Loosely Coupled TCP Acceleration Architecture

Leah Shalev, Vadim Makhervaks, Zorik Machulsky, Giora Biran, Julian Satran, Muli Ben-Yehuda, Ilan Shimony

IBM Haifa Research Lab, Haifa, Israel
leah@il.ibm.com

Abstract

We present a novel approach for scalable network acceleration. The architecture uses limited hardware support and preserves protocol processing flexibility, combining the benefits of TCP offload and onload. The architecture is based on decoupling the data movement functions, accelerated by a hardware engine, from complex protocol processing, controlled by an isolated software entity running on a central CPU. These operate in parallel and interact asynchronously. We describe a prototype implementation which achieves multi-gigabit throughput with extremely low CPU utilization.

1. Introduction

Server network stack performance problems are well known, and so are the controversial solutions that involve full protocol offload to the adapter. While partial offload techniques have gained wide acceptance, full TCP offload remains debatable. One of its main problems is lack of durable advantage over the host CPU, both in terms of performance and adequate protocol support. Solutions based on embedded processors can be quickly outperformed by the host CPU, while solutions based on custom hardware suffer from lack of flexibility in evolving protocol processing. In addition, full offload significantly decreases the OS control over the network processing, which is perceived as a threat to system robustness. The same set of problems exists for special forms of TCP offload adapters, such as RDMA NICs and iSCSI HBAs.

An evolving alternative to TCP offload is TCP onload, based on general-purpose hardware. With this approach, the system operates in asymmetric multiprocessing mode, where one of the main CPUs is dedicated to TCP/IP processing. Although this approach can yield significant performance improvement compared to a regular stack, it does not solve network scalability problem, since it does not address some of the fundamental limitations of current stack architecture.

We present a novel scalable network acceleration architecture that combines the efficient data movement capabilities of the offload approach with the protocol processing flexibility of the onload. A hardware data placement engine performs data-intensive operations and works in concert with a software stack that handles complex protocol processing, prone to future modifications. These components are loosely-coupled, i.e., they exchange state information asynchronously, without waiting for each other. The architecture is applicable to plain TCP offload, as well as RDMA or iSCSI implementation.

We prototyped the architecture using a programmable adapter with flexible multi-channel DMA capabilities that emulated the hardware engine and provided an efficient application interface. The protocol processing was implemented on one of the main CPUs of an SMP system, while applications executed on another CPU. Our results show that this architecture achieves high throughput with CPU utilization that is an order of magnitude lower than that of a standard stack, and several times lower than that of TCP onload ([15]).

This paper is structured as follows: Section 2 surveys full TCP offload, onload and other related work. Section 3 describes our architecture in detail. Section 4 presents a prototype implementation, and Section 5 evaluates its performance. Finally, we present our conclusions and future directions in Section 6.

2. Background and Related Work

Performance limitations of server TCP/IP networking are well-known ([3, 11, 12]). Increasing TCP/IP performance and scalability has been a major research area for the systems and networking communities ([1, 2]). Many incremental improvements, such as TCP checksum offload, were introduced and have since become widely adopted. However, these improvements only serve to keep the problem from getting worse over time, as they do not solve the network scalability problem caused by increasing disparity of improvement of CPU speed, memory bandwidth, memory latency and network bandwidth. At multi-gigabit data rates, TCP/IP processing is still a major source of system overhead. Moreover, this overhead in effect limits the achievable network bandwidth and server utilization.

For example, a recent work [15] indicates that TCP/IP processing on state-of-the-art platforms consumes one entire CPU to achieve 1 Gbit/s of throughput when transmitting data, or 750 Mbit/s when receiving data.

There are several recognized elements of TCP/IP processing that require improvement on modern systems ([8, 15]). One area for improvement is the cost of integration with OS mechanisms such as system calls and interrupt processing. A significant portion of this overhead is contributed by network buffer management and other per-packet overhead. On SMP systems, locking is also an important source of overhead.

Another issue is inefficient operation of memory subsystem, related to the overall stack architecture, and affecting both the software implementation and interaction between the CPU and the network adapter. Poor cache locality and excessive data copy operations, combined with high memory latency, make the existing stack architecture unsuitable for the server networking.

Three approaches to the problems described above are TCP/IP offload engines (*TOE*), Remote Direct Memory Access (*RDMA*), and TCP onload.

