Sockets vs RDMA Interface over 10-Gigabit Networks: An In-depth analysis of the Memory Traffic Bottleneck*

Pavan Balaji
Computer Science and Engg.,
The Ohio State University,
Columbus, OH 43210,
balaji@cse.ohio-state.edu

Hemal V. Shah
Intel Corporation,
PTL1, 5000 Plaza on the Lake,
Austin, Texas 78746,
hemal.shah@intel.com

D. K. Panda
Computer Science and Engg.,
The Ohio State University,
Columbus, OH 43210,
panda@cse.ohio-state.edu

Abstract

The compute requirements associated with the TCP/IP protocol suite have been previously studied by a number of researchers. However, the recently developed 10-Gigabit Networks such as 10-Gigabit Ethernet and InfiniBand have added a new dimension of complexity to this problem, Memory Traffic. While there have been previous studies which show the implications of the memory traffic bottleneck, to the best of our knowledge, there has been no study which shows the actual impact of the memory accesses generated by TCP/IP for 10-Gigabit networks. In this paper, we do an in-depth evaluation of the various aspects of the TCP/IP protocol suite including the memory traffic and CPU requirements, and compare these with RDMA capable network adapters, using 10-Gigabit Ethernet and InfiniBand as example networks. Our measurements show that while the host based TCP/IP stack has a high CPU requirement, up to about 80% of this overhead is associated with the core protocol implementation especially for large messages and is potentially offloadable using the recently proposed TCP Offload Engines. However, the host based TCP/IP stack also requires multiple transactions of data over the current moderately fast memory buses (up to a factor of four in some cases), i.e., for 10-Gigabit networks, it generates enough memory traffic to saturate a typical memory bus while utilizing less than 35% of the peak network bandwidth. On the other hand, we show that the RDMA interface requires up to four times lesser memory traffic and has almost zero CPU requirement for the data sink. These measurements show the potential impacts of having an RDMA interface over IP on 10-Gigabit networks.

1 Introduction

High-speed network interconnects that offer low latency and high bandwidth have been one of the main reasons attributed to the success of commodity cluster systems. Some of the leading high-speed networking interconnects include Ethernet [11, 1], InfiniBand [2], Myrinet [7] and Quadrics [18]. Two common features shared by these interconnects are *User-level networking* and *Remote Direct Memory Access (RDMA)*. Gigabit and 10-Gigabit Ethernet offer an excellent opportunity to build multi-gigabit per second networks over the existing Ethernet installation base due to their backward compatibility with Ethernet. InfiniBand Architecture (IBA) is a newly defined industry standard that defines a System Area Network (SAN) to enable a low latency and high bandwidth cluster interconnect. IBA mainly aims at reducing the system processing overhead by decreasing the number of copies associated with the message transfer and removing the kernel from the critical message passing path.

The Transmission Control Protocol (TCP) is one of the universally accepted transport layer protocols in today's networking world. The introduction of gigabit speed networks a few years back had challenged the traditional TCP/IP implementation in two aspects, namely performance and CPU requirements. In order to allow TCP/IP based applications achieve the performance provided by these networks while demanding lesser CPU resources, researchers came up with solutions in two broad directions: user-level sockets [19, 3] and TCP Offload Engines [20]. Both these approaches concentrate on optimizing the protocol stack either by replacing the TCP stack with zero-copy, OS-bypass protocols such as VIA, EMP or by offloading the entire or part of the TCP stack on to hardware.

The advent of 10-Gigabit networks such as 10-Gigabit Ethernet and InfiniBand has added a new dimension of complexity to this problem, *Memory Traffic*. While there have been previous studies which show the implications of the memory traffic bottleneck, to the best of our knowledge, there has been no study which shows the actual impact of the memory accesses generated by TCP/IP for 10-Gigabit networks.

In this paper, we evaluate the various aspects of the TCP/IP protocol suite for 10-Gigabit networks including the memory traffic and CPU requirements, and compare these with RDMA capable network adapters, using 10-Gigabit Ethernet and InfiniBand as example networks. Our measurements show that while the host based TCP/IP stack has a high CPU requirement, up to about 80% of this overhead is associated with the core protocol implementation especially

*This research is supported in part by National Science Foundation grants #CCR-0204429 and #CCR-0311542 to Dr. D. K. Panda at the Ohio State University.

for large messages and is potentially offloadable using the recently proposed TCP Offload Engines or user-level sockets layers.

Further, our studies reveal that for 10-Gigabit networks, the sockets layer itself becomes a significant bottleneck for memory traffic. Especially when the data is not present in the L2-cache, network transactions generate significant amounts of memory bus traffic for the TCP protocol stack. As we will see in the later sections, each byte transferred on the network can generate up to 4 bytes of data traffic on the memory bus. With the current moderately fast memory buses (e.g., 64bit/333MHz) and low memory efficiencies (e.g., 65%), this amount of memory traffic limits the peak throughput applications can achieve to less than 35% of the network's capability. Further, the memory bus and CPU speeds have not been scaling with the network bandwidth [4], pointing to the fact that this problem is only going to worsen in the future.

We also evaluate the RDMA interface of the InfiniBand architecture to understand the implications of having an RDMA interface over IP in two aspects: (a) the CPU requirement for the TCP stack usage and the copies associated with the sockets interface, (b) the difference in the amounts of memory traffic generated by RDMA compared to that of the traditional sockets API. Our measurements show that the RDMA interface requires up to four times lesser memory traffic and has almost zero CPU requirement for the data sink. These measurements show the potential impacts of having an RDMA interface over IP on 10-Gigabit networks.

The remaining part of the paper is organized as follows: In Section 2, we provide a brief background about InfiniBand, the RDMA interface and the TCP protocol suite. Section 3 provides details about the architectural requirements associated with the TCP stack. We present some experimental results in Section 4, other related work in Section 5 and draw our conclusions in Section 6.

2 Background

In this section, we provide a brief background about the InfiniBand Architecture, the RDMA interface and the TCP protocol suite.

2.1 InfiniBand Architecture

InfiniBand Architecture (IBA) is an industry standard that defines a System Area Network (SAN) to design clusters offering low latency and high bandwidth. In a typical IBA cluster, switched serial links connect the processing nodes and the I/O nodes. The compute nodes are connected to the IBA fabric by means of Host Channel Adapters (HCAs). IBA defines a semantic interface called as Verbs for the consumer applications to communicate with the HCAs.

IBA mainly aims at reducing the system processing overhead by decreasing the number of copies associated with a message transfer and removing the kernel from the critical message passing path. This is achieved by providing the consumer applications direct and protected access to the HCA.

