Connection Handoff Policies for TCP Offload Network Interfaces

Hyong-youb Kim and Scott Rixner
Rice University
Houston, TX 77005
{*hykim, rixner*}*@rice.edu*

Abstract

This paper presents three policies for effectively utilizing TCP offload network interfaces that support connection handoff. These policies allow connection handoff to reduce the computation and memory bandwidth requirements for packet processing on the host processor without causing the resource constraints on the network interface to limit overall system performance. First, prioritizing packet processing on the network interface ensures that its TCP processing does not harm performance of the connections on the host operating system. Second, dynamically adapting the number of connections on the network interface to the current load avoids overloading the network interface. Third, the operating system can predict connection lifetimes to select long-lived connections for handoff to better utilize the network interface. The use of the first two policies improves web server throughput by 12–31% over the baseline throughput achieved without offload. The third policy helps improve performance when the network interface can only handle a small number of connections at a time. Furthermore, by using a faster offload processor, offloading can improve server throughput by 33–72%.

1 Introduction

In order to address the increasing bandwidth demands of modern networked computer systems, there has been significant interest in offloading TCP processing from the host operating system to the network interface card (NIC). TCP offloading can potentially reduce the number of host processor cycles spent on networking tasks, reduce the amount of local I/O interconnect traffic, and improve overall network throughput [14, 18, 27, 30, 31]. However, the maximum packet rate and the maximum number of connections supported by an offloading NIC are likely to be less than those of a modern microprocessor due to resource constraints on the network interface [24, 28, 30]. Since the overall capabilities of an offloading NIC may lag behind modern microprocessors, the system should not delegate all TCP processing to the NIC. However, a custom processor with fast local memory on a TCP offloading NIC should still be able to process packets efficiently, so the operating system should treat such a NIC as an acceleration coprocessor and use as much resources on the NIC as possible in order to speed up a portion of network processing.

Connection handoff has been proposed as a mechanism to allow the operating system to selectively offload a subset of the established connections to the NIC [18, 22, 23]. Once a connection is handed off to the NIC, the operating system switches the protocol for that connection from TCP to a stateless bypass protocol that simply forwards application requests to send or receive data to the NIC. The NIC then performs all of the TCP processing for that connection. If necessary, the operating system can also reclaim the connection from the NIC after it has been handed off. Thus, the operating system retains complete control over the networking subsystem and can control the division of work between the NIC and host processor(s). At any time, the operating system can easily opt to reduce the number of connections on the NIC or not to use offload at all. Furthermore, with handoff, the NIC does not need to make routing decisions or allocate port numbers because established connections already have correct routes and ports.

While previous proposals have presented interfaces and implementations of TCP offload NICs that support connection handoff, they have not presented policies to utilize these NICs effectively. This paper shows that there are three main issues in systems that utilize connection handoff and evaluates policies to address these issues. First, the NIC must ensure that TCP processing on the NIC does not degrade the performance of connections that are being handled by the host. Second, the operating system must not overload the NIC since that would create a bottleneck in the system. Finally, when

the NIC is only able to store a limited number of connections, the operating system needs to hand off long-lived connections with high packet rates in order to better utilize the NIC.

Full-system simulations of four web workloads show that TCP processing on the NIC can degrade the performance of connections being handled by the host by slowing down packet deliveries to the host processor. The amount of time for a packet to cross the NIC increases from under 10 microseconds without handoff to over 1 millisecond with handoff. For three of the four workloads, the resulting request rate of the server is 14–30% lower than the baseline request rate achieved without handoff. The NIC can minimize delays by giving priority to those packets that must be delivered to the host. The use of this *host first* packet processing on the NIC increases the server's request rate over the baseline by up to 24%, when the NIC is not overloaded. However, the NIC still becomes overloaded when too many connections are handed off to the NIC, which reduces the request rate of the server by up to 44% below the baseline performance. The NIC can avoid overload conditions by dynamically adapting the number of connections to the current load indicated by the length of the receive packet queue. By using both techniques, handoff improves the request rate of the server by up to 31% over the baseline throughput. When the NIC can support a large number of connections, handing off connections in a simple first-come, first-served order is sufficient to realize these performance improvements. However, when the NIC has limited memory for storing connection state, handing off long-lived connections helps improve performance over a simple first-come, first-served handoff policy. Finally, using a NIC with a faster offload processor, handoff improves server throughput by 33–72%.

The rest of the paper is organized as follows. Section 2 briefly describes connection handoff. Section 3 presents techniques to control the division of work between the NIC and the operating system. Section 4 describes the experimental setup, and Section 5 presents results. Section 6 discusses related work. Section 7 draws conclusions.

