Trapeze/IP: TCP/IP at Near-Gigabit Speeds

Andrew Gallatin, Jeff Chase, and Ken Yocum[*]
Department of Computer Science
Duke University
{*gallatin, chase, grant*}*@cs.duke.edu*

Abstract

This paper presents experiences with high-speed TCP/IP networking on a gigabit-per-second Myrinet network, using a Myrinet messaging system called Trapeze. We explore the effects of common optimizations above and below the TCP/IP protocol stack, including zero-copy sockets, large packets with scatter/gather I/O, checksum offloading, message pipelining, and interrupt suppression. Our experiments use extended FreeBSD 4.0 kernels on a range of Intel and Compaq Alpha platforms. These experiments give a snapshot of the FreeBSD TCP/IP implementation running at bandwidths as high as 956 Mb/s. We also report some results using Gigabit Ethernet products from Alteon Networks, which yielded a TCP bandwidth of 988 Mb/s using zero-copy sockets on a 500 MHz Compaq Alpha 21264 workstation.

1 Introduction

Over the next few years, new high-speed network standards — primarily Gigabit Ethernet — will consolidate an order-of-magnitude gain in network performance already achieved with specialized cluster interconnects such as Myrinet and SCI. As these technologies gain acceptance in LANs and server farms, they will place new performance pressure on network software. Although the latest desktop-class computers are capable of outstanding I/O performance, there is little quantitative basis to: (1) predict the performance they will actually deliver using standard TCP/IP networking on the new generation of networks, (2) quantify the importance of proposed optimizations (e.g., Jumbo Frames, zero-copy buffering, checksum offloading) to achieving the potential hardware performance, or (3) judge when alternatives such as user-level networking (e.g., VIA) are justified. In most cases published performance results are based on research prototypes using previous-generation technology.

This paper presents experiences with high-speed TCP/IP networking on a gigabit-per-second Myrinet network [3]. Our work is based on the Trapeze messaging system [10, 5, 1, 9], which consists of a messaging library and custom firmware for Myrinet. Using Trapeze firmware, Myrinet delivers communication performance at the limit of I/O bus speeds on many platforms, closely approaching the full gigabit-per-second wire speed on the most powerful hosts. This makes Trapeze/Myrinet a good vehicle for probing the limits of both the hardware and the networking software. In the experiments presented here, we exercised a Trapeze/Myrinet network with a network device driver supporting a standard kernel-based TCP/IP protocol stack on a range of DEC Alpha and Intel-based platforms. Our purpose is to provide a quantitative snapshot of the current state of the art for point-to-point TCP/IP communication on short-haul networks with low error rates, low latency, and gigabit-per-second bandwidth.

The kernel used in our experiments is FreeBSD 4.0, a descendent of the Berkeley 4.4 BSD code base, which incorporates several years worth of TCP/IP refinements. It is now widely accepted that current TCP implementations are capable of delivering a high percentage of available link speeds with large transfers, reflecting the success of these earlier efforts. However, on gigabit-per-second networks the performance of even the best TCP/IP implementations is dependent on key optimizations for low-overhead data movement, both above and below the protocol stack. One goal of this paper is to provide quantitative data to support insights into the effects and importance of these optimizations on current workstation/PC technology.

This paper outlines the key optimizations important for obtaining hardware potential from TCP/IP, their implementation in the network interface, network driver,

[*]This work is supported by the National Science Foundation (CCR-96-24857, CDA-95-12356, and EIA-9870724) and equipment grants from Intel Corporation and Myricom.

and kernel socket code, and their effect on delivered TCP bandwidth, UDP latency, and CPU utilization at the sender and receiver. Below the TCP/IP stack, the Trapeze NIC firmware and network driver support page-aligned payload reception, interrupt suppression, large frames (MTUs), TCP checksum offloading, and an adaptive message pipelining scheme that balances low latency and high bandwidth. Above the protocol stack, at the socket layer, we have implemented new kernel support for zero-copy data movement in TCP as an extension to a zero-copy stream interface implemented by John Dyson. We show the effect of each of these factors on TCP/IP networking performance. We also report some results using similar features with Gigabit Ethernet adapters and switches from Alteon Networks.

Using Trapeze/Myrinet with zero-copy sockets, *netperf* attained a peak point-point bandwidth close to the link speed at 956 Mb/s on a 500 MHz Alpha 21264 PC platform equipped with prototype LANai-5 adapters from Myricom. At this speed, bandwidth is limited by the LANai-5 CPU. Newer controllers with upgraded CPUs promise still higher bandwidths. In fact, we measured bandwidth of 988 Mb/s on the same platform over the Alteon network, which uses a faster CPU on the adapters. The previous point-to-point record reported at *netperf.org* was 750 Mb/s, measured on a pair of mainframe-class SMP servers interconnected by HiPPI. We are not aware of any better result on public record.

This paper is organized as follows. Section 2 gives an overview of the Trapeze network interface, and Section 2.2 outlines the various optimizations for low-overhead TCP/IP communication. Section 3 presents performance results. We conclude in Section 4.

2 TCP/IP with Trapeze/Myrinet

This section presents background material important for understanding the performance results in Section 3. We first give an overview of the Trapeze messaging system, with a focus on the features relevant to TCP/IP networking. We then outline the optimizations used above and below the TCP/IP protocol stack to reduce data movement overhead and per-packet handling costs.

2.1 Trapeze Overview

The Trapeze messaging system consists of two components: a messaging library that is linked into the kernel or user programs, and a firmware program that runs on the Myrinet network interface card (NIC). The Trapeze firmware and the host interact by exchanging commands and data through a block of memory on the NIC, which is addressable in the host's physical address space using programmed I/O. The firmware defines the interface between the host CPU and the network device; it interprets commands issued by the host and controls the movement of data between the host and the network link. The host accesses the network using macros and procedures in the Trapeze library, which defines the lowest level API for network communication across the Myrinet. Since Myrinet firmware is customer-loadable, any Myrinet site can use Trapeze.

Trapeze was designed primarily to support fast kernel-to-kernel messaging alongside conventional TCP/IP networking. Trapeze currently hosts kernel-to-kernel RPC communications and zero-copy page migration traffic for network memory and network storage, a user-level communications layer for MPI and distributed shared memory, a low-overhead kernel logging and profiling system, and TCP/IP device drivers for FreeBSD and Digital UNIX. These drivers allow a native TCP/IP protocol stack to use a Trapeze network through the standard BSD *ifnet* network driver interface. Figure 1 depicts this structure.

