# Network I/O Acceleration in Heterogeneous Multicore Processors

Benjamin Wun and Patrick Crowley
*Applied Research Laboratory*
*Department of Computer Science and Engineering*
*Washington University in St. Louis*
*{bw6,pcrowley}@cse.wustl.edu*

## Abstract

*Chip multiprocessor (CMP) architectures are fast becoming the dominant design for general purpose processors. Whereas current generation server and desktop processors use homogenous CMP architectures, network processors (NPs) have used heterogeneous CMP architectures for years. At the same time, the failure of network stacks in traditional processors to scale with increased network bandwidths has spawned numerous proposals for new approaches to accelerate network processing. This paper looks at moving network stack processing from the main CPU to a series of smaller, closely coupled, and more efficient processors in a heterogeneous CMP by implementing such an architecture on an Intel IXP network processor. Our experiments show that the close coupling and flexible nature of the IXP's microengines allow them to greatly accelerate network processing for a small cost in area.*

## 1. Introduction

The trend in general-purpose processor (GPP) design is moving away from single, superscalar cores of increasing sophistication towards chip multiprocessor (CMP) designs. Among the high-level architectural options to consider is whether a CMP should be comprised of a few sophisticated cores, a large number of simpler cores, or a combination of both.

Concurrently, the mounting evidence that traditional software and hardware designs are inadequate to keep up with ever faster networking technologies [11][12] has sparked inquiry into new interfaces and architectures to exploit this newly available bandwidth.

Modern server NICs perform a variety of optimizations to speed network processing [13]. These include DMA, checksum offloading, large segment offload, interrupt coalescing and adaptive polling. These techniques eliminate or minimize the frequency of certain tasks, such as interrupt processing, or move other fairly expensive tasks, like memory copying, from the CPU to the NIC. While quite effective, these NICs are expensive, and can be bottlenecked by a slow I/O bus interface [1].

TCP Onloading [2] exploits the trend toward more cores by dedicating a processor in a SMP or CMP system to packet processing, which is more efficient than having networking code share each processor with other programs. Most proposals for TCP Onloading envision it in the setting of a homogenous CMP, but this approach may be inefficient. A group of smaller, simpler, more power and area efficient cores designed specifically for networking tasks, such as are found in network processors (NPs), could do the job at least as well and at a lower cost in terms of area, power, and complexity. This can be thought of as heterogeneous network onloading.

In this paper, we examine the proposition that the addition of small, simple cores to a general purpose CPU can accelerate standard sockets network I/O, either by employing the techniques of server NICs or through network onloading. To evaluate this proposal, we have built a prototype system using the Intel IXP network processor.

The remainder of this paper is organized as follows. Section 2 of this paper will give some background on the IXP. Section 3 will look at recent attempts to accelerate network stack processing. Section 4 lays out the architecture of our prototype, and section 5 analyzes its performance. Section 6 presents our ideas for future improvements. We conclude in section 7.

## 2. Packet Processing and the Intel IXP

IXP NPs feature two types of processors. The first is an ARM based XScale which boots a traditional OS and is typically used in management and slow path processing. The second processor type, the microengine (ME), is a small embedded core for line-rate packet processing. IXP NPs have a single XScale, and 4, 8, or 16 MEs, depending on the specific chip. In this work, we use the IXP2350, which is illustrated in Figure 1.

In our prototype, the XScale is the host CPU, and the MEs provide I/O acceleration. This is an atypical use of the XScale. In a usual application, the XScale would only receive exception packets, a relatively small fraction of traffic. In our system, an application on the XScale is the source and destination for every packet. We use the IXP for its convenient approximation of our architectural model. The XScale represents a high-performance GPP for which the MEs accelerate network I/O.

