An Evaluation of an Attempt at Offloading TCP/IP Protocol Processing onto an i960RN-based iNIC

Boon S. Ang
Computer Systems and Technology Laboratory
HP Laboratories Palo Alto
HPL-2001-8
January 9th, 2001*

TCP/IP networking, Intelligent Network Interface

This report presents an evaluation of a TCP/IP offload implementation that utilizes a 100BaseT Intelligent Network Interface Card (iNIC) equipped with a 100 MHz i960RN processor. The entire FreeBSD-derived networking stack from socket downward is implemented on the iNIC with the goal of reducing host processor workload. For large messages that result in MTU packets, the offload implementation can sustain wire-speed on receive but only about 80% of wire-speed on transmit. Utilizing hardware-based profiling of TTCP benchmark runs, our evaluation pieced together a comprehensive picture of transmit behavior on the iNIC. Our first surprise was the number of i960RN processor cycles consumed in transmitting large messages--around 17 thousand processor cycles per 1.5kbyte (Ethernet MTU) packet. Further investigation reveals that this high cost is due to a combination of i960RN architectural shortcomings, poor buffering strategy in the TCP/IP code running on the iNIC, and limitations imposed by the I20-based host-iNIC interface. We also found room for improvements in the implementation of the socket buffer data-structure. This report presents profiling statistics, as well as code-path analysis that back up these conclusions. Our results call into question the hypothesis that a specialized networking software environment coupled with cheap embedded processors is a cost effective way of improving system performance. At least in the case of the offload implementation on the i960RN-based iNIC, neither was the performance adequate nor the system cheap. This conclusion, however, does not imply that offload is a bad idea. In fact, measurements we made with Alacritech's SLIC NIC, which partially offloads TCP/IP protocol processing to an ASIC, suggests that offloading can confer advantages in a cost effective way. Taking the right implementation approach is critical

* Internal Accession Date Only
Approved for External Publication

1 Introduction

This report presents an evaluation of a TCP/IP implementation that performs network protocol stack processing on a 100BaseT intelligent network interface card (iNIC) equiped with an i960RN embedded processor. Offloading TCP/IP protocol processing from the host processors to a specialized environment was proposed as a means to reduce the workload on the host processors. The initial arguments profered were that network protocol processing is consuming an increasingly larger portion of processor cycles and that a specialized software envrionment on an iNIC can perform the same task more efficiently using cheaper processors.

Since the inception of the project, alternate motivations for offloading protocol stack processing to an iNIC have been proposed. One was the iNIC offers a point of network traffic control independent of the host -- a useful capability in the Distributed Service Utility (DSU) architecture [9] for Internet data-centers, where the host system is not necessarily trusted. More generally, the iNIC is viewed as a point where additional specialized functions, such as firewall and web caching, can be added in a way that scales performance with the number of iNIC's in a system.

The primary motivation of this evaluation is to understand the behavior of a specific TCP/IP offload design implemented in the Platform System Software department of HP Laboratories' Computer Systems and Technology Laboratory. Despite initial optimism, this implementation using Cyclone's PCI-981 iNIC, while able to reduce host processor cycles spent on networking, is unable to deliver the same networking performance as Windows NT's native protocol stack for 100BaseT Ethernet. Furthermore, transmit performance lags behind receive performance for reasons that were not well understood.[1]

Another goal of this work is to arrive at a good understanding of the processing requirements, implementation issues and hardware and software architectural needs of TCP/IP processing. This understanding will feed into future iNIC projects targetting very high bandwidth networking in highly distributed data center architectures. At a higher level, information from this project provides concrete data-points for understanding the merits, if any, of offloading TCP/IP processing from the host processors to an iNIC.

1.1 Summary of Results

Utilizing hardware-assisted profiling of TTCP benchmark runs, our evaluation pieced together a comprehensive picture of transmit behavior on the iNIC. All our measurements assume that checksum computation, an expensive operation on generic microprocessors, is done by specialized hardware in Ethernet MAC/Phy devices, as is the case with commodity devices appearing in late 2000.[2]

[1] The team that implemented the TCP/IP offload recently informed us that they found some software problem that was causing the transmit and receive performance disparity and had worked around it. Unfortunately, we were unable to obtain further detail in time for inclusion in this report.

[2] The Cyclone PCI-981 iNIC uses Ethernet devices that do not have checksum support. To factor out the cost of checksum computation, our offload implementation simply does not compute checksum on both transmit and receive during our benchmark runs. To accommodate this, machines receiving packets from

Our first surprise was the number of i960RN processor cycles consumed in transmitting large messages over TCP/IP -- around 17 thousand processor cycles per 1.5kbyte (Ethernet MTU) packet. This cost increases very significantly for smaller messages because aggregation (Nagle Algorithm) is done on the iNIC, thus incurring the overhead of handshake between host processor and iNIC for every message. In an extreme case, with host software sending 1-byte messages that are aggregated into approximately 940byte packets[3], each packet consumes 4.5 million i960RN processor cycles. Even transmitting a pure acknowledgement packet costs over 10 thousand processor cycles.

