allocation (and de-allocation in most cases) take place immediately in response to various system calls. However, because TCP operates asynchronously with respect to sockets (and those user processes that own them), a TCP control block (`struct tcpcb` in FreeBSD and `struct tcp_sock` in Linux) may ex-

ist without an associated socket. So, the kernel sometimes needs to be able to allocate or de-allocate TCP control blocks without user requests or associate a TCP control block with a socket or dissociate it from the socket (detach and attach in FreeBSD terminology), without explicit user requests. For example, when a remote machine establishes a connection to a listen (server) port, the kernel allocates a TCP control block for the new connection. However, the new control block does not have its own socket until the application requests for a new socket through the `accept` system call, which allocates a new socket and attaches the TCP control block to the new socket. Likewise, when the application closes and established connection through the `close` system call, the kernel detaches the corresponding socket from the TCP control block and can de-allocate the socket, but not the TCP control block. The kernel simply initiates the teardown procedure of TCP. Only when the teardown procedure completes, the TCP control block can be de-allocated.

Finally, TCP state variables are standardized, which helps facilitate handoff process. Various RFCs that define TCP operations typically provide reference pseudo-implementations. Actual TCP implementations tend to use the same set of state variables used in the reference implementations. So, much of the handoff process implemented in FreeBSD can be reused to implement handoff in Linux.

7.2 Implementation Difficulties

The previous section discussed similarities between the FreeBSD and Linux network stacks and how they enable both operating systems to support offload through connection handoff. The two network stacks do have distinct implementations, and specific implementation choices affect the implementation of connection handoff. This section examines some of the difficulties that arise from the differences.

First, as shown in Figure 7.1, FreeBSD and Linux both use data structures (`struct mbuf` in FreeBSD and `struct sk_buff`) that are customized specifically for representing socket buffers and packets. However, they have very different semantics. `struct mbuf` in FreeBSD represents a piece of contiguous memory buffer, and multiple `mbuf`'s

can be chained together to form an arbitrarily long socket buffer or packet. In contrast, `struct sk_buff` in Linux represents a packet, not a generic piece of memory buffer. Like `mbuf`, `sk_buff`'s can be chained together to form an arbitrarily long socket buffer, but not a packet. A single `sk_buff` represents a single packet. In order to represent a packet consisting of multiple disjoint memory buffers, `sk_buff` includes special fields `frags` that reference disjoint buffers. Using `mbuf`, such a packet will have multiple `mbuf`'s. Thus, when new user data is enqueued into the send socket buffer, FreeBSD simply allocates `mbuf`'s to represent the new data. In contrast, Linux breaks up the user data into maximum segment size (MSS) `sk_buff`'s, since each of such `sk_buff`'s will be transmitted as one packet by TCP. When manipulating socket buffers, connection hand-off essentially needs two methods: one method that removes a number of bytes from the head of a socket buffer, and the other that inserts a number of bytes to the tail of a socket buffer. These operations do not require a high-level construct like `sk_buff` that represents a packet, and low-level `mbuf`'s are easier to manipulate.

Second, there are a number of extensions to the basic TCP protocol, which are usually published in various RFCs, and the Linux network stack tends to support more features than FreeBSD. Commonly implemented protocol extensions include congestion control algorithms (slow start, congestion avoidance, fast retransmit, fast recovery) also known as Reno [4], an updated fast recovery (NewReno) [31], a newer congestion avoidance (TCP Vegas) [16], several extensions to address fast network speeds and long latencies such as window scaling and protection against wrapped sequence numbers (PAWS) [42], selective acknowledgment (SACK) [56], forward acknowledgment (FACK) [57], and an extension to perform well on lossy links like wireless networks (TCP Westwood) [55]. Some features such as SACK are negotiated during connection establishment. If the host operating system supports SACK, but the NIC does not, then handoff must be aborted in order to avoid confusing the remote machine. However, these extensions do not require changes to the handoff interface since their implementation is contained within TCP and do not alter the interaction between TCP and sockets.

