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

Pavan Balaji
Comp. and Info. Science,
The Ohio State University,
Columbus, OH 43210,
balaji@cis.ohio-state.edu

Hemal V. Shah
Embedded IA Division,
Intel Corporation,
Austin, Texas,
hemal.shah@intel.com

D. K. Panda
Comp. and Info. Science,
The Ohio State University,
Columbus, OH 43210,
panda@cis.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 performance, 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.

Keywords: Sockets, RDMA, TCP/IP, InfiniBand

*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.

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 [18, 1, 20, 16], InfiniBand [5, 2], Myrinet [11] and Quadrics [4, 28, 30, 29, 31]. 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) [36, 37] is one of the universally accepted transport layer protocols in today's networking world. Despite the development of other protocols such as VIA [12, 3, 14, 9, 10], FM [27], GM [15] and EMP [34, 35] that have been coming up in the wake of the rapidly changing industries and networking technologies, TCP continues its dominance due to its reliability, adaptability and robustness for a wide range of applications. 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 [7, 8, 33, 6, 24, 25] and TCP Offload Engines [38].

User-level sockets implementations rely on zero-copy, OS-

bypass high performance protocols built on top of the high speed interconnects. The basic idea of such implementations is to create a sockets-like interface on top of these high performance protocols. This sockets layer is designed to serve two purposes: (a) to provide a smooth transition to deploy existing applications on to clusters connected with high performance networks and (b) to sustain most of the performance provided by the high performance protocols. TCP Offload Engines, on the other hand, offload the TCP stack on to hardware in part or in whole. Earlier Gigabit Ethernet adapters offloaded TCP and IP checksum computations on to hardware. This was followed by the offload of message segmentation (LSO/TSO). In the recent past, a number of companies including Intel, Adaptec, Alacritec, etc., have been working on offloading the entire TCP stack on to hardware. In short, 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 and those generated by RDMA capable network adapters.

In this paper, we evaluate the various aspects of the TCP/IP protocol suite for 10-Gigabit networks including performance, 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 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 four 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, 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 give a brief background about InfiniBand and the RDMA interface and the TCP protocol suite. Section 3 provides details about the architectural requirements associated with the TCP stack. We describe the tools and utilities we used in Section 4. We present some experimental results in Section 5, other related work in Section 6 and draw our conclusions in Section 7.

2 Background

In this section, we provide a brief background about the InfiniBand Architecture and the RDMA interface, the TCP/IP protocol suite, user-level sockets implementations and TCP Offload Engines.

2.1 InfiniBand Architecture

InfiniBand Architecture (IBA) is an industry standard that defines a System Area Network (SAN) to design clusters offering a 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. The specifications for Verbs includes a queue-based interface, known as a Queue Pair (QP), to issue requests to the HCA. Figure 1 illustrates the InfiniBand Architecture model.

Each Queue Pair is a communication endpoint. A Queue Pair (QP) consists of the send queue and the receive queue. Two QPs on different nodes can be connected to each other to form a logical bi-directional communication channel. An application can have multiple QPs. Communica-

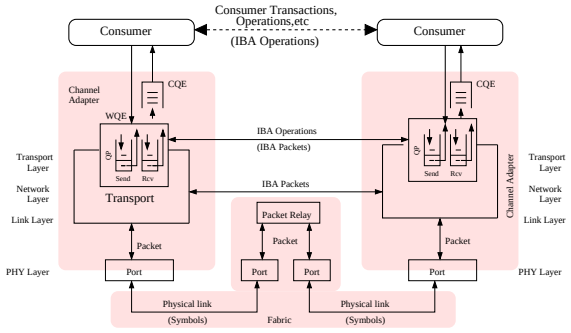


Figure 1. InfiniBand Architecture (Courtesy InfiniBand Specifications)

tion requests are initiated by posting Work Queue Requests (WQRs) to these queues. Each WQR is associated with one or more pre-registered buffers from which data is either transfered (for a send WQR) or received (receive WQR). The application can either choose the request to be a Signaled (SG) request or an Un-Signaled request (USG). When the HCA completes the processing of a signaled request, it places an entry called as the Completion Queue Entry (CQE) in the Completion Queue (CQ). The consumer application can poll on the CQ associated with the work request to check for completion. There is also the feature of triggering event handlers whenever a completion occurs. For Unsignaled request, no kind of completion event is returned to the user. However, depending on the implementation, the driver cleans up the the Work Queue Request from the appropriate Queue Pair on completion.

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. In this semantics, the send request itself contains both the virtual address for the local transmit buffer as well as that for the receive buffer on the remote end.

Most entries in the WQR are common for both the Send-Receive model as well as the RDMA model, except an additional remote buffer virtual address which has to be specified for RDMA operations.

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. IBA does not support scatter of data, hence the destination buffer in the case of RDMA Write and RDMA Read has to be contiguously registered buffer.

2.2 TCP/IP Protocol Suite

There have been a number of studies of the TCP/IP data transfer path [17, 19, 32]. In this section, we briefly reiterate on these previous studies on the Linux TCP/IP protocol suite.

Like most networking protocol suites, the TCP/IP protocol suite is a combination of different protocols at various levels, with each layer responsible for a different facet of the communications. The Socket abstraction was introduced to provide a uniform interface to network and interprocess communication protocols. The sockets layer maps protocol-independent requests from a process to the protocol-specific implementation selected when the process was created (Figure 2). For example, a "STREAM" socket corresponds to TCP, while a "Datagram" or "DGRAM" socket corresponds to UDP, and so on.

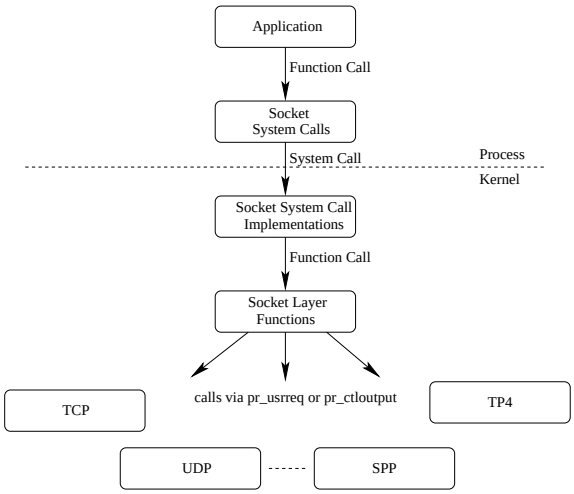


Figure 2. The sockets layer maps protocol-independent requests from the process to protocol-specific implementations

To allow standard Unix I/O system calls such as `read()` and `write()` to operate with network connections, the filesystem and networking facilities are integrated at the

system call level. Network connections represented by sockets are accessed through a descriptor (a small integer) in the same way an open file is accessed through a descriptor. This allows the standard filesystem calls such as `read()` and `write()`, as well as network-specific system calls such as `sendmsg()` and `recvmsg()`, to work with a descriptor associated with a socket.

A socket represents one end of a communication link and holds or points to all the information associated with the link. This information includes the protocol to use, state information of the protocol (which includes source and destination addresses), queues of arriving connections, data buffers and option flags. Figure 3 illustrates the communication between the layers of network input and output.

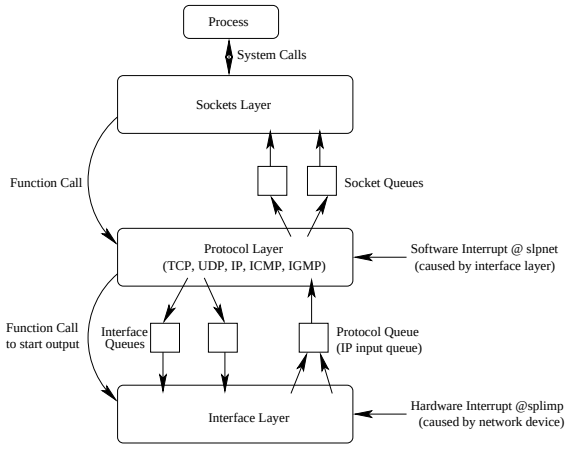


Figure 3. Communication between the layers of network input and output

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. [13]. 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. Figure 4 illustrates the sender and the receiver paths.