Trapeze messages are short *control messages* (maximum 128 bytes) with optional attached *payloads* typically containing application data not interpreted by the networking system, e.g., file blocks, virtual memory pages, or a TCP segments. The data structures in NIC memory include two message rings, one for sending and one for receiving. Each message ring is a circular array of 128-byte control message buffers and related state, managed as a producer/consumer queue shared with the host. From the perspective of a host CPU, the NIC produces incoming messages in the receive ring and consumes outgoing messages in the send ring.

Trapeze has several features useful for high-speed TCP/IP networking:

- **Separation of header and payload.** A Trapeze control message and its payload (if any) are sent as a single packet on the network, but the control message and payload are handled separately by the firmware and message system. In particular, payloads are transferred to and from aligned page frames of host memory, which the driver allocates from the virtual memory page pool. This enables the zero-copy optimizations described in Section 2.2.1, assuming the driver is able to place the TCP/IP headers in the control message portion of the packet.

- **Large MTUs with scatter/gather DMA.** Since Myrinet has no fixed maximum packet size (MTU), the maximum payload size of a Trapeze network is easily configurable. Trapeze supports scatter/gather DMA so that payload buffers may span multiple noncontiguous page frames. Scatter/gather allows

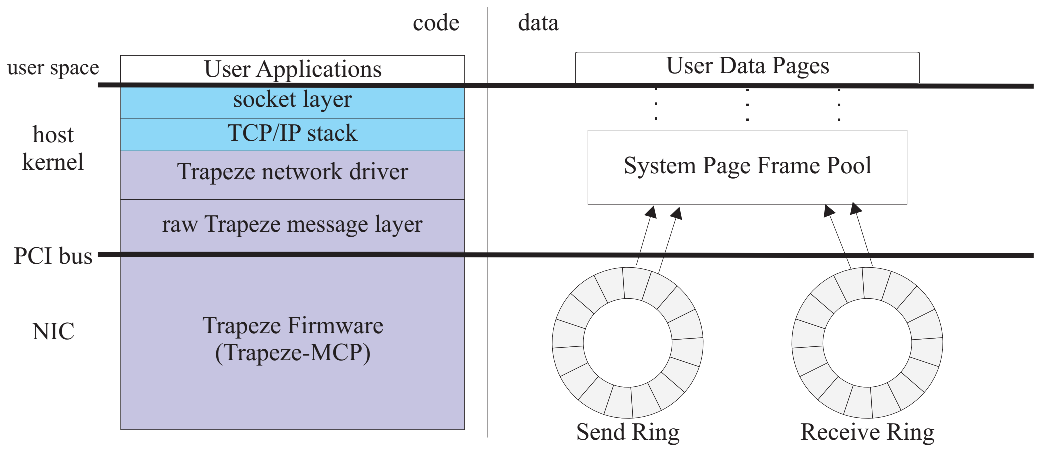

Figure 1: Using a Trapeze endpoint for kernel-based TCP/IP networking.

us to run TCP with MTUs of 64 KB or larger, yielding very low per-packet overheads in the networking code.

- **Adaptive message pipelining.** The Trapeze firmware pipelines DMA transfers on the I/O bus and network link to minimize large packet latency [10]. The pipelining scheme adaptively reverts to larger unpipelined DMA transfers in bandwidth-constrained scenarios [9]. This technique enables Trapeze to combine low large-packet latencies with high bandwidth under load.

One item missing from this list is *interrupt suppression*. Handling of incoming messages is interrupt-driven when Trapeze is used from within the kernel; incoming messages are routed to the destination kernel module (e.g., the TCP/IP network driver) by a common interrupt handler in the Trapeze message library. Interrupt handling imposes a per-packet cost that becomes significant with smaller MTUs. Some high-speed network interfaces reduce interrupt overhead by amortizing interrupts over multiple packets during periods of high bandwidth demand. For example, the Alteon Gigabit Ethernet NIC includes support for adaptive interrupt suppression, selectively delaying packet-receive interrupts if more packets are pending delivery. Trapeze implements interrupt suppression for a lightweight kernel-kernel RPC protocol [1], but we do not use receive-side interrupt suppression for TCP/IP because it yields little benefit for MTUs larger than 16KB at current link speeds.

2.2 Low-Overhead Data Movement

This section describes the optimizations used above and below the TCP/IP protocol stack to reduce data movement overhead for copying and checksumming data. These overheads increase with the volume of data moved per unit of time; at gigabit-per-second bandwidths, data movement overhead can consume a large share of CPU cycles. Unfortunately, faster CPUs do not help appreciably because copying is memory-intensive.

We describe the data movement optimizations as extensions to the conventional FreeBSD send/receive path, which is based on variable-sized kernel network buffers called *mbufs* [8]. Standard *mbufs* contain their own buffer space, while *external mbufs* hold pointers to other kernel buffers, e.g., file buffers or the virtual memory page frames used as Trapeze payload buffers. Packet data is stored in linked chains of *mbufs* passed between levels of the system; the TCP/IP protocol stack adds and removes headers and checksums by manipulating the *mbufs* in the chain. On a normal transmission, the socket layer copies IP message from a user memory buffer into a chain, which is passed through the TCP/IP stack to the network driver. On the receiving side, the driver constructs a chain containing each incoming packet header and payload, and passes the chain through the TCP/IP stack to the socket layer. When the receiving process accepts the data, e.g., with a *read* system call, a socket-layer routine (*soreceive*) copies the payload into user memory and frees the kernel *mbuf* chain.

2.2.1 Zero-Copy Sockets

Conventional TCP/IP communication incurs a high cost to copy data between kernel buffers and user process virtual memory at the socket layer. This situation has motivated development of techniques to reduce or eliminate data copying by *page remapping* between the user process and the kernel when size and alignment properties allow [6, 4, 7]. A page remapping scheme should preserve the copy semantics of the existing socket interface.

In general, zero-copy optimizations assume MTUs matched to the page size of the endstation hardware and operating system. Ideally, each packet payload is an even multiple of the page size, and is stored in buffers that naturally align on page boundaries. On the receive side, the NIC must separate the headers and payload into separate buffers, leaving the payload page-aligned. This can be done with special support to recognize TCP/IP packets on the NIC, or by constructing receive *mbuf* chains that optimistically assume that received packets are TCP packets. In Trapeze, the sending host explicitly separates header and payload portions of each packet; the Trapeze driver optimistically assumes that data in the first *mbuf* of an outgoing chain is header data, and places its data in the control message. The link layer preserves this separation on the receiving side.