2 Connection Handoff

TCP packet processing time is dominated by expensive main memory accesses, not computation [10, 17]. These main memory accesses occur when the network stack accesses packet data and connection data structures, which are rarely found in the processor caches. Accesses to packet data are due to data touching operations like data copies and checksum calculations. These accesses can be eliminated by using zero-copy I/O techniques to avoid data copies between the user and kernel memory spaces [9, 12, 26] and checksum offload techniques to avoid computing TCP checksums on the host processor [19]. These techniques have gained wide acceptance in modern operating systems and network interfaces. However, no such techniques exist to eliminate accesses to connection data structures. While these structures are small, around 1KB per connection, a large number of connections can easily overwhelm modern processor caches, significantly degrading performance. Previous experiments show that main memory accesses to connection data structures can degrade performance as much as data touching operations [17].

TCP offload can be used to reduce the impact of expensive main memory accesses to connection data structures. By moving TCP processing to the NIC, connection data structures can be efficiently stored in fast, dedicated memories on the NIC. Typically, all TCP processing is moved to the NIC. However, such full TCP offload is not scalable. First, the resource limitations of a peripheral device will limit the maximum processing capability and memory capacity of a TCP offload NIC. Second, TCP offload complicates the existing software architecture of the network stack, since the operating system and the NIC now need to cooperatively manage global resources like port numbers and IP routes [24]. Connection handoff solves these problems by enabling the host operating system to select a subset of the established connections and move them to the network interface [18, 22, 23]. Using handoff, the operating system remains in control of global resources and can utilize TCP offload NICs to accelerate as many TCP connections as the resources on the NIC will allow.

Using connection handoff, all connections are established within the network stack of the host operating system. Then, if the operating system chooses to do so, the connection can be handed off to the NIC. Once a connection is handed off to the NIC, the NIC handles all TCP packets for that connection. Figure 1 shows a diagram of a network stack that supports connection handoff. The left portion of the diagram depicts the regular (non-offload) stack, and the dotted lines show data movement. When a connection is offloaded to the NIC, the operating system switches its stack to the right (shaded) portion of the diagram. This new stack executes a simple bypass protocol on the host processor for offloaded connections, and the rest of the TCP/IP stack is executed directly on the NIC. The bypass protocol forwards socket operations between the socket layers on the host and the NIC via the device driver, as shown by the dashed lines in the figure. For an offloaded connection, packets are generated by the NIC and also completely consumed by the NIC, so packet headers are never transferred across the local I/O interconnect. The solid lines show the packet movement within the NIC. The lookup layer on the NIC

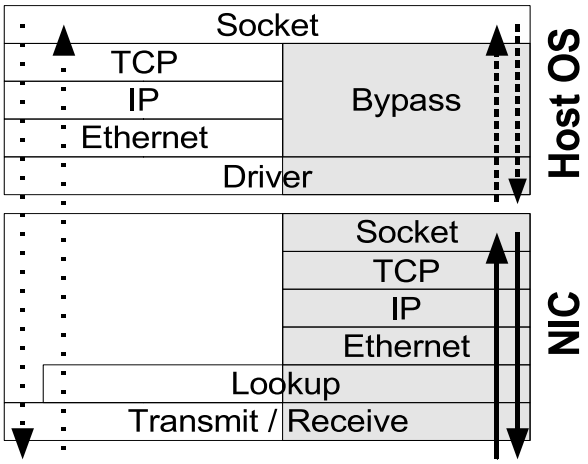

Figure 1: Modified network stack architecture to support connection handoff. The shaded region indicates the path used by connections on the NIC.

determines whether an incoming packet belongs to a connection on the NIC or to a connection on the host. It only adds a small amount of overhead to all incoming packets by using a hash-based lookup table [21].

The bypass layer communicates with the device driver using a connection handoff API. So, the operating system can transparently support multiple, heterogeneous NICs using the same API. Furthermore, the API also allows the actual socket buffer data to be stored in main memory in order to reduce the amount of buffering required on the NIC.

Previous experiments show that TCP offload, whether it is based on connection handoff or not, can reduce cycles, instructions, and cache misses on the host CPU as well as traffic across the local I/O interconnect [14, 18]. Since socket-level operations occur less frequently than Ethernet packet transmits and receives, handoff can reduce the number of message exchanges across the local I/O interconnect. For instance, because acknowledgment packets (ACKs) are processed by the NIC, the NIC may aggregate multiple ACKs into one message so that the host operating system can drop the acknowledged data in a single socket operation.