2.3 User-level Sockets Implementations

User Level Protocols achieve high performance by gaining direct access to the network interface in a protected manner. While User level protocols are beneficial for new applications, existing applications written using the sockets interface, have not been able to take advantage of these protocols. In order to allow these applications achieve the better performance provided by these networks, researchers came up with a number of solutions including user level sockets.

The basic idea of a user level sockets is to create a pseudo sockets-like interface to the application. This sockets layer is designed to serve two purposes: a) to provide a smooth transition to deploy existing application on to clusters connected with high performance networks and b) to sustain most of the performance provided by the high performance protocols.

2.4 TCP Offload Engines (TOEs)

The processing of TCP/IP over Ethernet is traditionally accomplished by software running on the central processor, CPU or microprocessor, of the server. As network connections scale beyond Gigabit Ethernet speeds, the CPU becomes burdened with the large amount of TCP/IP protocol processing required. Reassembling out-of-order packets, resource-intensive memory copies, and interrupts put a tremendous load on the host CPU. In high-speed networks, the CPU has to dedicate more processing to handle the network traffic than to the applications it is running. TCP Offload Engines (TOE) are emerging as a solution to limit the processing required by CPUs for networking links (Figure 5). A TOE may be embedded in a network interface card, NIC, or a host bus adapter, HBA.

The basic idea of a TOE is to offload the processing of TCP/IP protocols from the host processor to the hardware on the adapter or in the system. A TOE can be implemented with a network processor and firmware, specialized ASICs, or a combination of both. Most TOE implementations available in the market concentrate on offloading the TCP and IP processing.

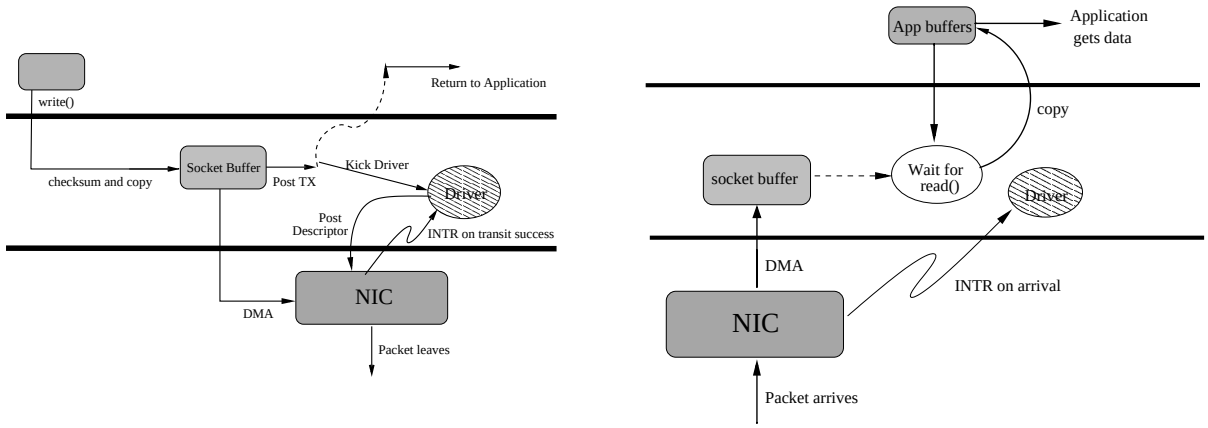


Figure 4. TCP Data Path: (a) Send; (b) Receive

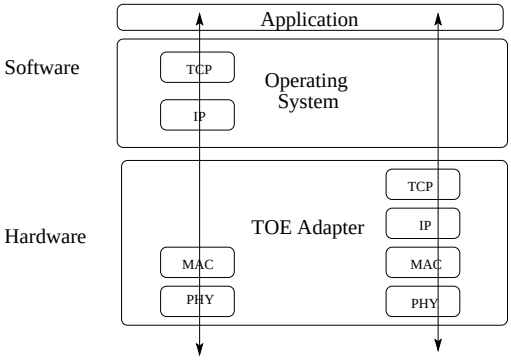


Figure 5. TCP Offload Engine

As a precursor to TCP offloading, some operating systems support features to offload some compute intensive features from the host to the underlying adapters. TCP and IP checksum offload implemented in some server network adapters is an example of a simple offload. But as Ethernet speeds increased beyond 100Mbps, the need for further protocol processing offload became a clear requirement. Some Gigabit Ethernet adapters complemented this requirement by offloading TCP segmentation on the transmission side on to the network adapter as well.

TOE can be implemented in different ways depending on end-user preference between various factors like deployment flexibility and performance. Traditionally, processor-based solutions provided the flexibility to implement new features, while ASIC solutions provided performance but were not flexible enough to add new features. Today, there is a new breed of performance optimized ASICs utilizing multiple processing engines to provide ASIC-like performance with more deployment flexibility.

In a firmware or processor based implementation, TOE is implemented using off-the-shelf components like a network processor or a microprocessor running a real time operating system (RTOS) and a MAC/PHY. The protocol processing from the host CPU is offloaded to the protocol stack in the RTOS, provided proper hooks are supplied by the hardware to offload these protocols from the host. The advantage of this implementation is the flexibility of the solution and the wide availability of the components.

In an ASIC-based implementation, TCP/IP processing is offloaded to performance-optimized hardware. ASIC implementations are customized for TCP/IP protocol offload, offering better performance than processor based implementations. There is a general agreement that the advantages of this implementation are performance and scalability, but at the expense of flexibility.

There are implementations which try to take advantage of both the processor-based implementation and the ASIC-based implementation. The intent is to be able to provide scalability and flexibility while maintaining performance.

In environments where dropped packets are infrequent and connections are maintained for long periods of time, the bulk of the overhead in the TCP/IP is the data transmission and reception. The offloading of this overhead if common referred to as a data path offloading. Data path offloading eliminates the TCP/IP overhead in the data transmission/reception phase. The host stack maintains responsibility for the remaining phases (i.e., connection establishment, closing and error handling).

Full offloading executes all phases of the TCP stack in hardware. With full offload, a TOE relieves the host not only from processing data, but also from connection management tasks. In environments where connection management or error handling are intensive tasks, there is a definitive advantage to full offload solutions.

Depending on the end-user application, data path offload or full offload may be equally effective in lowering the host CPU utilization and increasing the data throughput.

3 Understanding TCP/IP Requirements

At a high level, it is generally accepted that TCP/IP is a CPU and I/O intensive protocol stack. The stack in most operating systems is optimized to allow cache hits for the buffers involved in the data transmission and reception. The Linux TCP/IP stack uses advanced techniques such as header prediction to maximize cache hits in a single stream data transfer. However, with the increasing memory sizes, the gap between the cache and the memory sizes is growing, leading to more and more compulsory cache misses. 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 5, 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. These two cases lead to very different memory traffic analyses, which we will study in this section. Figures 6a and 6b 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. For the case when the application buffer fits in the cache, the following are the steps involved on the transmission side:

CPU reads the application buffer: Most micro-benchmark tests are written such that the application buffer is reused on every iteration. The buffer is fetched to cache once and it remains valid throughout the experiment. Subsequent reading of the buffer gets its copy from the cache. So, there is no data traffic on the FSB for this step.

CPU writes to the socket buffer: 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.

NIC does a DMA read of the socket buffer: Most current memory I/O controllers allow DMA reads to proceed from cache. Also, as an optimization, most controllers do an *implicit write back* of dirty cache lines to memory during a DMA read. Since the socket buffer is dirty at this stage, this generates one byte of memory traffic during the DMA operation.