2.1.1 RDMA Communication Model

IBA supports two types of communication semantics: channel semantics (send-receive communication model) and memory semantics (RDMA communication model).

In channel semantics, every send request has a corresponding receive request at the remote end. Thus there is one-to-one correspondence between every send and receive operation. Failure to post a receive descriptor on the remote node results in the message being dropped and if the connection is reliable, it might even result in the breaking of the connection. In memory semantics, Remote Direct Memory Access (RDMA) operations are used. These operations are transparent at the remote end since they do not require a receive descriptor to be posted. There are two kinds of RDMA operations: RDMA Write and RDMA Read. In an RDMA write operation, the initiator directly writes data into the remote node's user buffer. Similarly, in an RDMA Read operation, the initiator reads data from the remote node's user buffer.

2.2 TCP/IP Protocol Suite

The data processing path taken by the TCP protocol stack is broadly classified into the transmission path and the receive path. On the transmission side, the message is copied into the socket buffer, divided into MTU sized segments, data integrity ensured through checksum computation (to form the TCP checksum) and passed on to the underlying IP layer. Linux-2.4 uses a combined checksum and copy for the transmission path, a well known optimization first proposed by Jacobson, et al. [8]. The IP layer extends the checksum to include the IP header and form the IP checksum and passes on the IP datagram to the device driver. After the construction of a packet header, the device driver makes a descriptor for the packet and passes the descriptor to the NIC. The NIC performs a DMA operation to move the actual data indicated by the descriptor from the socket buffer to the NIC buffer. The NIC then ships the data with the link header to the physical network and raises an interrupt to inform the device driver that it has finished transmitting the segment.

On the receiver side, the NIC receives the IP datagrams, DMAs them to the socket buffer and raises an interrupt informing the device driver about this. The device driver strips the packet off the link header and hands it over to the IP layer. The IP layer verifies the IP checksum and if the data integrity is maintained, hands it over to the TCP layer. The TCP layer verifies the data integrity of the message and places the data into the socket buffer. When the application calls the `read()` operation, the data is copied from the socket buffer to the application buffer.

3 Understanding TCP/IP Requirements

In this section, we study the impact of cache misses not only on the performance of the TCP/IP protocol stack, but also on the amount of memory traffic associated with these cache misses; we estimate the amount of memory traffic for

a typical throughput test. In Section 4, we validate these estimates through measured values.

Memory traffic comprises of two components: Front Side Bus (FSB) reads and writes generated by the CPU(s) and DMA traffic generated through the I/O bus by other devices (NIC in our case). We study the memory traffic associated with the transmit path and the receive paths separately. Further, we break up each of these paths into two cases: (a) Application buffer fits in cache and (b) Application buffer does not fit in cache. In this section, we describe the path taken by the second case, i.e., when the application buffer does not fit in cache. We also present the final memory traffic ratio of the first case, but refer the reader to [4] for the actual data path description due to space restrictions. Figures 1a and 1b illustrate the memory accesses associated with network communication.

3.1 Transmit Path

As mentioned earlier, in the transmit path, TCP copies the data from the application buffer to the socket buffer. The NIC then DMAs the data from the socket buffer and transmits it. The following are the steps involved on the transmission side:

CPU reads the application buffer (step 1): The application buffer has to be fetched to cache on every iteration since it does not completely fit into it. However, it does not have to be written back to memory each time since it is only used for copying into the socket buffer and is never dirtied. Hence, this operation requires a byte of data to be transferred from memory for every byte transferred over the network.

CPU writes to the socket buffer (step 2): The default socket buffer size for most kernels including Linux and Windows Server 2003 is 64KB, which fits in cache (on most systems). In the first iteration, the socket buffer is fetched to cache and the application buffer is copied into it. In the subsequent iterations, the socket buffer stays in one of *Exclusive*, *Modified* or *Shared* states, i.e., it never becomes *Invalid*. Further, any change of the socket buffer state from one to another of these three states just requires a notification transaction or a Bus Upgrade from the cache controller and generates no memory traffic. So ideally this operation should not generate any memory traffic. However, the large application buffer size can force the socket buffer to be pushed out of cache. This can cause up to 2 bytes of memory traffic per network byte (one transaction to push the socket buffer out of cache and one to fetch it back). Thus, this operation can require between 0 and 2 bytes of memory traffic per network byte.

NIC does a DMA read of the socket buffer (steps 3 and 4): When a DMA request from the NIC arrives, the segment of the socket buffer corresponding to the request can be either in cache (dirtied) or in memory. In the first case, during the DMA, most memory controllers perform an implicit write back of the cache lines to memory. In the second case, the DMA takes place from memory. So, in either case, there would be one byte of data transferred either to or from memory for every byte of data transferred on the network.

Based on these four steps, we can expect the memory traffic required for this case to be between 2 to 4 bytes for every byte of data transferred over the network. Also, we can expect this value to move closer to 4 as the size of the application buffer increases (forcing more cache misses for the socket buffer).

Further, due to the set associative nature of some caches, it is possible that some of the segments corresponding to the application and socket buffers be mapped to the same cache line. This requires that these parts of the socket buffer be fetched from memory and written back to memory on every iteration. It is to be noted that, even if a cache line corresponding to the socket buffer is evicted to accommodate another cache line, the amount of memory traffic due to the NIC DMA does not change; the only difference would be that the traffic would be a memory read instead of an implicit write back. However, we assume that the cache mapping and implementation are efficient enough to avoid such a scenario and do not expect this to add any additional memory traffic.

3.2 Receive Path

The memory traffic associated with the receive path is simpler compared to that of the transmit path. The following are steps involved on the receive path:

NIC does a DMA write into the socket buffer (step 1): When the data arrives at the NIC, it does a DMA write of this data into the socket buffer. During the first iteration, if the socket buffer is present in cache and is dirty, it is flushed back to memory by the cache controller. Only after the buffer is flushed out of the cache is the DMA write request allowed to proceed. In the subsequent iterations, even if the socket buffer is fetched to cache, it would not be in a *Modified* state (since it is only being used to copy data into the application buffer). Thus, the DMA write request would be allowed to proceed as soon as the socket buffer in the cache is invalidated by the North Bridge (Figure 1), i.e., the socket buffer does not need to be flushed out of cache for the subsequent iterations. This sums up to one transaction to the memory during this step.

CPU reads the socket buffer (step 2): Again, at this point the socket buffer is not present in cache, and has to be fetched, requiring one transaction from the memory. It is to be noted that even if the buffer was present in the cache before the iteration, it has to be evicted or invalidated for the previous step.