3 Connection Handoff Framework

In addition to the features described in the previous section, both the operating system and the network interface must also implement policies to ensure that the performance of connections that are being handled by the host operating system is not degraded, that the network interface does not become overloaded, and that the appropriate connections are handed off to the network interface. Figure 2 illustrates the proposed framework for

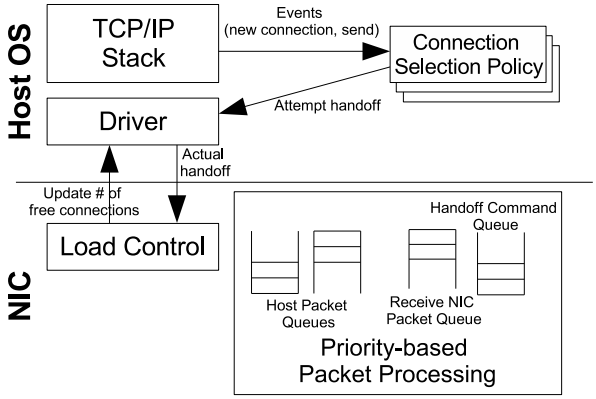

Figure 2: Framework for connection handoff and dynamically controlling the load on the NIC.

the implementation of such packet prioritization, load control, and connection selection policies. These policies are integral to the effective utilization of connection handoff network interfaces. The proposed framework and policies will be discussed in detail in the following sections—all supporting data was collected while running a simulated web server using the methodology that will be described in Section 4.

3.1 Priority-based Packet Processing

Traditional NICs act as a bridge between the network and main memory. There is very little processing required to send or receive packets from main memory (*host packets*), so the network interface can process each packet fairly quickly. However, with connection handoff, the NIC must perform TCP processing for packets that belong to connections that have been handed off to the network interface (*NIC packets*). Because NIC packets require significantly more processing than host packets, NIC packet processing can delay the processing of host packets, which may reduce the throughput of connections that remain within the host operating system. In the worst case, the NIC can become overloaded with TCP processing and drop host packets, which further reduces the throughput of host connections.

Table 1 illustrates the effect of connection handoff on host packet delays and overall networking performance for a web workload that uses 2048 simultaneous connections. 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 us 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

Offloaded Connections	Packet Priority	Host Packet Delay (usec) Send	Host Packet Delay (usec) Receive	Host Idle (%)	NIC Idle (%)	Requests/s
0	*no handoff*	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 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.

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 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 1, the NIC processes all packets on a first-come, first-served (FCFS) basis. As the load on the NIC increases, host packets suffer increasing delays. Since an offloading NIC is doing additional work for NIC packets, increased host packet delays are inevitable. However, such delays need to be minimized in order to maintain the performance of host connections. 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 2, enable such a prioritization. As discussed in Section 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. 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 1 shows the impact of using the priority queues to implement a *host first* packet processing policy on the NIC. With the *host first* policy on the NIC, the median packet delay of host packets is about 6–10 us 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 5.

3.2 Load Control

Packet prioritization can ensure that host packets are handled promptly, even when the NIC is heavily loaded. However, this will not prevent the NIC from becoming overloaded to the point where there are not enough processing resources remaining to process NIC packets. In such an overloaded condition, the network interface becomes a system bottleneck and degrades the performance of connections that have been handed off to the network interface. A weighted sum of the packet rate of NIC packets and the packet rate of host packets is an approximate measure of the load on the network interface. The number of connections that have been handed off to the NIC indirectly determines this load. In general, increasing the number of connections on the NIC increases the load on the NIC because they tend to increase overall packet rates. Likewise, decreasing the number of connections generally reduces the load on the NIC.

Due to the finite amount of memory on the NIC, there is a hard limit on the total number of connections that can be handed off to 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. Therefore, the network interface must dynamically control the number of connections that can be handed off based on the current load on the network interface.

As discussed in Section 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 acknowledgment 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 the values only depend on the hardware capabilities of the network interface, only the network interface manufacturer would need to tune the values, not the operating system. Finally, *Cnum* is the number of currently active connections on the NIC.

Figure 3 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

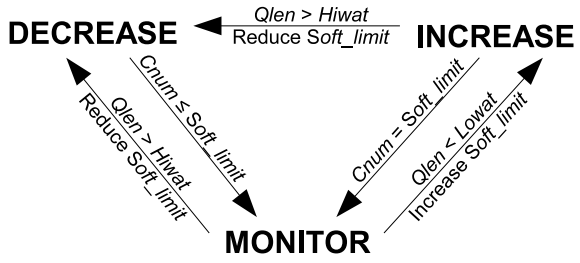

Figure 3: State machine used by the NIC firmware to dynamically control the number of connections on the NIC.