Based on these three steps, in the case where the application buffer fits in the cache, there is one byte of memory traffic for every byte of network data transmitted.

However, 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. In the worst case, this might sum up to as many as three additional memory transactions (one additional explicit write back and one fetch of the socket and the application buffers to cache). 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. Summarizing, in the case where the application buffer fits in cache, in theory there can be between 1-4 bytes of data transferred to or from memory for every byte of data transferred on the network. 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.

For the case when the application buffer does not fit into the cache, the following are the steps involved on the transmission side:

CPU reads the application buffer: The application buffer has to be fetched every time to cache 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: Again, we assume that the socket buffer is small enough to fit into cache. So, once the socket buffer is fetched to cache, it should stay valid throughout the experiment and require no additional 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).

NIC does a DMA read of the socket buffer: Similar to the case where the application buffer fits into the cache, the socket buffer is dirty at this point. When a DMA request

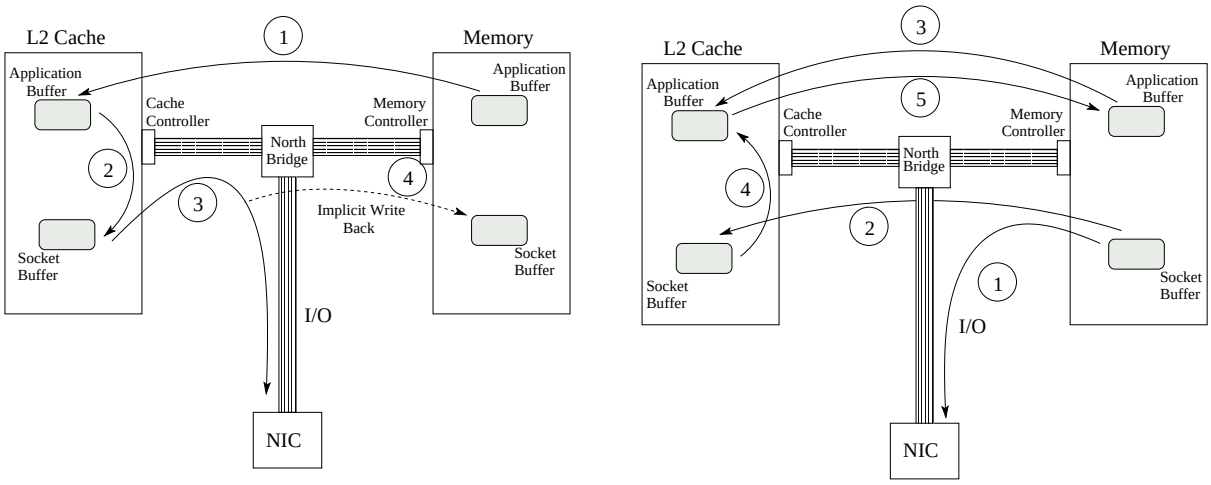


Figure 6. Memory Traffic for Sockets: (a) Transmit Path; (b) Receive Path

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, the memory controller does 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, 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).

As discussed earlier, due to the set-associative nature of the cache, it might be required that the socket buffer be fetched from and written back to memory on every iteration. Since the socket buffer being pushed out of cache and fetched back to cache is already accounted for (due to the large application buffer), the estimated amount of memory traffic does not change in this case.

3.2 Receive Path

The memory traffic associated with the receive path is simpler compared to that of the transmit path. We again consider 2 cases for the receive path: (a) Application buffer fits into cache and (b) Application buffer does not fit into cache. For the case when the application buffer fits into cache, the following are steps involved on the receive path:

NIC does a DMA write into the socket buffer: 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. We'll see in the next couple of steps that the socket buffer will be fetched to the cache so that the data be copied into the application buffer. So for all subsequent iterations, during the NIC DMA write, the socket buffer can be expected to be in the cache. Also, since it is only being used to copy data into the application buffer, it will not be dirtied. 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 6), 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: At this point, the socket buffer is not present in cache (even if the buffer was present in the cache before the iteration, it has to be evicted for the previous step). So, the CPU needs to read the socket buffer into memory. This requires one transaction to the memory during this step.

CPU writes to application buffer: Since the application buffer fits into cache, there is no additional memory traffic during this operation (again, we assume that the cache policy is efficient enough to avoid cache misses due to cache line mappings in set associative caches).

Based on these three steps, we can expect 2 bytes of memory traffic for every byte transferred over the network.

For the case when the application buffer does not fit into the cache, the following steps are involved on the receive path:

NIC does a DMA write into the socket buffer: Since this step involves writing data directly to memory, it is not affected by the cache size or policy and would be similar to the case when the application buffer fits into cache. Thus, this step would create one transaction to the memory.

CPU reads the socket buffer: Again, at this point the socket buffer is not present in cache, and has to be fetched, requiring one transaction from the memory.

CPU writes to application buffer: 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 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.

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 Tools, Benchmarks and Limitations

Ideally, we would have liked to perform all experiments on the same set of hardware, Operating Systems (OS), etc. However, due to limitations imposed by hardware, availability of tools, etc., we had to diverse to similar, but not-exactly-alike hardware configurations. We will highlight the different configurations of the hardware we used and the Operating System on which the different tools were used etc. through the course of this paper.

4.1 Tools and Benchmarks

This section deals with the tools we had used for attaining application and system information on Linux and Windows operating systems. We also briefly present some of the benchmark utilities we used for evaluation purposes.

Most of the evaluation on the Linux Operating System was done using standard system calls such as `getrusage()` which allow applications to collect resource usage information from the operating system. For the evaluation on Windows, we used the *Intel VTune^{TM} Performance Analyzer* which allows for all measurements related to the pareto analysis of the TCP/IP protocol stack, CPI measurements, etc. For measurements related to CPU Utilization, we used the Perfmon utility provided with Windows distributions and the data was re-verified using the *Intel VTune^{TM} Performance Analyzer*. Measurements pertaining to the memory traffic, Front Side Bus (FSB) analysis, Direct Memory Access (DMA), cache misses, etc. were obtained using the *Intel EMon performance tool*.

The *VTune^{TM} sampler* interrupts the processor at specified events (ex: 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. The *EMon event monitoring tool* is used to collect information on the processor and the chipset performance counters.

For the single stream throughput analysis, we used the *ttcp* micro-benchmark test for Linux and the equivalent *NTttcp* test for Windows. For multi-stream analysis, we used the *Chariot* benchmark test. Details regarding each of these benchmarks are provided in Section 5.

4.2 Limitations

The micro-benchmarks used in this paper are mainly data transfer oriented and do not exercise some of the other components of the TCP protocol stack such as connection management, etc. However, these benchmarks serve as ideal case estimates for the peak performance provided by the network and protocol stack combination. For measurement of the memory traffic, cache misses, etc., we used the *EMon performance tool*. However, at the time of the data collection, *EMon* was only available for Windows and for the Intel 82450NX chipsets; the chipset we used in our experiments (E7501) was not supported. Due to this limitation, we could measure only the memory transactions that were initiated by the CPU using *EMon*. For other memory transactions, such as DMA read and write operations, we have made an estimate of the memory traffic based on the CPU initiated memory transactions and the details provided in Section 3.

5 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.

5.1 10-Gigabit Ethernet

In this section, we evaluate the performance of the host TCP/IP stack over 10 Gigabit Ethernet. We have carried out tests in three broad directions: (1) Study of the performance of the TCP/IP stack in the form of micro-benchmark tests, both for single stream as well as multi-stream cases, (2) Study of the overall CPU requirements of the TCP/IP stack and the module wise analysis (pareto analysis) of the observed CPU usage and (3) Study of the memory traffic associated with the TCP/IP network traffic.