We implemented zero-copy TCP/IP extensions at the socket layer in the FreeBSD 4.0 kernel, using code developed by John Dyson for zero-copy I/O through the *read/write* system call interface. The zero-copy extensions require some buffering support in the network driver, but are otherwise independent of the underlying network, assuming that it supports sufficiently large MTUs and page-aligned sends and receives. Section 3 reports results from zero-copy TCP experiments on both Trapeze/Myrinet and Alteon Gigabit Ethernet hardware.

The page remapping occurs in a variant of the *uiomove* kernel routine, which directs the movement of data to and from the process virtual memory for all variants of the I/O read and write system calls. Our zero-copy socket code is implemented as a new case alongside Dyson's code in *uiomoveco*, which is invoked from socket-layer *sosend* and *soreceive* when a process requests the kernel to transfer a page or more of data to or from a page-aligned user buffer.

For a zero-copy read, *uiomoveco* maps kernel buffer pages directly into the process address space. If the *read* is from a file, it creates a copy-on-write mapping to a page in FreeBSD's unified buffer cache; the copy-on-write preserves the file data in case the user process stores to the remapped page. In the case of a receiver read from a socket, copy-on-write is unnecessary because there is no need to retain the kernel buffer after the read; ordinarily *soreceive* simply frees the kernel buffers once the data has been delivered to the user process. The remapping case instead frees just the *mbuf* headers and any physical page frames that previously backed remapped virtual pages in the user buffer. Thus most receive-side page remappings actually trade page frames between the process and the kernel buffer pool, preserving equilibrium.

On the send side, copy-on-write is used because the sending process may overwrite its send buffer once the send is complete. The send-side code maps each whole page from the user buffer into the kernel address space, references it with an external *mbuf*, and marks the page as copy-on-write. The *mbuf* chains and their pages are then passed through the TCP/IP stack to the network driver, which attaches them to outgoing messages as payloads. When each mbuf is freed on transmit complete, the external free routine releases the page's copy-on-write mapping. The new socket layer code handles only anonymous virtual memory pages; we do not support zero-copy transmission of memory backed by mapped files because this would duplicate the functionality of the *sendfile* routine already implemented by David Greenman.

2.2.2 Checksum Offloading

Checksum offloading eliminates host-side checksumming overheads by performing checksum computation with hardware assist in the NIC. TCP/IP checksum offloading is supported by Myricom's recently released LANai-5 adapter, and by other high-speed network interfaces including Alteon's Gigabit Ethernet NICs based on the Tigon-II chipset.

The NIC and the host-side driver must act in concert to implement checksum offloading. The LANai-5 and Alteon NICs support checksum offloading in the host-PCI DMA engine, which computes the raw 16-bit ones-complement checksum of each DMA transfer as it moves data to and from host memory. Using this checksum need not demand any significant change to the IP stack: simply setting a M_HWCKSUM flag in the header of an *mbuf* chain bypasses the software checksum computation in *in_cksum*. However, using hardware checksumming for IP protocol family is complicated by three factors:

- Movement of each packet may occur in multiple DMA transfers to or from distinct host memory buffers. If the hardware makes each partial checksum available to the NIC firmware separately (as Myricom's LANai-5 NIC), then firmware and/or host software must combine these partial checksums (using one's complement addition) to obtain a complete checksum.

- TCP or UDP checksumming actually involves two checksums: one for the IP header (including fields overlapping with the TCP or UDP header) and a second end-to-end checksum covering the TCP or UDP header and packet data. In a conventional system, TCP or UDP computes its end-to-end checksum before IP fills in its overlapping IP header fields (e.g., options) on the sender, and after the IP layer restores these fields on the receiver. Checksum offloading involves computing these checksums below the IP stack; thus the driver or NIC firmware must partially dismantle the IP header in order to compute a correct checksum.

- Since the checksums are stored in the headers at the front of each IP packet, a sender must complete the checksum before it can transmit the packet headers on the link. If the checksums are computed by the host-NIC DMA engine, then the last byte of the packet must arrive on the NIC before the firmware can determine the complete checksum.

Trapeze currently supports TCP checksum offloading only on LANai-5 receivers. Checksum offloading is not supported on the sending side, in part because Trapeze uses message pipelining to minimize latency of large packets. With message pipelining the front of a packet may be transmitted on the link before the tail of the packet arrives on the NIC, and therefore before the checksum can be determined. One solution is to append the end-to-end checksum to the tail of the outgoing packet; while this would depart from the standard IP packet format, it is transparent to the end hosts because the Trapeze firmware and driver can reconstruct the packet at the receiving side. Of course, this approach would compromise interoperability in a standards-based network containing some endstations that do not support checksum offloading. The alternative, apparently implemented in Alteon's NICs, is to use store-and-forward packet transmission at the sender, which increases large-packet latencies (see Section 3.4).

Trapeze uses the NIC DMA engines to checksum packet data, but header checksums are computed by special-case code for TCP/IP in the Trapeze network driver. The Trapeze firmware combines partial checksums for all DMA operations on the payload portion of the message, then passes the partial checksum to the host-side driver through a logical control register. The driver then computes an IP header checksum, computes the layer-4 header checksum using a scratch copy of the IP header, combines the layer-4 header checksum with the payload checksum to determine the complete end-to-end checksum, and compares the computed checksums with those transmitted in the packet. Our philosophy is that any instructions that manipulate the IP header should be executed on the fast host CPU rather than on the NIC. In contrast, the Alteon NICs perform both header and data checksums in the NIC firmware.

3 Experimental Results

We ran our experiments on four Intel and Alpha hardware configurations:

- **Pentium-II/440LX**. These are Dell Dimension XPS D-300 workstations containing a 300 MHz Intel Pentium-II processor and an Intel 440LX chipset. Each machine has 128MB of RAM and a Myricom Lanai 4.1 SAN adapter (M2M-PCI32C) connected to a 32-bit 33 MHz PCI slot.

- **Pentium-II/440BX**. These machines use a Pentium-II processor clocked at 450 MHz on an Asus P2B motherboard with an Intel 440BX chipset. Each machine has 128MB of RAM and a Myricom Lanai 4.1 LAN adapter (M2F-PCI32) connected to a 32-bit 33 MHz PCI slot.

