End-System Optimizations for High-Speed TCP

Jeffrey S. Chase, Andrew J. Gallatin, and Kenneth G. Yocum[*]
Department of Computer Science
Duke University
Durham, NC 27708-0129
{*chase, grant*}*@cs.duke.edu, gallatin@freebsd.org*

Abstract

Delivered TCP performance on high-speed networks is often limited by the sending and receiving hosts, rather than by the network hardware or the TCP protocol implementation itself. In this case, systems can achieve higher bandwidth by reducing host overheads through a variety of optimizations above and below the TCP protocol stack, given support from the network interface. This paper surveys the most important of these optimizations and illustrates their effects quantitatively with empirical results from an experimental network delivering up to two gigabits per second of end-to-end TCP bandwidth.

1 Introduction

Modern TCP/IP implementations can transfer data at a high percentage of available network link bandwidth, reflecting the success of many years of refinements. On the fastest networks, end-to-end (application-to-application) throughput is often limited by the capability of the end systems to generate, transmit, receive, and process the data at network speeds. Delivered performance is determined by a combination of factors relating to the host hardware, interactions between the host and the network adapter, and host system software.

This paper explores the end-system factors that can limit bandwidth for TCP on high-speed networks and the techniques to overcome those limitations. It is tempting to suppose that the advances in CPU power given by Moore's Law will render these limitations increasingly irrelevant, but this is not the case. The limiting factor is not CPU processing power but the ability to move data through the host I/O system and memory. Wider datapaths can improve raw hardware bandwidth, but more bandwidth is invariably more expensive for a given level of technology. Advances in network bandwidth follow a step function, but the fastest networks tend to stay close to the limits of the hosts. Given a sufficiently fast network, achievable TCP performance depends on optimizations to minimize networking overheads. With Gigabit Ethernet widely deployed and 10 Gb/s Ethernet on the horizon, these optimizations are highly relevant today.

This paper focuses on approaches to low-overhead networking that are now emerging into common practice. The key techniques discussed are interrupt coalescing, checksum offloading, and zero-copy data movement by page remapping. We also explore the effect of larger packet sizes, e.g., the Jumbo Frames standard for Ethernet. The network interface plays a key role for each of these features, and an increasing number of commercial network adapters support them. We quantify their performance potential and briefly outline alternative structures for high-speed TCP/IP networking that could offer similar benefits in a more general way for the next generation of adapters and systems: user-level network interfaces, scatter-gather I/O frameworks, TCP/IP protocol offloading, and Remote Direct Memory Access (RDMA).

This paper is organized as follows. Section 2 gives an overview of approaches to low-overhead TCP/IP communication, and their implications for the network interface and operating system software. Section 3 describes a complete system for high-speed TCP, with more detail on the interactions between the network interface and the operating system for zero-copy data movement and checksum offloading. Section 4 presents empirical results from two platforms on an experimental 2.5 Gb/s network, illustrating the impact of these techniques. The results provide a quantitative snapshot of the current state of the art for end-to-end TCP communication, yielding insights into networking behavior on the next generation of networks and hosts. Section 5 concludes.

[*]This work is supported by the National Science Foundation (EIA-9870724 and EIA-9972879), Intel Corporation, and Myricom.

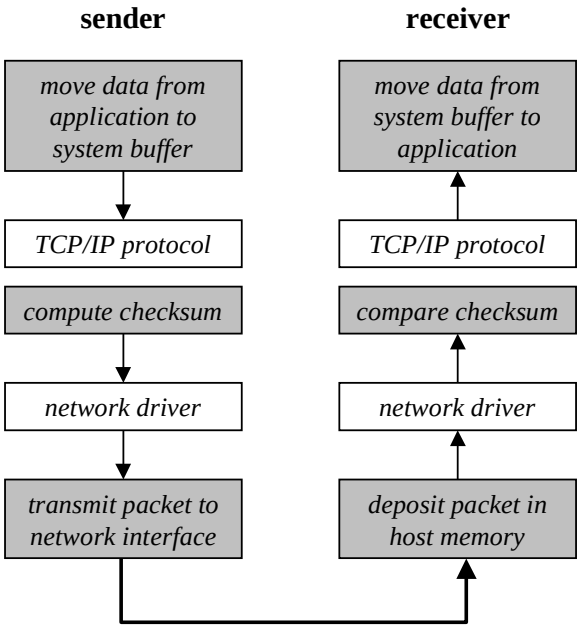

Figure 1: Sources of end-system overhead for TCP/IP.