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 described framework 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. Actively restoring connections in this manner was not necessary for the workloads studied in this paper, but it may help improve performance for other types of workloads.

While the receive packet queue length is easy to exploit, there are other measures of load such as idle time and packet rate. The network interface could calculate either of these metrics directly and use them to control the load. However, packet processing time is extremely dependent of the workload. Therefore, metrics such as the packet rate are difficult to use, as they are not directly related to the load on the NIC. This makes it more desirable to control the load based on resource use, such as the length of the receive queue or the idle time, than based on packet rate. The receive queue length was cho-

sen over idle time because it requires no NIC resources to compute.

3.3 Connection Selection

Whenever the network interface can handle additional connections, the operating system attempts to hand off established connections. The connection selection policy component of the framework depicted in Figure 2 decides whether the operating system should attempt to hand off a given connection. As described previously, the device driver then performs the actual handoff. The operating system may attempt handoff at any time after a connection is established. For instance, it may hand off a connection right after it is established, or when a packet is sent or received. If 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 consider 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, 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 the application, so a single selection policy may not 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.

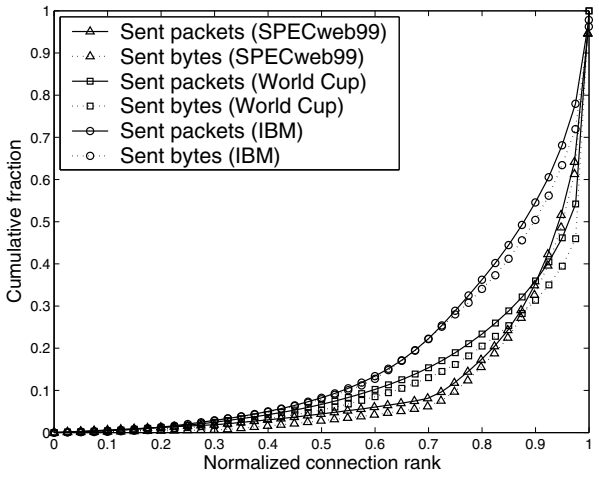

Figure 4: Distribution of connection lifetimes from SPECweb99 and the IBM and World Cup traces. Connection rank is based on the number of sent packets.

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 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. As shown in the figure, there are many short-lived connections, but the number of packets 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 for all three workloads. The other half of the connections send the remaining 90% of the packets. In fact, more than 45% of the total traffic is handled by less than 10% of the connections. The data shown in Figure 4 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 [2, 7, 8]. The operating system may exploit such distribution in order to identify and hand off long-lived connections. For instance, since the number of packets transfered 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 Experimental Setup

The authors have previously implemented connection handoff within FreeBSD 4.7 with the architecture described in Section 2 [18]. To evaluate the policies described in Section 3, this prototype is augmented with these policies using the existing framework. Since there are no offload controllers with open specifications, at least to the authors' best knowledge, an extended version of the full-system simulator Simics [20] is used for performance evaluations. Simics models the system hardware with enough detail that it can run complete and unmodified operating systems.

4.1 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 [29]. Table 2 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 [1]. 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 [15]. 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 [11]. 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

	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 10Gb/s wire

Table 2: Simulator configuration.

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 [5].

The testbed consists of a server and a client machine, directly connected through a full-duplex 10Gb/s wire. Both are simulated using Simics. The server uses the configuration shown in Table 2, while the client is completely functional, so will never be a performance bottleneck.

4.2 Web Workloads

The experiments use SPECweb99 and two real web traces to drive the Flash web server [25]. SPECweb99 emulates multiple simultaneous clients. Each client issues requests for both static content (70%) and dynamic content (30%) and tries to maintain its bandwidth between 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 of the 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 [4]. 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 where all requests are for static content.

For all experiments, the first 400000 packets are used to warm 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 [16]. In this paper, the warmup phase simulates timing, and 600000 packets lead to at least one second of simulated time for the experiments presented in the paper.

5 Experimental Results

5.1 Priority-based Packet Processing and Load Control

Figure 5 shows the execution profiles of the simulated web server using various configurations. The Y axis shows abbreviated system configurations (see Table 3 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 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 in 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 5 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.

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	*host first* 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 **T***n*: Threshold with value *n*
NIC MIPS	Instruction rate of the NIC in million instructions per second

Table 3: Configuration shorthands used in Section 5

Since the NIC has 62% idle time, handoff should be able to improve server performance. However, simply handing off many connection to the NIC can create a bottleneck at the NIC, as illustrated by *W-1024-N-N-FCFS-400*.