- **DEC Miata**. These are Digital Personal Workstation 500au platforms with a 500MHz Alpha 21164 CPU, a 96KB L2 cache, a 2MB L3 cache, and the Digital 21174 "Pyxis" chipset. These machines are configured with 512MB of RAM and a Myricom Lanai 4.1 SAN adapter connected to a 32-bit 33 MHz PCI slot. The Pyxis limits I/O bandwidth to approximately 103 MB/s on the sending side.

- **DEC Monet**. These are Compaq XP1000 Professional Workstations, with a 500 MHz Alpha 21264 CPU, a 4MB L2 cache, and the Digital 21272 "Tsunami" chipset. These machines are configured with 640MB of RAM and a Myricom Lanai 5.2 SAN adapter connected to a 64-bit 33 MHz PCI slot. The Lanai-5 is described in *http://www.myri.com:80/scs/PCI64X/PCI64X-spec.html*. We also report some Gigabit Ethernet measurements from this platform, using Alteon ACENIC adapters based on the Tigon-II chipset (firmware revision 12.3.8), interconnected through an ACEswitch 1080 (firmware revision 5.0.24).

All systems run kernels built from the same FreeBSD 4.0 source pool, which was current as of 4/15/99. The hosts are interconnected through diverse Myrinet switch models, which have no measurable effects on the results.

To take timings, we used *netperf* version 2.1pl3 built from the FreeBSD ports collection. We modified *netperf* to collect CPU utilization by reading the system timers directly from kernel memory via *libkvm*, in order to correctly charge interrupt overhead to the *netperf* process.

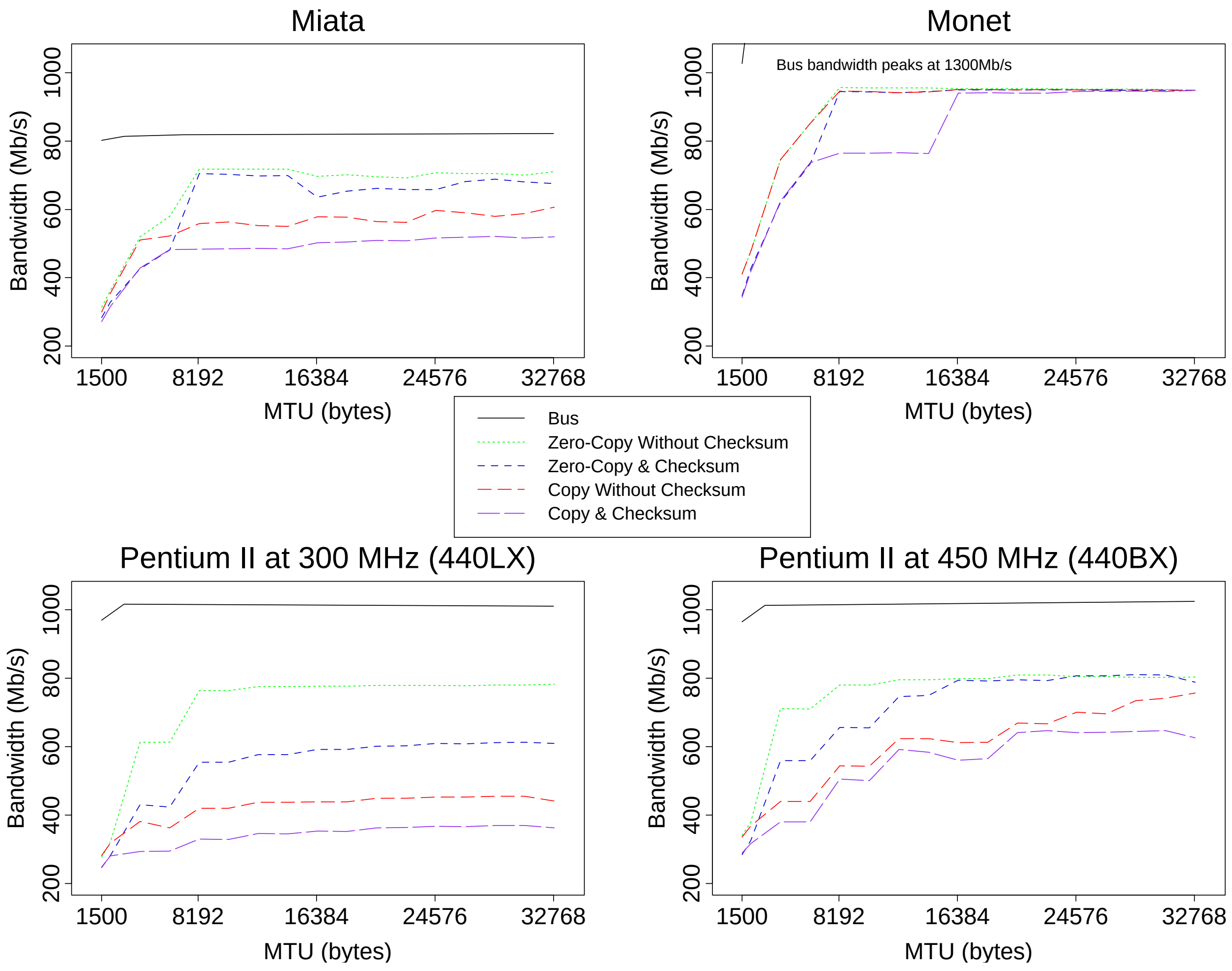

Figure 2: TCP Bandwidth

All tests were run on isolated machines, and the vast majority of the interrupts serviced came from the gigabit NIC.

3.1 TCP Bandwidth

We measured unidirectional point-to-point TCP bandwidth using *netperf -l60 -C -c*, which sends as much data as it can in 60 seconds, then computes the average bandwidth over the interval. Socket buffers were set to 512 KB for all tests with an MTU of 4KB or greater. Tests with smaller MTUs used 64 KB socket buffers. Note that the *netperf* sender and receiver do not access the data in these tests.

Figure 2 shows the average Trapeze TCP bandwidth on all four platforms as a function of the MTU. To show the effects of copying and checksumming, we tested with the zero-copy optimizations enabled and disabled, and with checksumming enabled and disabled. All checksumming is done in software except on the receiving side of the Monet configuration, which uses checksum offloading on the LANai-5 adapter as described in Section 2.2.2.