5.1.1 Single Stream Micro-Benchmarks

Figure 7a 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. Figure 8a shows the equivalent experiment on the Linux platform. We can see that 10-Gigabit Ethernet is able to achieve a latency of about 20.5μs on Linux with a CPU utilization of about 45% on each side.

Figure 7b shows the throughput achieved by 10-Gigabit Ethernet. 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 is provided in the later sections. The results for the linux platform (Figure 8b) are similar.

Figure 9 shows the impact of MTU size on the throughput achieved by 10-Gigabit Ethernet. In general, we observe an improvement in the throughput with the MTU size. This is attributed to the per-packet overhead associated with the TCP/IP stack such as per-packet interrupts, etc. An increase in the MTU size results in a lesser number of packets to be transmitted, thus leading to a lesser per-packet overhead.

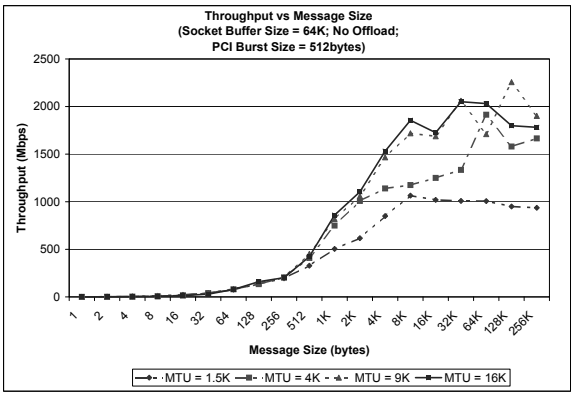
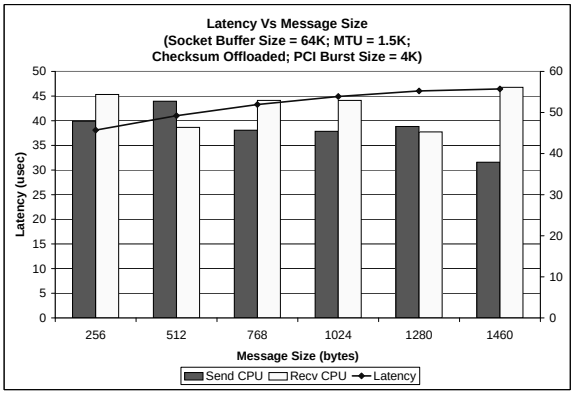


Figure 9. Impact of MTU on throughput (Windows)

Figure 10 shows the impact of the socket buffer size on the throughput achieved by 10-Gigabit Ethernet. We can observe that an increase in the receive buffer size results in an improvement in the performance while an increase in the send buffer size results in a drop in performance. We believe that this is due to the Silly Window Syndrome of TCP/IP. It is to be noted that the receiver side has a higher overhead in the TCP/IP stack than the sender side. So, if the sender window is larger than the receiver window (e.g., when the send socket buffer is 64Kbytes and the receive socket buffer is 128Kbytes), the sender pushes out some data and as soon as it receives an acknowledgment, which is not completely as large as the MTU, it sends out the next data segment and so on. This results in an inefficient usage of the available bandwidth. This trend is similar to the one observed by Feng, et. al, in [20].

Figure 11 shows the impact of checksum and segmentation offloads on the performance achieved by 10-Gigabit Ethernet. It is to be noted that the *segmentation* legend refers to both checksum and segmentation offloading while the



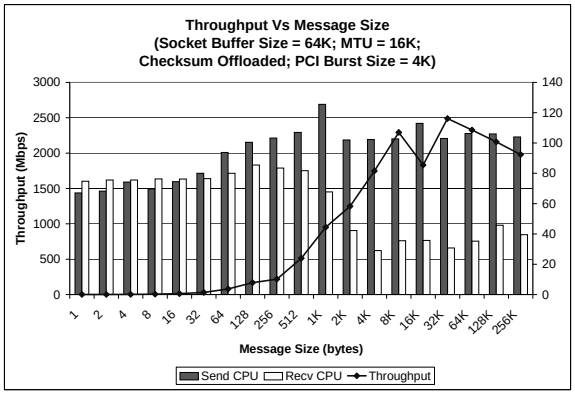
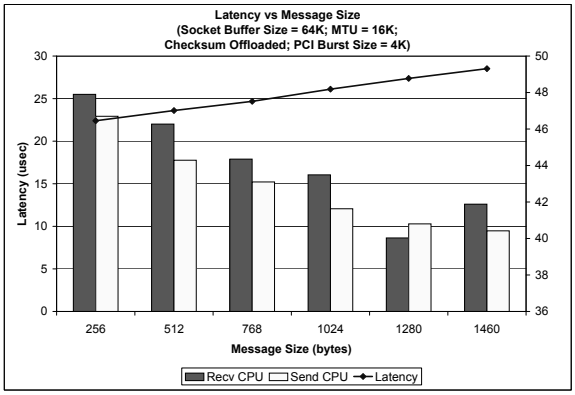


Figure 7. Micro-Benchmarks for the host TCP/IP stack over 10-Gigabit Ethernet on Windows: (a) One-Way Latency (MTU 1.5K); (b) Throughput (MTU 16K)



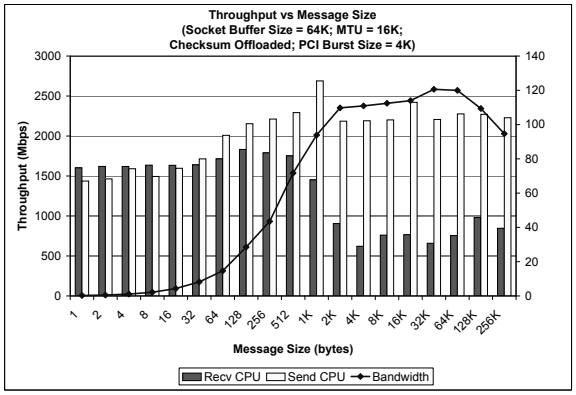


Figure 8. Micro-Benchmarks for the host TCP/IP stack over 10-Gigabit Ethernet (Linux): (a) One-Way Latency (MTU 1.5K); (b) Throughput (MTU 16K)

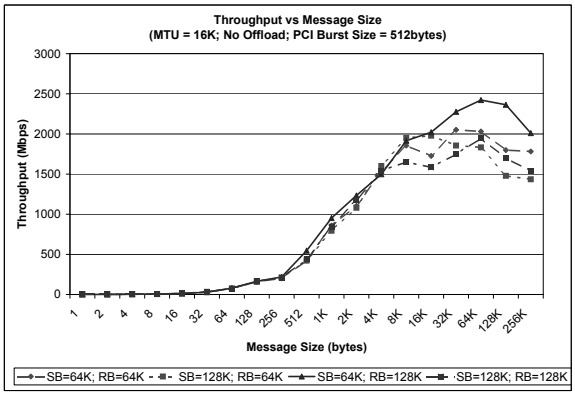


Figure 10. Impact of Socket buffer size on throughput (Windows)

checksum legend refers to the checksum offloading alone. We can see that checksum offload leads to a slight improvement in the performance. Also, we observe that segmentation offload leads to a drop in the performance for large messages. This is due to the entire message DMA carried out by the adapter. The adapter performs a DMA for a large chunk of the data and transmits one segment at a time. This results in some loss of pipelining for the message transmission. For small and moderate messages, the benefit due to segmentation offload overshadows the degradation due to the loss of pipelining, but for large messages this becomes observable.