Finally, there may be extensions to the TCP implementation that are specific to the operating system. For instance, Linux also implements Linux-specific socket options such as `TCP_CORK`. When the application sends data, the kernel enqueues the data to the send socket buffer and then TCP checks whether a new segment should be transmitted. With `TCP_CORK`, the kernel only enqueues the new data into the socket buffer but never transmit new segments. This option is internally implemented as an extension to the Nagle algorithm. If the NIC does not support such options, they may be ignored without compromising correctness. However, the NIC should support as many options as possible to ensure that the application achieves its intended effects.

7.3 Summary

The handoff interface described in Section 4.2 has been implemented in both the FreeBSD and Linux operating systems. Currently, the Linux prototype provides fewer functionalities than the FreeBSD prototype because of a limited amount of effort spent on the Linux prototype, not because of inherent incompatibilities between the Linux network stack implementation and the handoff interface. FreeBSD and Linux have distinct TCP implementations. FreeBSD network stack implementation derives from BSD, whereas Linux network stack implementation is largely independent from other operating systems. Another key difference between the two operating system is the data structures used to represent socket buffers and packets. Despite these differences, their interfaces between the socket layer and the TCP layer are similar because TCP functionalities are standardized and TCP's interactions with the user application through the sockets API are largely same in most operating systems. The connection handoff interface logically captures a portion of this interface in order to enable communication between the socket on the host operating system and the socket on the NIC, and the bypass layer simply relays user requests to the NIC through the connection handoff interface. So, the similarity in the interfaces between socket and TCP in both operating systems allows them to support offload using the same handoff interface.

Chapter 8

Conclusions

Network servers need to improve the performance of the TCP/IP network stack in order to handle a growing number of clients and networking applications, and increasing network speeds. However, as a result of the growing gap between the CPU and main memory speeds and the presence of main memory accesses within the network stack, general-purpose CPUs are becoming increasingly inefficient at executing the network stack. Processing a packet requires only a modest number of instructions, but they involve frequent memory accesses. Because each main memory access can take hundreds of CPU cycles, even a small number of main memory accesses can significantly increase packet processing time. Thus, in order to improve the network stack performance, the system should reduce main memory accesses as much as possible.

The well-known cause for main memory accesses is data touching operations such as checksum calculation and data copy. Fortunately, solutions exist. Checksum offload and zero-copy I/O techniques eliminate these operations and associated main memory accesses, and many systems now support them. However, there is another type of main memory accesses, which are due to connection states. As the number of connections increases, connection data structures saturate the CPU caches, which then leads to cache misses and main memory accesses. With data touching operations eliminated, these main memory accesses are now becoming a bottleneck within the network stack. Unfortunately, accesses to connection states are integral to packet processing, so they cannot be eliminated. Simply increasing the CPU cache size is not an effective solution because a majority of the CPU die is already dedicated to caches, and a few thousand connections easily saturate currently available CPU caches.

Offloading TCP processing from the host CPU to a special-purpose processor (the offloading processor) on the NIC can be an effective way to improve the networking performance of the system. The offloading processor can exploit its own fast memory to store connection states and quickly access them, which allows it to process packets more efficiently than the host CPU. Offloading reduces instruction and memory bandwidth on the host processor, so the host CPU will be able to spend more time executing applications and improve overall system performance. There are several ways to achieve offload. Typical full offload moves all TCP functionalities to the NIC, and TCP processing occurs only on the NIC. However, this approach creates serious problems. Because the NIC has inherently limited compute power, it can become a bottleneck in the system. Full offload also complicates the existing software architecture of the network stack because the operating system and the NIC now need to share global resources like ports and IP routes. Thus, offloading has the potential to improve system performance, but it should be achieved in a way that allows the system to avoid creating a bottleneck at the NIC and that does not overly complicate the existing network stack architecture.