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

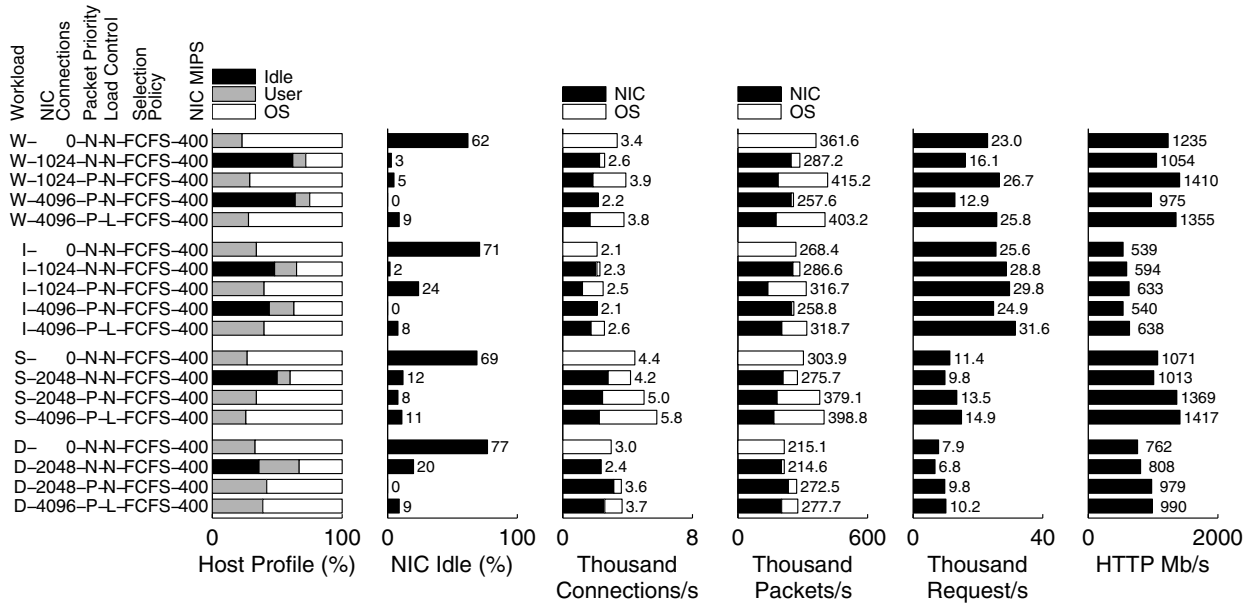

Figure 5: Impact of *host first* packet processing and load control on the simulated web server.

host, the request rate drops by 30%. 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 one 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. 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 it job, and host packets (mainly packets involved in new connection establishment) 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 3.2. Figure 6 shows how the NIC dynamically adjusts the number of connections during the experiment. Initially 2048 connections are handed off to

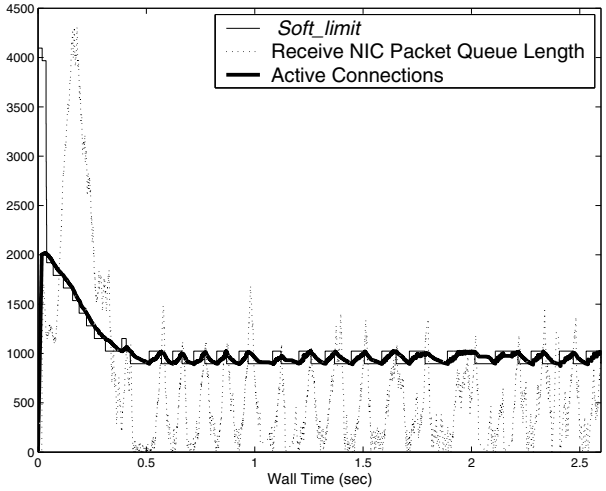

Figure 6: Dynamic adjustment of the number of connections on the NIC by the load control mechanism for configuration *W-4096-P-L-FCFS-400*.