CPU writes to application buffer (steps 3, 4 and 5): Since the application buffer does not fit into cache entirely, it has to be fetched in parts, data copied to it, and written back to memory to make room for the rest of the application buffer. Thus, there would be two transactions to and from the memory for this step (one to fetch the application buffer from memory and one to write it back).

This sums up to 4 bytes of memory transactions for every byte transferred on the network for this case. It is to be noted that for this case, the number of memory transactions

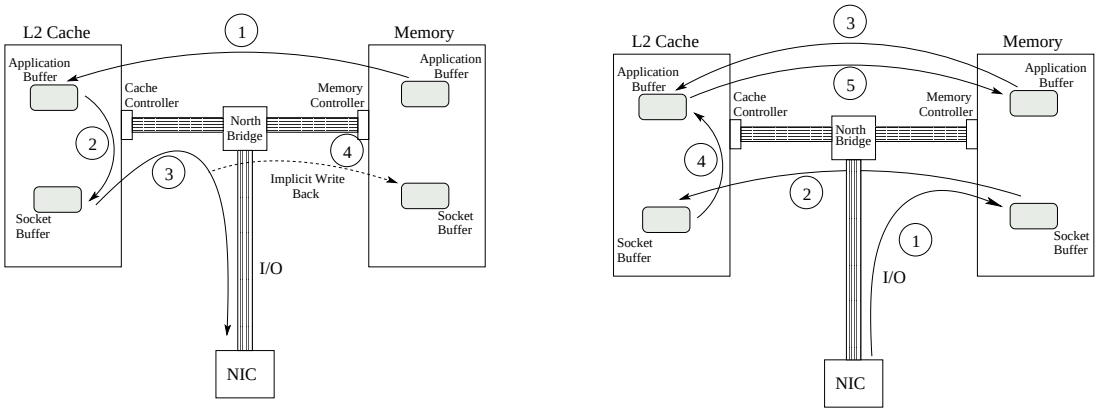


Figure 1. Memory Traffic for Sockets: (a) Transmit Path; (b) Receive Path

does not depend on the cache policy. Table 1 gives a summary of the memory transactions expected for each of the above described cases. *Theoretical* refers to the possibility of cache misses due to inefficiencies in the cache policy, set associativity, etc. *Practical* assumes that the cache policy is efficient enough to avoid cache misses due to memory to cache mappings. While the actual memory access pattern is significantly more complicated than the one described above due to the pipelining of data transmission to and from the socket buffer, this model captures the bulk of the memory transactions and provides a fair enough estimate.

Table 1. Memory to Network traffic ratio

	fits in cache	does not fit in cache
Transmit (Theoretical)	1-4	2-4
Transmit (Practical)	**1**	**2-4**
Receive (Theoretical)	2-4	4
Receive (Practical)	**2**	**4**

4 Experimental Results

In this section, we present some of the experiments we have conducted over 10 Gigabit Ethernet and InfiniBand.

The test-bed used for evaluating the 10-Gigabit Ethernet stack consisted of two clusters.

Cluster 1: Two Dell2600 Xeon 2.4 GHz 2-way SMP nodes, each with 1GB main memory (333MHz, DDR), Intel E7501 chipset, 32Kbyte L1-Cache, 512Kbyte L2-Cache, 400MHz/64-bit Front Side Bus, PCI-X 133MHz/64bit I/O bus, Intel 10GbE/Pro 10-Gigabit Ethernet adapters.

Cluster 2: Eight P4 2.4 GHz IBM xSeries 305 nodes, each with 256Kbyte main memory and connected using the Intel Pro/1000 MT Server Gigabit Ethernet adapters. We used Windows Server 2003 and Linux kernel 2.4.18-14smp for our evaluations. The multi-stream tests were conducted using a FoundryNet 10-Gigabit Ethernet switch.

The test-bed used for evaluating the InfiniBand stack consisted of the following cluster.

Cluster 3: Eight nodes built around SuperMicro SUPER P4DL6 motherboards and GC chipsets which include 64-bit 133 MHz PCI-X interfaces. Each node has two Intel Xeon 2.4GHz processors with a 512Kbyte L2 cache and a 400MHz front side bus. The machines are connected with Mellanox InfiniHost MT23108 DualPort 4x HCA adapter through an InfiniScale MT43132 Eight 4x Port InfiniBand Switch. The Mellanox InfiniHost HCA SDK version is thca-x86-0.2.0-build-001. The adapter firmware version is fw-23108-rel-1_17_0000-rc12-build-001. We used the Linux 2.4.7-10smp kernel version.

4.1 10-Gigabit Ethernet

In this section we present the performance delivered by 10-Gigabit Ethernet, the memory traffic generated by the TCP/IP stack (including the sockets interface) and the CPU requirements of this stack.

4.1.1 Micro-Benchmark Evaluation

For the micro-benchmark tests, we have studied the impacts of varying different parameters in the system as well as the TCP/IP stack including (a) Socket Buffer Size, (b) Maximum Transmission Unit (MTU), (c) Network adapter offloads (checksum, segmentation), (d) PCI burst size (PBS), (e) Switch Behavior, (f) TCP window size, (g) Adapter Interrupt delay settings, (h) Operating System (Linux and Windows Server 2003) and several others. Most of these micro-benchmarks use the same buffer for transmission resulting in maximum cache hits presenting the ideal case performance achievable by 10-Gigabit Ethernet. Due to this reason, these results tend to hide a number of issues related to memory traffic. The main idea of this paper is to study the memory bottleneck in the TCP/IP stack. Hence, we have shown only a subset of these micro-benchmarks in this paper. The rest of the micro-benchmarks can be found in [4].

Single Stream Tests: Figure 2a shows the one-way ping-pong latency achieved by 10-Gigabit Ethernet. We can see that 10-Gigabit Ethernet is able to achieve a latency of about 37μs for a message size of 256bytes on the Windows Server 2003 platform. The figure also shows the average CPU utilization for the test. We can see that the test requires about 50% CPU on each side. We have also done a similar analysis on Linux where 10-Gigabit Ethernet achieves a latency of about 20.5μs (Figure 3a).

Figure 2b shows the throughput achieved by 10-Gigabit

Ethernet on the Windows Server 2003 platform. The parameter settings used for the experiment were a socket buffer size of 64Kbytes (both send and receive on each node), MTU of 16Kbytes, checksum offloaded on to the network card and the PCI burst size set to 4Kbytes. 10-Gigabit Ethernet achieves a peak throughput of about 2.5Gbps with a CPU usage of about 110% (dual processor system). We can see that the amount of CPU used gets saturated at about 100% though we are using dual processor systems. This is attributed to the interrupt routing mechanism for the "x86" architecture. The x86 architecture routes all interrupts to the first processor. For interrupt based protocols such as TCP, this becomes a huge bottleneck, since this essentially restricts the transmission side to about one CPU. This behavior is also seen in the multi-stream transmission tests (in particular the fan-out test) which are provided in the later sections. The throughput results for the Linux platform are presented in Figure 3b.

