Server Network Scalability and TCP Offload

Doug Freimuth, Elbert Hu, Jason LaVoie, Ronald Mraz,
Erich Nahum, Prashant Pradhan, John Tracey
IBM T. J. Watson Research Center
Hawthorne, NY, 10532
{dmfreim,elbert,lavoie,mraz,nahum,ppradhan,traceyj}@us.ibm.com

Abstract

Server network performance is increasingly dominated by poorly scaling operations such as I/O bus crossings, cache misses and interrupts. Their overhead prevents performance from scaling even with increased CPU, link or I/O bus bandwidths. These operations can be reduced by redesigning the host/adapter interface to exploit additional processing on the adapter. Offloading processing to the adapter is beneficial not only because it allows more cycles to be applied but also of the changes it enables in the host/adapter interface. As opposed to other approaches such as RDMA, TCP offload provides benefits without requiring changes to either the transport protocol or API.

We have designed a new host/adapter interface that exploits offloaded processing to reduce poorly scaling operations. We have implemented a prototype of the design including both host and adapter software components. Experimental evaluation with simple network benchmarks indicates our design significantly reduces I/O bus crossings and holds promise to reduce other poorly scaling operations as well.

1 Introduction

Server network throughput is not scaling with CPU speeds. Various studies have reported CPU scaling factors of 43% [23], 60% [15], and 33% to 68% [22] which fall short of an ideal scaling of 100%. In this paper, we show that even increasing CPU speeds and link and bus bandwidths does not generate a commensurate increase in server network throughput. This lack of scalability points to an increasing tendency for server network throughput to become the key bottleneck limiting system performance. It motivates the need for an alternative design with better scalability.

Server network scalability is limited by operations heavily used in current designs that themselves do not scale well, most notably bus crossings, cache misses and interrupts. Any significant improvement in scalability must reduce these operations. Given that the problem is one of scalability and not simply performance, it will not be solved by faster processors. Faster processors merely expend more cycles on poorly scaling operations.

Research in server network performance over the years has yielded significant improvements including: integrated checksum and copy, checksum offload, copy avoidance, interrupt coalescing, fast path protocol processing, efficient state lookup, efficient timer management and segmentation offload, a.k.a. large send. Another technique, full TCP offload, has been pursued for many years. Work on offload has generated both promising and less than compelling results [1, 38, 40, 42]. Good performance data and analysis on offload is scarce.

Many improvements in server scalability were described more than fifteen years ago by Clark et al. [9]. The authors demonstrated that the overhead incurred by network protocol processing, per se, is small compared to both per-byte (memory access) costs and operating system overhead, such as buffer and timer management. This motivated work to reduce or eliminate data touching operations, such as copies, and to improve the efficiency of operating system services heavily used by the network stack. Later work [19] showed that overhead of non-data touching operations is, in fact, significant for real workloads, which tend to feature a preponderance of small messages. Today, per-byte overhead has been greatly reduced through checksum offload and zero-copy send. This leaves per-packet overhead, operating system services and zero-copy receive as the main remaining areas for further improvement.

Nearly all of the enhancements described by Clark et al. have seen widespread adoption. The one notable exception is an efficient network interface. This is a network adapter with a fast general-purpose processor that provides a much more efficient interface to the network than the current frame-based interface devised decades ago. In this paper, we describe an effort to develop a much more efficient network interface and to make this enhancement a reality as well.

Our work is pursued in the context of TCP for three reasons: 1) TCP's enormous installed base, 2) the methodology employed with TCP will transfer to other protocols, and 3) the expectation that key new architectural features, such as zero copy receive, will ultimately demonstrate their viability with TCP.

The work described here is part of a larger effort to improve server network scalability. We began by analyzing server network performance and recognizing, as others have, a significant scalability problem. Next, we identified specific operations to be the cause, specifically: bus crossings, cache misses, and interrupts. We formulated a design that reduces the impact of these operations. This design exploits additional processing at the network adapter, i.e. offload, to improve the efficiency of the host/adapter interface which is our primary focus. We have implemented a prototype of the new design which consists of host and adapter software components and have analyzed the impact of the new design on bus crossings. Our findings indicate that offload can substantially decrease bus crossings and holds promise to reduce other scalability limiting operations such as cache misses. Ultimately, we intend to evaluate the design in a cycle-accurate hardware simulator. This will allow us to comprehensively quantify the impact of design alternatives on cache misses, interrupts and overall performance over several generations of hardware.

This paper is organized as follows. Section 2 provides motivation and background. Section 3 presents our design, and the current prototype implementation is described in Section 4. Section 5 presents our experimental infrastructure and results. Section 6 surveys and contrasts related work, and Section 7 summarizes our contributions and plans for future work.

2 Motivation and Background

To provide the proper motivation and background for our work, we first describe the current best practices of techniques and optimizations for network server performance. Using industry standard benchmarks we then show that, despite these practices, servers are still not scaling with CPU speeds via several benchmarks. Since TCP offload has been a controversial topic in the research community, we review the critiques of offload, providing counterarguments to each point. How TCP offload addresses these scaling issues is described in more detail in Section 3.

2.1 Current Best Practices

Current high-performance servers have adopted many techniques to maximize performance. We provide a brief overview of them here.

Sendfile with zero copy. Most operating systems have a sendfile or transmitfile operation that allows sending a file over a socket without copying the contents of the file into user space. This can have substantial performance benefits [30]. However, the benefits are limited to send-side processing; it does not affect receive-side processing. In addition, it requires the server application to maintain its data in the kernel, which may not be feasible for systems such as application servers, which generate content dynamically.

Checksum offload. Researchers have shown that calculating the IP checksum over the body of the data can be expensive [19]. Most high-performance adapters have the ability to perform the IP checksum over both the contents of the data and the TCP/IP headers. This removes an expensive data-touching operation on both send and receive. However, adapter-level checksums will not catch errors introduced by transferring data over the I/O bus, which has led some to advocate caution with checksum offload [41].

Interrupt coalescing. Researchers have shown that interrupts are costly, and generating an interrupt for each packet arrival can severely throttle a system [28]. In response, adapter vendors have enabled the ability to delay interrupts by a certain amount of time or number of packets in an effort to batch packets per interrupt and amortize the costs [14]. While effective, it can be difficult to determine the proper trigger thresholds for firing interrupts, and large amounts of batching may cause unacceptable latency for an individual connection.

Large send/segmentation offload. TCP/IP implementers have long known that larger MTU sizes provide greater efficiency, both in terms of network utilization (fewer headers per byte transferred) and in terms of host CPU utilization (fewer per-packet operations incurred per byte sent or received). Unfortunately, larger MTU sizes are not usually available due to Ethernet's 1516 byte frame size. Gigabit Ethernet provides "jumbo frames" of 9 KB, but these are only useful in specialized local environments and cannot be preserved across the wide-area Internet. As an approximation, certain operating systems, such as AIX and Linux, provide large send or TCP segmentation offload (TSO) where the TCP/IP stack interacts with the network device as if it had a large MTU size. The device in turn segments the larger buffers into 1516-byte Ethernet frames and adjusts the TCP sequence numbers and checksums accordingly. However, this technique is also limited to send-side processing. In addition, as we demonstrate in Section 2.2, the technique is limited by the way TCP performs congestion control.