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-*

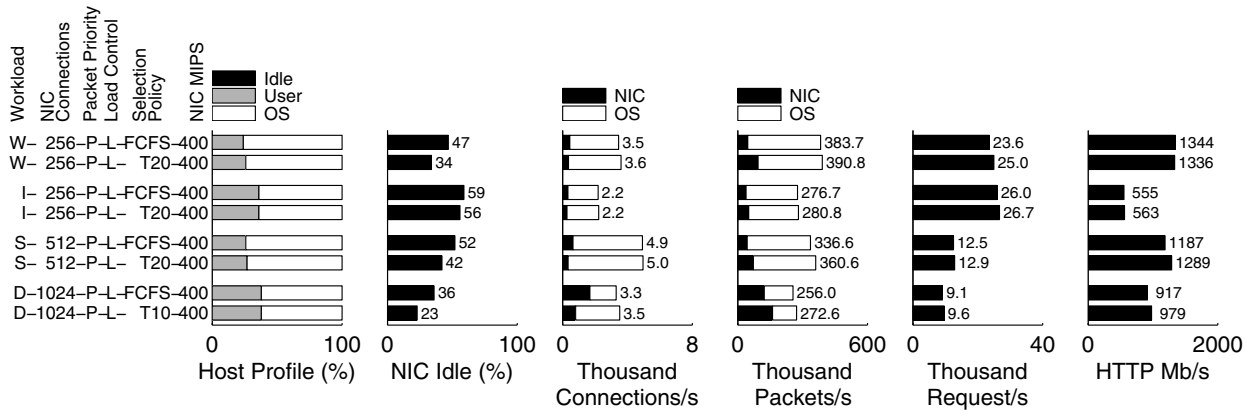

Figure 7: Impact of first-come, first-served and threshold-based connection selections on the simulated web server.

400, which indicates that the watermark values used in the load control mechanism are not optimal. Overall, handoff improves server throughput by 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 5 also 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, and the NIC has 69% idle time. When 2048 connections are handed off, the request rate actually drops slightly. 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, 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%.

5.2 Connection Selection Policy

As mentioned in Section 3.3, the system may use a threshold to differentiate long-lived connections that transfers many packets from short-lived ones. Handing off long-lived connections has the potential to improve server performance when the NIC has limited memory for a small number of connections. For instance, offload processors may use a small on-chip memory to store connections for fast access. In this case, it is necessary to be selective and hand off connections that transfer many packets in order to utilize the available compute power on the NIC as much as possible. On the other hand, when the NIC can handle a much larger number of connections, it is more important to hand off as many connections as possible, and a threshold-based selection policy has either negligible impact on server throughput or degrades it because fewer packets are processed by the NIC.

Figure 7 compares FCFS and threshold-based connection selection policies when the maximum number of connections on the NIC is much smaller than the value used in the previous section. 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 *T4*, the operating system attempts a handoff when the fourth en-

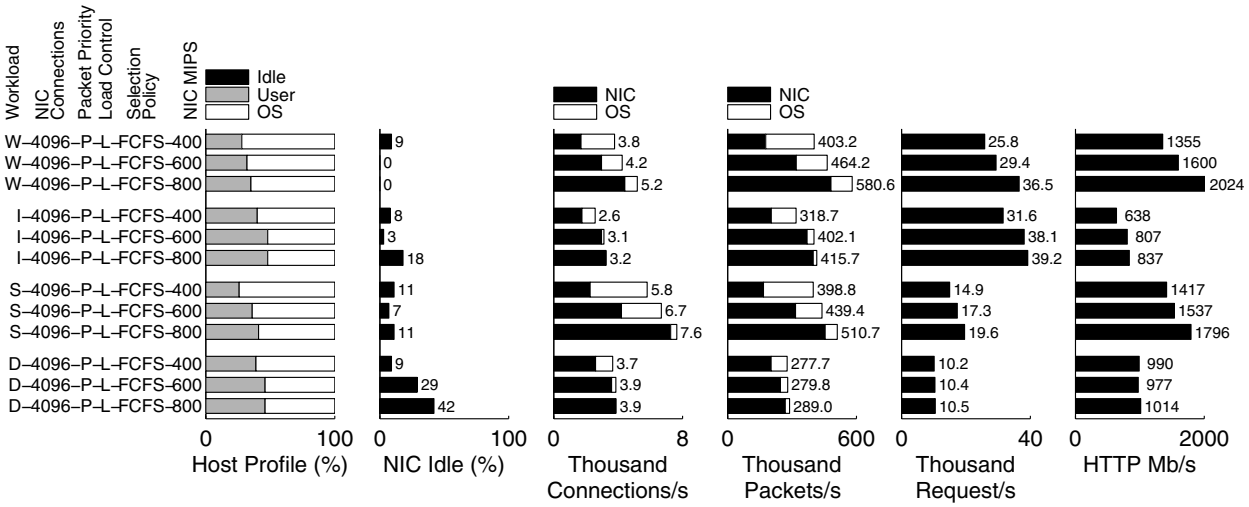

Figure 8: Impact of the instruction rate of the NIC on the simulated web server.

queue operation to the connection's send socket buffer occurs. As shown in the figure, the use of threshold enables the operating system to hand off longer connections than FCFS, but the resulting throughput improvements are small. For instance, *W-256-P-L-FCFS-400* shows a case in which the NIC can handle up to 256 connections and the operating system hands off connections on a FCFS basis. 13% of connections and 12% of packets are processed by the NIC, as expected. The NIC shows 47% idle time. When a threshold policy is used (*W-256-P-L-T20-400*), the NIC now processes 24% of packets, and the request rate improves by 6%. However, the NIC still has 34% idle time. The lifetime distribution shown in Figure 4 suggests that if the operating system were able to pick longest 10% of connections, the NIC would process over 60% of packets. Thus, with a more accurate selection policy, the NIC would be able to process a greater fraction of packets and improve system performance further.