The framework for utilizing the offloading NIC described in this dissertation uses connection handoff. Offload based on connection handoff achieves the performance benefits of offloading while avoiding the problems associated with full offload. Using handoff, the operating system establishes connections and hand them off the NIC as it wishes. The operating system can now control the division of work between the host operating system and the NIC by controlling the number of connections handed off to the NIC. Also, only the operating system needs to manage ports and IP routes. When handing off a connection, the operating system simply gives the NIC the correct port and IP route of that connection.

In addition, this dissertation presents three policies for effectively utilizing the offloading processor. First, the NIC prioritizes packet processing and gives high priority to packets that belong to the connections handled by the host CPU. Prioritization ensures that offload does not degrade performance of the connections handled by the host CPU. Second, the NIC monitors its current load and dynamically adjusts the number of connections in or-

der to avoid creating a bottleneck at the NIC. Third, the operating system can employ application-specific connection selection policy in order to better utilize the resources on the NIC. These policies allow the system to treat the offloading NIC as an acceleration coprocessor and utilize it as much as its resources will allow, without overloading it. The results presented in this dissertation show that offload based on handoff results in significant performance improvements for web workloads.

Finally, the handoff-based framework is independent of the particular location of the offloading processor. The prototype implementation used in this dissertation assumes that the offloading processor resides on the NIC, but it may also be placed in other locations such as the host CPU die, the chipset, and the system board. The operating system can still use the same framework to hand off connections to the offloading processor and control the division of work between the host CPU and the offloading processor.

There are several possible research opportunities as follow-up to this dissertation. First, building a real hardware NIC that is as capable as the simulated NICs used in this dissertation will help validate the results and also allow for more extensive evaluations. Second, exploring architectures for the offload processor will help understand how much processing capacity is realizable on the NIC. While some commercial offloading NICs show that they are able to achieve near 10Gb/s, they consume significantly more power than regular NICs. Designing a power-efficient offloading processor would provide architects with a challenging problem. It would also involve exploring potential hardware acceleration features that can reduce power consumption. More importantly, such architectural exploration would help determine how much compute power is realistically available on the NIC. This will in turn help decide whether offloading can be a viable solution in the long term. Third, implementing the offloading processor in the various places discussed in Chapter 6 will enable quantitative comparisons among the places and also expose implementation details that have not been covered in this dissertation. However, this would necessarily require simulations. Finally, moving TCP connections from one machine to another in a more a disciplined way can help improve interoperability between various operating systems and

offloading NICs. The operating system and the offloading NIC may use different implementations of TCP/IP network. Since the fundamental TCP operations are common to all implementations, handoff is theoretically possible between any two implementations. However, one implementation may provide a richer feature set than another or vice versa. In this case, handoff may be impossible without further support for handoff from both implementations. One can imagine developing a handoff process that would be able to negotiate implementation differences during handoff. Such a handoff process will not only help handoff-based offloading but other applications that need to migrate connection among different systems.

Bibliography

- [1] Advanced Micro Devices. *AMD Athlon Processor x86 Code Optimization Guide*, February 2002. Revision K.
- [2] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus ipc: The end of the road for conventional microarchitectures. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 248–259, June 2000.
- [3] Alaa R. Alameldeen, Carl J. Mauer, Min Xu, Pacia J. Harper, Milo M.K. Martin, Daniel J. Sorin, Mark D. Hill, and David A. Wood. Evaluating Non-deterministic Multi-threaded Commercial Workloads. In *Proceedings of the 5th Workshop On Computer Architecture Evaluation using Commercial Workloads*, February 2002.
- [4] M. Allman and V. Paxson W. Stevens. TCP Congestion Control. IETF RFC 2581, April 1992.
- [5] Alteon Networks. *Tigon/PCI Ethernet Controller*, August 1997. Revision 1.04.
- [6] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous Profiling: Where Have All the Cycles Gone? In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 1–14. ACM Press, 1997.
- [7] Martin F. Arlitt and Carey L. Williamson. Internet Web Servers: Workload Characterization and Performance Implications. *IEEE/ACM Transactions on Networking*, 5(5):631–645, October 1997.
- [8] P. Balaji, W. Feng, Q. Gao, R. Noronha, W. Yu, and D. K. Panda. Head-to-TOE Evaluation of High-Performance Sockets over Protocol Offload Engines. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2005.
- [9] Kalpana Banerjee, Aniruddha Bohra, Suresh Gopalakrishnan, Murali Rangarajan, and Liviu Iftode. Split-OS: An Operating System Architecture for Clusters of Intelligent Devices. Work-in-Progress Session at the 18th Symposium on Operating Systems Principles, October 2001.
- [10] Gaurav Banga and Peter Druschel. Measuring the Capacity of a Web Server. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