Efficient connection management. Early networked servers did not handle large numbers of TCP connections efficiently, for example by using a linear linked-list to manage state [26]. This led to operating systems using hash table based approaches [24] and separating table entries in the TIME_WAIT state [2].

Asynchronous interfaces. To maximize concurrency, high-performance servers use asynchronous interfaces as not to block on long-latency operations [33]. Server applications interact using an event notification interface such as `select()` or `poll()`, which in turn can have performance implications [5]. Unfortunately,

Machine	BIOS Release Date	Clock Speed (MHz)	Cycle Time (ns)	Bus Width (bits)	Bus Speed (MHz)	L1 Size (KB)	L2 Size (KB)	E1000 NICs (num)
Workstation-Class								
500 MHz P3	Jul 2000	500	2.000	32	33	32	512	1
933 MHz P3	Mar 2001	933	1.070	32	33	32	256	1
1.7 GHz P4	Sep 2003	1700	0.590	64	66	8	256	2
Server-Class								
450 MHz P2-Xeon	Jan 2000	450	2.200	64	33	32	2048	2
1.6 GHz P4-Xeon	Oct 2001	1600	0.625	64	100	8	256	3
3.2 GHz P4-Xeon	May 2004	3200	0.290	64	133	8	512	4

Table 1: Properties for Multiple Generations of Machines

these interfaces are typically only for network I/O and not file I/O, so they are not as general as they could be.

In-kernel implementations. Context switches, data copies, and system calls can be avoided altogether by implementing the server completely in kernel space [17, 18]. While this provides the best performance, in-kernel implementations are difficult to implement and maintain, and the approach is hard to generalize across multiple applications.

RDMA. Others have also noticed these scaling problems, particularly with respect to data copying, and have offered RDMA as a solution. Interest in RDMA and Infiniband [4] is growing in the local-area case, such as in storage networks or cluster-based supercomputing. However, RDMA requires modifications to both sides of a conversation, whereas Offload can be deployed incrementally on the server side only. Our interest is in supporting existing applications in an inter-operable way, which precludes using RDMA.

While effective, these optimizations are limited in that they do not address the full range of scenarios seen by a server. The main restrictions are: 1) that they do not apply to the receive side, 2) they are not fully asynchronous in the way they interact with the operating system, 3) they do not minimize the interaction with the network interface, or 4) they are not inter-operable. Additionally, many techniques do not address what we believe to be the fundamental performance issue, which is overall server scalability.

2.2 Server Scalability

The recent arrival of 10 gigabit Ethernet and the promise of 40 and 100 gigabit Ethernet in the near future show that raw network bandwidth is scaling at least as quickly as CPU speed. However, it is well-known that memory speeds are not scaling as quickly as CPU speed increases [16]. As a consequence of this and other factors, researchers have observed that the performance of host TCP/IP implementations is not scaling at the same rate as CPU speeds in spite of raw network bandwidth increases.

To quantify how performance scales over time, we ran a number of experiments using several generations of machines, described in detail in Table 1. We break the machines into 2 classes: desk-side workstations and rack-mounted servers with aggressive memory systems and I/O busses. The workstations include a a 500 MHz Intel Pentium 3, a 933 MHz Intel Pentium 3, and a a 1.7 GHz Pentium 4. The servers include a 450 MHz Pentium II-Xeon, a 1.6 GHz P4 Xeon, and a 3.2 GHz P4 Xeon. In addition, each of the P4-Xeon servers have 1 MB L3 caches. Each machine runs Linux 2.6.9 and has a number of Intel E1000 MT server gigabit Ethernet adapters, connected via a Dell gigabit switch. Load is generated by five 3.2 GHz P4-Xeons acting as clients, each using an E1000 client gigabit adapter and running Linux 2.6.5. We chose the E1000 MT adapters for the servers since these have been shown to be one of the highest-performing conventional adapters on the market [32], and we did not have access to a 10 gigabit adapter.

We measured the time to access various locations in the memory hierarchy for these machines, including from the L1 and L2 caches, main memory, and the memory-mapped I/O registers on the E1000. Memory hierarchy times were measured using LMBench [25]. To measure the device I/O register times, we added some modifications to the initialization routine of the Linux 2.6.9 E1000 device driver code. Table 2 presents the results. Note that while L1 and L2 access times remain relatively consistent in terms of processor cycles, the time to access main memory and the device registers is increasing over time. If access times were improving at the same rate as CPU speeds, the number of clock cycles would remain constant.

To see how actual server performance is scaling over time, we ran the static portion of SPECweb99 [12] us-

Machine	L1 Cache Hit		L2 Cache Hit		Main Memory		I/O Register Read		I/O Register Write	
	Time (ns)	Clock Cycles	Time (ns)	Clock Cycles	Time (ns)	Clock Cycles	Time (ns)	Clock Cycles	Time (ns)	Clock Cycles
Workstation-Class										
500 MHz P3	6	3	44	22	162	80	600	300	300	150
933 MHz P3	3.25	3	7.5	7	173	161	700	654	400	373
1.7 GHz P4	1.2	2	10.9	18	190	323	800	1355	100	169
Server-Class										
450 MHz P2-Xeon	6.75	3	38.3	17	207	93	800	363	200	90
1.6 GHz Xeon	1.37	2	11.57	18	197	315	900	1440	300	480
3.2 GHz Xeon	0.6	2	5.8	18	111	376	500	1724	200	668

Table 2: Memory Access Times for Multiple Generations of Machines

ing a recent version of Flash [33, 37]. In these experiments, Flash exploits all the available performance optimizations on Linux, including `sendfile()` with zero copy, TSO, and checksum offload on the E1000. Table 3 shows the results. Observe that server performance is not scaling with CPU speed, even though this is a heavily optimized server making use of all current best practices. This is not because of limitations in the network bandwidth; for example, the 3.2 GHz Xeon-based machine has 4 gigabit interfaces and multiple 10 gigabit PCI-X busses.

2.3 Offload: Critiques and Responses

In this paper, we study TCP offload as a solution to the scalability problem. However, TCP offload has been hotly debated by the research community, perhaps best exemplified by Mogul's paper, "TCP offload is a dumb idea whose time has come" [27]. That paper effectively summarizes the criticisms of TCP offload, and so, we use the structure of that paper to offer our counterarguments here.

Limited processing requirements. One argument is that Clark et al. [9] show that the main issue in TCP performance is implementation, not the TCP protocol itself, and a major factor is data movement; thus Offload does not address the real problem. We point out that Offload does not simply mean TCP header processing; it includes the entire TCP/IP stack, including poorly-scaling, performance-critical components such as data movement, bus crossings, interrupts, and device interaction. Offload provides an improved interface to the adapter that reduces the use of these scalability-limiting operations.