5.3 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 8 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 5 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 $\times$ 400Kb/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.

Finally, HTTP response times, measured as the amount of time elapsed between when a request is sent and when the full response is received, follow server request rates, as expected. For instance, the mean response time for the World Cup trace is 61ms without offload (*W-0-N-N-FCFS-400*). It increases to 99ms when 1024 connections are offloaded without *host first* packet processing or load control (*W-1024-N-N-FCFS-400*). The

use of both *host first* packet processing and load control drops the mean response time to 60ms (*W-1024-P-L-FCFS-400*). Increasing the instruction rate of the NIC from 400 to 600 and 800MIPS further reduces the mean response time to 53ms and 40ms, respectively. Mean response times for other workloads follow trends similar to that of the World Cup trace, except that mean response times for SPECweb99 are larger than those for the World Cup and IBM traces because of throttling and dynamic content generation.

6 Related Work

There are a number of previous studies on full TCP offload to both network interfaces and dedicated processors in the system. TCP servers was an early TCP offload design [27]. TCP servers, based on the Split-OS concept [3], splits TCP and the rest of the operating system and lets a dedicated processor or a dedicated system execute TCP. Brecht *et al.* expand this concept by providing an asynchronous I/O interface to communicate with the dedicated TCP processing resources [6]. Intel has dubbed such approaches, which dedicate a general-purpose processor to TCP processing, TCP *on*loading [28]. Regardless of the name, these approaches are effectively full TCP offload, as TCP and the rest of the system's processing are partitioned into two components.

Freimuth *et al.* recently showed that full offload reduces traffic on the local I/O interconnect [14]. They used two machines for evaluations, one acting as the NIC and the other as the host CPU. A central insight is that with offload, the NIC and the operating system communicate at a higher level than the conventional network interface, which gives opportunities for optimizations. Westrelin *et al.* also evaluated the impact of TCP offload [31]. They used a multiprocessor system in which one processor is dedicated to executing TCP, like TCP onloading, and show a significant improvement in microbenchmark performance. Finally, an analytical study on performance benefits of TCP offload shows that offload can be beneficial but its benefits can vary widely depending on application and hardware characteristics [30].

However, while these studies have shown the benefits of TCP offload, they have not addressed the problems that have been associated with full TCP offload. These 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) [24].

Connection handoff, which addresses some of these concerns, has been previously proposed and implemented. Microsoft has proposed to implement a device driver API for TCP offload NICs based on connection handoff in the next generation Windows operating system, as part of the Chimney Offload architecture [22]. Mogul *et al.* argued that exposing transport (connection) states to the application creates opportunities for enhanced application features and performance optimizations, including moving connection states between the operating system and offload NICs [23]. The authors have implemented both the operating system and network interface components of connection handoff, with the architecture described in Section 2 [18]. The policies presented in this paper apply to all of these previous proposals and implementations and will improve their efficiency and performance, and prevent the network interface from becoming a performance bottleneck.

A recent study shows that a commercial offloading NIC can achieve over 7Gb/s and substantially improve web server throughput [13]. This is an encouraging result since it shows that a specialized offload processor can handle high packet rates.

7 Conclusion

Offloading TCP processing to the NIC can improve system throughput by reducing computation and memory bandwidth requirements on the host processor. However, the NIC inevitably has limited resources and can become a bottleneck in the system. Offload based on connection handoff enables the operating system to control the number of connections processed by the host processor and the NIC, thereby controlling the division of work between them. Thus, 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.

A system that implements connection handoff can employ the policies presented in this paper in order to fully utilize the offload NIC without creating a bottleneck in the system. 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 that can be handed off. This avoids overloading the NIC, which would create a performance 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 offloading NICs that lack 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 po-

lices 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.

Acknowledgments

The authors thank Alan L. Cox for his interest and comments on the paper. The authors also thank Robbert van Renesse for shepherding and the reviewers for their valuable comments. This work is supported in part by a donation from Advanced Micro Devices and by the National Science Foundation under Grant Nos. CCR-0209174 and CCF-0546140.