Multi-Stream Tests: For the multi-stream results, we study the performance of the host TCP/IP stack in the presence of multiple data streams flowing from or into the node. The environment used for the multi-stream tests consisted of one node with a 10-Gigabit Ethernet adapter and several other nodes connected to the same switch using a 1-Gigabit Ethernet adapter.

Three main experiments were conducted in this category. The first test was a Fan-in test, where all the 1-Gigabit Ethernet nodes push data to the 10-Gigabit Ethernet node through the common switch they are connected to. The second test was a Fan-out test, where the 10-Gigabit Ethernet node pushes data to all the 1-Gigabit Ethernet nodes through the common switch. The third test was Dual test, where the 10-Gigabit Ethernet node performs the fan-in test with half the 1-Gigabit Ethernet nodes and the fan-out test with the other half. It is to be noted that the Dual test is quite different from a multi-stream bi-directional bandwidth test where the server node (10-Gigabit Ethernet node) does both a fan-in and a fan-out test with each client node (1-Gigabit Ethernet node). The message size used for these experiments is 10Mbytes. This forces the message *not to be in L2-cache* during subsequent iterations.

Figures 4a and 4b show the performance of the host TCP/IP stack over 10-Gigabit Ethernet for the Fan-in and the Fan-out tests. We can see that we are able to achieve a throughput of about 3.5Gbps with a 120% CPU utilization (dual CPU) for the Fan-in test and about 4.5Gbps with a 100% CPU utilization (dual CPU) for the Fan-out test. Further, it is to be noted that the server gets saturated in the Fan-in test for 4 clients. However, in the fan-out test, the throughput continues to increase from 4 clients to 8 clients. This again shows that with 10-Gigabit Ethernet, the receiver is becoming a bottleneck in performance mainly due to the high CPU overhead involved on the receiver side.

Figure 4c shows the performance achieved by the host TCP/IP stack over 10-Gigabit Ethernet for the Dual test. The host TCP/IP stack is able to achieve a throughput of about 4.2Gbps with a 140% CPU utilization (dual CPU).

4.1.2 TCP/IP CPU Pareto Analysis

In this section we present a module wise break-up (Pareto Analysis) for the CPU overhead of the host TCP/IP stack over 10-Gigabit Ethernet. We used the *NTttcp* throughput test as a benchmark program to analyse this. Like other micro-benchmarks, the NTttcp test uses the same buffer for all iterations of the data transmission. So, the pareto analysis presented here is for the ideal case with the maximum number of cache hits. For measurement of the CPU overhead, we used the *Intel VTune^{TM} Performance Analyzer*. In short, the *VTune^{TM} Analyzer* interrupts the processor at specified events (e.g., every 'n' clock ticks) and records its execution context at that sample. Given enough samples, the result is a statistical profile of the ratio of the time spent in a particular routine. More details about the *Intel VTune^{TM} Performance Analyzer* can be found in [4].

Figures 5 and 6 present the CPU break-up for both the sender as well as the receiver for small messages (64bytes) and large messages (16Kbytes) respectively. It can be seen that in all the cases, the kernel and the protocol stack add up to about 80% of the CPU overhead. For small messages, the overhead is mainly due to the per-message interrupts. These interrupts are charged into the kernel usage, which accounts for the high percentage of CPU used by the kernel for small messages. For larger messages, on the other hand, the overhead is mainly due to the data touching portions in the TCP/IP protocol suite such as checksum, copy, etc.

As it can be seen in the pareto analysis, in cases where the cache hits are high, most of the overhead of TCP/IP based communication is due to the TCP/IP protocol processing itself or due to other kernel overheads. This shows the potential benefits of having TCP Offload Engines in such scenarios where these components are optimized by pushing the processing to the hardware. However, the per-packet overheads for small messages such as interrupts for sending and receiving data segments would still be present inspite of a protocol offload. Further, as we'll see in the memory traffic analysis (the next section), for cases where the cache hit rate is not very high, the memory traffic associated with the sockets layer becomes very significant forming a fundamental bottleneck for all implementations which support the sockets layer, including high performance user-level sockets as well as TCP Offload Engines.

4.1.3 TCP/IP Memory Traffic

For the memory traffic tests, we again used the *VTune ^{TM} Performance Analyzer*. We measure the number of cache lines fetched and evicted from the processor cache to calculate the data traffic on the Front Side Bus. Further, we measure the data being transferred from or to memory from the North Bridge to calculate the memory traffic (on the Memory Bus). Based on these two calculations, we evaluate the amount of traffic being transferred over the I/O bus (difference in the amount of traffic on the Front Side Bus and the

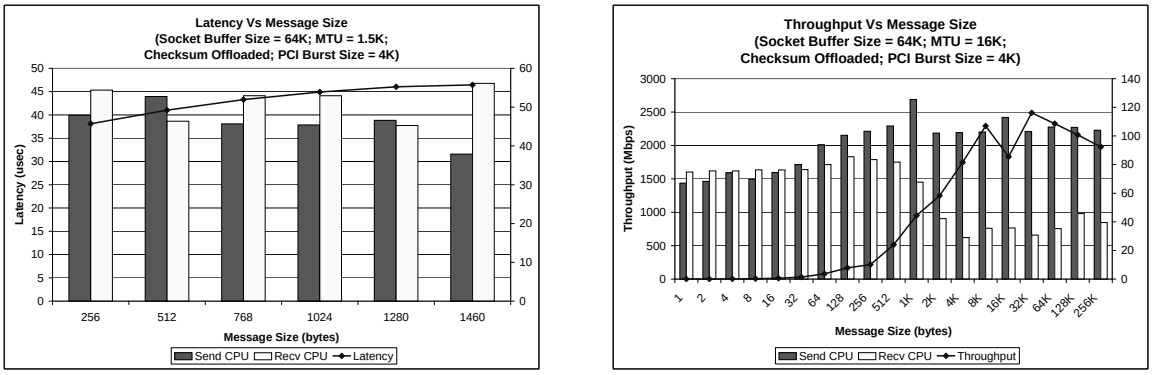


Figure 2. Micro-Benchmarks for the host TCP/IP stack over 10-Gigabit Ethernet on the Windows Platform: (a) One-Way Latency (MTU 1.5K); (b) Throughput (MTU 16K)