2 End-System Optimizations

This section outlines the factors that affect end-system performance for TCP, and gives an overview of the key optimizations proposed and studied in the literature.

Data transmission using TCP involves the host operating system (OS) facilities for memory and process management, as well as the TCP/IP protocol stack and the network device and its driver. Figure 1 depicts the key sources of overhead in transmitting data over a TCP connection in a typical system. End systems incur CPU overhead for processing each network packet or frame. These *per-packet* costs include the overhead to execute the TCP/IP protocol code, allocate and release memory buffers, and field device interrupts for packet arrival and transmit completion. The systems incur additional costs for each byte of data sent or received. These *per-byte* costs — incurred at the stages shown in grey in Figure 1 — include overheads to move data within the end system and to compute checksums to detect data corruption in the network.

At a given packet size, faster networks carry more packets as well as more bytes per unit of time, increasing both per-packet and per-byte overheads on the host. At high speeds these costs may consume a significant share of host CPU cycles and memory system bandwidth, siphoning off resources needed for application processing of the data. The end system ultimately saturates under the combined load of application processing and networking overhead, limiting delivered bandwidth. Faster CPUs alone do not help appreciably if the CPU spends most of its time waiting for memory. Given sufficient memory system bandwidth, multiprocessors or multithreading may improve performance to the extent that the software is able to extract parallelism from the network processing; in particular, multiprocessors may yield higher *aggregate* throughputs for servers handling multiple concurrent streams.

To deliver the maximum performance of the hardware, the network interface and OS software must cooperate to minimize both classes of overheads at the end systems. The following subsections give an overview of the approaches to reducing per-packet and per-byte overheads respectively, using optimizations on the network interface and in host system software above and below the TCP/IP protocol stack. This can improve the peak bandwidth of individual TCP streams as well as the aggregate throughput across multiple streams.

2.1 Reducing Per-Packet Overheads

Extended frames. One way to limit per-packet overheads is to use larger packets, reducing the number of packets needed to carry a given amount of data. For example, standard IEEE 802.3 Ethernet frame sizes impose a Maximum Transmission Unit (MTU) of 1500 bytes, but many Gigabit Ethernet vendors have followed a proposal by Alteon Networks to support "Jumbo" frame sizes up to 9000 bytes. Up-to-date TCP/IP implementations automatically discover and use the largest MTU accepted by the complete network path for a given connection. For bulk data transfer over TCP, the MTU is the typical unit of data exchange between the host and the adapter. It is this MTU that affects end-system overhead, independent of the transfer sizes actually carried on the links; for example, ATM networks may transparently segment and reassemble large IP packets for transport as 53-byte cells, but the small cell size does not impact TCP/IP performance. Section 4 quantifies end-system performance for MTU sizes up to 32KB.

Interrupt Coalescing. A second way to limit per-packet overhead is to amortize packet handling costs across multiple packets. For example, interrupts to notify the host of transmit-complete events and packet arrival can impose a significant overhead. Most high-speed network interfaces support *interrupt coalescing* or *interrupt suppression*, selectively delaying interrupts if more packets are pending transmission or delivery to the host [6]. This reduces the total number of interrupts delivered; the host interrupt handler may process multiple event notifications from each interrupt. While some interrupt coalescing schemes increase end-to-end latency slightly, they reduce interrupt costs during periods of high bandwidth demand, and may deliver better cache performance for processing the notifications.

2.2 Reducing Per-Byte Overheads

Per-byte costs dominate per-packet costs for bulk data transfer because they are fundamentally limited by the host memory system rather than the CPU. The only unavoidable per-byte costs occur in the application and in moving data between host memory and the network interface. An ideal system would entirely eliminate the remaining per-byte overheads.

Zero-copy networking. In principle, it is possible to eliminate all data copying within the host. Operating systems need not copy data to make it available to the network interface because advanced network devices support scatter/gather DMA, in which the data for a frame may span multiple regions of host memory. The OS can avoid other internal data copying with flexible mechanisms for buffer management.