Moore's Law: Moore's Law states that CPU speeds are doubling every 18 months, and thus one claim is that Offload cannot compete with general-purpose CPUs. Historically, chips used by adapter vendors have not increased at the same rate as general-purpose CPUs due to the economies of scale. However, offload can use commodity CPUs with software implementations, which we believe is the proper approach. In addition, speed needs only to be matched with the interface (e.g., 10 gigabit Ethernet), and we argue proper design reduces the code path relative to the non-offloaded case (e.g. with fewer memory copies). Sarkar et al. [38] and Ang [1] show that when the NIC CPU is under-provisioned with respect to the host CPU, performance can actually degrade. Clearly the NIC processing capacity must be sized properly. Finally, increasing CPU speeds does not address the scalability issue, which is what we focus on here.

Efficient host interface: Early critiques are that TCP Offload Engines (TOE) vendors recreated ”TCP over a bus”. Development of an elegant and efficient host/adapter interface for offload is a fundamental research challenge, one we are addressing in this paper.

Bad buffer management: Unless Offload engines understand higher-level protocols, there is still an application-layer header copy. While true, copying of application headers is not as performance-critical as copying application data. One complication is the application combining its own headers on the same connection with its data. This can only be solved by changing the application, which is already proposed in RDMA extensions for NFS and iSCSI [7, 8].

Connection management overhead: Unlike conventional NICs, offload adapters must maintain per-connection state. Opponents argue that offload cannot handle large numbers of connections, but Web server workloads have forced host TCP stacks to discover techniques to efficiently manage 10,000's of connections. These techniques are equally applicable for an interface-based implementation.

Resource management overhead: Critics argue that tracking resource management is ”more difficult” for offload. We do not believe this is the case. It is straight-

Machine	Throughput (ops/sec)	Requested Connections	Conforming Connections	Scale (achieved)	Scale (ideal)	Ratio (%)
Workstation-Class						
500 MHz P3	1231	375	375	1.00	1.00	100
933 MHz P3	1318	400	399	1.06	1.87	56
1.7 GHz P4	3457	1200	1169	3.20	3.40	94
Server-Class						
450 MHz P2-Xeon	2230	700	699	1.00	1.00	100
1.6 GHz P4-Xeon	8893	2800	2792	4.00	3.56	112
3.2 GHz P4-Xeon	11614	2500	3490	5.00	7.10	71

Table 3: SPECWeb99 Performance Scalability over Multiple Generations of Machines

forward to extend the notion of resource management across the interface without making the adapter aware of every process as we will show in Sections 3 and 4.

Event management: The claim is that offload does not address managing the large numbers of events that occur in high-volume servers. It is true that offload, per se, does not address *application visible* events, which are better addressed by the API. However, offload can shield *the host operating system* from spurious unnecessary *adapter events*, such as TCP acknowledgments or window advertisements. In addition, it allows batching of other events to amortize the cost of interrupts and bus crossings.

Partial offload is sufficiently effective: Partial offload approaches include checksum offload and large send (or TCP Segmentation Offload), as discussed in Section 2.1. While useful, they have limited value and do not fully solve the scalability problem as was shown in Section 2.2. Other arguments include that checksum offload actually masks errors to the host [41]. In contrast, offload allows larger batching and the opportunity to perform more rigorous error checking (by including the CRC in the data descriptors).

Maintainability: Opponents argue that offload-based approaches are more difficult to update and maintain in the presence of security and bug patches. While this is true of an ASIC-based approach, it is not true of a software-based approach using general-purpose hardware.

Quality assurance: The argument here is that offload is harder to test to determine bugs. However, testing tools such as TBIT [31] and ANVL [11] allow remote testing of the offload interface. In addition, software based approaches based on open-source TCP implementations such as Linux or FreeBSD facilitate both maintainability and quality assurance.

System management interface: Opponents claim that offload adapters cannot have the same management interface as the host OS. This is incorrect: one example is SNMP. It is trivial to extend this to an offload adapter.

Concerns about NIC vendors: Third-party vendors may go out of business and strand the customer. This has nothing to do with offload; it is true of any I/O device: disk, NIC, or graphics card. Economic incentives seem to address customer needs. In addition, one of the largest NIC vendors is Intel.

3 System Design

In this Section we describe our Offload design and how it addresses scalability.

3.1 How Offload Addresses Scalability

A higher-level interface. Offload allows the host operating system to interact with the device at a higher level of abstraction. Rather than simply queuing MTU-sized packets for transmission or reception, the host issues commands at the transport layer (e.g., `connect()`, `accept()`, `send()`, `close()`). This allows the adapter to shield the host from transport layer events (and their attendant interrupt costs) that may be of no interest to the host, such as arrivals of TCP acknowledgments or window updates. Instead, the host is only notified of meaningful events. Examples include a completed connection establishment or termination (rather than every packet arrival for the 3-way handshake or 4-way tear-down) or application-level data units. Sufficient intelligence on the adapter can determine the appropriate time to transfer data to the host, either through knowledge of standardized higher-level protocols (such as HTTP or NFS) or through a programmable interface that can provide an application signature (i.e., an application-level equivalent to a packet filter). By interacting at this higher level of abstraction, the host will transfer less data over the bus and incur fewer interrupts and device register accesses.

Ability to move data in larger sizes. As described in Section 2.1, the ability to use large MTUs has a significant impact on performance for both sending and re-

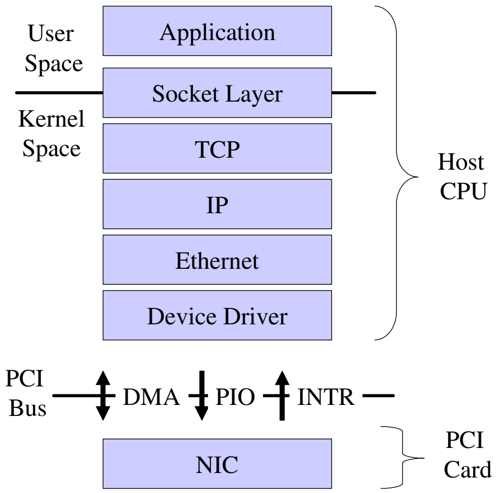

Figure 1: Conventional Protocol Stack

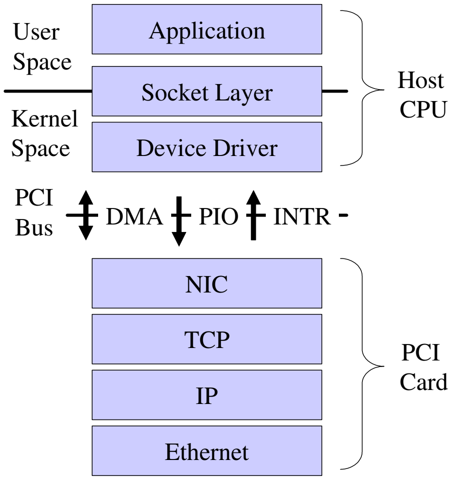

Figure 2: Offload Architecture

ceiving data. Large send/TSO only approximates this optimization, and only for the send side. In contrast, offload allows the host to send and receive data in large chunks unaffected by the underlying MTU size. This reduces use of poorly scaling components by making more efficient use of the I/O bus. Utilization of the I/O bus is not only affected by the data sent over it, but also by the DMA descriptors required to describe that data; offload reduces both. In addition, data that is typically DMA'ed over the I/O bus in the conventional case is not transferred here, for example TCP/IP and Ethernet headers.