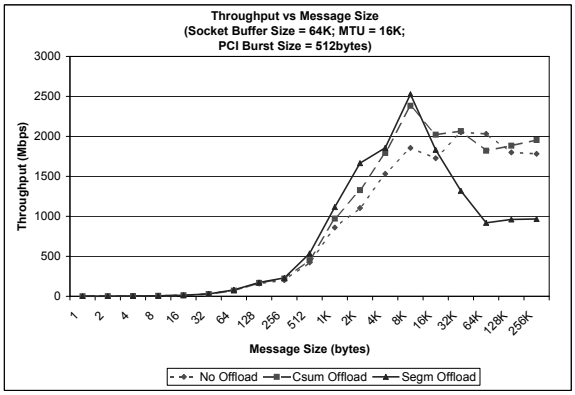


Figure 11. Impact of Offload on throughput (Windows)

Figure 12 shows the impact of the PCI burst size on the throughput achieved by 10-Gigabit Ethernet. A larger PCI burst size refers to the DMA engine performing DMA operations for larger segments of data. The 10-Gigabit Ethernet adapter we used supports PCI burst sizes of 512bytes and 4Kbytes. It is to be noted that though a large PCI burst size leads to lesser DMA operations, it could result in loss of pipelining between the DMA operation and the transmission operation (more detailed studies about the DMA operation and transmission pipelining are present in [22]). However, for fast networks such as 10-Gigabit Ethernet, the benefit from the number of DMA operations is more desirable due to the lesser speed of the I/O bus (PCI-X in our case) compared to the network speed.

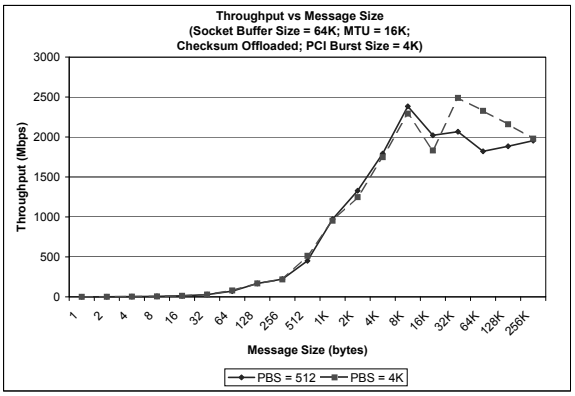


Figure 12. Impact of PCI Burst Size on throughput (Windows)

Figures 13 to 15 show a similar analysis for the Linux platform. We see that the trends are similar to the Windows platform.

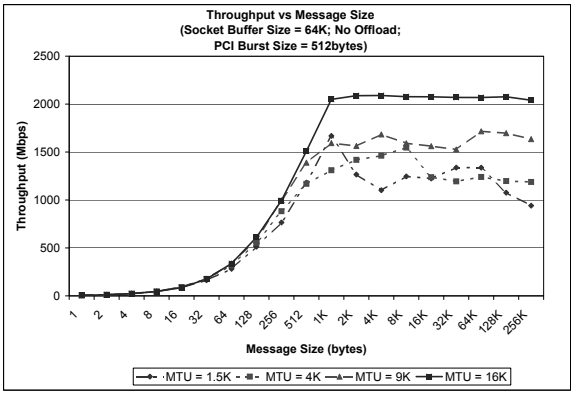


Figure 13. Impact of MTU on throughput (Linux)

5.1.2 Single Stream 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 analyze this. Like other

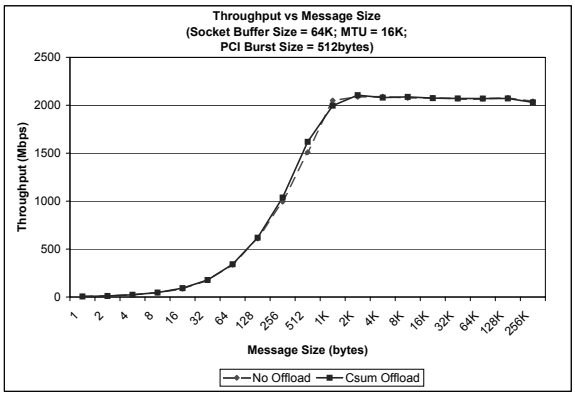


Figure 14. Impact of Offload on throughput (Linux)

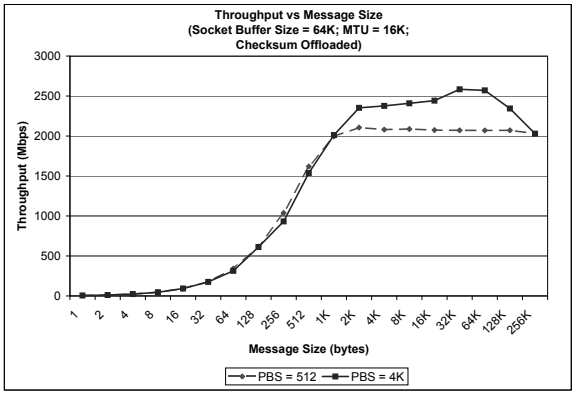


Figure 15. Impact of PCI Burst Size on throughput (Linux)

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.

Figures 16 and 17 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.

5.1.3 Single Stream Memory Traffic

Figure 18 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

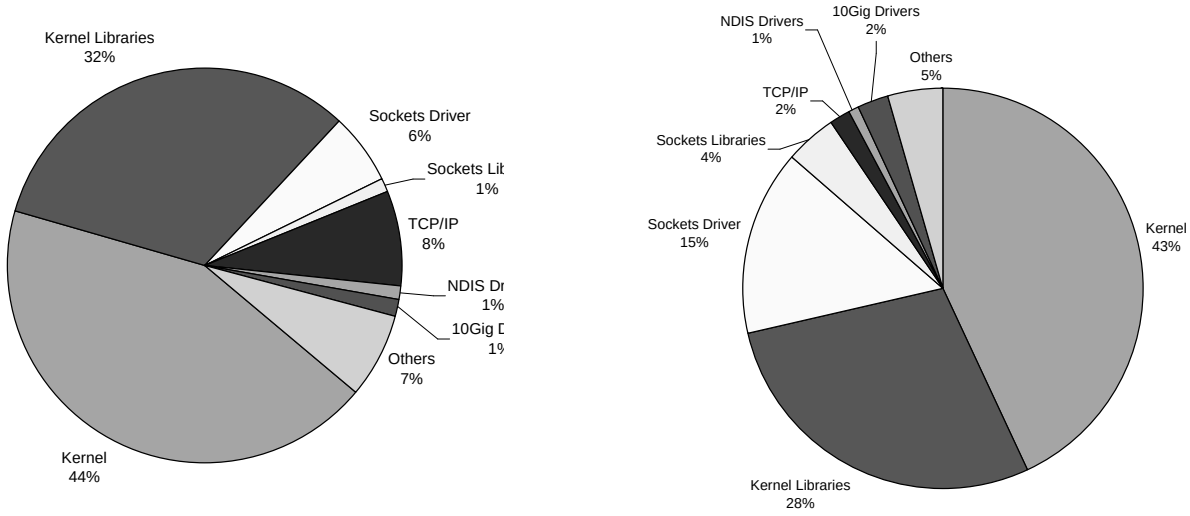


Figure 16. Throughput Test: CPU Pareto Analysis for small messages (64bytes): (a) Transmit Side, (b) Receive Side

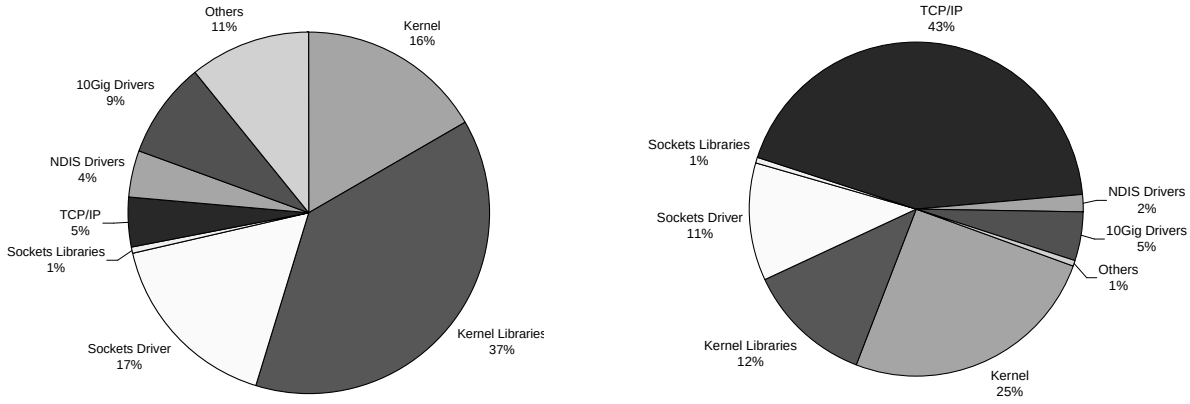


Figure 17. Throughput Test: CPU Pareto Analysis for large messages (16Kbytes): (a) Transmit Size, (b) Receive Side

side. We can see that the actual memory to network traffic ratio follows this trend.