- [11] Nathan L. Binkert, Erik G. Hallnor, and Steven K. Reinhardt. Network-Oriented Full-System Simulation using M5. In *Proceedings of the Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, pages 36–43, February 2003.
- [12] Nathan L. Binkert, Lisa R. Hsu, Ali G. Saidi, Ronald G. Dreslinski, Andrew L. Schultz, and Steven K. Reinhardt. Analyzing NIC Overheads in Network-Intensive Workloads. In *Proceedings of the Eighth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, February 2005.
- [13] Nathan L. Binkert, Lisa R. Hsu, Ali G. Saidi, Ronald G. Dreslinski, Andrew L. Schultz, and Steven K. Reinhardt. Performance Analysis of System Overheads in TCP/IP Workloads. In *Proceedings of the 14th Conference on Parallel Architectures and Compilation Techniques*, September 2005.
- [14] Mats Björkman and Per Gunningberg. Performance Modeling of Multiprocessor Implementations of Protocols. *IEEE/ACM Transactions on Networking*, 6(3):262–273, 1998.
- [15] Daniel Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly Media, Inc., 3rd edition, 2005.
- [16] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *Proceedings of the ACM SIGCOMM '94 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 24–35, 1994.
- [17] Tim Brecht, G. (John) Janakiraman, Brian Lynn, Vikram Saletore, and Yoshio Turner. Evaluating Network Processing Efficiency with Processor Partitioning and Asynchronous I/O. In *Proceedings of the EuroSys 2006*, pages 265–278, April 2006.
- [18] 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.
- [19] Doug Burger, James R. Goodman, and Alain Kägi. Limited Bandwidth to Affect Processor Design. *IEEE MICRO*, 17(6):55–62, November 1997.
- [20] David Callahan, Ken Kennedy, and Allan Porterfield. Software Prefetching. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52. ACM Press, 1991.
- [21] 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.
- [22] Jonathan Chang, Stefan Rusu, Jonathan Shoemaker, Simon Tam, Ming Huang, Mizan Haque, Siufu Chiu, Kevin Truong Mesbah Karim, Gloria Leong, Kiran Desai, and

- Richard Goe Sandhya Kulkarni. A 130-nm Triple- V_t 9-MB Third-Level On-Die Cache for the 1.7-GHz Itanium 2 Processor. *IEEE Journal of Solid-State Circuits*, 40(1):195–203, January 2005.
- [23] H. K. Jerry Chu. Zero-Copy TCP in Solaris. In *Proceedings of the 1996 Annual USENIX Technical Conference*, pages 253–264, 1996.
- [24] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine*, pages 23–29, June 1989.
- [25] David D. Clark and David L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *Proceedings of the ACM SIGCOMM '90 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 200–208, 1990.
- [26] Lawrence T. Clark, Eric J. Hoffman, Jay Miller, Manish Biyani, Yuyun Liao, Stephen Strazdus, Michael Morrow, Kimberley E. Velarde, and Mark A. Yarch. An Embedded 32-b Microprocessor Core for Low-Power and High-Performance Applications. *IEEE Journal of Solid-State Circuits*, 36(11):1599–1608, November 2001.
- [27] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly Media, Inc., 3rd edition, 2005.
- [28] 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.
- [29] Dave Dunning, Greg Regnier, Gary McAlpine, Don Cameron, Bill Shubert, Frank Berry, Anee Marie Merritt, Ed Gronke, and Chris Dodd. The Virtual Interface Architecture. *IEEE MICRO*, 18(2):66–76, March 1998.
- [30] W. Feng, P. Balaji, C. Baron, L. N. Bhuyan, and D. K. Panda. Performance Characterization of a 10-Gigabit Ethernet TOE. In *Proceedings of the 13th IEEE Symposium on High-Performance Interconnects*, 2005.
- [31] S. Floyd, T. Henderson, and A. Gurtov. The NewReno Modification to TCP's Fast Recovery Algorithm. IETF RFC 3782, April 2004.
- [32] Sally Floyd and Van Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.
- [33] Doug Freimuth, Elbert Hu, Jason LaVoie, Ronald Mraz, Erich Nahum, Prashant Pradhan, and John Tracey. Server Network Scalability and TCP Ooffload. In *Proceedings of the 2005 Annual USENIX Technical Conference*, pages 209–222, April 2005.
- [34] Mor Harchol-Balter and Allen B. Downey. Exploiting Process Lifetime Distributions for Dynamic Load Balancing. *ACM Transactions on Computer Systems*, 15(3):253–285, August 1997.