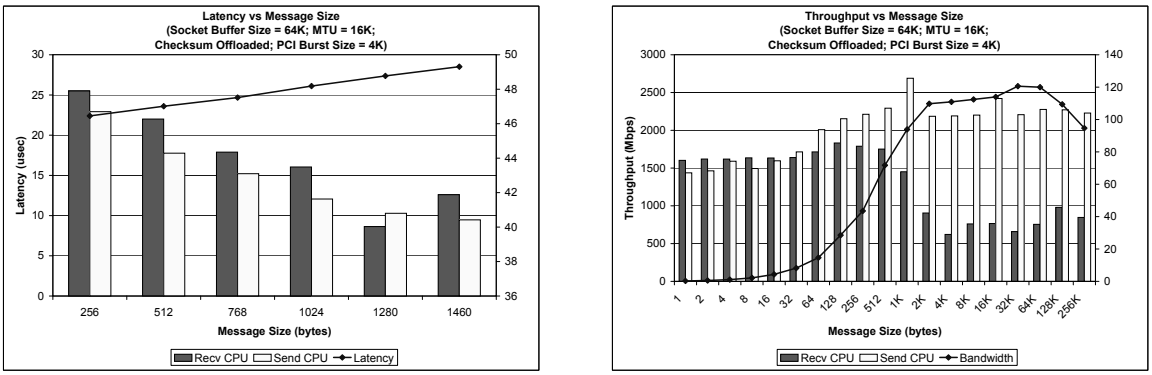


Figure 3. Micro-Benchmarks for the host TCP/IP stack over 10-Gigabit Ethernet on the Linux Platform: (a) One-Way Latency (MTU 1.5K); (b) Throughput (MTU 16K)

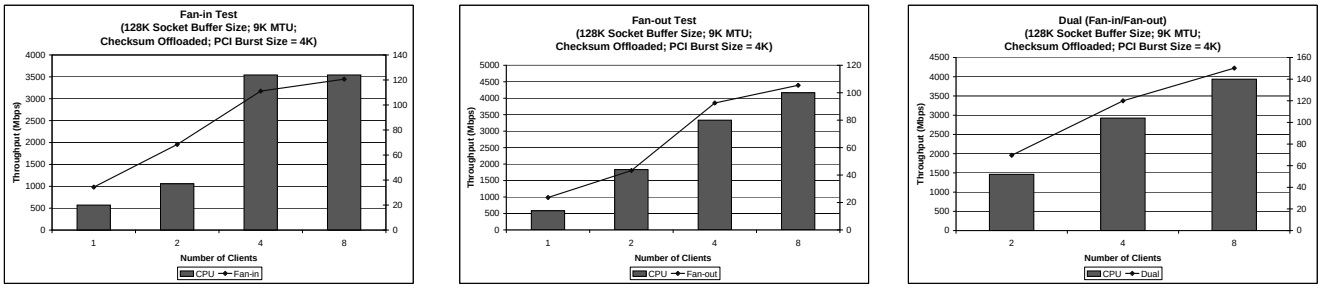


Figure 4. Multi-Stream Micro-Benchmarks: (a) Fan-in, (b) Fan-out, (c) Dual (Fan-in/Fan-out)

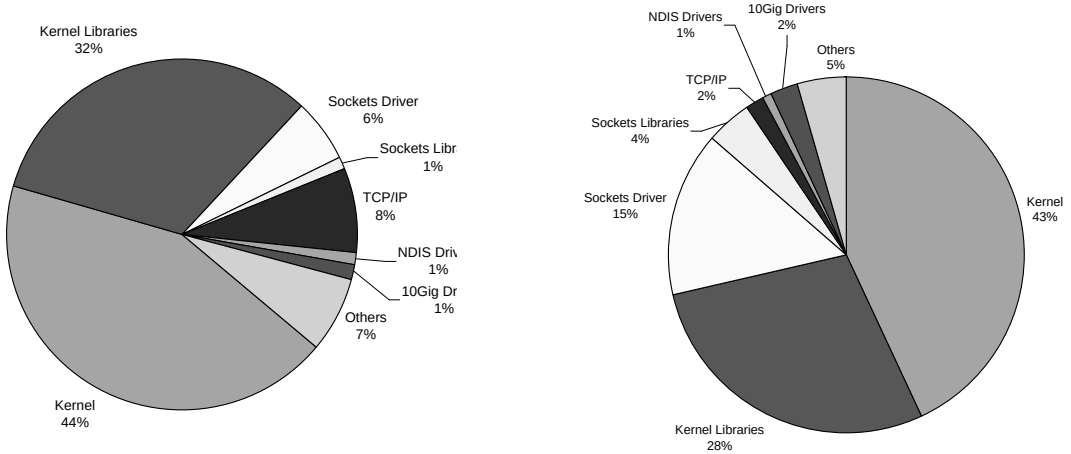


Figure 5. Throughput Test: CPU Pareto Analysis for small messages (64bytes): (a) Transmit Side, (b) Receive Side

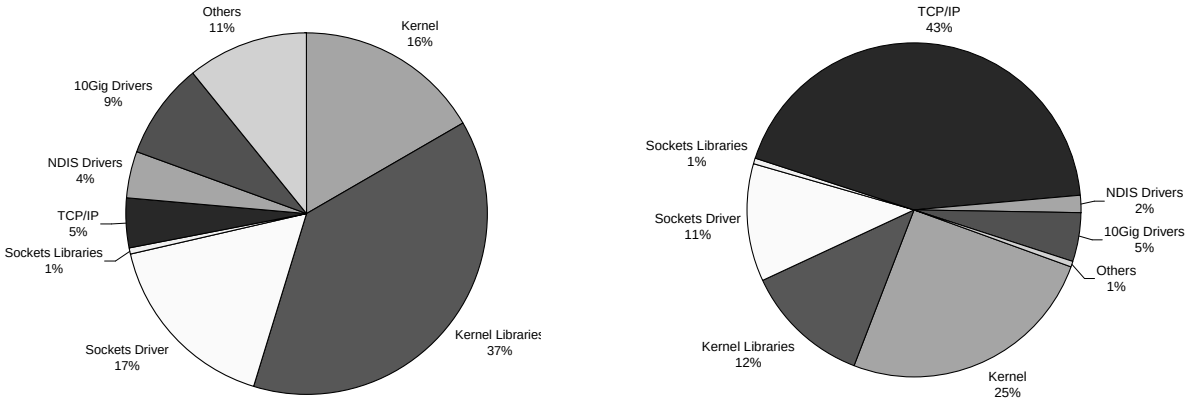


Figure 6. Throughput Test: CPU Pareto Analysis for large messages (16Kbytes): (a) Transmit Size, (b) Receive Side

Memory Bus).