The graphs show the bandwidth costs of copying and checksumming, primarily on the older platforms with less memory system bandwidth. The effect is most apparent on the P-II/440LX, which is capable of a peak bandwidths of only 450 Mb/s if it is forced to copy the data, while peak bandwidth almost doubles to 780 Mb/s when the zero-copy optimizations are enabled. The cost of checksumming is more pronounced with zero-copy enabled, since the checksum code must bring the data into the CPU cache. On the P-II/440BX, superior memory system bandwidth allows the system to achieve close to its peak bandwidth even while copying or checksumming, but not both, and only for very large MTUs when the CPU is not already busy with packet-handling overheads. This is also visible on the Miata, which has comparable memory system bandwidth, but the effect is less pronounced because the I/O bus limits the achievable bandwidth. The Monet has adequate memory system bandwidth to deliver a peak bandwidth of 956 Mb/s for sufficiently large MTUs, even while copying and checksumming. Even so, copying and checksumming have a significant effect on the available CPU cycles remaining to process the data at these speeds, as Section 3.2 shows.

Figure 2 also shows the difficulty of achieving high bandwidth using the small 1500-byte MTUs of the Gigabit Ethernet standard. In addition to increasing packet handling overheads, small MTUs defeat the zero-copy optimizations. The combined effect causes the host CPU to saturate at bandwidths as low as 300 Mb/s, and none of the platforms is capable of using more than half of the available link speed. Section 3.2 examines the overheads in more detail. All platforms are capable of achieving most of their peak bandwidth at MTUs large enough to contain a TCP/IP header and a page of data; the Intel platform bandwidths rise faster because zero-copy kicks in at the 4KB page size, while the Alpha platforms use an 8KB page size.

While we were pleased with the Trapeze bandwidth results on Monet, which we believed to be an open-source record, we measured even higher bandwidths with Alteon's new Gigabit Ethernet products. The Monet delivers point-to-point TCP bandwidth of 988 Mb/s with zero-copy sockets over Alteon. The higher bandwidth is apparently due to lower overheads in the Alteon controller, which sports dual 100 MHz MIPS R4000-like processors delivering several times the processing capacity of the LANai-5 CPU. We anxiously await the LANai-7 from Myricom.

3.2 CPU Utilization

The potential for high bandwidth has little value in practice if communication overheads leave no CPU power to process the data. CPU utilization is just as important as bandwidth, since bandwidths will drop if application processing saturates the CPU. All the optimizations we explore are fundamentally directed at reducing overhead; they increase the delivered bandwidth only indirectly by delaying saturation of the host CPUs. Overheads are reduced by reducing packet handling costs with larger MTUs, reducing interrupt costs with larger MTUs or interrupt suppression, and reducing data-touching costs through zero-copy page remapping or checksum offloading.

Figures 3 and 4 show the average CPU utilizations on the sender and receiver respectively for the bandwidth tests reported in Section 3.1. Some of the results vary noticeably due to several factors. On the receiver, this may be affected by incomplete support for page coloring in the Alpha FreeBSD port; the runs show a bimodal distribution on the Alpha receiver configurations when copying is used. In the zero-copy sender results, irregularities result when *netperf* reuses a send buffer page before the driver determines that the previous transmit on that buffer is complete. The Trapeze driver detects this case and conservatively copies the page, although *netperf* does not actually store to the buffer in this experiment; if the process did store to the page then a copy-on-write would result. The sender-side zero-copy optimizations trigger with varying probabilities on different configurations, and are affected by CPU speed, the process send buffer size, and the Trapeze ring size, given that Trapeze suppresses transmit-complete notices until a send ring entry is reused. Some step behavior results from the TCP implementation selecting packet sizes that are integral multiples of the page size for odd MTUs; these effects are less pronounced on the Intel platforms, which use 4KB rather than 8KB pages. The numbers presented here are averages of 20 runs.

On the receiving side, all graphs show a trend of declining CPU utilization with large MTUs, with much lower CPU utilizations when data-movement costs such as copying and checksumming are eliminated. The downward trend with larger MTUs is most pronounced on the faster platforms, since the bandwidth of older platforms such as 440LX is initially limited by the CPU; reduced overheads result in higher bandwidth rather than lower CPU utilization. Similarly, CPU utilizations initially increase with larger MTUs on the sending side. This is because the larger MTUs allow higher bandwidth at the receiver, driving the sender to transmit faster. Once peak bandwidth is attained, the CPU utilizations begin to drop with increasing MTU. The graphs reflect the higher CPU costs on the receiving side, primarily due to lower interrupt overheads at the sender.

These graphs show that copying and checksumming optimizations are extremely important even on the platforms that are capable of achieving peak bandwidth without them. Any reduction in overhead translates directly into lower CPU utilization, leaving more cycles available for application processing at a given bandwidth. Note also that disabling checksumming yields little benefit on the Monet receiver because of checksum offloading: the small incremental CPU cost is due to checksumming the headers in the driver.

The receiver utilizations in Figure 3 again reinforce the importance of the Jumbo Frames standard promulgated by Alteon and Microsoft, which would increase the Gigabit Ethernet MTU to 9000 bytes. The Intel receiver CPUs are saturated at 1500-byte MTUs,showing that the bandwidth limitation near 300 Mb/s for standard Ethernet MTUs is due to receiver CPU saturation caused by the overhead of handling the larger number of packets. Slightly higher 1500-byte bandwidths are achieved on the Alphas due to the faster host CPUs: on