- [35] Thomas F. Herbert. *The Linux TCP/IP Stack: Networking for Embedded Systems*. Charles River Media, Inc., 2004.
- [36] Yatin Hoskote, Bradley A. Bloechel, Gregory E. Dermer, Vasantha Erraguntla, David Finan, Jason Howard, Dan Klowden, Siva Narendra, Greg Ruhl, James W. Tschanz, Sriram Vangal, Venkat Veeramachaneni, Howard Wilson, Jianping Xu, and Nitin Borkar. A TCP Offload Accelerator for 10 Gb/s Ethernet in 90-nm CMOS. *IEEE Journal of Solid-State Circuits*, 38(11):1866–1875, November 2003.
- [37] Lisa R. Hsu, Ali G. Saidi, Nathan L. Binkert, and Steven K. Reinhardt. Sampling and Stability in TCP/IP Workloads. In *Proceedings of the First Annual Workshop on Modeling, Benchmarking, and Simulation (MoBS)*, pages 68–77, 2005.
- [38] Ram Huggahalli, Ravi Iyer, and Scott Tetrick. Direct Cache Access for High Bandwidth Network I/O. In *Proceedings of the 32th International Symposium on Computer Architecture*, 2005.
- [39] HyperTransport Technology Consortium. *HyperTransport I/O Link Specification*, April 2006. Revision 3.0.
- [40] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*, 2002.
- [41] Intel Corporation. *Intel I/O Acceleration Technology*, 2006. Technology Brief.
- [42] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. IETF RFC 1323, May 1992.
- [43] Ron Kalla, Balaram Sinharoy, and Joel M. Tendler. IBM Power5 Chip: A Dual-Core Multithreaded Processor. *IEEE MICRO*, 24(2):40–47, 2004.
- [44] Dongkeun Kim, Steve Shih wei Liao, Perry H. Wang, Juan del Cuillo, Xinmin Tian, Xiang Zou, Hong Wang, Donald Yeung, Milind Girkar, and John P. Shen. Physical Experimentation with Prefetching Helper Threads on Intel's Hyper-Threaded Processors. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, pages 27–38, 2004.
- [45] Hyong-youb Kim and Scott Rixner. *Performance Characterization of the FreeBSD Network Stack*. Computer Science Department, Rice University, June 2005. Technical Report TR05-450.
- [46] Hyong-youb Kim and Scott Rixner. Connection Handoff Policies for TCP Offload Network Interfaces. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, November 2006.
- [47] Hyong-youb Kim and Scott Rixner. TCP Offload through Connection Handoff. In *Proceedings of the EuroSys 2006*, pages 279–290, April 2006.