Single Stream Tests: Figure 7a shows the memory traffic associated with the data being transferred on the network for the sender and the receiver sides. As discussed in Section 3, for small message sizes (messages which fit in the L2-cache), we can expect about 1 byte of memory traffic per network byte on the sender side and about 2 bytes of memory traffic per network byte on the receiver side. However, the amount of memory traffic seems to be large for very small messages. The reason for this is the TCP control traffic and other noise traffic on the memory bus. Such traffic would significantly affect the smaller message sizes due to the less amount of memory traffic associated with them. However, when the message size becomes moderately large (and still fits in L2-cache), we can see that the message traffic follows the trend predicted.

For large message sizes (messages which do not fit in the L2-cache), we can expect between 2 and 4 bytes of memory traffic per network byte on the sender side and about 4 bytes of memory traffic per network byte on the receiver side. We can see that the actual memory to network traffic ratio follows this trend.

These results show that even without considering the host CPU requirements for the TCP/IP protocol stack, the memory copies associated with the sockets layer can generate up to 4 bytes of memory traffic per network byte for traffic in each direction, forming what we call the *memory-traffic* bottleneck. It is to be noted that while some TCP Offload Engines try to avoid the memory copies in certain scenarios, the sockets API can not force a zero copy implementation for all cases (e.g., transactional protocols such as RPC, File I/O, etc. first read the data header and decide the size of the buffer to be posted). This forces the memory-traffic bottleneck to be associated with the sockets API.

Multi-Stream Tests: Figure 7b shows the actual memory traffic associated with the network data transfer during the multi-stream tests. It is to be noted that the message size used for the experiments is 10Mbytes, so subsequent transfers of the message need the buffer to be fetched from memory to L2-cache.

The first two legends in the figure show the amount of bytes transferred on the network and the bytes transferred on the memory bus per second respectively. The third legend shows 65% of the peak bandwidth achievable by the memory bus. 65% of the peak memory bandwidth is a general rule of thumb used by most computer companies to estimate the peak practically sustainable memory bandwidth on a memory bus when the requested physical pages are non-contiguous and are randomly placed. It is to be noted that though the virtual address space could be contiguous, this doesn't enforce any policy on the allocation of the physical address pages and they can be assumed to be randomly placed.

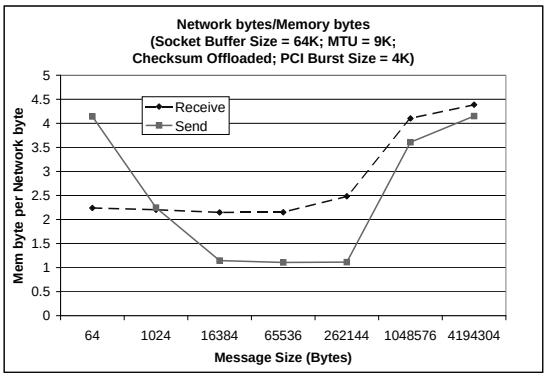
It can be seen that the amount of memory bandwidth required is significantly larger than the actual network bandwidth. Further, for the Dual test, it can be seen that the memory bandwidth actually reaches within 5% of the peak practically sustainable bandwidth.

4.2 InfiniBand Architecture

In this section, we present briefly the performance achieved by RDMA enabled network adapters such as InfiniBand.

Figure 8a shows the one-way latency achieved by the RDMA write communication model of the InfiniBand stack for the polling based approach for completion as well as an event based approach. In the polling approach, the application continuously monitors the completion of the message by checking the arrived data. This activity makes the polling based approach CPU intensive resulting in a 100% CPU utilization. In the event based approach, the application goes to sleep after posting the descriptor. The network adaptor raises an interrupt for the application once the message arrives. This results in a lesser CPU utilization for the event based scheme.

We see that RDMA write achieves a latency of about 5.5μs for both the polling based scheme as well as the event based scheme. The reason for both the event based scheme and the polling based scheme performing alike is the receiver transparency for RDMA Write operations. Since, the RDMA



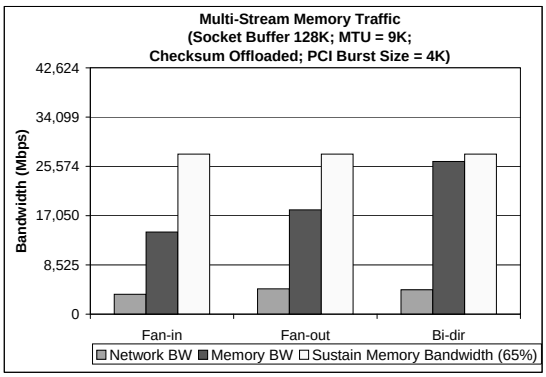
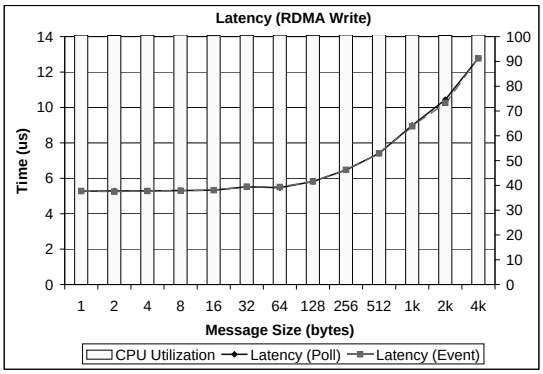


Figure 7. Throughput Test Memory Traffic Analysis: (a) Single Stream, (b) Multi Stream



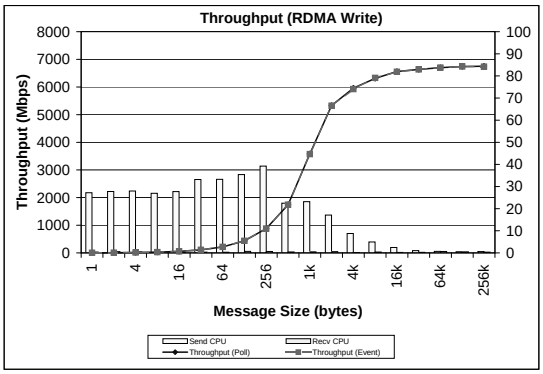
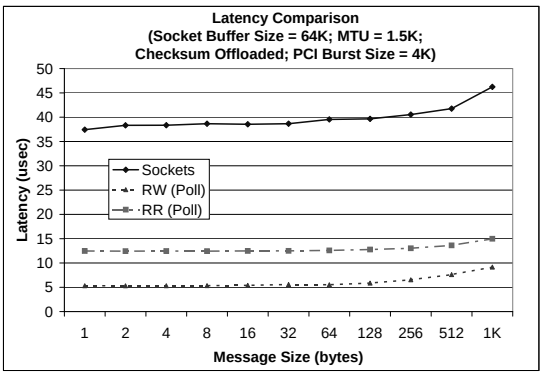