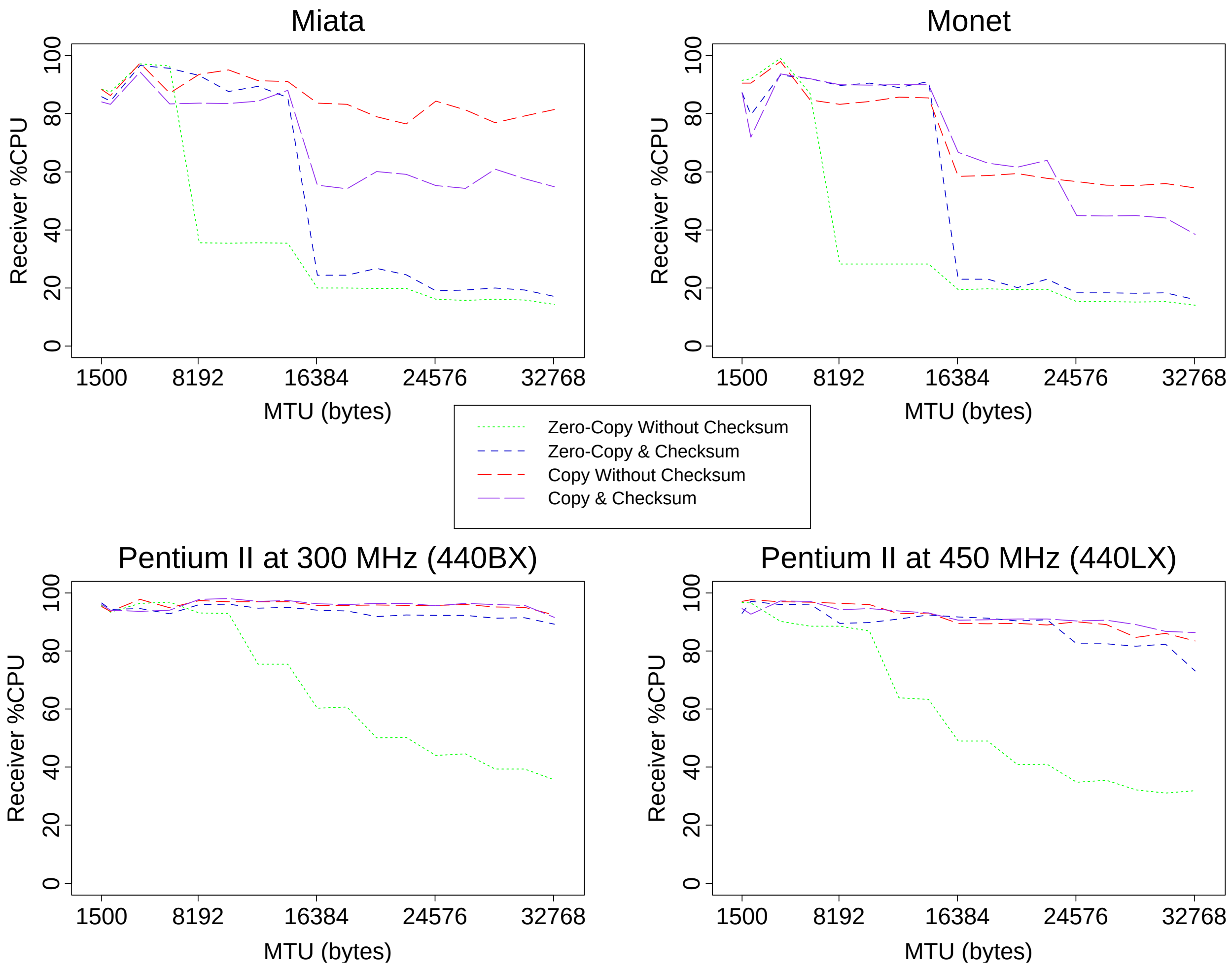

Figure 3: Receiver CPU Utilization

Miata the 1500-byte bandwidth is limited at 313 Mb/s by saturation of the CPU on the LANai-4 NIC, while the faster LANai-5 NIC delivers bandwidths closer to 410 Mb/s before saturating. However, at this speed packet handling overheads push the Monet's Alpha 21264 host CPU to 90% utilization, even while it is driving less than half of the link speed. The Monet's receiver utilization drops below 30% with 8KB packet sizes when zero-copy and checksum offloading are enabled, even as the delivered bandwidth more than doubles to 956 Mb/s.

Looking further, the results show that the 9000-byte MTU of the Jumbo Frames standard is sufficient to achieve near-peak bandwidth on all platforms. However, Figure 3 shows that if other host overheads are present, per-packet overheads can constrain peak bandwidth even if Jumbo Frames are used. The 440BX platform does not attain its peak bandwidth until 16KB MTUs, and does not achieve its minimal CPU utilization until the MTUs reach 57KB. Receiver CPU utilization on this platform drops from 88% to 48% as the MTU grows from 8KB to 16KB.

3.3 TCP Overhead

To better understand the costs responsible for the CPU utilizations presented in Figures4 and 3, we used *iprobe* to derive a breakdown of receiver overheads on Miata for selected MTU sizes at bandwidth levels held in the 300-400 Mb/s range by a slow sender. Iprobe (Instruction Probe) is an on-line profiling tool developed by the performance group (High Performance Servers/Benchmark Performance Engineering) of Digital/Compaq. It uses the Digital Alpha on-chip performance counters to report detailed execution breakdowns with low overhead (3%-5%), using techniques similar to those reported in [2]. We gathered our data using a local port of *iprobe_suite-4.0* to FreeBSD. This port will be integrated into the next release of *iprobe*.

Figure 5 shows the breakdown of receiver overhead into five categories: data movement overheads for copy-