2.1 TCP Offload and RDMA NICs

Full TCP offload attempts to solve the problem by moving all TCP/IP processing to the network interface adapter. This technique has been pursued for a long time, but has not gained acceptance so far. Even though some encouraging results were reported (e.g., *QPIP* [13]), the approach remains highly controversial ([6, 7]). A summary of the criticisms of TCP offload can be found in [4]; a detailed response can be found in [8].

One particular point of the TOE debate is the issue of a potential bottleneck created by an offload adapter. TOE implementations range from mostly-firmware to hardware-only solutions, with varying levels of special-purpose hardware on the adapter. TCP offload adapters that make extensive use of embedded CPUs typically suffer more performance problems, in particular because they face scalability problems similar to those of the host CPU. On the other hand, a hardware state machines implementation is potentially more efficient and less power-consuming, but also more expensive to develop and not flexible enough to support protocol modifications and bug fixes.

The lack of flexibility in the protocol processing hinders the acceptance of TOE implementations. Even if the adapter's internal implementation is flexible, only the device vendor can utilize this flexibility. Most operating systems do not even provide a standard way to integrate a full offload solution with the native OS stack.

RDMA NIC (RNIC) approach can be viewed as an extension of the TOE technique. TCP processing is done within the adapter, where the adapter is operated through a standardized interface, which simplifies the integration with the OS. The RDMA adapter interface is an asynchronous direct-access interface, similar to the Infiniband interface [14]. In RDMA mode, instead of using streaming semantics, the applications use message semantics and memory semantics. Boundaries of messages posted by the host on transmit side are identified on the wire and correspond to boundaries of messages delivered to the receiver. In addition, the sender can specify the destination in the receiver memory. This allows a much more efficient interface between the applications and the adapter, as compared to plain TCP offload.

The main drawback of this approach is lack of interoperability with "legacy" TCP applications. Also, as in the case of TOE, RDMA solutions may suffer from a lack of flexibility or a lack of OS control over TCP processing.

2.2 TCP Onload

Recently, "TCP onload" was proposed as a new approach to improve the "conventional" implementation of a network stack ([9, 10, 15]). The concept is based on an asymmetric multi-processing mode, when at least one of the CPUs on a multiprocessor system is dedicated to network stack processing. The *TCP Servers* project [5] is an earlier work that demonstrated the value of a similar approach.

A significant drawback of the onload approach is that it does not reduce the load on the memory subsystem; in particular, it does not eliminate receive data copy. At most, it can hide its latency, for example by using a general-purpose DMA engine, which executes the memory copy asynchronously. This can solve the problem of CPU stalls and pollution of the cache. However, the copy operations, although executed in the background, still affect the memory bus performance, which in turn affects the CPU performance, and ultimately limits the ability to scale both the number of CPUs and achievable network bandwidth.

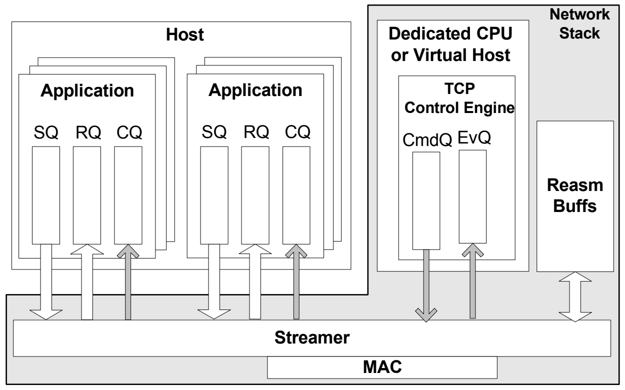
3. Asynchronous Split Architecture

We propose a hybrid approach that combines the benefits of both offload and onload while eliminating their drawbacks. As in the offload approach, the application CPU uses a hardware interface to interact with an "offloaded" network stack. However, network stack processing is not fully offloaded to the network interface adapter. Instead, we use an "asynchronous split" stack architecture. All data processing is performed by a dedicated hardware acceleration engine, while the protocol control operations are done by software, executed on a dedicated main CPU, resembling the onload approach.

The adapter provides hardware support to achieve true zero copy operation, which involves minimal protocol awareness. Most of the protocol processing, prone to future changes, is performed by assisting software, loosely coupled with the hardware. The split of responsibilities allows independent operation of both parts, thus avoiding bottlenecks related to their interaction. In particular, the architecture allows lock-free operation.

3.1 Major Components

The architecture, shown in Figure 1, includes three major components: consumer interface, hardware acceleration engine called *Streamer*, and TCP Control Engine (*TCE*) software.