Figure 8. IBA Micro-Benchmarks for RDMA Write: (a) Latency and (b) Throughput



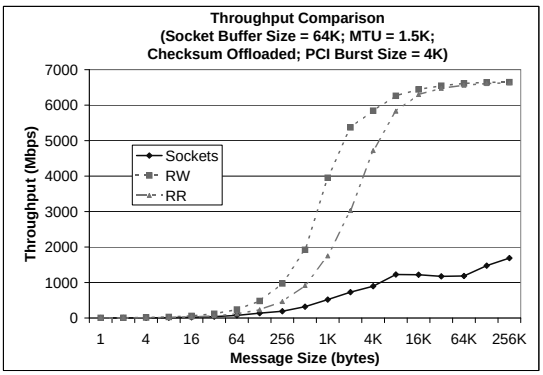
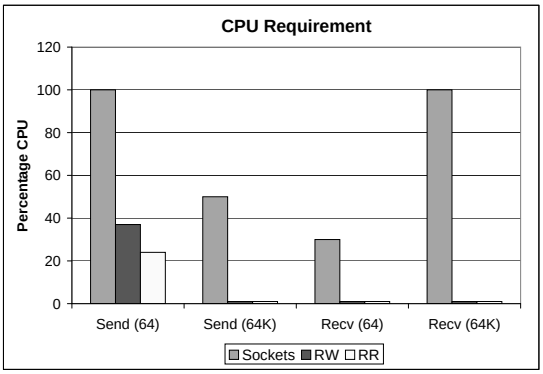


Figure 9. Latency and Throughput Comparison: Host TCP/IP over 10-Gigabit Ethernet Vs InfiniBand



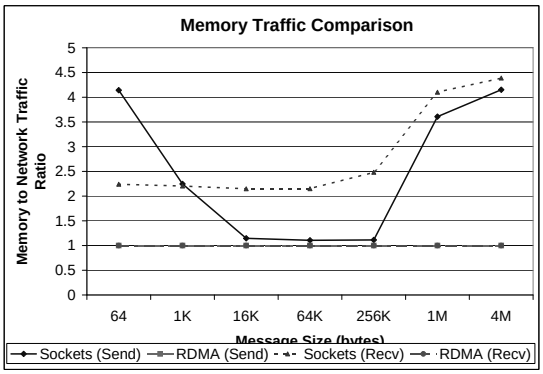


Figure 10. CPU Requirement and Memory Traffic Comparisons: Host TCP/IP over 10-Gigabit Ethernet Vs InfiniBand

write operation is completely receiver transparent, the only way the receiver can know that the data has arrived into its buffer is by polling on the last byte. So, in an event-based approach only the sender would be blocked on send completion using a notification event; the notification overhead at the sender is however parallelized with the data transmission and reception. Due to this the time taken by RDMA write for the event-based approach is similar to that of the polling based approach. Due to the same reason, the CPU overhead in the event-based approach is 100% (similar to the polling based approach).

RDMA Read on the other hand achieves a latency of about 12.5μs for the polling based scheme and about 24.5μs for the event based scheme. The detailed results for RDMA read and the other communication models such as send-receive and RDMA write with immediate data can be found in [4].

Figure 8 shows the throughput achieved by RDMA write. Again, results for both the polling based approach as well as the event-based approach are presented.

Both approaches seem to perform very close to each other giving a peak throughput of about 6.6Gbps. The peak throughput is limited by the sustainable bandwidth on the PCI-X bus. The way the event-based scheme works is that, it first checks the completion queue for any completion entries present. If there are no completion entries present, it requests a notification from the network adapter and blocks while waiting for the data to arrive. In a throughput test, data messages are sent one after the other continuously. So, the notification overhead can be expected to be overlapped with the data transmission overhead for the consecutive messages. This results in a similar performance for the event-based approach as well as the polling based approach.

The CPU utilization values are only presented for the event-based approach; those for the polling based approach stay close to 100% and are not of any particular interest. The interesting thing to note is that for RDMA, there is nearly zero CPU utilization for the data sink especially for large messages.

4.3 10-Gigabit Ethernet/InfiniBand Comparisons

Figures 9a and 9b show the latency and throughput comparisons between IBA and 10-Gigabit Ethernet respectively. In this figure we have skipped the event based scheme and shown just the polling based scheme. The reason for this is the software stack overhead in InfiniBand. The performance of the event based scheme depends on the performance of the software stack to handle the events generated by the network adapter and hence would be specific to the implementation we are using. Hence, to get an idea of the peak performance achievable by InfiniBand, we restrict ourselves to the polling based approach.

We can see that InfiniBand is able to achieve a significantly higher performance than the host TCP/IP stack on 10-Gigabit Ethernet; a factor of three improvement in the latency and a up to a 3.5 times improvement in the throughput. This improvement in performance is mainly due to the offload of the network protocol, direct access to the NIC and direct placement of data into the memory.

Figures 10a and 10b show the CPU requirements and the memory traffic generated by the host TCP/IP stack over 10-Gigabit Ethernet and the InfiniBand stack. We can see that the memory traffic generated by the host TCP/IP stack is much higher (more than 4 times in some cases) as compared to InfiniBand; this difference is mainly attributed to the copies involved in the sockets layer for the TCP/IP stack. This result points to the fact that inspite of the possibility of an offload of the TCP/IP stack on to the 10-Gigabit Ethernet network adapter TCP's scalability would still be restricted by the sockets layer and its associated copies. On the other hand, having an RDMA interface over IP together with the offloaded TCP stack can be expected to achieve all the advantages seen by InfiniBand.

Some of the expected benefits are (1) Low overhead interface to the network, (2) Direct Data Placement (significantly reducing intermediate buffering), (3) Support for RDMA semantics, i.e., the sender can handle the buffers allocated on the receiver node and (4) Most importantly, the amount of memory traffic generated for the network communication will be equal to the number of bytes going out to or coming in from the network, thus improving scalability.

5 Related Work

Several researchers have worked on implementing high performance user-level sockets implementations over high performance networks. Kim, Shah, Balaji and several others have worked on such pseudo sockets layers over Gigabit Ethernet [5], GigaNet cLAN [16, 19, 6] and InfiniBand [3]. However, these implementations try to maintain the sockets API in order to allow compatibility for existing applications and hence still face the memory traffic bottlenecks discussed in this paper.