Further investigation reveals that this high cost is due to a combination of i960RN architectural shortcomings, poor buffering strategy in the TCP/IP code running on the iNIC, and limitations imposed by the I2O-based host-iNIC interface. We also found room for improvements in the implementation of the socket buffer data-structure and some inefficiency due to the gcc960 compiler.

Our study of the host-side behavior is done at a coarser level. Using NT's Performance Monitor tool, we quantified the processor utilization and the number of interrupts during each TTCP benchmark run. We compare these metrics for our offload implementation with those of NT's native TCP/IP networking code and another partially offloaded iNIC implementation from Alacritech. To deal with the fact that different implementations achieve different networking bandwidth, the metrics are accumulated over the course of complete runs transferring the same amount of data.

The measurements show that compared against native NT implementation, our offload implementation achieves significantly lower processor utilization for large messages, but much higher processor utilization for small messages, with crossover point at around 1000byte messages. The interrupt statistics shows similar trend, though with crossover at a smaller message size. Furthermore, the number of interrupts is reduced by a much more significant percentage than the reduction in host processor utilization, suggesting that costs other than interrupt processing contributes quite significantly to the remaining host-side cost in our offload implementation. Based on other researcher's results [1], we believe that host processor copying data between user and system buffer is the major remaining cost.

The Alacritech NIC is an interesting comparison because it represents a very lean and low cost approach. Whereas the Cyclone board is a full-length PCI card that is essentially a complete computer system decked with a processor, memory and supporting logic chips, the Alacritech NIC looks just like another normal NIC card, except that its MAC/Phy ASIC has additional logic to process TCP/IP protocol for "fast path" cases. A

---

our offload implementation run specially doctored TCP/IP stacks that do not verify checksum. Error rates on today's networking hardware in a local switched network are low enough that running TTCP benchmark is not a problem.

[3] The aggregation is not controlled by a fixed size threshold, and thus the actual packet size varies dynamically, subjected to an MTU of 1460 data bytes.

limitation of this approach is it does not allow any modification or additions to the offloaded functions once the hardware is designed.

Our measurement shows that an Alacritech NIC is able to sustain network bandwidth comparable to that of Native NT for large messages, which is close to wire-speed. Its accumulated host processor utilization, while lower than native NT's, is higher than that with our offload implementation. Its performance degrades when messages are smaller than 2k bytes because it has no means of aggregating out-going messages (i.e. no Nagel Algorithm).

This study calls into question the hypothesis that a specialized software environment together with a cheap embedded processor can effectively offload TCP/IP protocol stack processing. The i960RN-based implementation studied in this work is unable to adequately handle traffic even at the 100Mbit/s level, much less at 1 Gigabit/s or 10 Gigabit/s levels that are the bandwidths of interest in the near future. While bad buffering strategy is partly responsible for the less than satisfactory performance on our offload implementation, its BSD-derived TCP/IP protocol stack is in fact better than NT's when executed on the same host platform. Clearly, the "better" stack and the advantage from the specialized interrupt-handling environment are insufficient to make up for the loss of a good processor and the additional overhead of interactions between the host and the iNIC. Ultimately, this approach is at best moving work from one place to another without conferring any advantage in efficiency, performance or price, and at worse, a performance limiter.

This conclusion does not imply that offload is a bad idea. In fact, the performance of the Alacritech NIC suggests that it can confer advantages. What is critical is taking the right implementation approach. We believe there is still unfinished research in this area, that a fixed hardware implementation, such as that from Alacritech, is not the solution. From a functional perspective, having a flexible and extensible iNIC implementation not only enables tracking changes to protocol standards, but also allows additional functions to be added over time. The key is a cost effective, programmable iNIC micro-architecture that pays attention to the interface between iNIC and host. There are a number of promising alternate micro-architecture components, ranging from specialized queue management hardware, to field programmable hardware, to multi-threaded and/or multi-core, possibly Systolic-like, processors. This is the research topic of our next project.