Improving memory reference behavior. We believe offload will not only increase available cycles to the application but improve application memory reference behavior. By reducing cache and TLB pollution, cache hit rates and CPI will improve, increasing application performance.

3.2 Current Adapter Designs

Perhaps the simplest way to understand an architecture that offloads all TCP/IP processing is to outline the ways in which offload differs from conventional adapters in the way it interacts with the OS. Figure 1 illustrates a conventional protocol architecture in an operating system. Operating systems tend to communicate with conventional adapters only in terms of data transfer by providing them with two queues of buffers. One queue is made up of ready-made packets for transmission; the other is a queue of empty buffers to use for packet reception. Each queue of buffers is identified, in turn, by a descriptor table that describes the size and location of each buffer in the queue. Buffers are typically described in physical memory and must be pinned to ensure that they are accessible to the card, i.e., so that they are not paged out. The adapter provides a memory-mapped I/O interface for telling the adapter where the descriptor tables are located in physical memory, and provides an interface for some control information, such as what interrupt number to raise when a packet arrives. Communication between the host CPU and the adapter tends to be in one of three forms, as is shown in Figure 1: DMA's of buffers and descriptors to and from the adapter; reads and writes of control information to and from the adapter, and interrupts generated by the adapter.

3.3 Offloaded Adapter Design

An architecture that seeks to offload the full TCP/IP stack has both similarities and differences in the way it interacts with the adapter. Figure 2 illustrates our offload architecture. As in the conventional scenario, queues of buffers and descriptor tables are passed between the host CPU and the adapter, and DMA's, reads, writes and interrupts are used to communicate. In the offload architecture, however, the host and the adapter communicate using a higher level of abstraction. Buffers have more explicit data structures imposed on them that indicate both control and data interfaces. As with a conventional adapter, passed buffers must be expressed as physical addresses and must be in pinned memory. The control interface allows for the host to command the adapter (e.g., what port numbers to listen on) and for the adapter to instruct the host (e.g., to notify the host of the arrival of a new connection). The control interface is invoked, for example, by conventional socket functions that control connections: `socket()`, `bind()`, `listen()`, `connect()`, `accept()`, `setsock-`

opt(), etc. The data interface provides a way to transfer data on established connections for both sending and receiving and is invoked by socket functions such as send(), sendto(), write(), writev(), read(), readv(), etc. Even the data interface is at a higher layer of abstraction, since the passed buffers consist of application-specific data rather than fully-formed Ethernet frames with TCP/IP headers attached. In addition, these buffers need to identify which connection that the data is for. Buffers containing data can be in units much larger than the packet MTU size. While conceptually they could be of any size, in practice they are unlikely to be larger than a VM page size.

As with a conventional adapter, the interface to the offload adapter need not be synchronous. The host OS can queue requests to the adapter, continue doing other processing, and then receive a notification (perhaps in the form of an interrupt) that the operation is complete. The host can implement synchronous socket operations by using the asynchronous interface and then block the application until the results are returned from the adapter. We believe asynchronous operation is key in order to ameliorate and amortize fixed overheads. Asynchrony allows larger-scale batching and enables other optimizations such as polling-based approaches to servers [3, 28].

The offload interface allows supporting conventional user-level APIs, such as the socket interface, as well as newer APIs that allow more direct access to user memory such as DAFS, SDP, and RDMA. In addition, offload allows performing *zero-copy* sends and receives *without changes to the socket API*. The term zero-copy refers to the elimination of memory-to-memory copies by the host. Even in the zero-copy case, data is still transferred across the I/O bus by the adapter via DMA.

For example, in the case of a send using a conventional adapter, the host typically copies the data from user space into to a pinned kernel buffer, which is then queued to the adapter for transmission. With an intelligent adapter, the host can block the user application and pin its buffers, then invoke the adapter to DMA the data directly from the user application buffer. This is similar to previous “single-copy” approaches [13, 20], except that the transfer across the bus is done by the adapter DMA and not via an explicit copy by the host CPU.

Observe from Figure 2 that the interaction between the host and the adapter now occurs between the socket and TCP layers. A naive implementation may make unnecessary transfers across the PCI bus for achieving socket functionality. For example, accept() would now cause a bus crossing in addition to a kernel crossing, as could setsockopt() for actions such as changing the send or receive buffer sizes or the Nagle algorithm. However, each of these costs can be amortized via batching multiple requests into a single request that crosses the bus. For example, multiple arrived connections can be aggregated into a single accept() crossing which then translates into multiple accept() system calls. On the other hand, certain events that would generate bus crossings with a conventional adapter might not do so with a offload adapter, such as ACK processing and generation. The relative weight of these advantages and disadvantages depends on the implementation and workload of the application using the adapter.

4 System Implementation

To evaluate our design and the impact of design decisions, we implemented a software prototype. Our decision to implement the prototype purely in software, rather than building or modifying actual adapter hardware, was motivated by several factors. Since our goal is to study not just performance, but scalability, we ultimately intend to model different hardware characteristics, for both the host and adapter, using a cycle accurate hardware simulator. Limiting our analysis to only currently available hardware would hinder our evaluation for future hardware generations. Ultimately, we envision an adapter with a general purpose processor, in addition to specialized hardware to accelerate specific operations such as checksum calculation. Our prototype software is intended to serve as a reference implementation for a production adapter.

Our prototype is composed of three main components:

- OSLayer, an operating system layer that provides the socket interface to applications and maps it to the descriptor interface shared with the adapter;
- Event-driven TCP, our offloaded TCP implementation;
- IOLib, a library that encapsulates interaction between OSLayer and Event-driven TCP.

At the moment, OSLayer is implemented as a library that is statically linked with the application. Ultimately, it will be decomposed into two components: a library linked with applications and a component built in the kernel. Event-driven TCP currently runs as a user-level process that accesses the actual network via a raw socket. It will eventually become the main software loop on the adapter. The IOLib implementation currently communicates via TCP sockets, but the design allows for implementations that communicate over a PCI bus or other interconnects such as Infiniband. This provides a vehicle for experimentation and analysis and allows us to measure bus traffic without having to build a detailed simulation of a PCI bus or other interconnect.

We used the Flash Web server for our evaluation, with Flash and OSLayer running on one machine and Event-driven TCP running on another. We use httperf [29] running on a separate machine to drive load. To compare

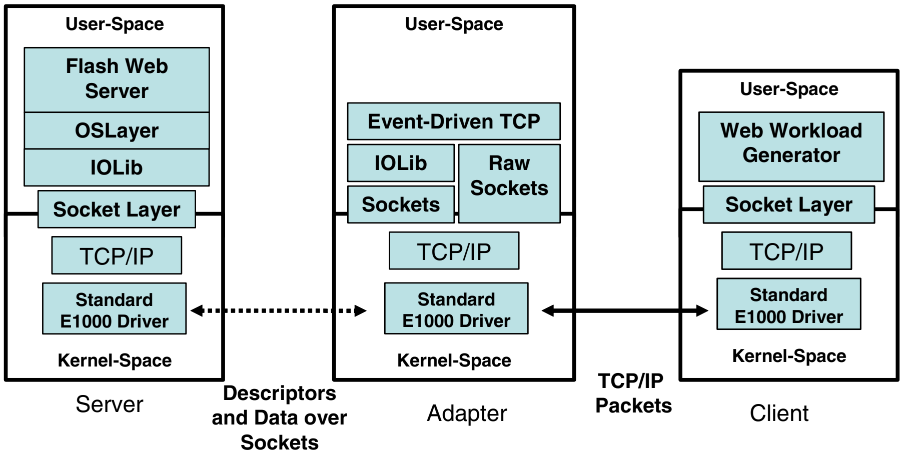

Figure 3: Offload Prototype

the behavior of the prototype with the conventional case, we evaluate a similar client-server configuration using an E1000 device driver that has been instrumented to measure bus traffic. Figure 3 illustrates how the components fit together. The implementation is described in more detail below.

