# TCP Offload through Connection Handoff

Hyong-youb Kim and Scott Rixner
Rice University
Houston, TX 77006
{hykim, rixner}@rice.edu

## ABSTRACT

This paper presents a connection handoff interface between the operating system and the network interface. Using this interface, the operating system can offload a subset of TCP connections in the system to the network interface, while the remaining connections are processed on the host CPU. Offloading can reduce computation and memory bandwidth requirements for packet processing on the host CPU. However, full TCP offloading may degrade system performance because finite processing and memory resources on the network interface limit the amount of packet processing and the number of connections. Using handoff, the operating system controls the number of offloaded connections in order to fully utilize the network interface without overloading it. Handoff is transparent to the application, and the operating system may choose to offload connections to the network interface or reclaim them from the interface at any time. A prototype system based on the modified FreeBSD operating system shows that handoff reduces the number of instructions and cache misses on the host CPU. As a result, the number of CPU cycles spent processing each packet decreases by 16–84%. Simulation results show handoff can improve web server throughput (SEPCweb99) by 15%, despite short-lived connections.

## Categories and Subject Descriptors

D.4.4 [**Operating Systems**]: Communications Management – Network Communication

## General Terms

Experimentation, Performance

## Keywords

TCP Offload, Connection Handoff, Operating System, Programmable Network Interface

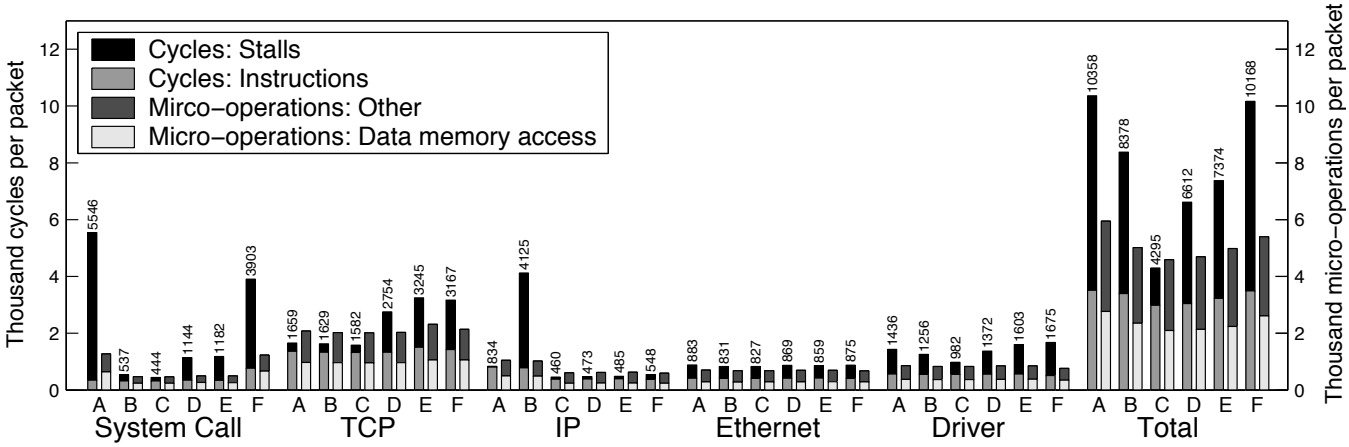
## 1. INTRODUCTION

Main memory accesses have been a long-standing bottleneck in TCP processing. Data touching operations such as data copy and checksum calculations were identified as a bottleneck early on [2]. On a modern computer, processing each TCP packet requires about 3100 instructions, but more than half of those instructions involve data memory accesses. Techniques like zero-copy I/O and checksum offload have reduced processing time by eliminating expensive memory accesses to packet data [3, 7, 16]. Now, memory accesses to connection data structures are becoming a bottleneck in TCP packet processing. As the number of connections increases, connection structures start to overwhelm the CPU caches, causing main memory accesses that take over two hundred CPU cycles. For instance, on an Athlon-based system, processing a TCP packet takes 10358 cycles when a single connection is used. Zero-copy I/O and checksum offload reduce the number to 4295. However, when 1024 connections are used, the time increases again to 7374 cycles, showing that main memory accesses to connection data structures are expensive.