More problematic is the copying of data between application memory and OS buffers. On the sender, copying the data protects it from modification if the process reuses the memory before the transmit of the old data completes. On the receiver, copying allows the OS to place the incoming data at arbitrary virtual addresses specified by the application. Avoiding this copying involves complex relationships among the network hardware and software, OS buffering schemes, and the application programming interface (API).

A basic technique for copy avoidance is to use virtual memory to change buffer access or address bindings without copying the data to a new buffer. For example, virtual memory allows the system to protect against modifications to buffers with pending transmits, even if they are accessible in the process address space. Section 3.1 discusses a key technique — *page remapping* or *page flipping* — in detail, and outlines an implementation that is compatible with a standard *socket* API for networking. Research systems have used page remapping for several years [1, 7], and it is now emerging into more common use. IO-Lite [9] generalizes this technique by replacing sockets with a new scatter/gather I/O API in which the OS rather than the application selects the addresses for incoming data.

Checksum offloading. To detect data corruption in the network, TCP/IP generates an end-to-end checksum for each packet covering all protocol headers and data. In most implementations, software running on the host CPU computes these checksums on both sending and receiving sides. This requires the CPU to load all of the data through the memory hierarchy for a sequence of add operations. One way to eliminate this cost is to compute the checksum in hardware on the network adapter's DMA interface as the packet passes through on its way to or from host memory [8]. The TCP/IP stack and the network device driver must cooperate in checksum offloading, as discussed in Section 3.2.

Integrated copy/checksum. Another approach to reducing per-byte costs is to coalesce multiple operations that touch the data, completing multiple steps in a single traversal of the packet [4]. For example, some TCP/IP implementations integrate checksumming with the data copy to or from application memory. A similar approach could apply to other processing steps such as compression or encryption.

2.3 The Role of the Network Interface

Most of the optimizations outlined above require some support from the network adapter [6]. Advanced network interfaces can help reduce end-system overheads by supporting larger packets at the host interface, segmenting and reassembling packets as needed to match the link frame sizes, depositing packets in host memory intelligently, coalescing interrupts, or incorporating checksum hardware. There is a rebirth of interest in supporting TCP/IP protocol functions directly on the adapter (*TCP offloading*), with several products on the market and under development.

All techniques to avoid copying of incoming data rely on the network interface to process the incoming packet stream and deposit the data in suitable host memory buffers. For page remapping with the Unix socket API, the adapter must (at minimum) separate incoming packet headers from their payloads and deposit the payloads into buffers aligned to the system virtual memory page size. The OS may then deliver the data to an application buffer by updating virtual mappings to reference the payload's physical location, if the virtual address of the application buffer is also suitably aligned. If the MTU is smaller than a page, the adapter would need to recognize the TCP connection for each incoming packet and "pack" payloads into page-size buffers for each connection.

Several recent proposals promise more general support for low-overhead networking by a comprehensive restructuring of the end systems. IO-Lite's redesigned API allows more flexible data placement and reduces copying for inter-process communication and storage access as well [9]. Another approach termed *Remote Direct Memory Access (RDMA)* inserts directives into the packet stream to name the buffer destinations for incoming data; the receiving adapter recognizes these directives and "steers" the data directly into its buffers. RDMA is similar to *sender-based memory management* [2]: it handles arbitrary MTUs and buffer alignments, and supports protocols layered above TCP, e.g., for network storage access. RDMA features are already present in the Virtual Interface Architecture (VI); VI supports *user-level networking* [10] in which applica-

tions interact directly with the network adapter, bypassing the kernel buffering system entirely. Adapters supporting the VI standard over TCP/IP networks have been announced.

3 A System for High-Speed TCP

This section outlines a complete experimental system for high-speed TCP, in order to illustrate the OS issues and host/interface interactions for zero-copy networking and checksum offloading.

The experimental system is based on FreeBSD 4.0, a free Unix system descending from the Berkeley 4.4 BSD code base. We chose FreeBSD in part because it has a high-quality network subsystem incorporating all relevant TCP/IP refinements. We modified FreeBSD to support zero-copy sockets and checksum offloading, and added code for integrated copy/checksum from a recent Linux release.