SQ - Send Queue
RQ - Receive Queue
CQ - Completion Queue

CmdQ - TCE Command Queue
EvQ - Streamer Event Queue

Figure 1. System Architecture

- Consumer interface - the interface used by the user or kernel space applications. It is implemented as an access library, which may provide different types of semantics. An asynchronous queue-based mechanism is used to submit application work requests directly to the Streamer, and to receive completion notifications.
- Streamer - hardware acceleration engine which, in conjunction with the TCE, provides network acceleration services to the consumers. This engine is responsible for handling data intensive operations.
- TCP Control Engine (TCE) - a software component that implements most of the protocol processing. It carries out the decision-making part of the TCP protocol. It is not involved in data movement for most data packets. It does not need to keep track of message or packet boundaries, and does not maintain any equivalent of *mbuf* or *skbuf* data structures.

The Streamer and the TCE use an asynchronous interface to exchange information without handshake or locking. This interface consists of two queues: *Command Queue* is used to pass commands from the TCE to the Streamer and *Event Queue* is used to pass information on the events processed by the Streamer to the TCE. Also, the TCE does not need to use locks internally, because it is executed as a single loop; it polls for new entries in the Event Queue, instead of using interrupt-based notifications.

The Streamer and TCE perform TCP processing asynchronously; each uses its own copy of the relevant connection context fields. Consider an event generated by the Streamer that has to be processed by the TCE to accomplish a part of TCP processing. The Streamer does not wait until the TCE processes the event. Instead, it just updates its own view of the (partial) connection state. The connection state replicas are “weakly consistent”, where at any given time the TCE view of connection state may be different from that of the Streamer. Processing of independent TCP events may be reordered, for example the Streamer can handle data that arrives after an ACK before the TCE handled that ACK. The architecture assumes that many commands or events, possibly pertaining to the same connection, may be accumulated before they are actually processed. The context fields and the contents of Events and Commands exchanged by the TCE and the Streamer were designed to allow parallel TCP processing. In the sections below we describe the split of responsibility between the Streamer and TCE, separately for receive and transmit flows.

3.2 Transmit Flow

Data transmission requests originate from the consumer, which posts the requests to its Send Queues and notifies the Streamer of the new requests. The Streamer examines the connection context, decides whether the connection should be served, and schedules it for transmit execution. It then calculates the amount of data that can be sent for the given connection. This calculation is based on transmit window and the amount of data available in the consumer buffers. The transmit window information, namely the send window and the congestion window, is kept in the connection context, and is indirectly updated by the TCE via the Command Queue, using context update commands.

Once the connection is scheduled for actual transmit operation, the Streamer fetches the required amount of data, performs segment generation using a small subset of TCP connection context, and transmits the generated segments via the MAC interface.

The Streamer may transmit more than one packet for the selected connection; when it is done handling the connection, the Streamer provides to the TCE, via the Event Queue, a transmit event for this connection, carrying the connection ID and the TCP sequence numbers range of the transmitted data.

The TCE tracks transmitted data using the transmit events described above, ACK receive events described in *Section 3.3*, and internal timer events. When the TCE detects a retransmission condition, it passes a retransmit request to the Streamer via the Command Queue, providing the connection ID, the sequence number from which to perform retransmit and the amount of data to retransmit.

3.3 Receive Flow

Each arriving packet is first handled by the Streamer, which places the data either directly on the destination buffers or on reassembly buffers, and passes a receive event to the TCE. The TCE then completes the packet processing.

3.3.1 Streamer Receive Handling. For each arriving TCP segment, the Streamer first performs its basic validation and classification. This includes TCP checksum validation, TCP four-tuple lookup, and identifying whether the received segment belongs to the fast path. The meaning of fast-path is much broader than in other systems, as described below.

A segment is identified as a fast-path segment if it passes a basic TCP validation sequence, which includes the following checks: the timestamp option is valid, the data sequence number is within the receive window, the flags are valid, and the Ack number is valid (i.e., acknowledges recently sent data). This validation involves access to a small subset of TCP connection context. It is sufficient to allow data placement to consumer buffers and delivery to the consumer without TCE involvement. Note that a valid out-of-order segment is handled as a fast-path segment. Note also that a valid packet that carries both data and new ACK information is handled as a fast-path segment, unlike in classical header prediction fast path.