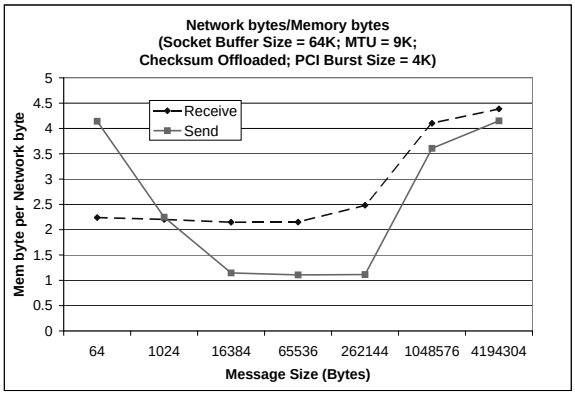


Figure 18. Single Stream Throughput Test: Memory Traffic Analysis

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.

5.1.4 Multi Stream Micro-Benchmarks

In this section, 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 19a and 19b 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 19c 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).

5.1.5 Multi Stream Memory Traffic

Figure 20 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.

5.2 InfiniBand Architecture

In this section, we evaluate the various communication models over the InfiniBand network with respect to ideal case Performance, CPU Utilization and Memory traffic associated with network traffic.

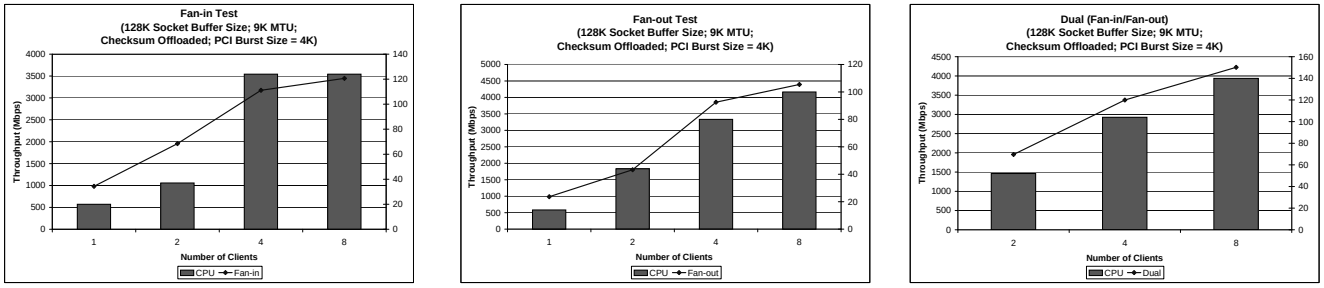


Figure 19. Multi-Stream Micro-Benchmarks: (a) Fan-in, (b) Fan-out, (c) Dual (Fan-in/Fan-out)

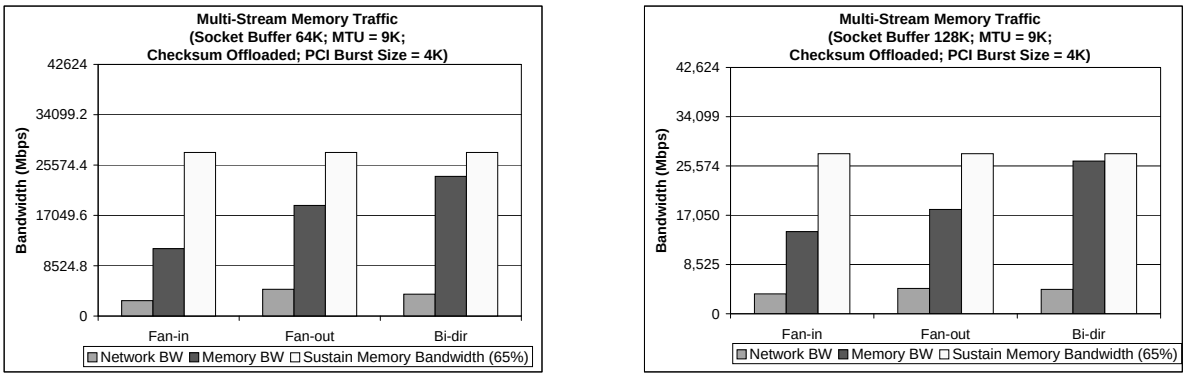


Figure 20. Multi Stream Throughput Test: Memory Traffic Analysis: (a) Socket Buffer Size = 64Kbytes; (b) Socket Buffer Size = 128Kbytes

5.2.1 One-Way Latency

Figures 21a to 24a show the one-way latencies achieved by the different communication models of the InfiniBand stack for a 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 completion descriptor. 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 adapter raises an interrupt for the application once the message arrives. This results in a lesser CPU utilization for the event based scheme.

As seen in Figure 21a, the Send/Receive communication model of the InfiniBand stack both for a polling based approach for completion as well as an event based approach. The polling based approach gives a latency of about 7.5μs as compared to a 20.5μs latency achieved by the event-based approach. However, the polling based approach requires a 100% CPU utilization (not shown in the figure) as compared to a 25% CPU utilization for the event-based approach.

Further, 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 (Figure 22a). The reason for both the event based scheme ans the polling based scheme performing alike is the receiver transparency for RDMA Write operations. Since, the RDMA 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 block 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). The CPU results have been skipped in this figure because of this behavior of the event-based approach of the RDMA Write operation.

Figure 23a also shows the one-way latency achieved by the RDMA Write with Immediate Data communication model of InfiniBand. Again, results for both the polling based model and the event-based model are provided. The latencies achieved by the two models are similar to the send-receive model, i.e., about 7.5μs for the polling based scheme and about 20.5μs for the event based scheme.

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 (Figure 24a). The higher latency for the RDMA Read scheme compared to the other schemes is due to round-trip semantics of the RDMA read operation, i.e., the local network adapter has to send the initiation message to the remote network adapter, which in turn returns the required data, thus requiring a round-trip latency.

5.2.2 Single Stream Throughput

Figures 21b to 24b show the throughputs achieved by the different communication models of the InfiniBand stack for a polling based approach for completion as well as an event based approach. All 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.

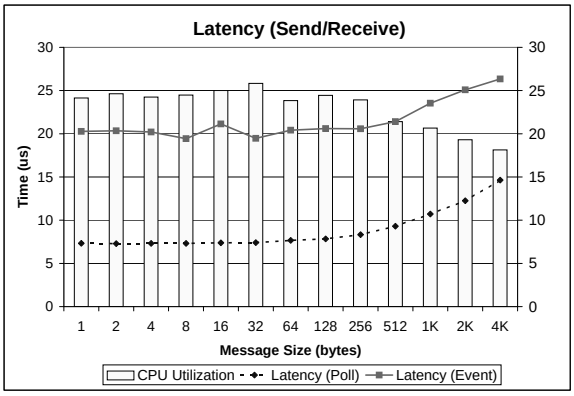
The CPU utilization values are only presented for the event-based approach; those for the polling based approach stay close to 100% and are not of any particular interest. The interesting thing to note is that for RDMA, there is nearly zero CPU utilization for the data sink especially for large messages.