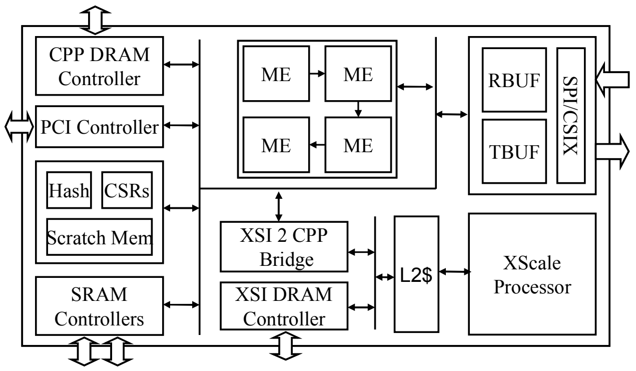

**Figure 1: Organization of the IXP 2350 NP.**

Each IXP ME provides hardware support for 8 hardware thread contexts, including register storage, multithreading ISA extensions, and a thread arbiter. Each ME has its own local data and instruction storage, both implemented as SRAMs. An ME communicates asynchronously with other units via I/O commands and transfer registers. A DRAM read, for example, is carried out by sending a read operation to the DRAM controller (via the Command Outlet FIFO) that specifies the desired address as well as the target incoming transfer registers to which the data should be delivered. Hardware signals are specified in the ISA and are asserted when requested operations have completed. This message-passing style and the use of hardware signals allow ME software to initiate multiple external requests without blocking, as long as subsequent computation does not depend on the completion of these requests. This interface provides both a more efficient way to access memory and a way to hide memory latencies.

Other units provide critical functions or resources in hardware, including a configurable hash unit, 16KB of on-chip scratch memory and 128KB of message SRAM. The IXP 2350 include a DDR SDRAM and a QDR SRAM controller on-chip as channels for bulk and latency-sensitive data storage, respectively. A separate channel of SDRAM is used by the OS and programs on the XScale. Both the MEs and XScale are clocked at 900 MHz.

All IXP processors contain a Media Switch Fabric (MSF) to facilitate high speed communication between the MEs and MACs. The IXP2350 uses the MSF to interface to its two, 1Gbps on-chip MACs. Having the MAC located on-chip over a high speed interface is a great advantage for scalable, high speed networking [1].

## 3. Related Work

TCP offload engines (TOEs) are being used to accelerate specific tasks, such as storage area networking or for use with protocols such as RDMA [4][5]. Though commercial implementations exist, it is inconclusive whether TOEs are actually an effective solution, with some studies showing the TOE itself to be the actual bottleneck [6][7]. Our approach differs from that taken with TOEs, as IXP MEs are on-chip, fully-programmable, and closely coupled with the CPU, thus bypassing the major problems with TOEs and providing additional opportunities for optimization. Furthermore, our interest is in accelerating general purpose networking, whereas most TOEs are used to accelerate a specific task.

Binkert et. al., in a simulation based study, have examined the efficacy of moving the NIC's location relative to the CPU [1]. They found that putting the NIC on a direct HyperTransport like channel and eliminating the I/O bus bottleneck greatly increased system throughput. Locating the NIC on chip produced further improvements for the receive path and also allowed packet data to be directly written into cache, a potential accelerator for certain network workloads. While the IXP has no mechanism for direct cache access by the MEs, the MEs, MSF, and MACs are located on-chip with a dedicated off-chip connection to the PHY.

The ETA [3] project demonstrates a new interface for communication between a host processor and an associated packet processing engine (PPE). Their interface allows for asynchronous operation, whereby a user program can send off a packet for transmission or request notification of arriving packets without blocking. The traditional sockets interface semantics require a program to block until the packet is sent or until a packet arrives. For the prototype ETA system, the PPE was a Xeon processor in an SMP system. This prototype showed both improved network throughput and an increase in surplus cycles for the host processor. This differs from our research because ETA uses a state of the art superscalar out of order processor with a high clock rate, deep pipeline and prodigious amounts of cache as the packet processing engine, whereas our project uses the smaller, in order, single issue MEs of the IXP. The smaller, simpler MEs are more power and area efficient for this task than the Xeon.