The Streamer processes the TCP payload of a fast-path segment without waiting for the TCE. It retrieves the location of consumer buffers, places the payload on these buffers, and generates a completion.

If a segment does not pass the validation, or buffers for the direct data placement are not available, it is treated as a slow-path segment. The Streamer places the payload of slow-path segments on the reassembly area; the Streamer is responsible for reassembly buffers management. The TCE can later instruct the Streamer to process the valid data from the reassembly buffer, i.e. place it to the consumer buffers and deliver it to the consumer.

For both fast and slow paths, in parallel with the data placement, the Streamer passes to the TCE a receive event entry via the Event Queue. The event entry carries the TCP header and additional information, such as the connection ID and packet status.

3.3.2 TCE Receive Handling. The TCE retrieves the TCP headers from the Event Queue, and performs control processing, such as TCP congestion control, window management and RTTM estimation. As a part of the ACK processing, it can pass the Streamer a transmit context update command. As a part of the data header processing, it either passes the Streamer an ACK generation command or schedules a delayed ACK.

For the slow-path segments, the TCE performs an extended validation sequence, covering various corner cases (e.g., ambiguous timestamp information). It also implements reassembly control for valid slow-path segments; when reassembly is completed, the TCE passes a command to the Streamer via the Command Queue, asking it to fetch the data from the reassembly buffers, process it, and place it on the destination buffers. This relieves the TCE from copying the data or alternatively waiting until an asynchronous copy completes.

4. Prototype

In order to evaluate the architecture, we built a prototype on an SMP machine with one of the processors dedicated for TCP control processing. We used dual-CPU servers with Xeon 1.8 GHz CPUs; hyperthreading was disabled to simplify the analysis. The servers ran Redhat Linux, patched in such a way as to dedicate a single CPU to TCP Control Engine processing. We used the Linux *setaffinity* system calls to bind all userspace tasks to CPU#0. Additionally, we disabled the IRQ balancing code so that all interrupts were delivered to CPU#0. CPU#1 was dedicated to running the TCP Control Engine code.

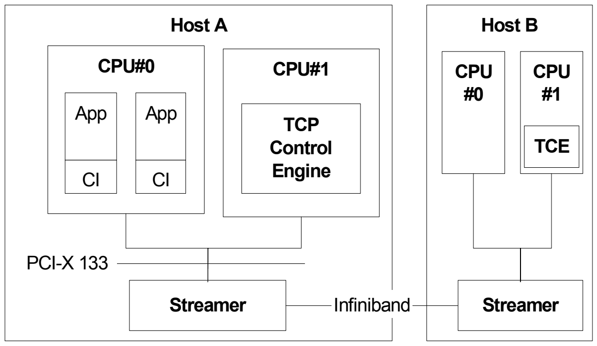


Figure 2. Prototype environment

To facilitate the prototyping, we used an available high-speed programmable adapter with multi-channel DMA capabilities. We implemented the missing hardware features in the adapter firmware. The adapter which we used was a dual port 4x[1] Infiniband HCA. The microcode on the adapter emulated Ethernet (i.e., packetized the data) and provided the Streamer functions described in the previous section, using the existing DMA support.

4.1 TCE Implementation

We implemented the TCE as a kernel thread that took over a CPU. It polled the protocol Event Queue and a timer event queue in a loop. When the queues were empty, the TCE stopped polling for 100 μs to allow the queues to fill. Conceptually, it could yield the CPU in such cases. In practice, in order to avoid scheduling problems, the TCE did not actually yield the CPU, but just performed a busy wait. The

[1] Single data rate 4X link that carries 10 Gbit/s of raw data, or 8 Gbit/s of user data due to 8/10 encoding.

busy wait time was counted as idle time.

The TCE supported a wide set of TCP features, as expected from a modern TCP implementation. The features affecting the regular path execution included retransmission timers, delayed ack timer, NewReno congestion control, windows scaling, timestamp option, and dynamic round-trip time (RTT) estimation. The TCE implementation did not directly use any kernel services, except infrequent timer operations, as described below.

A timer handler routine was used to generate "TCP clock ticks" each 100 ms; it placed the events to a timer queue, implemented as a simple cyclic buffer. The TCE periodically checked for new "tick" events on the timer queue, examined timeout control data structures and handled the detected timeout expiration events.