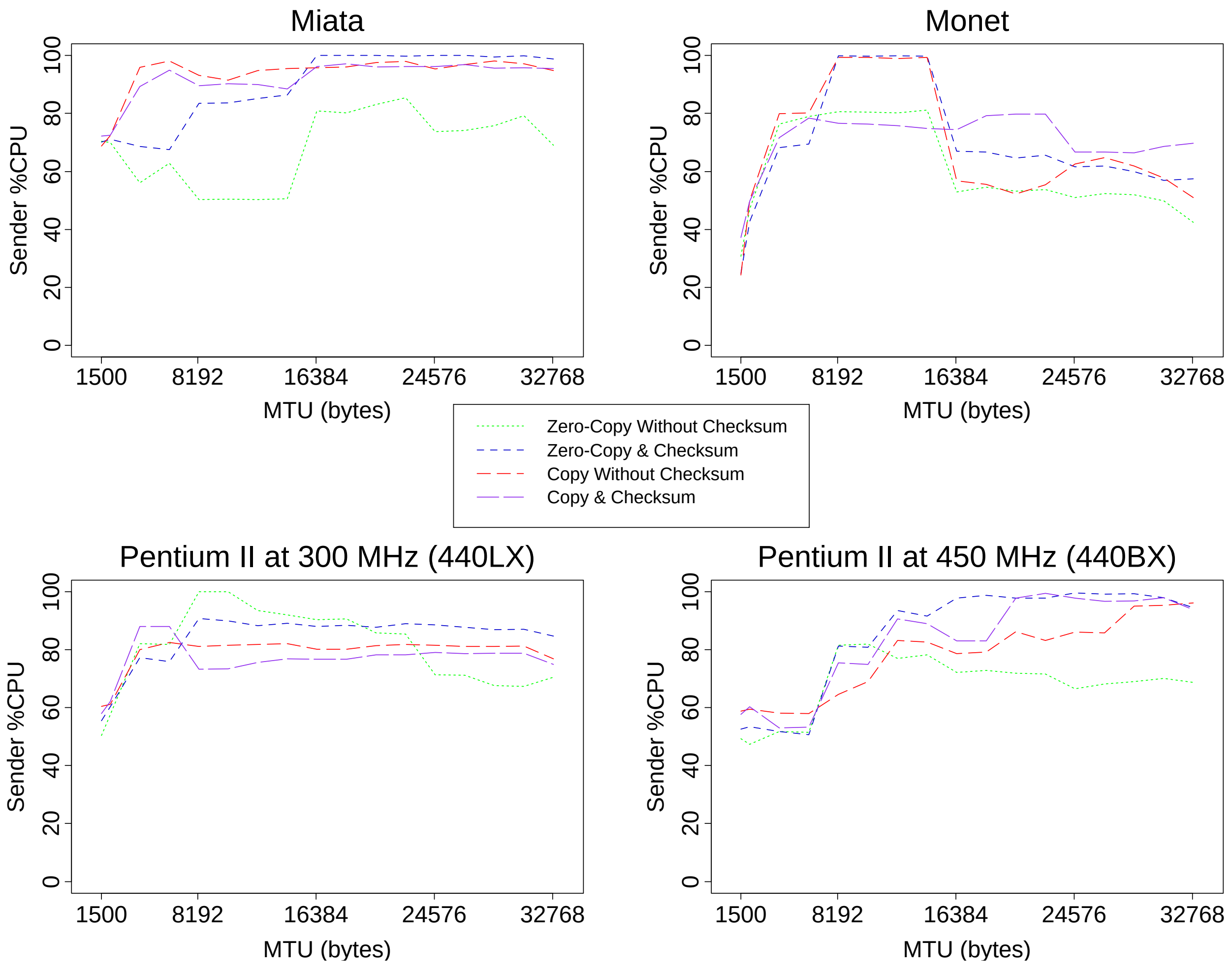

Figure 4: Sender CPU Utilization

ing and checksumming, interrupt handling, virtual memory costs (buffer page allocation and/or page remapping), Trapeze driver overheads, and TCP/IP protocol stack overheads. With a 1500-byte MTU the Miata is near 80% saturation at a bandwidth of 300 Mb/s. While the overhead can be reduced somewhat by checksum offloading and interrupt suppression, about 55% of CPU time is spent on unavoidable packet-handling overheads in the driver and TCP/IP stack, and data movement costs at the socket layer. With an 8KB payload, the bandwidth level has increased to 360 Mb/s, while CPU time spent in packet handling has dropped from 55% to 24%. Data movement overheads grow slightly due to the higher bandwidth, but the larger MTU introduces the opportunity to almost fully eliminate these overheads by enabling zero-copy optimizations. While the zero-copy optimization has some cost in VM page remapping, the reduced memory system contention causes other overheads to drop slightly, leaving utilizations in the 24% range if checksums are disabled (checksum offloading is not supported on the LANai-4 NICs used in this experiment). Again, these measurements reinforce the inadequacy of the 1500-byte standard Ethernet for high-speed networking, and the importance of the Jumbo Frames standard.

The rightmost set of bars in Figure 5 shows the overhead breakdown for 57K MTUs at a bandwidth of 390 Mb/s. While data movement overheads increase slightly due to the higher bandwidth, these costs can be eliminated with page remapping, which increases VM overheads but again causes other non-VM overheads to drop slightly due to reduced memory system contention. In the zero-copy experiment, the larger MTU does not affect VM page remappings at all relative to the 8KB MTU, since these costs are proportional to the number of pages of data transferred. However, per-packet TCP/IP and driver overheads drop from 8% to just 3% of CPU, even as bandwidth increases by about 10%. The Miata can handle the 390 Mb/s of bandwidth with a comfortable 10% CPU utilization.

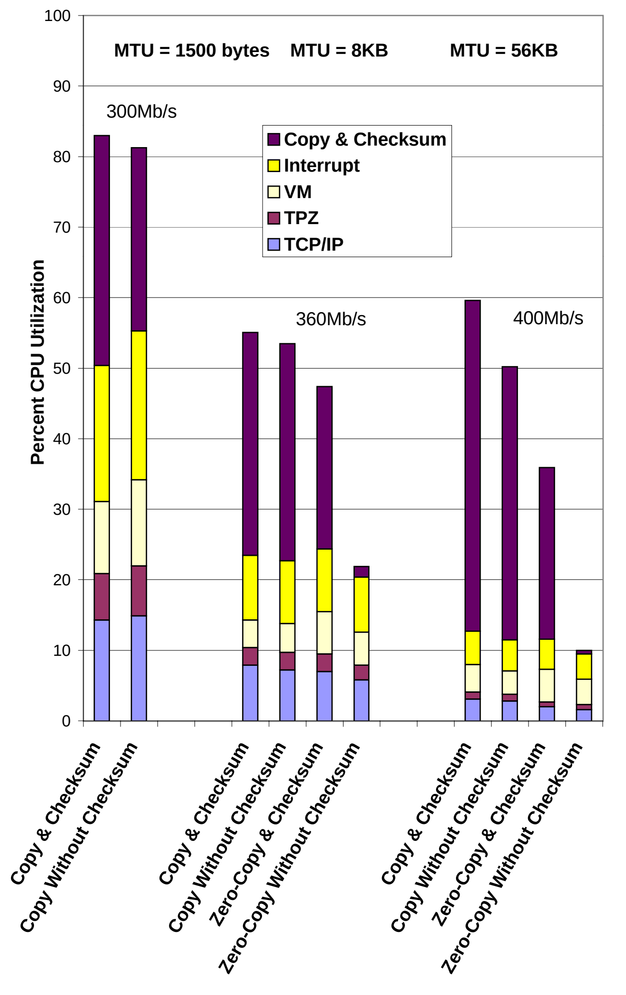

Figure 5: TCP Receiver CPU Utilization Breakdown

3.4 UDP Latency

Figure 6 shows the one-way UDP latency for various packet sizes using Trapeze and the Alteon Gigabit Ethernet on Monet. These results were obtained with *netperf -tUDP_RR -l60*, which ping-pongs a packet of the requested size for one minute. We plotted points for one half of the average round-trip latency for one-byte packets and for packet sizes of 1KB to 8KB in 1KB increments. For these experiments we disabled checksum offloading on the Alteon after observing unexpectedly high latencies with checksum offloading enabled. The comparison is fair because Trapeze does not support checksum offloading on this platform. The two sets of lines for each configuration show the latency with software checksums and with checksums disabled. Zero-copy sockets are responsible for the lower latency in all configurations when the 8KB page size is reached.