TCP offload schemes can reduce memory accesses to connection data structures by offloading connections to the network interface card (NIC) and running the network stack on the NIC. However, the typical approach (full TCP offload) in which all packet processing is performed on the NIC has several drawbacks. Because the NIC has limited computation and memory resources, it can degrade rather than improve system performance. Saturating a full-duplex 10 Gb/s link using TCP packets requires about 8 billion operations per second, which is likely to exceed the available computational capacity of a NIC without resorting to specialized, inflexible hardware. Likewise, memory is limited on the NIC which will place hard limits on the number of connections that the NIC can support. Running TCP on the NIC also 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 [12].

Connection handoff has previously been suggested as a possible alternative to full offload [11]. This paper introduces an actual design and implementation of a connection handoff interface between the operating system and the NIC that can achieve the benefits of TCP offload while avoiding the drawbacks. Using the handoff interface, the operating system offloads established connections to the NIC, as appropriate. Once a connection is offloaded, the NIC processes TCP packets for that connection, and 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 operating system can also reclaim a connection from the NIC, if necessary. Thus, the operating system retains complete control over the networking subsystem and can control the division of work between the NIC and host CPU. The proposed connection handoff interface also allows the NIC to decide to return a connection to the operating system. When the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*EuroSys'06,* April 18–21, 2006, Leuven, Belgium.
Copyright 2006 ACM 1-59593-322-0/06/0004 ...$5.00.




| | Connections | Zero-copy | Checksum offload | Dummynet latency | Benchmark |
|---|---|---|---|---|---|
| A | 1 | No | No | 0ms | TCP Send |
| B | 1 | Yes | No | 0ms | TCP Send |
| C | 1 | Yes | Yes | 0ms | TCP Send |
| D | 1024 | Yes | Yes | 0ms | TCP Send |
| E | 1024 | Yes | Yes | 20ms | TCP Send |
| F | 256 | Yes + No | Yes | 0ms | SPECweb99 |

**Figure 1: Execution profiles of the network stack under various configurations.**

NIC runs low on resources, it can move connections back to the operating system to avoid performance losses. The operating system can easily opt to reduce the number of connections on the NIC or not to use offload entirely. With handoff, the NIC does not need to make routing decisions or allocate port numbers because established connections already have correct routes and ports.

A prototype is built using a modified version of the FreeBSD operating system and a programmable Gigabit Ethernet controller. Experimental results show that handoff reduces L2 cache misses as well as instructions in the host network stack. Consequently, the number of cycles spent in the host network stack for each packet decreases by 16–84%. Moreover, the volume of message traffic across the local I/O interconnect also decreases with handoff. Currently, the prototype is limited by the slow programmable controller. Full-system simulations of a system with a faster controller show that these reductions improve SPECweb99 web server throughput by 15%, even though the workload comprises many small files and short-lived connections. The modifications to the operating system are confined within the network stack below the socket layer. Other operating systems that employ BSD based network stacks should be able to support the handoff interface without much difficulty.

The rest of the paper is organized as follows. Section 2 describes performance issues of TCP/IP packet processing on modern processors. Section 3 describes a connection handoff interface between the operating system and the NIC. Section 4 describes the experimental setup, and Section 5 presents results. Section 6 discusses related work, and Section 7 draws conclusions.

## 2. NETWORK STACK PERFORMANCE

Most layers of the network stack, including the IP, Ethernet, and device driver layers, 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 [2]. Rather, it is expensive main memory accesses that dominate packet processing time. Figure 1 illustrates the impact of such main memory accesses. The execution profiles of the network stack shown in the figure are collected using an instrumented FreeBSD 4.7 network stack running on a machine with a Gigabit Ethernet NIC and a single AMD Athlon XP 2600+ processor that runs at 2.1GHz and has 256KB of L2 cache. The profiles show the number of processor cycles, instructions, micro-operations, and data memory accesses per TCP packet in each layer of the network stack for several configurations denoted by A–F. A–E are generated by a microbenchmark program that sends maximum-sized (1460 byte) TCP segments at 100 Mb/s to another machine evenly across a number of connections. Zero-copy and Checksum offload indicate whether the system call layer performs data copy and whether the stack calculates checksums, respectively. Also, dummynet [21] is used to control packet delays. 0ms latency means that dummynet is not used. F is generated by SPECweb99 with 256 clients (connections). Each client sends both static and dynamic requests to the Flash web server [15] and maintains a fixed bandwidth of 400 Kb/s. Static HTTP response data is sent through zero-copy I/O while other data is copied between the user and kernel memory spaces. 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.