In the initial stages of the implementation, profiling revealed that the TCP processing itself took much less time than the interaction with the Streamer, due to cache misses. Therefore, we paid special attention to optimization of the Command Queue and Event Queue in terms of cache behavior. For example, we found that writing commands consumed a significant part of the TCE CPU (4.5% out of approximately 16%). Cache misses upon the command entry access caused CPU stalls because the number of fields that were written to one command entry exceeded the capacity of the CPU to handle multiple outstanding write misses. To solve the problem, after filling a command entry, we performed a seemingly needless write operation to the next (free) command entry. This caused the CPU to initiate a prefetch of the next entry in background. The overhead of writing the commands dropped from 4.5% to less than 0.5%. This type of optimizations is possible because the TCE "owns" the CPU, and its cache usage patterns are nearly deterministic.

5. Experimental Results

We performed measurements for a single sender application and a single receiver application, which pass equal-sized messages of configurable size over 1-8 connections, using an asynchronous API. In order to isolate consumer interface costs from other overhead, we used an application that performed only basic transfer operations and polled for data transfer completions.

We measured the CPU utilization using oprofile ([16]), a statistical profiler based on the Pentium performance counters. Oprofile was configured to generate an interrupt every 300000 clock ticks. We measured bandwidth by having the application use a processor timestamp clock. Measurements were taken only during data transfer phase, after the connection setup.

The baseline standard stack performance on our machines, over a state-of-the-art 1 Gbit/s Ethernet adapter, was similar to that reported in [15]. For example, when receiving 4 KB messages at 1 Gbit/s rate, CPU utilization of both CPUs was 70% (140% total).

Figure 3 demonstrates our prototype performance on the receive side for different message sizes. The analysis is somewhat complicated, due to throughput decline for smaller messages which is caused by the emulated Streamer, as described in *Section 5.1*. As expected, the application CPU utilization is proportional to the number of messages (not shown on the graph), i.e., it decreases when message size is increased. The TCE CPU utilization is proportional to the number of received data packets (MTU-size segments).

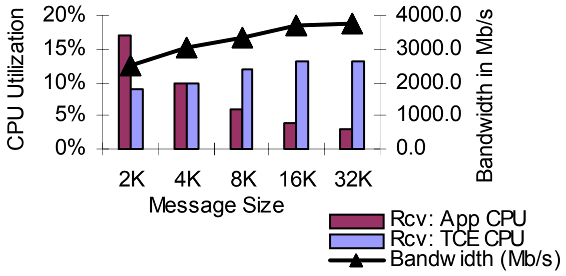


Figure 3. Receive Performance

The CPU utilization of both CPUs is strikingly low. Total utilization of both CPUs is an order of magnitude lower than that of an application using a standard stack. The utilization of the application CPU is consistent with the expectations for a direct-access adapter.

Figure 4 provides details on the time spent by the TCE for different activities, when the achieved bandwidth is approximately 3.7 Gb/s. The TCE "yields" the CPU 88% of time; 6.1% is spent on reading the events from memory or polling for new events. 5.9% is spent on actual event processing, which also includes posting of the ack generation commands to the Streamer. This is a very low number for such bandwidth, but it is consistent with the expectations for pure TCP control implementation, which is not burdened with the data movement or the application interface.

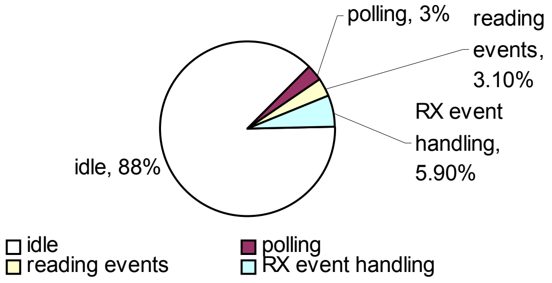


Figure 4. Rx TCE Processing

Figure 5 shows the CPU utilization on the transmit side. The behavior of the application CPU is almost the same as on the receive side.

The transmit TCE CPU utilization is slightly different from the receive case, due to a different pattern of Streamer events, because the number of transmit events generated by the Streamer decreases for larger messages, since more segment transmission indications are coalesced.

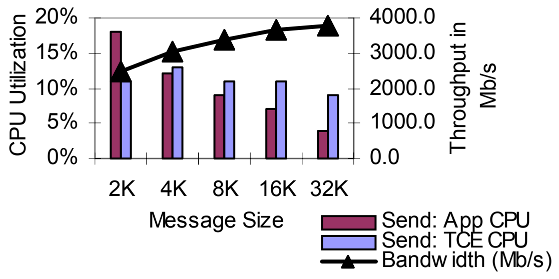


Figure 5. Transmit Performance

Figure 6 provides more details on the TCE utilization for transmit traffic.