The authors of ETA have advocated TCP Onloading by combining ETA with a memory aware reference stack (MARS) [2]. MARS is an attempt to mitigate memory access latencies by using asynchronous memory copies, light-weight threading, and direct cache access. The first two are already present in the IXP.

## 4. Network Acceleration

For this project, we have explored two ways of using the MEs to accelerate network processing in the Linux kernel running on the XScale. The first, the "**softnic**" approach, is to have the MEs emulate a high end server NIC [13], while leaving the networking stack on the XScale unmodified. Our second system is an **onload engine** that does moves the networking stack to the MEs, and only

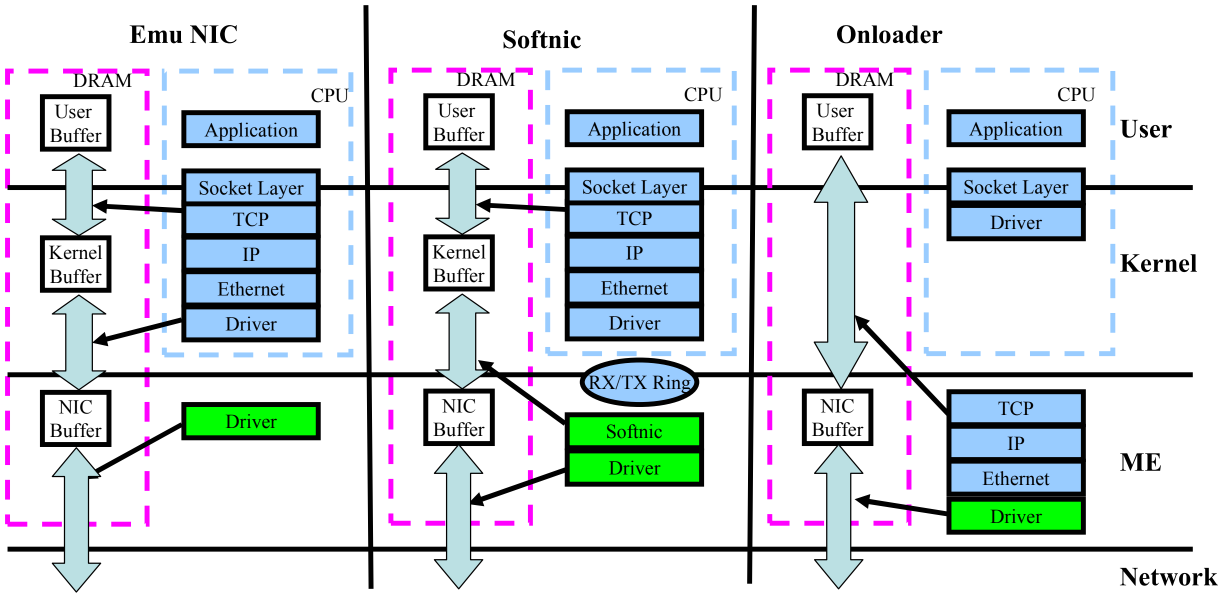

**Figure 2: Architecture Comparison.**

performs high level interface functions on the XScale. Both systems either enhance or replace the kernel's networking stack and support the sockets interface for communicating with user programs.

## 4.1 Software NIC

Figure 2 contains an illustration of the **softnic's** architecture. To transmit a packet from the **softnic**, a user program calls the *sendmesg* system call. The upper, unmodified layers of the network stack in the Linux kernel will copy the data to be sent into a kernel space buffer, determine the interface the packet should be sent out on, and add headers. At this point, the fully formed packet is passed to the driver layer code, which is where our **softnic** modifications take over. The driver code places the buffer on a ring to the MEs for transmission. The kernel on the XScale is now finished with this packet and can go on to process the next one. An ME is constantly polling this ring (a hardware controlled scratchpad ring) for work. When it dequeues a packet buffer, it copies the contents into an internal buffer and raises an interrupt, letting the Xscale know it can now free that buffer. The internal buffer is then passed to another ME in a pipelined fashion for transmission.