As configuration C in Figure 1 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, almost 50% of the micro-operations are data memory accesses. Typical integer applications have much less frequent memory accesses. For exam-

ple, 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.

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 10358 cycles to 4295 (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. This impact of data touching operations is well-known [2]. However, the figure also shows that main memory accesses to connection state, such as TCP control blocks, have similar impacts. 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 cache size. Consequently, accesses to such structures suffer from long main memory latencies, which are easily over 200 cycles on modern processors. C and D show that 1024 connections require 54% 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 12%. 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.

Finally, F shows that the network stack behavior is essentially the same for both the microbenchmark and web server except for the system call layer. The behavior of the system call layer is different simply because Flash and the microbenchmark program use different system calls to access the network. The TCP layer's profile is very similar to D, indicating that main memory access to connection data structures can be a performance issue for real servers that must handle a large number of connections.

In summary, it is crucial to reduce the frequent, expensive main memory accesses incurred by 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 10 Gb/s link (assuming that the per-packet processing requirements remain as they are for configuration C in Figure 1). 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 10 Gb/s link with 1024 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. Conventional latency hiding techniques, such as software prefetching, do not work well [5]. 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.

# 3. TCP ACCELERATION

As the processor/memory speed gap continues to grow, the memory latency bottleneck in TCP processing becomes increasingly worse. Offloading the network stack 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. 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:

**Computation resources.** As discussed previously, it can take upwards of 8 billion operations per second to saturate a full-duplex 10 Gb/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 rather than a programmable processor. This is not only more costly, but also greatly 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 greatly complicates the software architecture of the network subsystem [12]. 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.

## 3.1 Connection Handoff

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 interface 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 pro-

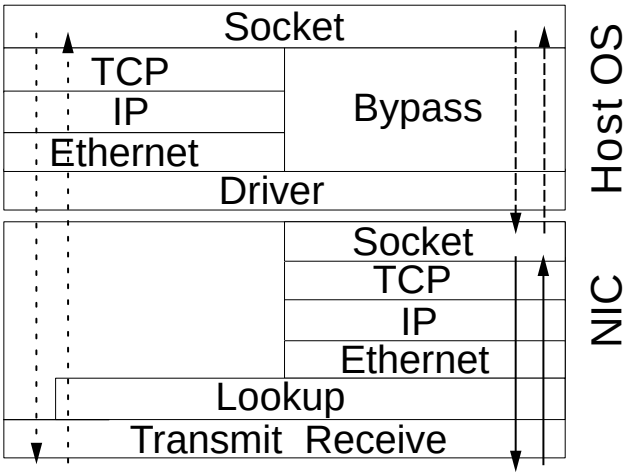



**Figure 2: Extended network stack for connection handoff.**

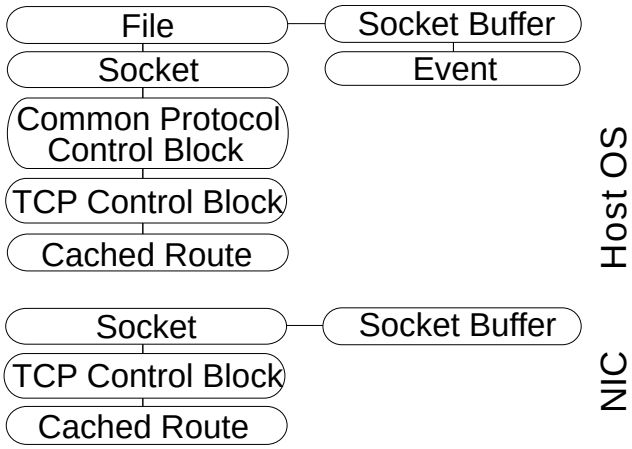



**Figure 3: Important data structures of a connection.**

cessing 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.

## 3.2 Architecture

Figure 2 shows the network stack modified to support connection handoff. The left half of the figure represents the traditional network stack, while the right half 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 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 2 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.

Even though the lookup layer of the NIC is shown in the figure, the system may choose not to employ it. Without the lookup layer, a received TCP packet flows through the Ethernet and IP layers, and connection lookup is performed in the TCP layer. If the packet does not belong to an offloaded connection, then it is transferred to the main memory, at which point it can either be completely reprocessed or can skip directly to the TCP layer.