The network interface used in our experiments is Trapeze/Myrinet [3]. Myrinet (*http://www.myri.com*) is a reliable short-haul interconnect with very low latency, configurable maximum frame sizes, and link speeds up to 2.5 Gb/s in each direction. Trapeze is a firmware program for Myrinet adapters. Trapeze and its associated network driver incorporate support for the optimizations discussed in Section 2.1 and 2.2. While Trapeze/Myrinet is not representative of networks encountered by TCP systems "in the wild," it is ideal for probing the limits of end-system hardware and software for high-speed networking.

The FreeBSD extensions described in this section are compatible with network interfaces other than Trapeze. In particular, we have tested the system with an Alteon Gigabit Ethernet network, yielding TCP bandwidths just under the maximum link speed.

3.1 Zero-Copy Sockets

The primary support for zero-copy TCP/IP (page remapping) resides at the socket layer in the OS kernel. The optimizations trigger only if the data transfer sizes requested by the application are larger than the page size. On the receiver, page remapping requires an MTU matched to the page size, page-aligned application buffers, and a network interface that deposits packet data on a page boundary. In general, this last requirement means that the network interface must split the packet header from its data for common protocols such as TCP.

Page remapping extends the conventional FreeBSD send/receive path, which is based on variable-sized kernel network buffers (*mbufs*). Standard *mbufs* contain their own buffer space, while an *external mbuf* holds a reference to another kernel buffer, e.g., a file buffer or a physical page frame. 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 buffers in the chain. On a normal transmission, the socket layer copies data from user memory 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 copies the payload into user memory and frees the chain.

Page remapping is implemented in a variant of the kernel routine *uiomove*, 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. The new code activates when a process requests the kernel to transfer a page or more of data between a network socket and a page-aligned user buffer. Instead of copying data to or from the *mbuf* chain, *uiomove* passes the data by reference.

On the sender, *uiomove* creates a new external *mbuf* that references the page frame backing the application's virtual buffer. The *mbuf* chain and its pages are then passed through the TCP/IP stack to the network driver, which attaches the pages to outgoing messages as payloads. In Unix systems, the page remapping scheme must preserve the copy semantics of the existing socket interface [1, 7]. The application may overwrite its send buffer after it requests the send but before the TCP acknowledgement is received, completing the transmit. To handle this case, the kernel marks pages with pending transmit as *copy-on-write*, disabling write access to the page. The copy-on-write mapping is released when the receiver acknowledges the data.

On the receiver, *uiomove* installs a new virtual translation for the page frame containing the data, as referenced by the external *mbuf* passed up through the protocol stack from the network driver. Copy-on-write is unnecessary because there is no need to retain the kernel buffer after the read; *uiomove* simply releases the *mbuf* headers, leaving the buffer page mapped into the application address space. It also releases any pages left unmapped by the remapping operations, preserving equilibrium between the process and the kernel buffer pool.

3.2 Checksum Offloading

The network interface and its driver act in concert to implement checksum offloading. The Myrinet and Alteon devices support checksum offloading in the host-PCI DMA engine, which computes the raw 16-bit one's complement checksum of each DMA transfer as it moves

data to and from host memory. It is not necessary to change the TCP/IP stack to use this checksum; simply setting a flag in the *mbuf* chain bypasses the software checksum. However, three factors complicate hardware checksumming for IP:

- A packet's data may span multiple host buffers; the device accesses each buffer with a separate DMA. On the sender, Trapeze and Alteon adapters combine these partial checksums in firmware using one's complement addition.
- TCP and UDP actually use 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.

The last issue may require a compromise between host overhead and packet latency, since the adapter must handle large packets in a store-and-forward fashion.