5.3 10-Gigabit Ethernet/InfiniBand Comparisons

Figures 25a and 25b 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 26a and 26b 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



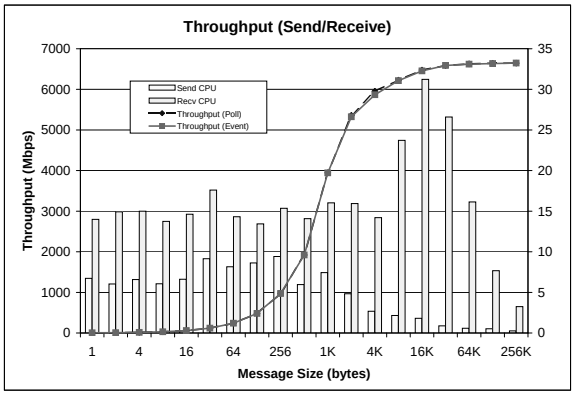
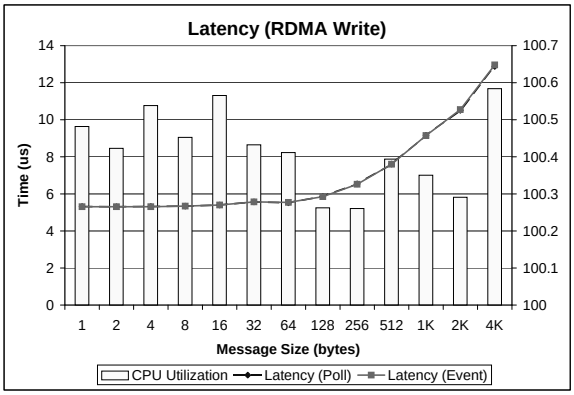


Figure 21. IBA Micro-Benchmarks for Send-Receive: (a) Latency and (b) Throughput



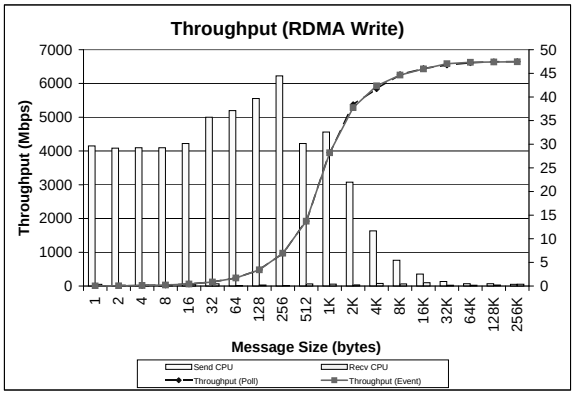
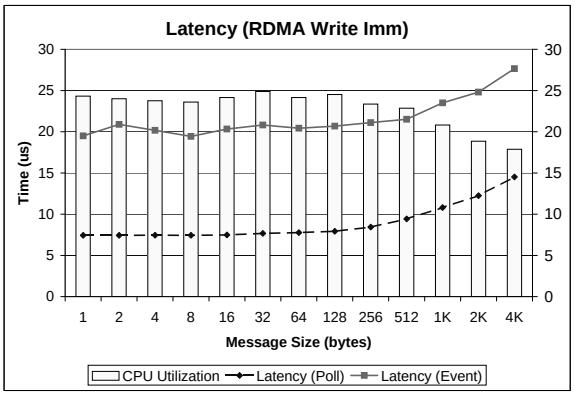


Figure 22. IBA Micro-Benchmarks for RDMA Write: (a) Latency and (b) Throughput



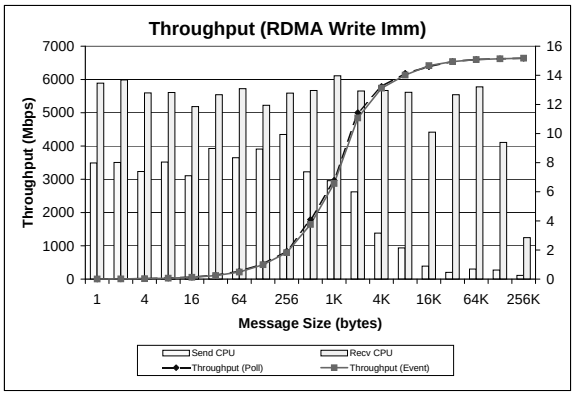
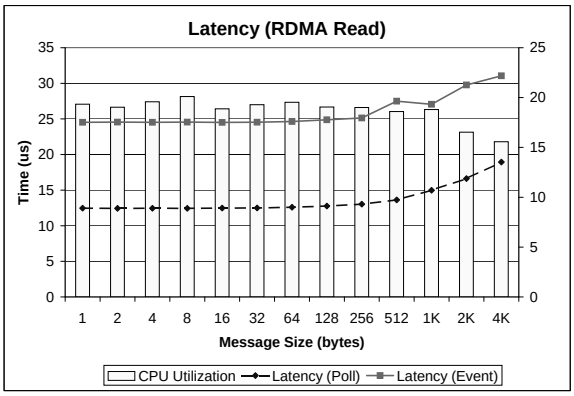


Figure 23. IBA Micro-Benchmarks for RDMA Write with Immediate Data: (a) Latency and (b) Throughput



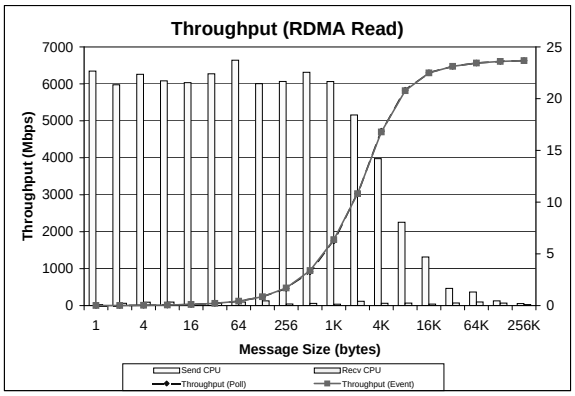
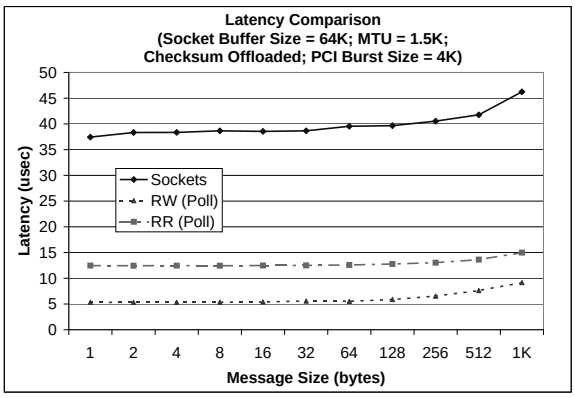


Figure 24. IBA Micro-Benchmarks for RDMA Read: (a) Latency and (b) Throughput



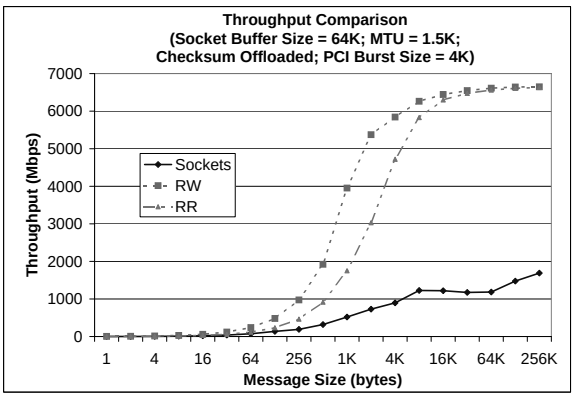
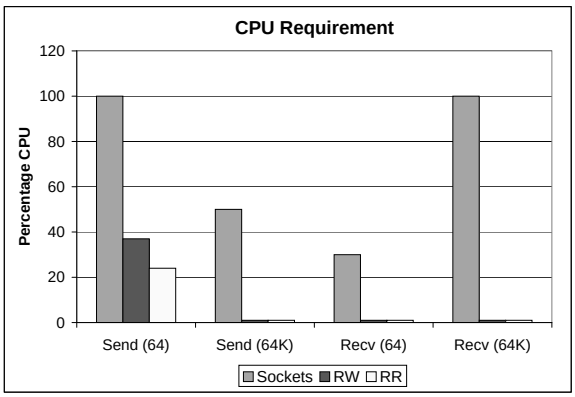