Figure 3 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 implement 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. 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

- `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: Commands provided by the NIC.**

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 3, events are attached to the sockets and are independent of the layers below the socket layer.

## 3.3 Interface

The handoff interface resides between the bypass layer and the device driver shown in Figure 2 and 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 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 a main memory buffer that is pre-allocated for messages, and interrupts the host CPU. The operating system then processes the message.

Figure 4 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 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 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 5 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 3.2, 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

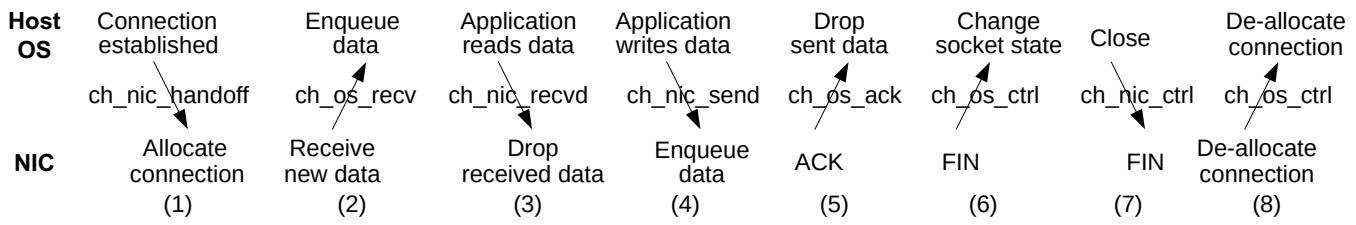



**Figure 6: Example sequence of events and commands exchanged between the OS and the NIC.**

- `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 5: Commands provided by the operating system.**

`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. 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. Most operating systems can cope with route changes transparently. So, 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. 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`.

## 3.4 Example

Figure 6 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 onto the receive socket buffer in the operating system using `ch_os_recv`. 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_recvd`. 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 stack, this changes the socket state to note 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. EXPERIMENTAL METHODOLOGY

## 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 3.3 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.

| | Configuration |
|---|---|
| CPU | Functional single issue 2GHz x86 CPU |
| | Instantaneous instruction fetch |
| L1 cache | 64KB data cache |
| | Line size: 64B, associativity: 2-way |
| | Hit latency: 3 cycles |
| L2 cache | Unified cache of size 256KB |
| | Line size: 64B, associativity: 16-way |
| | Hit latency: 17 cycles |
| | Next-line prefetch on a miss |
| DRAM | DDR266 SDRAM of size 2GB |
| | Access latency: 240 cycles |
| NIC | Functional 450MHz CPU |
| | 1Gb/s Ethernet wire |

**Table 1: Simulator configuration.**

Currently, the maximum TCP throughput over an offloaded connection is only about 110Mb/s (using 1460B TCP segments) on the 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. A simulator that models a controller similar to Tigon reveals that it needs to sustain about 450 million instructions per second to achieve full-duplex Gigabit bandwidth, whereas Tigon has a theoretical peak rate of only 176 million instructions per second. The actual instruction rate achieved on Tigon is much lower than the theoretical peak, as will be shown later.

## 4.2 Simulation

A full-system simulator, Simics [8], is used to to measure the impact of a faster NIC with more memory than the one used in the prototype. Simics is functional but allows external modules to enforce timing. For the purposes of this paper, it has been extended with a memory system simulator that models caches and DRAM and the NIC simulator mentioned above. The CPU core itself remains functional and is configured to execute one x86 instruction per cycle unless there are memory stalls. The MIPS core of the NIC simulator is set to execute 450 million instructions per second at all times. Also, it models 16MB of local memory, so the firmware is configured to handle up to 8192 connections. Except these differences in configuration, the simulator executes the same firmware and the host operating system as those used in the actual prototype. However, the local I/O interconnect is not modeled. The simulator configuration is summarized in Table 1. The cache configuration is similar to that of AMD Athlon XP 2600+. The DRAM simulator is cycle accurate and models both the memory controller and DRAM [20]. It also models contention for resources, latencies, and bandwidth.