4.1 OSLayer

OSLayer is essentially the socket interface decoupled from the TCP protocol implementation. OSLayer is a library that exposes an asynchronous socket interface to network applications. As seen in Figure 3, the application employs the socket API and OSLayer communicates with Event-driven TCP via IOLib through descriptors, discussed in more detail in Section 4.3. After creating the descriptor appropriate to the particular API function, the call is returned.

OSLayer and Event-driven TCP interact via a byte stream abstraction. Towards that end, OSLayer transfers buffers of 4 KB to Event-driven TCP. For large transfers, this reduces the number of DMA's and bus crossings significantly.

To further limit bus crossings and increase scalability, OSLayer employs several techniques. Descriptors can be batched before transferring them to the card. In the current implementation, we allow batching using a configurable batching level. However, a timer is used to ship over descriptors that have been waiting sufficiently long before the batching level has been reached. Even if batching is set to one, descriptors can still batch significantly with large data transfers. In addition, OSLayer performs buffer coalescing (similar to the TCP_CORK socket option on Linux), which is utilized by the Flash Web server. When using the `sendfile()` operation, this allows HTTP headers and data to be aggregated, thus sharing a descriptor and therefore a transfer. While the conventional Linux stack is limited to two sk_bufs, OSLayer can combine any number of sk_buffs into one.

4.2 Event Driven TCP

Event-driven TCP (EDT) performs the majority of the TCP processing for the adapter and was derived from the Arsenic user-level TCP stack [34]. Normally a TCP stack running on the host is animated by three types of events: a calling user-space application process or thread, a packet arrival, or a timer interrupt. Since there are no applications on the adapter, an event-driven architecture was chosen since it scales better than a process or thread-based approach. EDT is thus a single-threaded event-based closed loop, implemented as a stand-alone user-space process. On every iteration of the loop, each of the following are checked: pending packets, new descriptors, DMA completion and TCP timers. Execution is thus animated by packet and descriptor arrivals, DMA completions, and TCP timer firings.

Event-driven TCP does not necessarily notify OSLayer of every packet. For example, instead of informing OSLayer about every acknowledgment, OSLayer is only alerted when an entire transfer completes. OSLayer receives only a single event for a connection establishment or termination, rather than each packet of the TCP handshake. This reduces the number of descriptors (and their corresponding events) to be transferred and processed by the host.

EDT communicates to OSlayer by passing descriptors through IOLib, discussed in Sections 4.3 and 4.4. Since it is a user-space process, EDT sends and receives packets over the network using raw sockets and libpcap [21].

4.3 Descriptors

Descriptors are a software abstraction intended to capture the hardware-level communication mechanism that occurs over the I/O bus between host and adapter. Descriptors are primarily typed by how they are used: *request* descriptors for issuing commands (e.g., CONNECT, SEND), and *response* descriptors for the results of those commands (e.g., success, failure, retry, etc.). Descriptors are further categorized as control (SOCKET, BIND, etc.) or data (SEND, RECV). Two separate sets of tables are used for each transfer direction.

When an application calls send, a SEND request descriptor is transferred from OSLayer to Event-driven TCP, containing the address and length of the buffer to be sent. A request for DMA is queued at Event-driven TCP and the next descriptor is processed. After the DMA completes, the event is picked up by Event-driven TCP, and a response descriptor is created with the address of the buffer. This descriptor informs OSLayer that the buffer is no longer used. Upon receipt of this SEND response, OSLayer cleans up the send buffer. Of course, many send descriptors can be sent to Event-driven TCP at once. Buffers described by a SEND descriptor can be up to 4 KB. We chose 4 KB since this is the standard page size in most architectures; however, our implementation has the ability to transfer up to 64K bytes.

When receive() is called, the RECV descriptor is transferred from OSLayer to Event-driven TCP, containing the address and length of the buffer to DMA received data into. If data is available on Event-driven TCP's receive queue, a DMA is immediately initiated. Later, after DMA is finished, a RECV response descriptor is created, notifying OSLayer that the data is available, and Event-driven TCP can free its own sk_buff. If data is not available upon receipt of a RECV descriptor, the buffer is placed on a receive buffer queue for that connection. When the data does arrive later on the appropriate connection, the buffer is removed from the queue, the DMA is performed, and a RECV response descriptor is created and sent to the host.

Most of the control descriptors work in a relatively straightforward manner; however, the CLOSE operation is worth describing in more detail. A CLOSE descriptor is transferred from OSLayer to Event-driven TCP when the application initiates a close. After sending out all of the sk_buffs on the write queue, Event-driven TCP will signal the close to the remote peer via sending a packet with the FIN bit set. After the final ACK is sent a response descriptor is created. In the event the other side closes the connection, a CLOSE command descriptor is created in Event-driven TCP and sent to OSLayer. OSLayer need not reply with a CLOSE response descriptor in this case; OSLayer just notifies the application and cleans up appropriately.

All DMA's involving data are initiated by Event-driven TCP, allowing EDT to control the flow up to the host. Note that a DMA is not necessarily performed immediately. Since the request for a DMA is queued, it may be some time before a response to a descriptor is received by OSLayer.

This queued DMA approach required some changes to the TCP stack because pending sends were not preventing a CLOSE descriptor from being processed before the send's DMA competed. Since it was difficult to determine how many sends were queued for DMA (and when they were finished), "empty" sk_buffs are placed on the write queue with a flag set indicating that the data is not yet present. When the DMA completes, this flag is set to true, and the sk_buff is ready for sending. Thus, this flag is checked before sending any sk_buff. This caused changes in several components in the TCP stack. For example, close processing is now split into two pieces. The first part indicates the connection is in the process of closing; The second part actually completes the close, after the last DMA is complete and the buffer is sent.

4.4 IOLib

IOLib provides a communication library to the OSlayer and Event- driven TCP code by abstracting the I/O layer to a generic Put/Get interface. We chose this approach for ease of porting the offload prototype to bus, fabric or serial communication interfaces. Thus, only IOLib needs to understand the specific properties of the underlying communication link, while the calls within OSLayer and Event-driven TCP remain unchanged.

The IOLib Put/Get library has an asynchronous queuing interface for sending and receiving data. This interface is augmented by virtual interface registers that can be used for base address references traditionally used in the PCI bus interface. Communications support for the Put/Get interface can be provided by several types of communication: shared memory, message passing, etc. Figure 3 shows an example of how the server and adapter components communicate using IOLib, where support for the Put/Get interface is provided over a standard TCP/IP socket.