When a packet arrives at the MEs in the **softnic**, its checksum is verified, and it is copied into a kernel packet buffer, a pool of which has been preallocated for the MEs' use. The filled buffer is put on a ring for delivery to the XScale, and an interrupt is raised. The interrupt handler on the XScale will turn off interrupts, pull packets off the receive ring, and enqueue them for processing by higher levels in the kernel. Interrupts are re-enabled when the receive ring has been emptied. This is the adaptive polling technique. Control devolves to the unmodified Linux stack and a soft interrupt is raised, invoking the protocol processing code.

## 4.2 Onloader

Figure 2 also illustrates the organization of the **onloader**. When *sendmsg* is called in the **onloader**, the kernel prepares the user buffer for DMA and signals the MEs that a buffer is ready for processing. The MEs copy data directly from the user buffer into an internal buffer. The MEs then add headers and transmit the packet.

When a packet arrives at the **onloader**, an ME verifies the checksum, examines the headers, looks up the control block for that connection, and enqueues the packet for the proper connection. An interrupt is raised only if there is an idle process waiting for that packet to arrive. The only work the Xscale needs to do is to notify the waiting process that a packet is now available.

Our **onload** engine currently only supports UDP over IP. We believe that our results will also apply to TCP, as most of the OS infrastructure, such as interrupts, DMA, sockets interface etc. are common between them. The only major difference is the protocol processing step, which is a demonstrably small component of packet processing [10].

## 4.3 Emulated NIC

To determine how well the **softnic** and onload engine accelerate networking, we compare them to a base case wherein the MEs perform the minimum possible work to get packets to and from the MSF and most tasks are left to the XScale (called the **emu nic** in the graphics). The left side of Figure 2 illustrates the organization of the

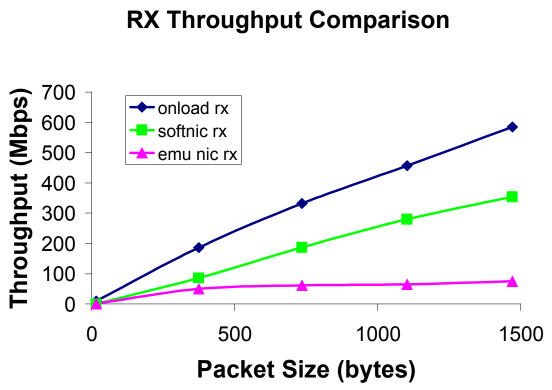

**Figure 3: Receive Throughput Comparison.**

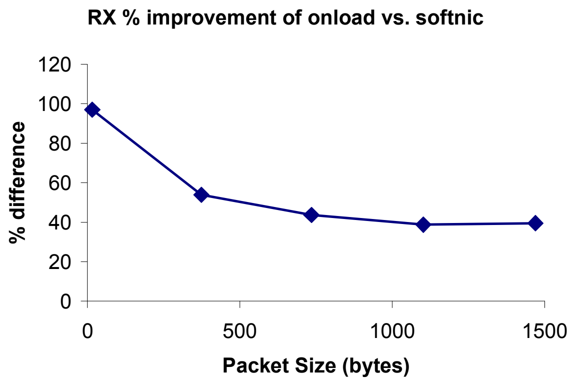

**Figure 4: Receive Percentage Difference.**

**emulated nic**. The **emu nic** corresponds to a low end NIC in a desktop system. The main difference between the softnic and this base case is that the driver code on the XScale must compute checksums and do all data copies between kernel buffers and device buffers. Additionally, interrupt handling is more expensive, because interrupts are raised for every packet on reception instead of adaptively polling after the first one.

## 5. Results

### 5.1 Experiment Setup

Our hardware setup consists of an IXP2350 system, connected through a gigabit Ethernet switch to a PC running Linux 2.4.19. The PC sends packets to the IXP to test the IXP's receive throughput, and receives packets from the IXP to determine the IXP's send throughput. We have determined by sending packets between two PCs that the PC does not represent a bottleneck.