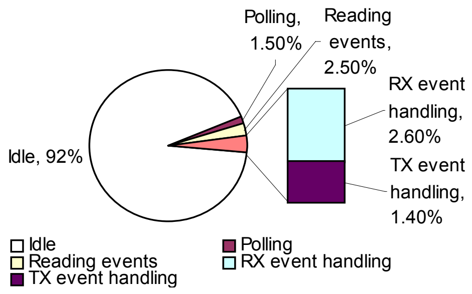


Figure 6. Tx TCE Processing

5.1 Prototype Limitations

Due to limitations of the available hardware, we could not achieve throughput close to the wire speed. We emulated the Streamer by extending the adapter microcode, which was a bottleneck to begin with. For 32 KB messages, the original microcode provided a throughput of 4.0 Gbit/s. The microcode extensions, reduced the throughput further to approximately 3.7 Gbit/s.

The throughput provided by the adapter decreased as the message size decreased, because of per-message overhead in the microcode. The throughput decline, which shows up in our measurement results, closely correlates with the throughput decline of the original adapter.

Despite these limitations, we were able to achieve multi-gigabit throughput, that allowed us to estimate the software scalability in our architecture.

6. Conclusions and Future Work

We built a highly efficient loosely coupled architecture for TCP acceleration, which requires minimal hardware support. It eliminates most of the OS integration costs due to a direct access hardware interface, hardware-managed data buffers and interrupt and lock-free processing. The memory bus is relieved from data copy operations, and cache performance is improved by decoupling the TCP execution environment from the application execution environment. The architecture preserves flexibility in protocol implementation, and provides the ability to sustain protocol modifications. Our prototype implementation achieves 3.7 Gb/s throughput using less than 20% of CPU time.

The ability to use a virtual CPU needs to be examined. On single CPU systems, or on systems with highly variant network bandwidth requirements, it may be impossible or undesirable to dedicate an entire CPU to TCE processing. In such systems, the TCE can be executed on a virtual CPU implemented through one of the CPU virtualization techniques. The prototype can be extended to operate in such an environment, to check the impact of virtualization overhead and of longer latencies of event processing.

7. References

[1] J. Chase et.al. End system optimizations for high-speed TCP. *IEEE Communications,* April 2001.

[2] D. Clark et.al. An analysis of TCP processing overhead. *IEEE Communications*, June 1989.

[3] J. Kay et.al. The importance of non-data touching processing overheads in TCP/IP. *ACM SIGCOMM,* September 1993.

[4] J. Mogul. TCP offload is a dumb idea whose time has come. *HotOS*. May 2003.

[5] M. Rangarajan et.al. TCP Servers: Offloading TCP Processing in Internet Servers–Design, Implementation and Performance. Tech. Rep., Rutgers University, 2002.

[6] P. Sarkar et.al. Storage over IP: When does hardware support help? *2nd USENIX Symposium on File and Storage Technologies (FAST)*, March 2003.

[7] P. Shivam et.al. Promises and reality: On the elusive benefits of protocol offload. *NICELI at ACM SigComm,* August 2003.

[8] D. Freimuth et.al. Server Network Scalability and TCP Offload. *USENIX Annual Technical Conference*, April 2005.

[9] G. Regnier et.al. ETA: Experience with an Intel Xeon Processor as a Packet Processing Engine. *HOT Interconnects*, 2003.

[10] D. McAuley et.al. A case for Virtual Channel Processors. *NICELI at ACM SigComm*, August 2003.

[11] A. Foong et.al. TCP Performance Re-Visited. *IEEE IPASS 03*, 2003.

[12] E. Markatos. Speeding-up TCP/IP: faster processors are not enough. *21st IEEE Int'l Performance, Computing, and Communications Conference (IPCCC 2002)*, April 2002.

[13] P. Buonadonna et.al. Queue-pair IP: A hybrid architecture for system area networks. *29th Ann. Int'l Symp. on Computer Architecture*, May 2002.

[14] The Infiniband Trade Association. The Infiniband Architecture. *http://www.infinibandta.org/specs*.

[15] G. Regnier et.al. TCP onloading for data center servers. *IEEE Computer,* November 2004.

[16] OProfile Manual. *http://oprofile.sourceforge.net/doc.*