Since IOLib provides the interface between the host and the adapter, it is a natural place to monitor traffic between the two. To facilitate comparisons to conventional adapter implementations, we instrumented IOLib to measure three different aspects of I/O traffic: number of DMA's requested, number of bytes transferred, and number of I/O bus cycles consumed by a transfer. The model for capturing the number of bus cycles consumed is based on a 133 MHz, 64 bit PCI-X bus and is calculated as follows:

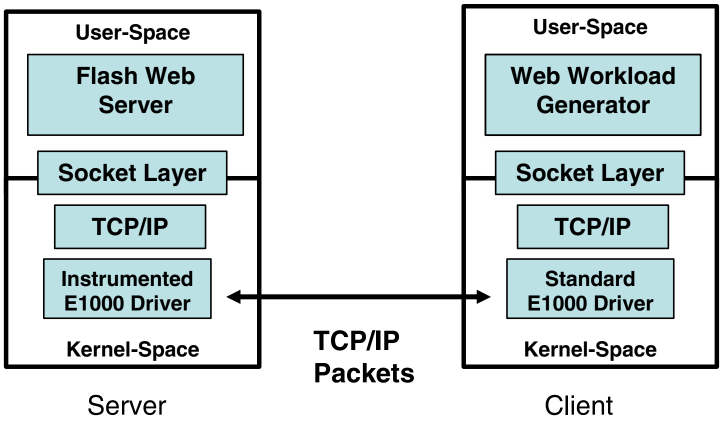

Figure 4: Baseline Configuration

$$cycles = 4 + ((transfer\_size + 7)/8)$$

This is because four cycles are required for initiation and termination [10], the bus is eight bytes (64 bits) wide, and transfers that are less than a full multiple of eight consume the bus for the entire cycle. Charges are incurred for all data transferred; not only the packet buffers transferred but also for the DMA descriptors that describe them.

4.5 Limitations of the Prototype

OSLayer is still under active development. Many of the socket options not used by Flash are not fully implemented. OSLayer requires a single-threaded application because there is no current mechanism to distinguish descriptors between threads or processes. A feedback mechanism is still required so that OSLayer knows how many send buffers are available in Event-driven TCP. If there are no send buffers available, then OSLayer can return a failure code to the application invoking the send.

More work can be done to improve and extend Event-driven TCP. For example, it could be made current with the latest version of the Linux TCP stack. We believe performance would be improved if sk_buffs could reference multiple noncontiguous pages.

Certain non-essential descriptors are not yet implemented. An immediate mode descriptor, that is, one with the data embedded directly in the descriptor, would also reduce the number of bus crossings. Descriptors for sending status (e.g., the number of available send buffers) and option querying could also improve performance and allow more dynamic behavior. Finally, descriptors to cancel a send or a receive are needed.

Several new operations with associated descriptors are planned. A batching ACCEPT operation will allow OSLayer to instruct Event-driven TCP to wait for N connections to be established before returning a response to the host. A single response descriptor would contain all of the requisite information about each connection. In the appropriate scenarios, this should reduce ACCEPT descriptor traffic. The next logical step is to provide the option of delaying the connection notification until the arrival of the first data on a newly-established connection. Another item is the addition of a “close” option to a SEND descriptor that lets a close operation be combined with a send. This eliminates the need for a separate close descriptor, and can increase the likelihood that the FIN bit is piggybacked on the final data segment.

We are also designing cumulative completion descriptors. Instead of completing each send or receive request individually with its own SEND/RECV complete descriptor, we intend to have a send complete descriptor that indicates completion of all requests up to and including that one. This change requires no syntactic changes to the descriptors; it simply changes the semantics of the response so that completion of a send/receive implicitly indicates completion of any previous sends. This approach is employed by OE [42], and we believe the benefits can be achieved in our stack as well.

4.6 E1000 Driver

To provide comparisons with a baseline system, we modified the Linux 2.6.9 Intel E1000 device driver code to measure the same three components of bus traffic as was done for IOlib: DMA’s requested, bytes transferred, and bus cycles consumed. The bus model is the same as is described in Section 4.4. Sends are measured in `e1000_tx_queue()`; receives are monitored

	1 KB			64 KB			512 KB		
	E1000	Offload	Diff	E1000	Offload	Diff	E1000	Offload	Diff
	(num)	(num)	(%)	(num)	(num)	(%)	(num)	(num)	(%)
Recv DMA Count	12	11	08	62	39	37	363	189	48
Recv Bus Cycles Consumed	119	78	34	559	269	52	3260	1394	57
Recv Bytes Transferred	538	252	53	2385	821	66	13900	4706	66
Send DMA Count	9	9	0	159	56	64	1222	370	70
Send Bus Cycles Consumed	237	200	16	9366	8532	9	74244	67683	8
Send Bytes Transferred	1572	1301	17	69474	66389	4	552132	529131	4

Table 4: Comparing IO Traffic for E1000 and Offload

in e1000_clean_rx_irq(). Figure 4 shows how the instrumented driver is used in our experiments.

5 Experimental Results

In this Section, we present the results of our prototype described in Section 4 and compare it to a baseline implementation meant to represent the current state of the art in conventional (i.e., non-offloaded) systems. The goal is to show that offload provides a more efficient interface between the host and the adapter for the metric we are able to measure, namely, I/O traffic. We again use a simple Web server workload to evaluate our prototype. For software we use the Flash Web server [33] and the httperf [29] client workload generator. We use multiple nodes within an IBM Blade Center to produce the offload prototype configuration depicted in Figure 3. The baseline configuration is shown in Figure 4.

We examine three scenarios: transferring a small file (1 KB), a moderately-sized file (64 KB), and a large file (512 KB). This is intended to capture a spectrum of data transfer sizes and vary the ratio of per-connection costs to per-byte costs. We measure transfers in both directions (send and receive) using three metrics for utilizing the I/O bus: *DMA count*, which counts the number of times a DMA is requested from the bus; *bus cycles*, which measures the number of cycles consumed on the bus (based on the model in Section 4.4); and *bytes transferred*, to determine the raw number of bytes sent over the bus.

5.1 Baseline Results

Table 4 shows our results. Overall, we see that offload is effective at reducing bus activity, with improvements up to 70 percent. We look at each transfer size in turn. Examining the results for 1 KB transfers in Table 4, note that there is a significant improvement on the receive path, mainly due to shielding the host from ACK packets. However, this is a send-side test, and the number of bytes sent from application to application do not change. Even so, we see a moderate reduction (4-17 %) in bytes transferred on the send side. This is partly because TCP, IP and Ethernet headers are not transferred over the bus in the offload prototype, whereas they are in the baseline case. Note that the number of DMAs and the utilization of the bus are also reduced, up to 70 % and 16 %, respectively.

Looking at the results for 64 KB transfers, again we see significant improvement on the receive side. A larger amount of data is being sent in this experiment, and thus the byte savings on the send side are relatively small at four percent. However, note that the efficiency of the bus has greatly improved: the number of send DMA's requested falls by 64 percent, and the bus utilization is reduced by 9 percent. The amount of bus cycles consumed has also improved by 9 percent. These trends are also reflected in the 512 KB results.