# 5. EXPERIMENTAL RESULTS

## 5.1 Prototype

Figure 7 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 rep-

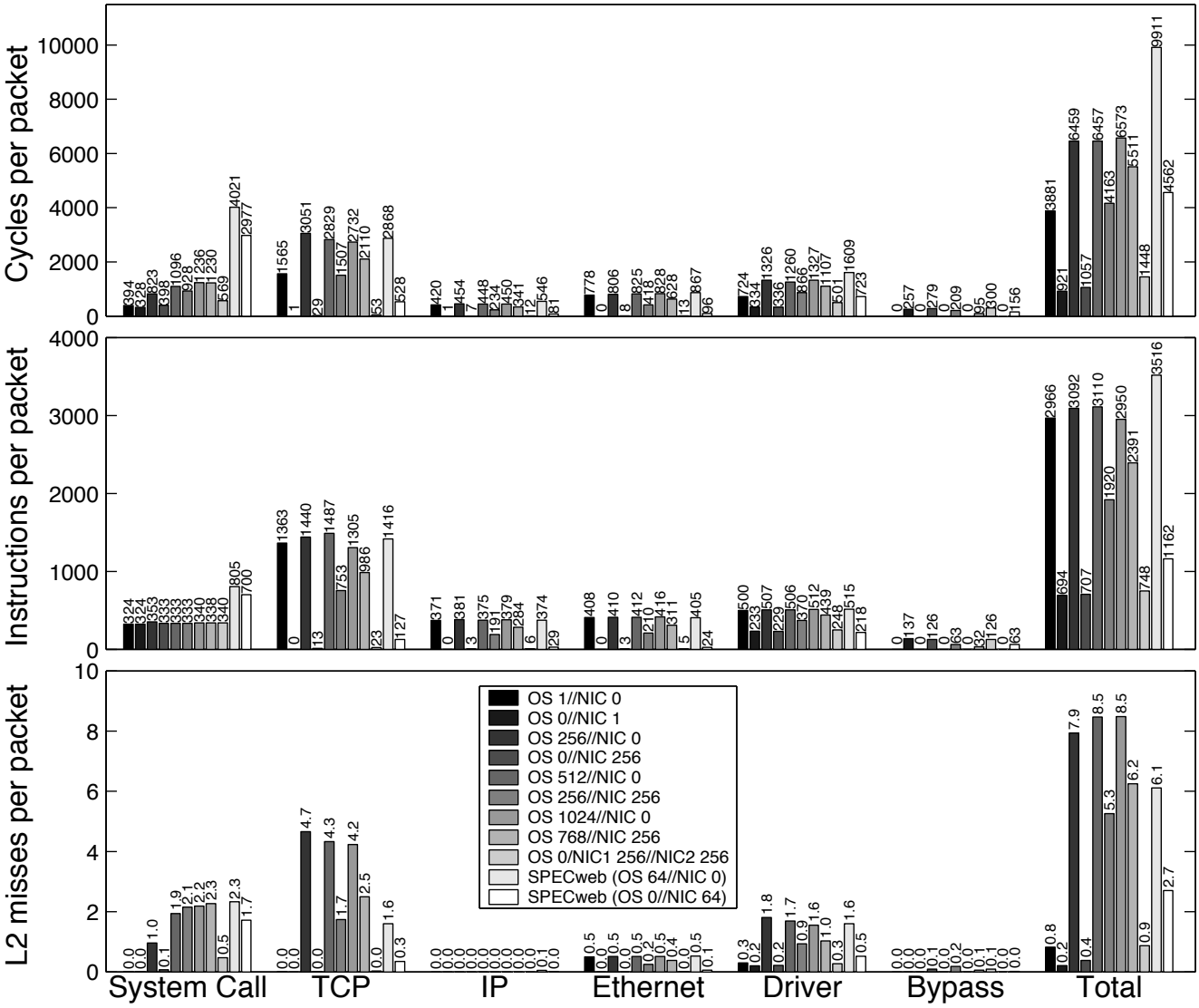



**Figure 7: Execution profiles of the network stack of the prototype.**

resenting 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 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 reduce 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