As will be seen, applications executing on the 900 MHz XScale processor cannot receive or transmit packets at rates greater than 500 Mbps. While the packet processing code on the MEs can sustain approximately 2 Gbps, this rate cannot be delivered to the XScale and the applications it hosts. The main challenge facing the end-host system is data copying. Data not only has to be copied from the network into internal buffers and out again, but also into and out of user buffers within the system. As we will see, the cost of this is due not only to moving bytes, but also to pinning and aligning with virtual pages. Normally, router applications implement their fast path on the MEs alone and use the XScale for exceptions, but since we must interface with user programs on the XScale, we incur the overheads of sharing it with other OS functions, such as timer interrupts, or task scheduling.

While the XScale on the IXP2350 cannot perform end-host network processing tasks at gigabit rates, we note that our goal is not absolute performance, but to validate our idea that small, simple, efficient cores attached to a general purpose processor can accelerate network processing. Hence, our use of the **emu nic** as a base case for performance.

We ran our experiments using the Iperf benchmark [14] in UDP mode. In server mode, Iperf waits for a client to connect, and counts the number of bytes received until a termination packet is received. A timestamp is taken after reception of the first packet and reception of the termination packet for determining the achieved throughput. The client program sends fixed sized packets for a given amount of time, followed by a termination packet, and keeps track of the number of bytes sent and the elapsed time. This is a test of throughput in a bulk data movement application.

### 5.2 Receive Path

Figure 3 shows the achievable receive throughput for the 3 cases. We can see that the both the **softnic** and **onloader** are clearly superior to the base case. For large packets, a nearly 10-fold improvement is seen. This is mainly due to the MEs' superior ability to move memory from buffer to buffer. The base case NIC suffers from the XScale's more limited bandwidth when copying between two buffers. Furthermore, while the number of data copies is the same between the base NIC and the **softnic** (from the MSF to an internal buffer, to a kernel buffer, to a user buffer), the **softnic** handles the copy from the internal to kernel buffer asynchronously on the MEs. The **onloader** avoids the copy to a kernel buffer altogether and copies data directly from its internal buffers into the user program's buffers.

Figure 4 shows the percentage improvement of the **onloader** over the **softnic** for different packet sizes. Between the **softnic** and the **onloader**, the **onloader** has superior receive performance, with throughput increase between 100% and 40%. This difference is especially true for smaller packets, where per packet overheads, such as header processing and control buffer lookups, dominate execution time. The main reason for the improvement is

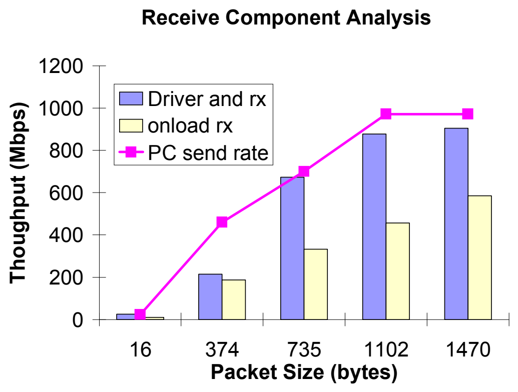

**Figure 5: Receive Components.**

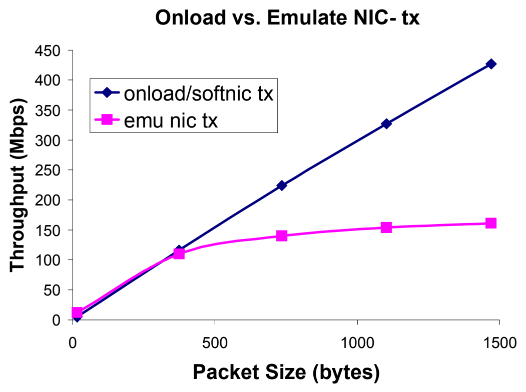

**Figure 6: Transmit Throughput.**

that the onloader can asynchronously receive and enqueue packets while the XScale can be dedicated to other tasks, such as running the userspace benchmarking program. For the softnic, the XScale must split its time between packet processing and other tasks. With larger packets, the per byte costs of checksumming and data copying are dominant, and as this is done on the MEs in both the softnic and onloader, the difference between them becomes quite small, about 20%.