References

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

[2] M. F. Arlitt and C. L. Williamson. Internet Web Servers: Workload Characterization and Performance Implications. *IEEE/ACM Transactions on Networking*, 5(5):631–645, Oct. 1997.

[3] K. Banerjee, A. Bohra, S. Gopalakrishnan, M. Rangarajan, and L. Iftode. Split-OS: An Operating System Architecture for Clusters of Intelligent Devices. Work-in-Progress Session at the 18th Symposium on Operating Systems Principles, Oct. 2001.

[4] G. Banga and P. Druschel. Measuring the Capacity of a Web Server. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Dec. 1997.

[5] N. L. Binkert, E. G. Hallnor, and S. 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, Feb. 2003.

[6] T. Brecht, G. J. Janakiraman, B. Lynn, V. Saletore, and Y. Turner. Evaluating Network Processing Efficiency with Processor Partitioning and Asynchronous I/O. In *Proceedings of EuroSys 2006*, pages 265–278, Apr. 2006.

[7] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Schenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of IEEE INFOCOM '99*, volume 1, pages 126–134, Mar. 1999.

[8] P. Cao and S. Irani. Cost-Aware WWW Proxy Caching Algorithms. In *Proceedings of the 1997 USENIX Symposium on Internet Technology and Systems*, pages 193–206, Dec. 1997.

[9] H. K. J. Chu. Zero-Copy TCP in Solaris. In *Proceedings of the 1996 Annual USENIX Technical Conference*, pages 253–264, 1996.

[10] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine*, pages 23–29, June 1989.

[11] L. T. Clark, E. J. Hoffman, J. Miller, M. Biyani, Y. Liao, S. Strazdus, M. Morrow, K. E. Velarde, and M. 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, Nov. 2001.

[12] P. Druschel and L. 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, Dec. 1993.

[13] 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.

[14] D. Freimuth, E. Hu, J. LaVoie, R. Mraz, E. Nahum, P. Pradhan, and J. Tracey. Server Network Scalability and TCP Offload. In *Proceedings of the 2005 Annual USENIX Technical Conference*, pages 209–222, Apr. 2005.

[15] Y. Hoskote, B. A. Bloechel, G. E. Dermer, V. Erraguntla, D. Finan, J. Howard, D. Klowden, S. Narendra, G. Ruhl, J. W. Tschanz, S. Vangal, V. Veeramachaneni, H. Wilson, J. Xu, and N. Borkar. A TCP Offload Accelerator for 10 Gb/s Ethernet in 90-nm CMOS. *IEEE Journal of Solid-State Circuits*, 38(11):1866–1875, Nov. 2003.

[16] L. R. Hsu, A. G. Saidi, N. L. Binkert, and S. 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.

[17] H. Kim and S. Rixner. *Performance Characterization of the FreeBSD Network Stack*. Computer Science Department, Rice University, June 2005. Technical Report TR05-450.

[18] H. Kim and S. Rixner. TCP Offload through Connection Handoff. In *Proceedings of EuroSys 2006*, pages 279–290, Apr. 2006.

[19] K. Kleinpaste, P. Steenkiste, and B. 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, Aug. 1995.

[20] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hállberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *Computer*, 35(2):50–58, 2002.

[21] P. E. McKenney and K. 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.

[22] Microsoft Corporation. *Scalable Networking: Network Protocol Offload – Introducing TCP Chimney*, Apr. 2004. WinHEC Version.

[23] J. Mogul, L. Brakmo, D. E. Lowell, D. Subhraveti, and J. Moore. Unveiling the Transport. *ACM SIGCOMM Computer Communication Review*, 34(1):99–106, 2004.

[24] J. 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.

[25] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, pages 199–212, June 1999.

[26] V. S. Pai, P. Druschel, and W. 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, Feb. 1999.

[27] M. Rangarajan, A. Bohra, K. Banerjee, E. V. Carrera, R. Bianchini, L. Iftode, and W. Zwaenepoel. *TCP Servers: Offloading TCP/IP Processing in Internet Servers. Design, Implementation, and Performance*. Computer Science Department, Rutgers University, Mar. 2002. Technical Report DCR-TR-481.

[28] G. Regnier, S. Makineni, R. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. TCP Onloading for Data Center Servers. *Computer*, 37(11):48–58, Nov. 2004.

[29] S. Rixner. Memory Controller Optimizations for Web Servers. In *Proceedings of the 37th Annual International Symposium on Microarchitecture*, pages 355–366, Dec. 2004.

[30] P. Shivam and J. S. Chase. On the Elusive Benefits of Protocol Offload. In *Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence*, pages 179–184, 2003.

[31] R. Westrelin, N. Fugier, E. Nordmark, K. Kunze, and E. 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, Aug. 2004.