There has been some previous work done by Foong et. al. [10] which does a similar analysis of the bottlenecks associated by the TCP/IP stack and the sockets interface. This research is notable in the sense that this was the one of the first to show the implications of the memory traffic associated with the TCP/IP stack. However, this analysis was done using much slower networks, in particular Gigabit Ethernet adapters following which the memory traffic did not show up as a fundamental bottleneck and the conclusions of the work were quite different from ours. Wu-Chun Feng, et. al. [12, 9], have done some initial performance evaluation of 10-Gigabit Ethernet adapters. Their work focuses only on the peak performance deliverable by the adapters and does not consider the memory traffic issues for the TCP/IP stack present.

We would also like to mention some previous research to optimize the TCP stack [8, 13, 17, 15, 14]. However, in this paper, we question the sockets API itself and propose issues associated with this API. Further, we believe that most of the previously proposed techniques would still be valid for the proposed RDMA interface over TCP/IP and can be used

in a complementary manner.

6 Concluding Remarks and Future Work

The compute requirements associated with the TCP/IP protocol suite have been previously studied by a number of researchers. However, the recently developed 10 Gigabit networks such as 10-Gigabit Ethernet and InfiniBand have added a new dimension of complexity to this problem, *Memory Traffic*. While there have been previous studies which show the implications of the memory traffic bottleneck, to the best of our knowledge, there has been no study which shows the actual impact of the memory accesses generated by TCP/IP for 10-Gigabit networks.

In this paper, we first do a detailed evaluation of various aspects of the host-based TCP/IP protocol stack over 10-Gigabit Ethernet including the memory traffic and CPU requirements. Next, we compare these with RDMA capable network adapters, using InfiniBand as an example network. Our measurements show that while the host based TCP/IP stack has a high CPU requirement, up to 80% of this overhead is associated with the core protocol implementation especially for large messages and is potentially offloadable using the recently proposed TCP Offload Engines. However, the current host based TCP/IP stack also requires multiple transactions of the data (up to a factor of four in some cases) over the current moderately fast memory buses, curbing their scalability to faster networks; for 10-Gigabit networks, the host based TCP/IP stack generates enough memory traffic to saturate a 333MHz/64bit DDR memory bandwidth even before 35% of the available network bandwidth is used.

Our evaluation of the RDMA interface over the InfiniBand network tries to nail down some of the benefits achievable by providing an RDMA interface over IP. In particular, we try to compare the RDMA interface over InfiniBand not only in performance, but also in other resource requirements such as CPU usage, memory traffic, etc. Our results show that the RDMA interface requires up to four times lesser memory traffic and has almost zero CPU requirements for the data sink. These measurements show the potential impacts of having an RDMA interface over IP on 10-Gigabit networks.

As a part of the future work, we would like to do a detailed memory traffic analysis of the 10-Gigabit Ethernet adapter on 64-bit systems and for various applications such as SpecWeb and multimedia streaming servers.

7 Acknowledgments

We would like to thank Annie Foong for all the help she provided while using the performance tools. We would also like to thank Gary Tsao, Gilari Janarthanan and J. L. Gray for the valuable discussions we had during the course of the project.

References

[1] 10 Gigabit Ethernet Alliance. http://www.10gea.org/.

[2] InfiniBand Trade Association. http://www.infinibandta.org.

[3] P. Balaji, S. Narravula, K. Vaidyanathan, S. Krishnamoorthy, J. Wu, and D. K. Panda. Sockets Direct Protocol over InfiniBand in Clusters: Is it Beneficial? In *ISPASS '04*.

[4] P. Balaji, H. V. Shah, and D. K. Panda. Sockets vs RDMA Interface over 10-Gigabit Networks: An In-depth analysis of the Memory Traffic Bottleneck? Technical Report OSU-CISRC-2/04-TR11, The Ohio State University, 2003.

[5] P. Balaji, P. Shivam, P. Wyckoff, and D. K. Panda. High Performance User Level Sockets over Gigabit Ethernet. In *Cluster '02*.

[6] P. Balaji, J. Wu, T. Kurc, U. Catalyurek, D. K. Panda, and J. Saltz. Impact of High Performance Sockets on Data Intensive Applications. In *HPDC '03*.

[7] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. K. Su. Myrinet: A Gigabit-per-Second Local Area Network. http://www.myricom.com.

[8] D. Clark, V. Jacobson, J. Romkey, and H.Salwen. An Analysis of TCP processing overhead. *IEEE Communications*, June 1989.

[9] W. Feng, J. Hurwitz, H. Newman, S. Ravot, L. Cottrell, O. Martin, F. Coccetti, C. Jin, D. Wei, and S. Low. Optimizing 10-Gigabit Ethernet for Networks of Workstations, Clusters and Grids: A Case Study. In *Proceedings of the IEEE International Conference on Supercomputing*, Phoenix, Arizona, November 2003.

[10] A. Foong, H. Hum, T. Huff, J. Patwardhan, and G. Regnier. TCP Performance Revisited. In *ISPASS '03*.

[11] H. Frazier and H. Johnson. Gigabit Ethernet: From 100 to 1000Mbps.

[12] J. Hurwitz and W. Feng. End-to-End Performance of 10-Gigabit Ethernet on Commodity Systems. *IEEE Micro*, January 2004.

[13] V. Jacobson. 4BSD Header Prediction. In *SIGCOMM '90*.

[14] Hyun-Wook Jin, Chuck Yoo, and Sung-Kyun Park. Stepwise Optimizations of UDP/IP on a Gigabit Network. In *The Proceedings of the 8th International Euro-Par Conference on Parallel Processing (Euro-Par 2002)*, August 2002.

[15] J. Kay and J. Pasquale. The Importance of Non-Data Touching Processing Overheads in TCP/IP. In *SIGCOMM '93*.

[16] J. Kim, K. Kim, and S. Jung. SOVIA: A User-level Sockets Layer Over Virtual Interface Architecture. In *Cluster Computing*, 2001.

[17] P. E. McKenney and K. F. Dove. Efficient Demultiplexing of Incoming TCP Packets. In *SIGCOMM '92*.

[18] F. Petrini, W. C. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network (QsNet): High-Performance Clustering Technology. In *the Proceedings of the IEEE International Conference on Hot Interconnects*, August 2001.

[19] H. V. Shah, C. Pu, and R. S. Madukkarumukumana. High Performance Sockets and RPC over Virtual Interface (VI) Architecture. In *CANPC '99*.

[20] E. Yeh, H. Chao, V. Mannem, J. Gervais, and B. Booth. Introduction to TCP/IP Offload Engine (TOE). http://www.10gea.org, May 2002.