Figure 5 shows the throughput of various **onloader** receive components. The top line shows the sending rate of the PC. The first set of bars is the receive throughput of the driver and receive blocks of the **onloader,** with no user program consuming the received packets. These blocks receive packets from the network, move them into an internal buffer, verify checksums, parse the headers, lookup control blocks and enqueue the packets for future reference. They achieve a receive throughput very close the sender's sending rate. The second set of bars shows throughput of the entire **onloader** system, including the driver and receive block, as well as an XScale component that calls the *recvmsg* system call and moves the packet payloads into a user buffer. As the driver and receive blocks receive packets about as fast as they are being sent, we must conclude that the movement of data into user space using sockets is our receive bottleneck.

### 5.3 Transmit Path

On the transmit side, the **onloader** and **softnic** are again superior to the base case, as demonstrated in Figure 6. The main reason is because the onloader and softnic take advantage of the MEs' superior ability to move memory, whereas the base case is hampered by the XScale's limited memory throughput. There is no clear advantage for either the softnic or onloader on the transmit side, as the main bottleneck here is memory copying and checksumming, which are offloaded to the MEs in both cases. As with the receive path, the softnic on transmit has the advantage of asynchronously performing a DMA from kernel buffers to ME buffers on the MEs while the onloader copies directly from user buffers to internal ME buffers.

## 6. Design Improvements

Our experience designing the **onloader** has demonstrated some aspects of the IXP design that should be modified in a system attempting onloading. These observations apply to any on-chip I/O acceleration technique. One such weakness is the lack of an MMU. In order for the MEs to copy data directly to and from user space buffers into internal buffers, achieving the equivalent of zero copy semantics, the kernel on the XScale must first ensure that all pages of the user buffer exist in memory, walk the system's page tables to find their locations, and clean or invalidate the cache in order to keep consistency. This is very inefficient; in early implementations, we found this cost to account for a third of the per packet overhead. In order to get around this problem, we cached previous mappings of user space buffers for each connection, so that if a program reuses the same buffer (as the Iperf benchmark does), this costly overhead does not have to be incurred again. This resulted in a 1.5X increase in throughput for the receive path, which is reflected in our numbers for the previous section. Without this optimization, the **onloader** is actually slower than the **softnic**, despite eliminating one buffer to buffer copy. Giving the MEs a TLB would effectively do the same job in hardware, without having to incur page walking overheads on the XScale.

Another optimization would be to have the MEs participate fully in the XScale's cache coherence protocol. The ability of the MEs to push data into the L2 on writes to DRAM was a big improvement for both the **onloader** and **softnic**. If, instead of using the push feature (which is optional), the XScale were to invalidate those addresses in its cache and reload the data from memory, our experiments show that the **softnic** would suffer a 35%

degradation in performance. As of the current model IXP2350s, this is the only feature available for cache coherence. A full coherence protocol, which would include letting the MEs snoop the XScale cache on a memory read, would prove advantageous for the transmit path.

One problem with supporting the sockets interface is that the system has no control over where the application allocates its user buffers. Thus when moving data into these buffers, the DMA mechanism must contend with both crossing page boundaries and DRAM word alignment. For DRAM alignment, a read-modify-write may be necessary to avoid overwriting other data that shares a given DRAM word with the user buffer.

Some elements of the IXP design proved extremely useful in this setting. The hardware supported queues for the IXP memory controller made buffer management easy and fast, whereas it can be a significant source of overhead in traditional systems [10]. With memory copying being a major bottleneck for large packets, the asynchronous memory access models and hardware threading were a great help in hiding those latencies and increasing achievable memory bandwidth. Our benchmarks showed that under the right conditions, the MEs can move data from one DRAM bank to another at up to 10x faster than the XScale.