- [48] 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.
- [49] Ravi Kokku, Ramakrishnan Rajamony, Lorenzo Alvisi, and Harrick Vin. Half-pipe Anchoring: An Efficient Technique for Multiple Connection Handoff. In *Proceedings of the 10th IEEE International Conference on Network Protocols*, pages 12–21, November 2002.
- [50] Jonathan Lemon. Kqueue: A generic and scalable event notification facility. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, June 2001.
- [51] Yue Luo, Juan Rubio, Lizy Kurian John, Pattabi Seshadri, and Alex Mericas. Benchmarking Internet Servers on Superscalar Machines. *Computer*, 36(2):34–40, February 2003.
- [52] Terry Lyon, Eric Delano, Cameron McNairy, and Dean Mulla. Data Cache Design Considerations for the Itanium 2 Processor. In *Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 356–362, 2002.
- [53] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A Full System Simulation Platform. *Computer*, 35(2):50–58, 2002.
- [54] Srihari Makineni and Ravi Iyer. Architectural Characterization of TCP/IP Packet Processing on the Pentium M Microprocessor. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, pages 152–162, February 2004.
- [55] Saverio Mascolo, Claudio Casetti, Mario Gerla, M. Y. Sanadidi, and Ren Wang. TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, pages 287–297, 2001.
- [56] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. IETF RFC 2018, October 1996.
- [57] Matthew Mathis and Jamshid Mahdavi. Forward Acknowledgment: Refining TCP Congestion Control. In *Proceedings of the ACM SIGCOMM '96 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 281–291, 1996.
- [58] Paul E. McKenney and Ken F. Dove. Efficient Demultiplexing of Incoming TCP Packets. In *Proceedings of the ACM SIGCOMM '92 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 269–279, 1992.

- [59] Cameron McNairy and Rohit Bhatia. Montecito: A Dual-Core Dual-Thread Itanium Processor. *IEEE MICRO*, 25(2):10–20, 2005.
- [60] Microsoft Corporation. *Scalable Networking: Network Protocol Offload – Introducing TCP Chimney*, April 2004. WinHEC Version.
- [61] Ron Minnich, Dan Burns, and Frank Hady. The Memory-Integrated Network Interface. *IEEE MICRO*, 15(1):11–19, February 1995.
- [62] Jeffrey Mogul, Lawrence Brakmo, David E. Lowell, Dinesh Subhraveti, and Justin Moore. Unveiling the Transport. *ACM SIGCOMM Computer Communication Review*, 34(1):99–106, 2004.
- [63] Jeffrey C. Mogul. TCP offload is a dumb idea whose time has come. In *Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems*, pages 25–30, 2003.
- [64] Shubhendu S. Mukherjee, Babak Falsafi, Mark D. Hill, and David A. Wood. Coherent Network Interfaces for Fine-Grain Communication. In *Proceedings of the 23th International Symposium on Computer Architecture*, pages 247–258, 1996.
- [65] Erich Nahum, David Yates, Jim Kurose, and Don Towsley. Cache Behavior of Network Protocols. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 169–180. ACM Press, 1997.
- [66] Erich M. Nahum, Marcel-Catalin Rosu, Srinivasan Seshan, and Jussara Almeida. The Effects of Wide-Area Conditions on WWW Server Performance. In *Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 257–267. ACM Press, 2001.
- [67] Erich M. Nahum, David J. Yates, James F. Kurose, and Don F. Towsley. Performance Issues in Parallelized Network Protocols. In *Proceedings of the First Operating Systems Design and Implementation*, pages 125–137, 1994.
- [68] Vivek S. Pai, Mohit Aron, Gaurov Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 205–216, 1998.
- [69] 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.
- [70] 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.