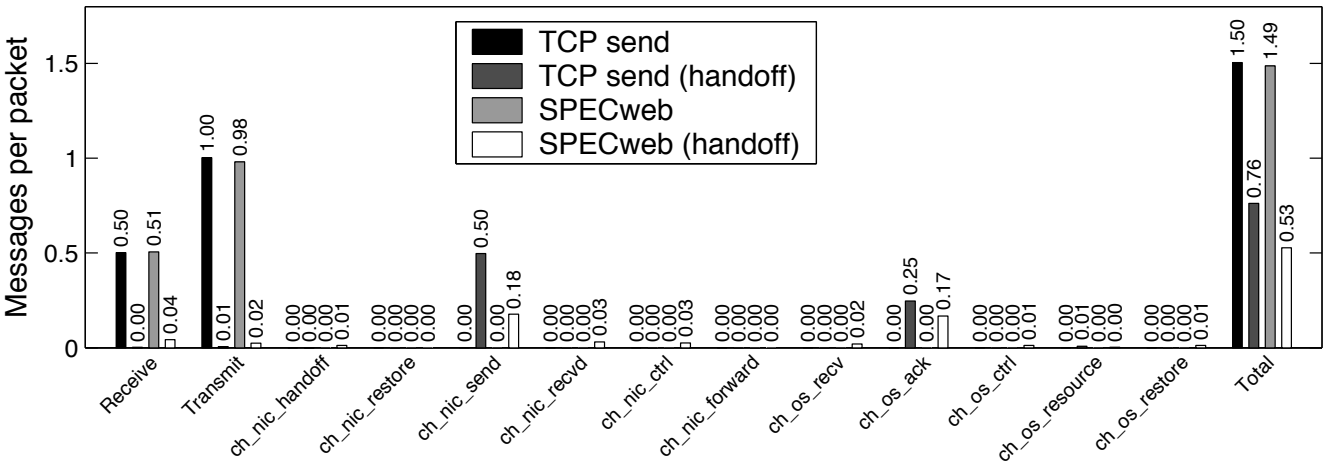



**Figure 8: Messages exchanged across the local I/O interconnect.**

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 8 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.

In Figure 8, 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 [4]. 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. Thus, they are not analyzed in this paper.

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

**Table 2: Profile of the firmware.**

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 2 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.

## 5.2 Simulation

The prototype shows that handoff can substantially reduce the number of processor cycles, instructions, and cache misses required for packet processing on the host CPU. However, the NIC is currently the bottleneck in the system because the outdated Tigon controller lacks the necessary compute power.