4 Empirical Results

This section presents quantitative results showing the impact of zero-copy networking, checksum offloading, integrated copy/checksum, and MTU size (extended frames) on TCP performance. We focus on point-to-point bandwidth in a small Myrinet LAN, where the stresses on the end systems are greatest; of course the results apply to WAN networks if they deliver sufficient bandwidth to stress the hosts. We tested two hardware configurations:

- **Dell PowerEdge**. The Dell PowerEdge 4400 servers use a 733 MHz Intel Xeon CPU (32KB L1 cache, 256KB L2 cache), and a ServerWorks ServerSet III LE chipset. Each machine has 2-way interleaved RAM and an M2M-PCI64B Myrinet adapter connected to a 66 MHz 64-bit PCI slot. The Myrinet adapter is a pre-production prototype based on the LANai-9 chip; it has a 133 MHz processor supporting the Myrinet-2000 link speed of 250 MB/s (2 Gb/s). The prototype adapters run in a test mode with a link speed of 320 MB/s.
- **Compaq Monet**. The Compaq XP1000 Professional Workstations have a 500 MHz Alpha 21264 CPU, a 4MB L2 cache, and the Digital 21272 "Tsunami" chipset. Each machine has an M2M-PCI64A-2 Myrinet adapter connected to a 33 MHz 64-bit PCI slot and an M2M-SW16 Myrinet switch. The Myrinet adapter is based on the LANai-7 chip; it has a 66 MHz processor supporting a link speed of 160 MB/s (1.28 Gb/s).

All machines run extended FreeBSD 4.0 kernels with Trapeze firmware on the Myrinet adapters, as described in Section 3. The machines are configured with sufficient buffer memory to maintain the link speed (512 KB socket buffers). All results are from a unidirectional data transfer over a single TCP connection. A single connection delivers enough bandwidth to saturate the receiver, so multiple streams cannot deliver better aggregate performance. The application is a modified version of *netperf* version 2.1pl3, a standard tool for benchmarking TCP/IP performance. The machines are isolated and otherwise idle.

4.1 TCP Bandwidth

Figure 2 shows TCP bandwidth on the PowerEdge platform as a function of the MTU size. The MTUs allowed payloads of 1500 bytes (standard Ethernet), 4KB (FDDI), 8KB (Ethernet with Jumbo Frames), and multiples of 8KB up to 32 KB. The points are averages from 50 *netperf* runs that each send data as fast as the system allows for 60 seconds, then compute the average bandwidth over the interval. The left-hand graph is from a standard *netperf* that does not access its data; the right-hand graph is from a modified version that accesses all data using longword stores on the sender and longword loads on the receiver.

For the baseline bandwidth results (labeled *no optimizations*) the end systems copy the data in the conventional fashion and compute all checksums in software. The copy and checksum overhead saturates the host CPUs well below the link speed; bandwidth peaks at 824 Mb/s, and stays below 630 Mb/s when *netperf* accesses its data.

The lines labeled *zero-copy* show the effect of enabling page remapping. The size and alignment of *netperf*'s buffers and transfer requests allow the systems to avoid almost all copy operations for MTUs equal to or

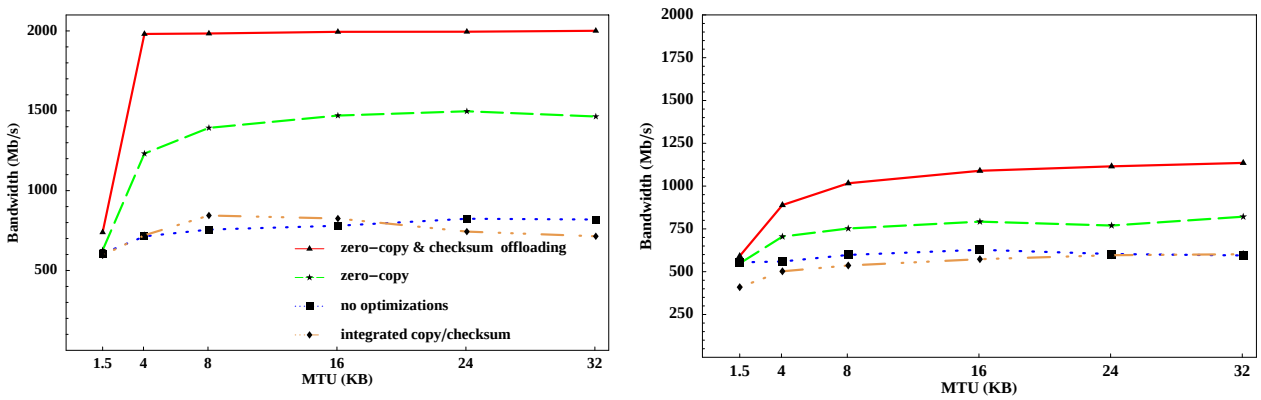

Figure 2: TCP Bandwidth on the PowerEdge 4400 platform. The left-hand graph shows bandwidth when the application process does not access the data. The right-hand graph shows the impact of this access on delivered bandwidth.