- [71] Athanasios E. Papathanasiou and Eric Van Hensbergen. KNITS: Switch-based Connection Hand-off. In *Proceedings of IEEE INFOCOM '02*, volume 1, pages 332–341, 2002.
- [72] Murali Rangarajan, Kalpana Banerjee, Jihwang Yeo, and Liviu Iftode. *MemNet: Efficient: Offloading of TCP/IP Processing Using Memory-Mapped Communication*. Computer Science Department, Rutgers University, March 2002. Technical Report DCS-TR-485.
- [73] Murali Rangarajan, Aniruddha Bohra, Kalpana Banerjee, Enrique V. Carrera, Ricardo Bianchini, Liviu Iftode, and Willy Zwaenepoel. *TCP Servers: Offloading TCP/IP Processing in Internet Servers. Design, Implementation, and Performance*. Computer Science Department, Rutgers University, March 2002. Technical Report DCS-TR-481.
- [74] R. Recio, P. Culley, D. Garcia, and J. Hilland. An RDMA Protocol Specification. Version 1.0, October 2002.
- [75] Greg Regnier, Srihari Makineni, Ramesh Illikkal, Ravi Iyer, Dave Minturn, Ram Huggahalli, Don Newell, Linda Cline, and Annie Foong. TCP Onloading for Data Center Servers. *Computer*, 37(11):48–58, November 2004.
- [76] Greg Regnier, Dave Minturn, Gary McAlpine, Vikram A. Saletore, and Annie Foong. ETA: Experience with an Intel Xeon Processor as a Packet Processing Engine. *IEEE MICRO*, 24(1):24–31, January 2004.
- [77] Scott Rixner. Memory Controller Optimizations for Web Servers. In *Proceedings of the 37th Annual International Symposium on Microarchitecture*, pages 355–366, December 2004.
- [78] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM SIGCOMM Computer Communication Review*, 27(1):31–41, 1997.
- [79] Vincent Roca, Torsten Braun, and Christophe Diot. Demultiplexed Architectures: A Solution for Efficient STREAMS-Based Communication Stacks. *IEEE Network*, 11(4):16–26, 1997.
- [80] Stefan Rusu, Jason Stinson, Simon Tam, Justin Leung, Harry Muljono, and Brian Cherkauer. A 1.5-GHz 130-nm Itanium 2 Processor With 6-MB On-die L3 Cache. *IEEE Journal of Solid-State Circuits*, 38(11):1887–1895, November 2003.
- [81] 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.
- [82] Hemal Shah, James Pinkerton, Renato Recio, and Paul Culley. Direct Data Placement over Reliable Transports. Version 1.0, October 2002.

- [83] Piyush Shivam and Jeffrey S. Chase. On the Elusive Benefits of Protocol Offload. In *Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence*, pages 179–184. ACM Press, 2003.
- [84] Alex Snoeren, David G. Andersen, and Hari Balakrishnan. Fine-Grained Failover Using Connection Migration. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, March 2001.
- [85] Perry H. Wang, Jamison D. Collins, Hong Wang, Dongkeun Kim, Bill Greene, Kai-Ming Chan, Aamir B. Yunus, Terry Sych, Stephen F. Moore, and John P. Shen. Helper Threads via Virtual Multithreading On An Experimental Itanium 2 Processor-based Platform. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 144–155, 2004.
- [86] Klaus Wehrle, Frank Pählke, Hartmut Ritter, Daniel Muller, and Marc Bechler. *The Linux Network Architecture: Design and Implementation of Network Protocols in the Linux Kernel*. Prentice Hall, 2004.
- [87] Roland Westrelin, Nicolas Fugier, Erik Nordmark, Kai Kunze, and Eric Lemoine. Studying Network Protocol Offload With Emulation: Approach And Preliminary Results. In *Proceedings of the 12th Annual IEEE Symposium on High Performance Interconnects*, pages 84–90, August 2004.
- [88] Paul Willmann, Scott Rixner, and Alan L. Cox. An Evaluation of Network Stack Parallelization Strategies in Modern Operating Systems. In *Proceedings of the 2006 Annual USENIX Technical Conference*, pages 91–96, 2006.
- [89] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated*, volume 2. Addison-Wesley Publishing Company, 1994.
- [90] David J. Yates, Erich M. Nahum, James F. Kurose, and Don Towsley. Networking Support for Large Scale Multiprocessor Servers. In *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 116–125, 1996.