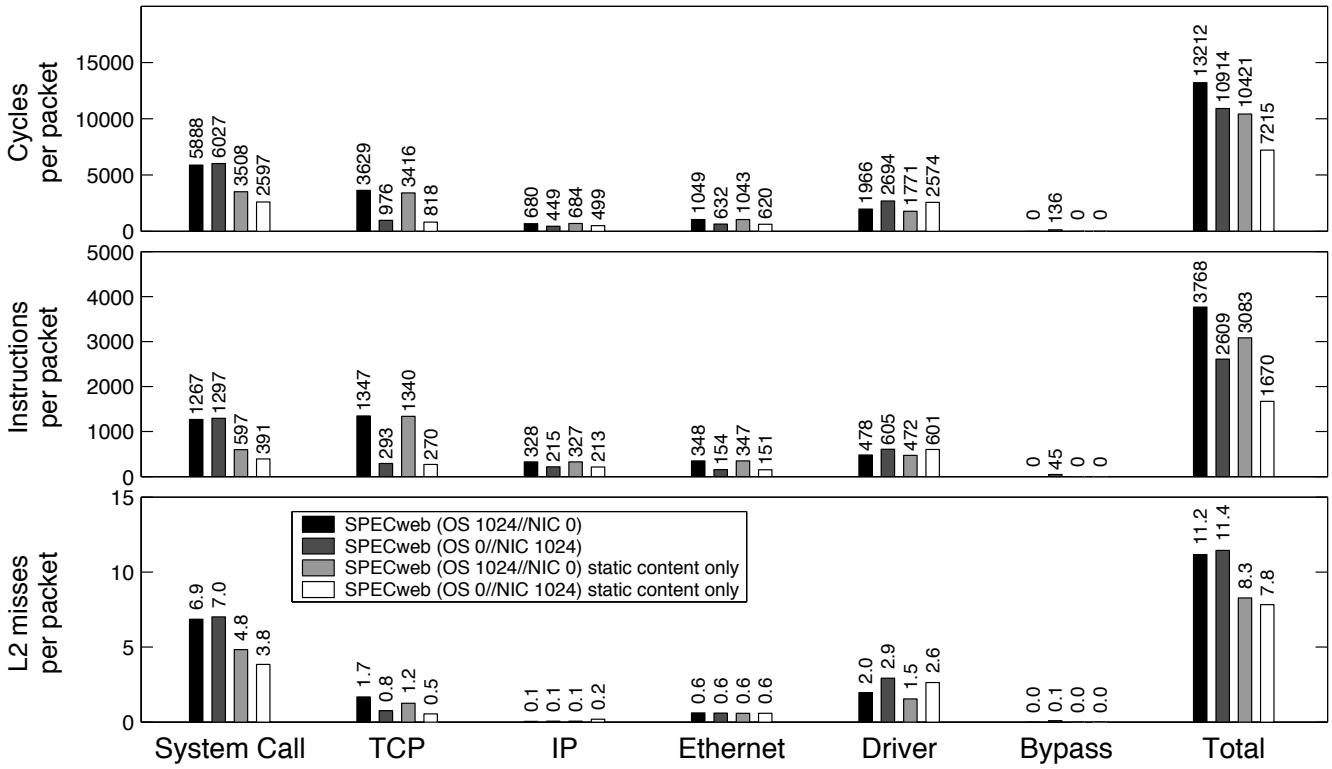



**Figure 9: Execution profiles of the network stack of the simulated machine.**

Figure 9 shows the profiles of the host network stack running on the simulated machine with a faster NIC that can sustain 450 million instructions per second. The profiles were taken on the server while running SPECweb99 benchmark with 1024 connections. The caches are warmed up during the first 400000 packets, and the profiles are collected during the next 200000 packets. The left two bars in each group show the profiles when the default SPECweb configuration is used. By default, there are both dynamic and static content requests. Without handoff, the web server achieves 192Mb/s of HTTP throughput. With handoff, all connections are offloaded to the NIC. As expected, handoff reduces the number of instructions, cache misses, and cycles in the TCP layer. The SPECweb99 workload consists of many short lived connections as well as CGI processing for generating dynamic content. Dynamic content in particular leads to data copies by the system call layer, which explains a high number of cache misses. The system spends about 17% fewer cycles per packet with handoff. Consequently, HTTP throughput improves by about 15% to 222Mb/s. The right two bars in each group show the profiles when only static content requests are used. Since generating static content responses requires much less work than dynamic content responses, the server's throughput improves, and the server now spends more time executing the network stack. Without handoff, HTTP throughput is 307Mb/s. Handoff now reduces cycles by about 31% and improves throughput by 27% to 391Mb/s, close to the maximum 410Mb/s achievable using 1024 connections at 400Kb/s per connection. This result indicates that by using a more capable NIC than the prototype, real machines should be able to see improved performance with handoff.

## 6. RELATED WORK

Makineni and Iyer analyzed execution profiles of a real machine for various TCP workloads. They show that cache misses can significantly affect the network stack performance [9]. However, they ignore the impact of a large number of connections and communication latencies that servers experience and consequently do not investigate their negative impact on performance. Nahum *et al.* reported that caches have a big impact on packet processing time spent in the network stack [13]. They show that larger, higher associativity caches reduce processing time, and that the protocol processing time would scale with the processor performance as long as the caches provide the data necessary for protocol processing. However, as the number of connections increases, the cache size would have to grow proportionally to store the connections.

Several studies evaluated the impact of moving TCP processing to a dedicated processor [18, 19, 23]. Rangarajan *et al.* developed TCP servers [18]. This design, based on the Split-OS concept proposed by the same authors [1], splits TCP and the rest of the operating system and lets a dedicated processor or a dedicated system execute TCP. They report various performance gains for web workloads. This approach amounts to full offload since TCP and the rest of processing are partitioned into two components. So, it suffers from the same drawbacks outlined in Section 3. Others have proposed TCP onloading (as opposed to offload to the NIC), in which a general-purpose processor in a multi-processor system is dedicated for TCP processing [19]. They report various potential performance gains and emphasize that the use of a dedicated processor can eliminate interrupts by simply polling the NIC. Despite its name, this technique is essentially full offload to another processor in the system, rather than the NIC. Westrelin *et al.* also used a multiprocessor system in which one processor is dedicated to executing TCP, like TCP onloading, and show a significant improvement in microbenchmark performance [23].