## 7. Conclusions

In this paper we have explored several ways of exploiting small, simple, low power processor cores to accelerate network processing in end host systems. Our prototypes, built on an IXP 2350 NP, show that the MEs are capable of greatly accelerating network processing by using a combination of methods used in modern server NICs and by onloading part of the stack to the MEs. While there is a slight advantage for receive side onloading over the softnic, the important point is that whatever method is chosen, the MEs can be used to accelerate network processing at the processor for a slight increase in complexity without adding the cost of an expensive server NIC.

We have also uncovered several architectural improvements that would make the MEs better able to accelerate network processing. A full cache coherence protocol and onboard MMUs would speed up buffer copies, which are the current bottleneck for medium to large size packets.

Finally, we have developed a Linux kernel module and IXP ME code that we will to make available for others to experiment with. Thus, our implementation is both a prototype and a useful platform. With our scheme, network I/O to Linux applications on the XScale processor have been increased nearly ten fold for receive and by a factor of three for transmit, compared to the standard techniques. We expect that this infrastructure will be useful to IXP2350 developers, both academic researchers and industrial productions alike.

## 8. Acknowledgments

This work was supported by a gift from Intel Corporation and by NSF grant CCF-0430012.

## 9. References

[1] N. Binkert, L. Hsu, A. Saidi, R. Dreslinski, A. Schultz, S. Reinhardt. "Performance Analysis of System Overheads in TCP/IP Workloads." In Proc. 14th Int'l Conf. on Parallel Architectures and Compilations Techniques (PACT), Sept. 2005.

[2] Greg Regnier, Srihari Makineni, Ramesh Illikkal, Ravi Iyer, Dave Minturn, Ram Huggahalli, Don Newell, Linda Cline, and Annie Foong. *TCP onloading for data center servers*. IEEE Computer, 37(11):48--58, November 2004.

[3] G. Regnier, D. Minturn, G. McAlping, V. Saletore, A. Foong. "ETA: Experience with an Intel Xeon Processor as a Packet Processing Engine." *IEEE Micro*, Jan/Feb 2004, pp. 23-31.

[4] J. Mogul. "TCP Offload is a Dumb idea Whose Time has Come." In Proc. Of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX), May 2003.

[5] P. Shivam, J. Chase, "On the Elusive Benefits of Protocol Offload," In Proc. Of the ACM Special Interest Group on Data Communication (SIGCOMM), Aug. 2003.

[6] Boon S. Ang, "An Evaluation of an Attempt at Offloading TCP/IP Protocol Processing on to an i960RN-based iNIC," Hewlett-Packard Technical Report HPL-2001-8, 2001.

[7] P. Sarkar, S. Uttamchandani, K. Voruganti, "Storage Over IP: When Does Hardware Support Help?" in Proc. Of 2nd USENIX Conference on File and Storage Technologies (FAST), April 2003.

[8] A. Foong, T. Huff, J. Patwardhan, G. Regnier, "TCP Performance Re-Visited," 2003 International Symposium on Performance Analysis of Systems and Software, Austin, TX, March 2003.

[9] Infiniband Trade Associations, http://www.infinibandta.org.

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

[11] Srihari Makineni and Ravi Iyer. Measurement-based analysis of TCP/IP processing requirements. In *10th International Conference on High Performance Computing (HiPC 2003)*, Hyderabad, India, Dec. 2003.

[12] Evangelos P. Markatos. Speeding up TCP/IP: Faster processors are not enough. In *Proceedings 21st IEEE International Performance, Computing, and Communication Conference (IPCCC),* pgs. 341-345, Phoenix, AZ, April 2002.

[13] J. S. Chase, A. J. Gallatin, and K. G. Yocum, "*End-System Optimizations for High-Speed TCP*," IEEE Communications, vol. 39, no. 4, pp. 68--74, 2001.

[14] NLANR/DAST, Iperf: The TCP/UDP Bandwidth Measurement Tool, http://dast.nlanr.net/Projects/Iperf/, 2003.