5.2 Batching Descriptors

One obvious method to reduce bus crossings is to transfer multiple descriptors at a time rather than one. The results presented in Table 5 provide experimental results for a minimum batching level of ten descriptors at a time, using a idle timeout value of ten milliseconds. These can be tuned to the transfer size, but are held constant for these experiments.

Observe that the numbers have improved for the 1K, 64K and 512K transfers over the previous comparison in Table 4. The improvements are limited since increasing the minimum batching threshold and timeouts did not significantly help for this type of traffic. This is because multiple response and socket descriptor messages are provided at nearly the same time. This technique is similar in concept to interrupt coalescing in adapters; the distinction is that information batched at a higher level of abstraction.

6 Related Work

Several performance studies on TCP offload have been conducted using an emulation approach which partitions an SMP and uses a processor as an offload engine. These

	1 KB			64 KB			512 KB		
	E1000 (num)	Offload (num)	Diff (%)	E1000 (num)	Offload (num)	Diff (%)	E1000 (num)	Offload (num)	Diff (%)
Recv DMA Count	12	7	42	62	9	85	363	24	93
Recv Bus Cycles Consumed	119	60	50	559	134	76	3260	648	80
Recv Bytes Transferred	538	244	55	2385	761	68	13900	4375	69
Send DMA Count	9	5	44	159	21	87	1222	148	88
Send Bus Cycles Consumed	237	183	23	9366	8385	11	74244	66683	10
Send Bytes Transferred	1572	1294	18	69474	66319	5	552132	528686	4

Table 5: Comparing IO Traffic for E1000 and Offload Batching Descriptor Traffic.

studies have shown that offloading TCP processing to an intelligent interface can provide significant performance improvements when compared to the standard TCP/IP networking stack. However, the study by Westrelin et al. [42] lacks an effective way to model the I/O bus traffic that occurs between the host and offload adapter. They use the host memory bus to emulate the I/O bus, but this emulation lacks the characteristics necessary to capture the performance impact of an I/O bus such as PCI. In practice, a high-speed memory bus is not representative of the performance seen by an I/O bus. Our implementation is designed with a modular I/O library that can be used to model different I/O bus types. The focus of our paper considers multiple performance impacts on server scalability including I/O. Additionally, when using a partitioned SMP emulation approach, there is coherency traffic necessary to keep the memory state consistent between processors. This coherency overhead can affect the results, since it perturbs the interaction between the host and offload adapter and includes overhead that will not exist in a real system. Our modeled offload system does not suffer from this issue.

Rangarajan et al. [35], and Regnier et al. [36] also use a partitioned SMP approach and show greater absolute performance when dedicating a processor to packet processing. This approach can measure server scalability with respect to the CPU but does not address the underlying scalability issues that exist in other parts of the system, such as the memory bus.

TCP offload designs that do not address the scalability issues discussed in this paper might improve CPU utilization on the host for large block sizes but harm throughput and latency for small block sizes. The current generation of offload adapters in the market have simply moved the TCP stack from the host to the offload adapter without the necessary design considerations for the host and adapter interface. For some workloads this creates a bottleneck on the adapter [38]. Handshaking across the host and adapter interface can be quite costly and reduce performance especially for small messages. Additionally, Ang [1] found that there appears to be no cheap way of moving data between host memory and an intelligent interface.

Performance analysis of current generation network adapters only reveals the characteristics of networking at a given point in time. In order to understand the performance impacts of various design tradeoffs, all of the components of the system need to be modeled so that performance characteristics that change over time can be revealed. Binkert et al. [6] propose the execution-driven simulator M5 to model network-intensive workloads. M5 is capable of full system simulation including the OS, the memory model, caching effects, DMA activity and multiple networked systems. M5 faithfully models the system so it can boot an unmodified OS kernel and execute applications in the simulated environment. In Section 7 we describe the use of Mambo, an instruction level simulator for the PowerPC®, in order to faithfully model network-intensive workloads.

Shivam and Chase [40] showed that offload can enable direct data placement, which can serve to eliminate some communication overheads, rather than of shifting them from the host to the adapter. They also provide a simple model to quantify the benefits of offload based on the ratio of communication to computation and the ratio of the host CPU processing power to the NIC processing power. Thus a workload can be characterized based on the parameters of the model and one can determine whether offload will benefit that workload. This paper can be seen as an application of Amdahl's Law to TCP offload. Their analysis suggests that offload best supports low-lambda applications such as storage servers.

Foong et al. [15] found that performance is scaling at about 60 percent of CPU speeds. This implies that generally accepted rule of thumb that states 1 bps of network link requires 1 Hz of CPU processing will not hold up over time. They point out that as CPU speed increases the performance gap widens between it and the memory and I/O bus. However, their study did not generate an implementation and their results are from using an emulated offload system. Our work has focused on these server scalability issues and created a design and implementation to study them.

7 Summary and Future Work

We have presented experimental evidence that quantifies how poorly server network throughput is scaling with CPU speed even with sufficient link and I/O bandwidth. We argue the scalability problem is due to specific operations that limit scalability, in particular bus crossings, cache misses and interrupts. Furthermore, we have shown experimental evidence that quantifies how bus crossings and cache misses are scaling poorly with CPU speed. We have designed a new host/adapter interface that exploits additional processing at the network interface to reduce scalability-limiting operations. Experiments with a software prototype of our offloaded TCP stack show that it can substantially reduce bus crossings. By allowing the host to deal with network data in fewer pieces, we expect our design to reduce cache misses and interrupts as well. Work is ongoing to continue development of the prototype and extend our analysis to study the effects on cache misses and interrupts.

As described in Section 4.5, the current prototype does not yet implement all aspects of the design. We are continuing development with an emphasis on further aggregation and reduction of operations that limit scalability. Future additions include a batching accept operation, an accept that returns after data arrives on the connection, a send-and-close function, and cumulative completion semantics.

We are also preparing to evaluate our prototype in Mambo, a simulation environment for PowerPC® systems [39]. Running in Mambo provides the ability to measure cache behavior and quantify the impact of hardware parameters such as processor clock rates, cache sizes, associativity and miss penalties. Mambo allows us to run the OSLayer (host) and Event-driven TCP (adapter) portions of the prototype on distinct simulated processors. We can thus determine the hardware resources needed on the adapter to support a given host workload.

Finally, we intend to extend the prototype and simulation to encompass low-level device interaction. This will entail replacing the socket-based version of IOLib with a version that communicates across a hardware interconnect such as PCI or InfiniBand®. This will allow us to predict throughput and latency on simulated next-generation interconnects.

References

[1] Boon S. Ang. An evaluation of an attempt at offloading TCP/IP protocol processing onto an i960rn-based NIC. Technical Report 2001-8, HP Labs, Palo Alto, CA, Jan 2001.

[2] Mohit Aron and Peter Druschel. TCP implementation enhancements for improving Webserver performance. Technical Report TR99-335, Rice University Computer Science Dept., July 1999.

[3] Mohit Aron and Peter Druschel. Soft timers: Efficient microsecond software timer support for network processing. *ACM Transactions on Computer Systems*, 18(3):197–228, 2000.