Freimuth *et al.* recently showed that full offload reduces traffic on the local interconnect [4]. 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. The results in Section 5 show that handoff also reduces local interconnect traffic.

Hoskote *et al.* built a prototype programmable network controller that can receive and send TCP packets at 10Gb/s rate [6]. This result is encouraging in that specialized programmable architectures can process TCP efficiently. However, their protocol implementation is incomplete, and the testbed does not include the host operating system. There are several commercial NICs that provide the operating system with TCP offload. The hardware typically includes one or more specialized programmable processors for executing protocols and off-chip memory for storing data including connections and packets. These NICs implement full TCP offload, and thus suffer from the limitations outlined in Section 3.

Microsoft recently proposed to implement a driver API for offload NICs in the next generation Windows operating system [10]. This API, a part of the Chimney Offload architecture, 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 very similar to those described in Section 3. The design presented in this paper is developed independently, and the performance analysis given in this paper is new. The Sun Microsystems Solaris 10 operating system is expected to implement a driver API for offload in the future as well.

TCP connection handoff techniques have been used in various systems. Pai *et al.* used a connection handoff mechanism in their request distribution system for a cluster of web servers [14]. In this system, a client first establishes a TCP connection to the front-end request distributor, and then the distributor hands off the connection to an appropriate back-end server. 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 [17]. Handoff can also be used to provide fault tolerance to long-lived connections by replicating connection states among redundant servers [22].

# 7. CONCLUSION

Main memory access, not computation, is the major bottleneck in TCP processing. Several techniques, such as zero-copy I/O and checksum offload, are widely utilized to eliminate accesses to packet data. However, main memory accesses to connection data structures are now becoming a significant performance bottleneck in TCP packet processing. When a large number of connections saturate the processor caches, accesses to those connection data structures incur the full main memory latency. Offloading TCP processing to the NIC can relieve the memory pressure on the host CPU, while the NIC can access connection data structures within its fast local memory to quickly process packets. However, limited computation and memory resources on the NIC, as well as software architecture issues involved with running TCP on the NIC, pose serious challenges. The limited computation and memory resources on the NIC would slow the entire system down if the NIC is required to handle too many connections, while global resource management issues like port allocation and IP routing complicate the implementation of TCP offloading. Thus, the typical full TCP offload in which all TCP processing occurs on the NIC is not a viable solution.

The connection handoff interface presented in this paper aims to achieve the benefits of offload while avoiding its drawbacks. Using this interface, the operating system can handoff established connections to the NIC and also restore them to the operating system, as necessary. Thus, the operating system retains full control over the work done on the NIC and can avoid the performance issues of full TCP offload. Handoff also simplifies the implementation because established connections already have correct port numbers and IP routes. A prototype built using a modified FreeBSD operating system is able to reduce the total cycles spent in the host network stack by 16–84% per packet by reducing both the number of instructions executed and the number of cache misses. Furthermore, the simulated throughput of SPECweb99 increases by 15% with handoff. This improvement, despite the prevalence of short packets and short-lived connections, indicates that handoff can improve overall system performance. The combination of a prototype implementation and a simulation study validates that while the outdated Tigon is unable to support line rate with connection handoff firmware, it would be practical to build a network interface that is able to do so in modern process technologies. Thus, connection handoff can be used to exploit the network interface to accelerate packet processing, without requiring unreasonable resources on the network interface.

## Acknowledgments

The authors thank Alan L. Cox for his interest and comments on the paper. This work is supported in part by a donation from Advanced Micro Devices and by the National Science Foundation under Grant No. CCR-0209174.

# 8. REFERENCES

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

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

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] S. Makineni and R. 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, Feb. 2004.

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

[11] 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.

[12] 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.

[13] E. Nahum, D. Yates, J. Kurose, and D. 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.

[14] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. 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.

[15] 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.

[16] 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.

[17] A. E. Papathanasiou and E. V. Hensbergen. KNITS: Switch-based Connection Hand-off. In *Proceedings of IEEE INFOCOM '02*, volume 1, pages 332–341, 2002.

[18] 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.

[19] 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.

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

[21] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM SIGCOMM Computer Communication Review*, 27(1):31–41, 1997.

[22] A. Snoeren, D. G. Andersen, and H. Balakrishnan. Fine-Grained Failover Using Connection Migration. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, Mar. 2001.

[23] 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.