larger than the 4KB system page size. Moving from a 1500-byte to a 4KB MTU enables page remapping, yielding a sudden jump in bandwidth. This illustrates the maximum benefit from page remapping or other zero-copy networking enhancements on this platform. Bandwidth is lower for MTUs that are not a multiple of the page size because the socket API must copy the "odd" data; Figure 2 omits these points for clarity.

Since checksums are computed in software for the *zero-copy* line, host CPU saturation again limits bandwidth, this time at a higher peak of 1.5 Gb/s. The right-hand graph shows that the percentage improvement from copy avoidance is lower when *netperf* accesses its data, yielding a peak of 820 Mb/s. This results from two factors. First, the additional load drives the CPU to saturation at a lower bandwidth, thus the CPU is spending less of its time copying data, diminishing the potential benefit from avoiding the copy. Second, memory caches create a synergy between copying and application processing; each may benefit from data left in cache by the other. Despite these factors, avoiding the copy yields a 26% bandwidth improvement at the 8KB MTU. We omitted the lines for checksum offloading with copy because the results are similar to *zero-copy*, although the benefit is slightly lower.

The lines labeled *zero-copy & checksum offloading* show the combined effect of page remapping and use of checksum hardware. In the left-hand graph the host CPUs never touch the data. Bandwidth jumps to 2 Gb/s at the 4KB MTU as page remapping replaces copying. At this point the host CPUs are near saturation from per-packet overheads including buffer management, interrupt handling, and TCP/IP protocol costs. In the right-hand graph *netperf* touches the data, yielding a much lower peak bandwidth of 1.18 Gb/s for *zero-copy & checksum offloading* at a 32KB MTU. This is somewhat slower than the left-hand *zero-copy* points (which checksum the data rather than touching it) because the sender stores to the data; on this Xeon P6-based platform, the CPU can store to memory at only 215 MB/s, but it can read at 504 MB/s. Of course these bandwidths are lower when the CPU is competing with the I/O system for memory bandwidth, as it is in this case.

The lines labeled *integrated copy/checksum* show the effect of combining the conventional copy and checksum into a single software loop, as implemented in Linux 2.3.99-pre8 kernels for this platform. Microbenchmark results confirm a bandwidth limitation of roughly 100 MB/s for this loop, about half the 4400's memory copy bandwidth of 207 MB/s. The integrated loop may actually reduce bandwidth with large MTUs on this platform, but we have not investigated why. What is important is that the expected benefits of integrating the copy and checksum can be elusive unless the loop is carefully tuned for specific platforms. In the left-hand graph, the integrated copy/checksum appears to offer a benefit at 8KB MTUs, but this effect disappears if the application touches the data.

4.2 Effect of Larger MTUs

Moving beyond 4KB along either x-axis in Figure 2 shows the effect of reducing total per-packet overhead by increasing the MTU. Larger MTUs mean fewer packets, and hence fewer interrupts and less protocol processing overhead. Varying MTU size also reveals the likely benefits of other approaches to reducing per-packet overheads, e.g., interrupt coalescing or TCP offloading.

The data clearly shows a "sweet spot" at the 8KB MTU, which approximates the Jumbo Frames proposal for Ethernet. Standard Ethernet 1500-byte MTUs never yield a bandwidth above 740 Mb/s in these experiments. In addition to reducing per-packet overheads, larger MTUs enable page remapping optimizations for sockets

on the receiver, potentially reducing per-byte overheads.