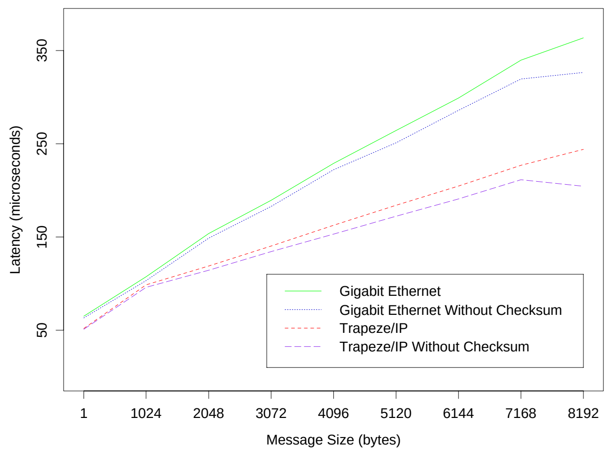

Figure 6: UDP One-Way Message Latency

In the Trapeze runs, the slope change at 1KB results from use of a payload buffer. The one-byte Trapeze packets are sent in a control message with no payload buffer, resulting in lower latency. For the 1KB-7KB sizes, the data is copied into a payload buffer and sent as a Trapeze payload.

The latency results show the benefits of message pipelining in Trapeze, which overlaps transfers on the link with transfers on the sender and receiver I/O bus. This overlap causes packet latencies to grow at a slower rate as packet size increases. In fact, the experiment understates these benefits because message pipelining is supported only on the receiver on the LANai-5 NIC used in this configuration, due to a change in the meaning of certain control registers on the LANai-5.

4 Conclusion

The experiments reported in this paper give a quantitative snapshot of the state of the art for TCP/IP networking performance on current-generation desktop-class PCs and workstations and gigabit-per-second networks.

Our measurements are taken from the standard high-quality TCP/IP implementation in the FreeBSD 4.0 kernel, supplemented with support for a range of techniques to reduce communication overheads. These include zero-copy sockets and several features implemented in the Trapeze firmware for Myrinet, including large MTUs with scatter/gather I/O, page-aligned payload buffers, adaptive message pipelining, interrupt suppression, and checksum offloading. With these features, we have measured TCP/IP bandwidths of 956 Mb/s using Trapeze/Myrinet and 988 Mb/s using an Alteon Gigabit Ethernet network. These are the highest TCP bandwidths on public record at present. The 500 MHz Alpha 21264 platform is capable of handling these bandwidths with CPU overheads as low as 20%.

5 Acknowledgments

The equipment used in this work was provided by the National Science Foundation, Intel Corporation, and Myricom Inc. We would like to thank Bob Felderman of Myricom for his frequent technical assistance, and Myricom for making LANai-5 prototypes available. We also thank Ted Schroeder and Alteon Networks for their support and for releasing Tigon-II documentation and firmware to research groups.

The FreeBSD port to the Alpha platforms is due to Doug Rabson. Don Rice at Compaq Workstation Engineering has given us invaluable assistance with the FreeBSD/Alpha Monet platform support. Bill Paul wrote the FreeBSD driver for the Tigon-II Gigabit Ethernet cards. We thank Phillip Ezolt and Digital/Compaq's High Performance Servers/Benchmark Performance Engineering group for making iprobe available as open source.

Jenise Swall introduced us to S-Plus and provided much-needed assistance in coercing this tool to produce our graphs.

6 Availability

Trapeze is available in source form with a BSD-style copyright from *http://www.cs.duke.edu/ari/trapeze*. Our *iprobe* port to FreeBSD/Alpha is available from *http://www.cs.duke.edu/ari/iprobe.html*. FreeBSD extensions (including Monet platform support, Alteon driver extensions, and zero-copy sockets) are incorporated into the FreeBSD code base or are available from the Trapeze Web site.

References

[1] Darrell Anderson, Jeffrey S. Chase, Syam Gadde, Andrew J. Gallatin, Kenneth G. Yocum, and Michael J. Feeley. Cheating the I/O bottleneck: Network storage with Trapeze/Myrinet. In *Proceedings of the 1998 Usenix Technical Conference*, June 1998. Duke University CS Technical Report CS-1997-21.

[2] Jennifer Anderson, Lance Berc, Jeff Dean, Sanjay Ghemawat, Monika Henzinger, Shun-Tak Leung, Mark Vandevoorde, Carl Waldspurger, and Bill Weihl. Continuous profiling: Where have all the cycles gone? In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles (SOSP)*, October 1997.

[3] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W-K Su. Myrinet - a gigabit-per-second local area network. *IEEE Micro*, February 1995.

[4] José Carlos Brustoloni and Peter Steenkiste. Effects of buffering semantics on I/O performance. In *Proc. of the Second Symposium on Operating Systems Design and Implementation*, pages 277–291, Seattle, WA, October 1996. USENIX Assoc.

[5] Jeffrey S. Chase, Andrew J. Gallatin, Alvin R. Lebeck, and Kenneth G. Yocum. Trapeze messaging API. Technical Report CS-1997-19, Duke University, Department of Computer Science, November 1997.

[6] Zubin D. Dittia, Guru M. Parulkar, and Jerome R. Cox. The APIC approach to high performance network interface design: Protected DMA and other techniques. In *Proceedings of IEEE Infocom*, 1997. WUCS-96-12 technical report.

[7] Hsiao-Keng and Jerry Chu. Zero-copy TCP in Solaris. In *Proceedings of the USENIX 1996 Annual Technical Conference*, January 1996.

[8] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Unix Operating System*. Addison Wesley, Reading, MA, 1996.

[9] Kenneth G. Yocum, Darrell C. Anderson, Jeffrey S. Chase, Syam Gadde, Andrew J. Gallatin, and Alvin R. Lebeck. Adaptive message pipelining for network memory and network storage. Technical Report CS-1998-10, Duke University Department of Computer Science, April 1998.

[10] Kenneth G. Yocum, Jeffrey S. Chase, Andrew J. Gallatin, and Alvin R. Lebeck. Cut-through delivery in Trapeze: An exercise in low-latency messaging. In *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing (HPDC-6)*, pages 243–252, August 1997.