[4] The Infiniband Trade Association. The Infiniband architecture. `http://www.infinibandta.org/specs`.

[5] Gaurav Banga, Jeffrey Mogul, and Peter Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the USENIX 1999 Technical Conference*, Monterey, CA, June 1999.

[6] Nathan L. Binkert, Erik G. Hallnor, and Steven Reinhardt. Network-oriented full-system simulation with M5. In *Proceedings Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads CAECW*, Anaheim, CA, Feb 2003.

[7] Brent Callaghan, Theresa Lingutla-Raj, Alex Chiu, Peter Staubach, and Omer Asad. NFS over RDMA. In *Proceedings ACM SigComm Workshop on Network-I/O Convergence (NICELI)*, Karlsruhe, Germany, Aug 2003.

[8] Mallikarjun Chadalapaka, Uri Elzur, Michael Ko, Hemal Shah, and Patricia Thaler. A study of iSCSI extensions for RDMA (iSER). In *Proceedings ACM SigComm Workshop on Network-I/O Convergence (NICELI)*, Karlsruhe, Germany, Aug 2003.

[9] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6), June 1989.

[10] Intel Corporation. Small packet traffic performance optimization for 8255x and 8254x Ethernet controllers. Technical Report Application Note (AP-453), Sept 2003.

[11] Ixia Corporation. ANVL TCP testing tool. `http://www.ixiacom.com/`.

[12] The Standard Performance Evaluation Corporation. SPECWeb99. `http://www.spec.org/osg/web99`, 1999.

[13] Chris Dalton, Greg Watson, David Banks, Costas Clamvokis, Aled Edwards, and John Lumley. Afterburner. *IEEE Network*, 11(2):36–43, July 1993.

[14] Peter Druschel, Larry Peterson, and Bruce Davie. Experiences with a high-speed network adaptor: A software perspective. In *ACM SIGCOMM Symposium*, London, England, August 1994.

[15] Annie P. Foong, Thomas R. Huff, Herbert H. Hum, Jaidev P. Patwardhan, and Greg J. Regnier. TCP performance re-visited. In *Proceedings International Symposium on Performance Analysis of Systems and Software ISPASS*, Austin, TX, March 2003.

[16] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach (2nd Edition)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 1995.

[17] Red Hat Inc. The Tux WWW server. `http://people.redhat.com/~mingo/TUX-patches/`.

[18] Philippe Joubert, Robert King, Richard Neves, Mark Russinovich, and John Tracey. High-performance memory-based Web servers: Kernel and user-space performance. In *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, June 2001.

[19] Jonathan Kay and Joseph Pasquale. Profiling and reducing processing overheads in TCP/IP. *IEEE/ACM Transactions on Networking*, 4(6):817–828, December 1996.

[20] Karl Kleinpaste, Peter Steenkiste, and Brian Zill. Software support for outboard buffering and checksumming. In *ACM SIGCOMM Symposium*, pages 196–205, Cambridge, MA, August 1995.

[21] The libpcap Project. `http://sourceforge.net/projects/libpcap/`.

[22] Srihari Makineni and Ravi Iyer. Measurement-based analysis of TCP/IP processing requirrments. In *10th International Conference on High Performance Computing (HiPC 2003)*, Hyderabad, India, December 2003.

[23] Evangelos P. Markatos. Speeding up TCP/IP: Faster processors are not enough. In *Proceedings 21st IEEE International Performance, Computing, and Communication Conference IPCCC*, pages 341–345, Phoenix, AZ, April 2002.

[24] Paul E. McKenney and Ken F. Dove. Efficient demultiplexing of incoming TCP packets. In *ACM SIGCOMM Symposium*, pages 269–279, Baltimore, Maryland, August 1992. ACM.

[25] Larry McVoy and Carl Staelin. LMBENCH: Portable tools for performance analysis. In *USENIX Technical Conference of UNIX and Advanced Computing Systems*, San Diego, CA, January 1996.

[26] Jeffrey C. Mogul. Operating systems support for busy Internet servers. In *Proceedings Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, Orcas Island, WA, May 1995.

[27] Jeffrey C. Mogul. TCP offload is a dumb idea whose time has come. In *USENIX Workshop on Hot Topics on Operating Systems (HotOS)*, Hawaii, May 2003.

[28] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, 1997.

[29] David Mosberger and Tai Jin. httperf – a tool for measuring Web server performance. In *Proceedings 1998 Workshop on Internet Server Performance (WISP)*, Madison, WI, June 1998.

[30] Erich M. Nahum, Tsipora Barzilai, and Dilip Kandlur. Performance issues in WWW servers. *IEEE/ACM Transactions on Networking*, 10(2):2–11, Feb 2002.

[31] Jitendra Padhye and Sally Floyd. On inferring TCP behavior. In *ACM SIGCOMM Symposium*, pages 287–298, 2001.

[32] Vijay Pai, Scott Rixner, and Hyong-Youb Kim. Isolating the performance impacts of network interface cards through microbenchmarks. In *Proceedings ACM Sigmetrics*, New York, NY, June 2004.

[33] Vivek Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable Web server. In *USENIX Annual Technical Conference*, Monterey, CA, June 1999.

[34] Ian Pratt and Keir Fraser. Arsenic: A user-accessible gigabit ethernet interface. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, Anchorage, Alaska, April 2001.

[35] Murali Rangarajan, Aniruddha Bohra, Kalpana Banerjee, Enrique V. Carrera, Ricardo Bianchini, and Liviu Iftode. TCP servers: Offloading TCP processing in Internet servers, design, implementation and performance. Technical Report DCS-TR-481, Rutger University, Department of Computer Science, Piscataway,NJ, March 2003.

[36] Greg Regnier, Dave Minturn, Gary McAlpine, Vikram Saletore, and Annie Foong. ETA: Experience with an intel xeon processor as a packet processing engine. In *11th Annual Symposium on High Performance Interconnects*, Palo Alto, CA, August 2003.

[37] Yaoping Ruan and Vivek Pai. Making the “box” transparent: System call performance as a first-class result. In *USENIX Annual Technical Conference*, Boston, MA, June 2004.

[38] Prasenjit Sarkar, Sandeep Uttamchandani, and Kaladhar Voruganti. Storage over IP: When does hardware support help? In *USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, March 2003.

[39] H. Shafi, P.J. Bohrer, J. Phelan, C.A. Rusu, and J.L. Peterson. Design and validation of a performance and power simulator for POWERPC systems. *IBM Journal of Research and Development*, 47(5/6):641–651, September/November 2003.

[40] Piyush Shivam and Jeffrey S. Chase. On the elusive benefits of protocol offload. In *ACM SigComm Workshop on Network-IO Convergence (NICELI)*, Germany, August 2003.

[41] Jonathan Stone, Michael Greenwald, Craig Partridge, and James Hughes. Performance of checksums and CRC’s over real data. *IEEE/ACM Transactions on Networking*, 6(5):529–543, 1998.

[42] R. Westrelin, N. Fugier, E. Nordmark, K. Kunze, and E. Lemoine. Studying network protocol offload with emulation: Approach and preliminary results. In *12th Annual IEEE Symposium on High Performance Interconnects*, Stanford, CA, Aug 2004.