Figure 25. Latency and Throughput Comparison: Host TCP/IP over 10-Gigabit Ethernet Vs InfiniBand



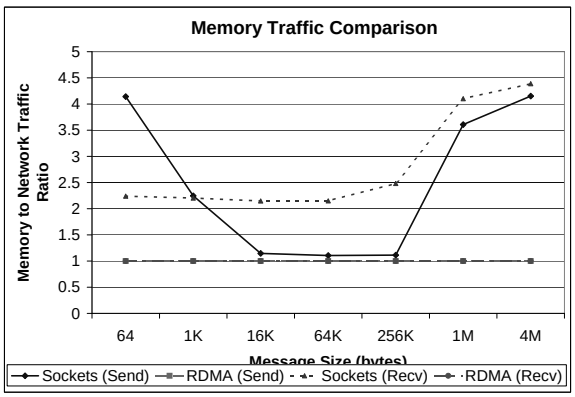


Figure 26. CPU Requirement and Memory Traffic Comparisons: Host TCP/IP over 10-Gigabit Ethernet Vs InfiniBand

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.

6 Related Work

Several researchers have worked on implementing high performance user-level sockets implementations over high performance networks. Balaji, Shah, and several others have worked on such pseudo sockets layers over Gigabit Ethernet, GigaNet cLAN [33] and InfiniBand [6]. 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. [17] 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.

We would also like to mention some previous research to optimize the TCP stack [13, 21, 26, 23]. 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.

7 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 performance, 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 requirement 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.

8 Acknowledgments

We would like to thank Annie Foong for all the help she provided while using the VTune and EMon 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 Specifications. http://www.infinibandta.org/estore.html.

[3] M-VIA: A High Performance Modular VIA for Linux. http://www.nersc.gov/research/FTG/via.

[4] Quadrics Supercomputers World Ltd. http://www.quadrics.com/.

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

[6] Pavan Balaji, Sundeep Narravula, Karthikeyan Vaidyanathan, Savitha Krishnamoorthy, Jiesheng Wu, and Dhabaleswar K. Panda. Sockets Direct Protocol over InfiniBand in Clusters: Is it Beneficial? In *the Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Austin, Texas, March 10-12 2004.

[7] Pavan Balaji, Piyush Shivam, Pete Wyckoff, and Dhabaleswar K. Panda. High Performance User Level Sockets over Gigabit Ethernet. In *the Proceedings of the IEEE International Conference on Cluster Computing*, pages 179–186, Chicago, Illinois, September 23-26 2002.

[8] Pavan Balaji, Jiesheng Wu, Tahsin Kurc, Umit Catalyurek, Dhabaleswar K. Panda, and Joel Saltz. Impact of High Performance Sockets on Data Intensive Applications. In *the Proceedings of the IEEE International Conference on High Performance Distributed Computing (HPDC)*, pages 24–33, Seattle, Washington, June 22-24 2003.

[9] M. Banikazemi, B. Abali, L. Herger, and D. K. Panda. Design Alternatives for VIA and an Implementation on IBM Netfinity NT Cluster. *Special Issue of the Journal of Parallel and Distributed Computing (JPDC), Vol. 61, No. 11, pp. 1512-1545*, November 2001.

[10] M. Banikazemi, V. Moorthy, L. Hereger, D. K. Panda, and B. Abali. Efficient Virtual Interface Architecture Support for IBM SP switch-connected NT clusters. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, 2000.

[11] 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.

[12] P. Buonadonna, A. Geweke, and D. E. Culler. BVIA: An Implementation and Analysis of Virtual Interface Architecture. In *Proceedings of Supercomputing*, 1998.

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

[14] GigaNet Corporations. cLAN for Linux: Software Users' Guide.

[15] Myricom Corporations. The GM Message Passing System.

[16] 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.

[17] Annie Foong, Herbet Hum, Tom Huff, Jaidev Patwardhan, and Greg Regnier. TCP Performance Revisited. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Austin, Texas, 2003.

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

[19] G. Herrin. Linux IP Networking: A Guide to the Implementation and Modification of the Linux Protocol Stack. http://kernelnewbies.org/documents/ipnetworking, May 2000.

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

[21] V. Jacobson. 4BSD Header Prediction. In *ACM SIGCOMM*, pages 13–15, April 1990.

[22] Hyun-Wook Jin, Pavan Balaji, Chuck Yoo, Jin-Yong Choi, and Dhabaleswar K. Panda. Exploiting NIC Architectural Support for Enhancing IP based Protocols on High Performance Networks. Technical report, Ohio State University, Columbus, Ohio, May 2004.

[23] J. Kay and J. Pasquale. The Importance of Non-Data Touching Processing Overheads in TCP/IP. In *ACM SIGCOMM*, San Francisco, September 1993.

[24] Jin-Soo Kim, Kangho Kim, and Sung-In Jung. Building a High-Performance Communication Layer over Virtual Interface Architecture on Linux Clusters. In *the Proceedings of the IEEE International Conference on Supercomputing (ICS)*, pages 335–347, Naples, Italy, June 16-21 2001.

[25] Jin-Soo Kim, Kangho Kim, and Sung-In Jung. SOVIA: A User-level Sockets Layer Over Virtual Interface Architecture. In *the Proceedings of the IEEE International Conference on Cluster Computing*, pages 399–408, California, USA, October 8-11 2001.

[26] P. E. McKenney and K. F. Dove. Efficient Demultiplexing of Incoming TCP Packets. In *ACM SIGCOMM*, pages 269–280, Baltimore, MD, August 1992.

[27] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM). In *Proceedings of Supercomputing*, 1995.

[28] 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.

[29] Quadrics Supercomputers World Ltd. Elan Programming Manual. 1999.

[30] Quadrics Supercomputers World Ltd. Elan Reference Manual. 1999.

[31] Quadrics Supercomputers World Ltd. Elite Reference Manual. 1999.

[32] A. Rubini and J. Corbet. Linux Device Drivers. O'reilly, 2001.

[33] Hemal V. Shah, Calton Pu, and Rajesh S. Madukkarumukumana. High Performance Sockets and RPC over Virtual Interface (VI) Architecture. In *the Proceedings of the CANPC workshop (held in conjunction with HPCA Conference)*, pages 91–107, 1999.

[34] Piyush Shivam, Pete Wyckoff, and Dhabaleswar K. Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing. In *the Proceedings of the IEEE International Conference on Supercomputing*, pages 57–64, Denver, Colorado, November 10-16 2001.

[35] Piyush Shivam, Pete Wyckoff, and Dhabaleswar K. Panda. Can User-Level Protocols Take Advantage of Multi-CPU NICs? In *the Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, Fort Lauderdale, Florida, April 15-19 2002.

[36] W. Richard Stevens. *TCP/IP Illustrated, Volume I: The Protocols*. Addison Wesley, 2nd edition, 2000.

[37] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated, Volume II: The Implementation*. Addison Wesley, 2nd edition, 2000.

[38] Eric Yeh, Herman Chao, Venu Mannem, Joe Gervais, and Bradley Booth. Introduction to TCP/IP Offload Engine (TOE). http://www.10gea.org, May 2002.