On this platform, bandwidth improvements from increasing MTU size beyond 8KB are modest. This is due to two factors. For the bottom two lines on both graphs, performance is dominated by per-byte overheads, memory system bandwidth, and caching behaviors. In the left-hand *zero-copy & checksum offloading* line, larger MTUs do not improve bandwidth beyond 2 Gb/s because the bottleneck shifts to the network interface CPU; however, host CPU utilization at this speed drops to 45% with the 32KB MTU. Larger MTUs do yield improvements for the three remaining *zero-copy* cases, but this effect diminishes with larger packet sizes as data movement overheads increasingly dominate. The effect of larger MTUs is more apparent from a breakdown of how the CPU spends its time.

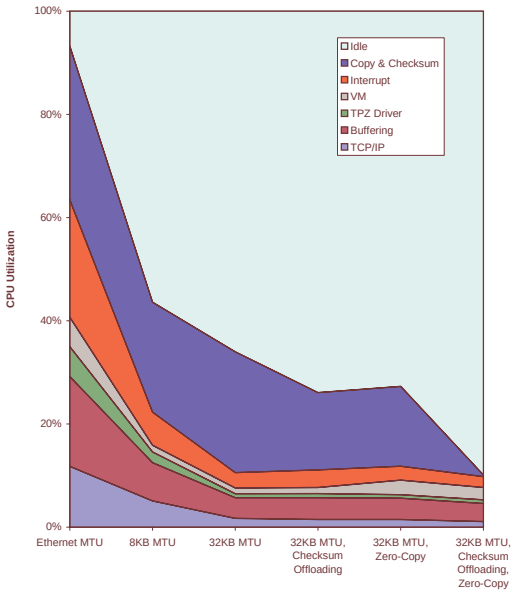

Figure 3: TCP Receiver CPU Utilization Breakdown

4.3 CPU Utilization

Figure 3 shows a breakdown of receiver CPU utilization on the Monet platform in order to better illustrate the overheads for high-speed TCP. The data is from *iprobe_suite-4.3* (Instruction Probe), an on-line profiling tool from Compaq that uses the Alpha on-chip performance counters to report detailed execution profiles with low overhead (3%-5%). We do not report bandwidth results from Monet because its I/O bus is slower than the Myrinet-2000 link speed, but the effects of networking overhead are qualitatively similar on the two platforms.

Figure 3 classifies overheads into five categories: data access for copying and checksums, interrupt handling, virtual memory costs (buffer page allocation and/or page remapping), network buffer management, the TCP/IP protocol stack, and the Trapeze network driver. The y-axis in Figure 3 gives the CPU utilization attributed to each overhead category. The x-axis gives six different configurations to show the effect of varying the MTU size and enabling page remapping and checksum offloading. For these experiments TCP bandwidth was held constant at 370 Mb/s by a slow sender, so it is possible to directly compare receiver CPU overheads for different configurations.

With a standard Ethernet MTU the Monet is near 95% saturation. The CPU spends about 65% of its time processing 30,000 packets per second through the network driver and TCP/IP stack. Total packet handling costs drop to 22% with an 8KB MTU, and to 10% with a 32KB MTU. *Iprobe* attributes approximately 20% of CPU time to interrupt dispatching at a 1500-byte MTU, and 5% at an 8KB MTU; in each case it is 20% to 25% of total overhead on the Alpha, showing that interrupt coalescing is an important optimization at higher bandwidths, even with Jumbo Frames. It is interesting to note that actual TCP/IP protocol overhead accounts for at most 12% of CPU time even with a 1500-byte MTU. Protocol costs are significant but are not the limiting factor for a well-implemented TCP/IP stack [5].

The last three configurations in Figure 3 show the effects of checksum offloading and page remapping on per-byte overheads, with per-packet overheads held low by the 32KB MTU. Avoiding the copy or offloading the checksum drops the total CPU time spent on data access from 24% to roughly 15%. For page remapping (*zero-copy*), this effect is partially offset by per-page costs to manipulate translations in the virtual memory system; this overhead totals about 3% of CPU time and is independent of MTU size. Applying both optimizations effectively eliminates all data access costs, and the reduced memory contention causes other overheads to drop slightly; the Monet handles the 370 Mb/s of bandwidth with a comfortable 10% CPU utilization.

5 Conclusion

Technology trends suggest that the fastest networks will continue to carry link bandwidths close to the memory and I/O subsystem limits on most hosts. Thus the end systems remain the bottlenecks to high-speed TCP/IP performance in current and future hardware.

To achieve end-to-end TCP/IP networking performance on the fastest networks, it is necessary to minimize the host overheads to handle packets and their data. This paper reviews the techniques used by operating systems and high-speed network interfaces to reduce end-system overheads. The approaches to reducing overheads include larger frame sizes (MTUs), inter-

rupt coalescing, copy avoidance by page remapping, integrated copy/checksum, and hardware checksum computation. These optimizations increase delivered bandwidth by delaying saturation of the host CPUs, while leaving more CPU power for the application.

The experiments reported in this paper give a quantitative snapshot of a high-quality TCP/IP stack supplemented with these optimizations, running at bandwidths up to 2 Gb/s on current hardware. At these speeds, 8KB frame sizes (e.g., Jumbo Frames for Ethernet) yield bandwidth improvements up to 70%, relative to standard 1500-byte Ethernet frame sizes. However, there is little benefit to increasing frame sizes beyond 16KB. In our system, per-packet overheads are dominated by operating system buffer management, which is more expensive than packet processing in the network driver and TCP/IP protocol stack. Interrupt handling constitutes up to 20% of the per-packet overhead on Alpha systems with an 8KB frame size, showing the value of interrupt coalescing. With 8KB and larger MTUs, per-byte costs dominate per-packet costs on the platforms we tested, showing that copy avoidance and hardware checksum offloading are the most important optimizations. Taken together, these yield bandwidth improvements up to 70%, and can double bandwidth if application processing is ignored.

Acknowledgments

The equipment used in this work was provided by the National Science Foundation, Intel Corporation, and Myricom Inc. We thank Bob Felderman of Myricom for his frequent technical assistance, and Myricom for making Myrinet-2000 prototypes available. Darrell Anderson and the anonymous IEEE reviewers offered valuable comments on the paper.

Biographical Summaries

Jeffrey S. Chase joined the Computer Science faculty at Duke University in 1995. He holds a Bachelor's degree in Mathematics and Computer Science from Dartmouth College, and a Ph.D. in Computer Science from the University of Washington. His research interests are operating systems, distributed systems, networking, and storage systems.

Andrew J. Gallatin is a research staff member in the Network Storage Lab at Duke University. He holds an M.S. degree in Computer Science from Rensselaer Polytechnic Institute and a Bachelor's degree from SUNY Buffalo. He is a frequent contributor to the FreeBSD Project's Alpha port, and is working to incorporate the features described in this article into FreeBSD-5.0

Kenneth G. Yocum is a doctoral student in Computer Science at Duke University. His research focuses on high-speed network storage and Internet service virtualization. He holds a Bachelor's degree in Computer Science from Stanford University. He is the primary developer of the Trapeze networking system.

References

[1] J. Brustoloni. Interoperation of copy avoidance in network and file I/O. In *Proceedings of IEEE Infocom*, pages 534–542, 1999.

[2] Greg Buzzard, David Jacobson, Milon Mackey, Scott Marovich, and John Wilkes. An implementation of the Hamlyn sender-managed interface architecture. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*. USENIX Association, October 1996.

[3] Jeffrey S. Chase, Darrell C. Anderson, Andrew J. Gallatin, Alvin R. Lebeck, and Kenneth G. Yocum. Network I/O with Trapeze. In *1999 Hot Interconnects Symposium*, August 1999.

[4] David Clark and David Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the SIGCOMM Conference*, September 1990.

[5] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of TCP processing overhead. *IEEE Communications*, 27(6):23–29, June 1989.

[6] Peter Druschel, Larry L. Peterson, and Bruce S. Davie. Experiences with a high-speed network adaptor: A software perspective. In *Proceedings of the ACM SIGCOMM Conference*, pages 2–13, August 1994.

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

[8] K. Kleinpaste, P. Steenkiste, and B. Zill. Software support for outboard buffering and checksumming. In *Proceedings of the ACM SIGCOMM Conference*, August 1995.

[9] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. *ACM Transactions on Computer Systems (TOCS)*, 18(1):37–66, February 2000.

[10] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A user-level network

interface for parallel and distributed computing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, 1995.

Keywords: *TCP, high-speed network, network adapter, zero-copy, copy avoidance, checksum, operating system, memory management, socket*.