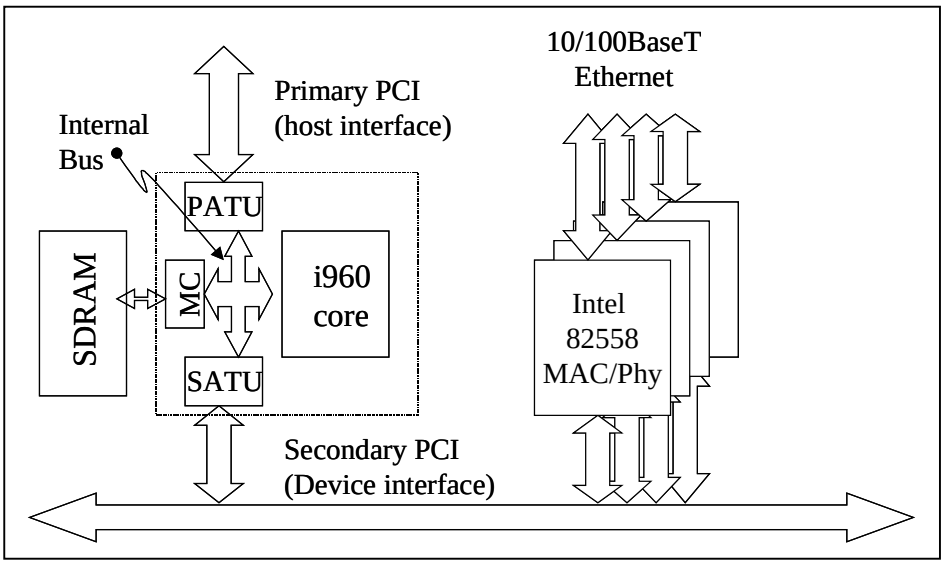
1.2 Organization of this Report

The next section gives an overview of our i960RN-based TCP/IP offload implementation. We briefly cover both the hardware and the software aspects of this implementation to pave the background for the rest of this report. Section 3 examines the behavior on the iNIC. Detailed profiling statistics giving breakdowns for various steps of the processing, and utilization of the hardware resources is presented. This is followed in Section 4 with an examination of the host side statistics. Section 5 presents some related work covering both previous studies of TCP/IP implementations and other offload implementations. Finally, we conclude in Section 6 with what we learned from this study and areas for future work.

2 TCP/IP Offload Implementation Overview

Our TCP/IP offload implementation uses the Cyclone PCI-981 iNIC described in Section 2.1. The embedded processor on this iNIC runs a lean, HP custom designed run-time/operating system called RTX described in Section 2.2.1. The networking protocol stack code is derived from FreeBSD's Reno version of the TCP/IP protocol code. Interaction between the host and the iNIC occurs through hardware implemented I2O messaging infrastructure. Section 2.2.2 presents saliant features of the networking code.

2.1 Cyclone PCI-981 iNIC hardware



The Cyclone PCI-981 iNIC, illustrated in the above figure, supports four 10/100BaseT Ethernet ports on a private 32/64-bit, 33MHz PCI bus which will be referred to as the secondary PCI bus. Although the bus is capable of 64-bit performance, each Ethernet devices only supports a 32-bit PCI interface, so that this bus is effectively 32-bit, 66-MHz. The iNIC presents a 32/64-bit, 33MHz PCI external interface which plugs into the host PCI bus, referred to as the primary PCI bus. In our experiments, the host is only equipped with a 32-bit PCI bus, thus constraining the iNIC's interface to operate at 32-bit, 33MHz. Located between the two PCI buses is an i960RN highly integrated embedded processor, marked by the dashed box in the above figure. It contains a i960 processor core running at 100MHz, a primary address translation unit (PATU) interfacing with the primary PCI bus, a secondary address translation unit (SATU) interfacing with the secondary PCI bus, and a memory controller (MC) used to control 66MHz SDRAM DIMMs. (Our evaluation uses an iNIC equiped with 16Mbytes of 66MHz SDRAM.) An internal 64-bit, 66MHz bus connects these four components together.

The PATU is equiped with two DMA engines in addition to bridging the internal bus and the primary PCI bus. It also implements I2O messaging and door bell facilities in hardware. The SATU has one DMA engine and bridges the internal bus and the secondary PCI bus. The i960 processor core implements a simple single-issue processing pipeline with none of the fancy superscalar, branch prediction, and out-of-order capabilities of today's main stream processors. Not shown in the above figure are a PCI-to-PCI bus bridge and an application accelerator in the i960RN chip. These are not used in our offload implementation. Further details of the i960RN chip can be found in the i960 RM/RN I/O Processor Developer's Manual [3].

2.2 Offload Software

Two pieces of software running on the i960 processor are relevent to this study. One is a run-time system, called RTX, that defines the operating or execution environment. This is described in the next section. The other piece of software is the networking protocol code itself, which is described in Section 2.2.2.

2.2.1 Execution Environment

RTX is designed to be a specialized networking environment that avoids some well-known system-imposed networking costs. More specifically, interrupts are not structured into the layered, multiple invocation framework found in most general purpose operating systems. Instead, interrupt handlers are allowed to run-to-completion. The motivation is to avoid the cost of repeatedly storing aside information and subsequently re-invocaking processing code at a lower priority. RTX also only supports a single address space without a clear notion of system vs. user-level address spaces.

RTX is a simple, ad hoc run-time system. Although it provides basic threading and pre-emptive multi-threading support, the lack of enforeable restrictions on interrupt handling makes it impossible to guarantee any fair share of processor cycles to each thread. In fact, with the networking code studied in this report, interrupt handlers disable all forms of interrupts for the full duration of its execution, including timer interrupts[4]. Coupled with the fact that interrupt handlers sometimes run for very very long periods (e.g., we routinely observer the Ethernet device driver running for tens of thousands of processor clocks), this forces processor scheduling to be hand coded into the Ethernet device interrupt handling code in the form of explicit software polls for events. Without these, starvation is a real problem.

Overall the execution of networking code is either invoked by interrupt when no other interrupt handler is running, or by software polling for the presence of masked interrupts. Our measurement shows that for TTCP runs, most invocations are through software polling -- for every hardware interrupt dispatched invocation, we saw several tens of software polling dispatched invocations.

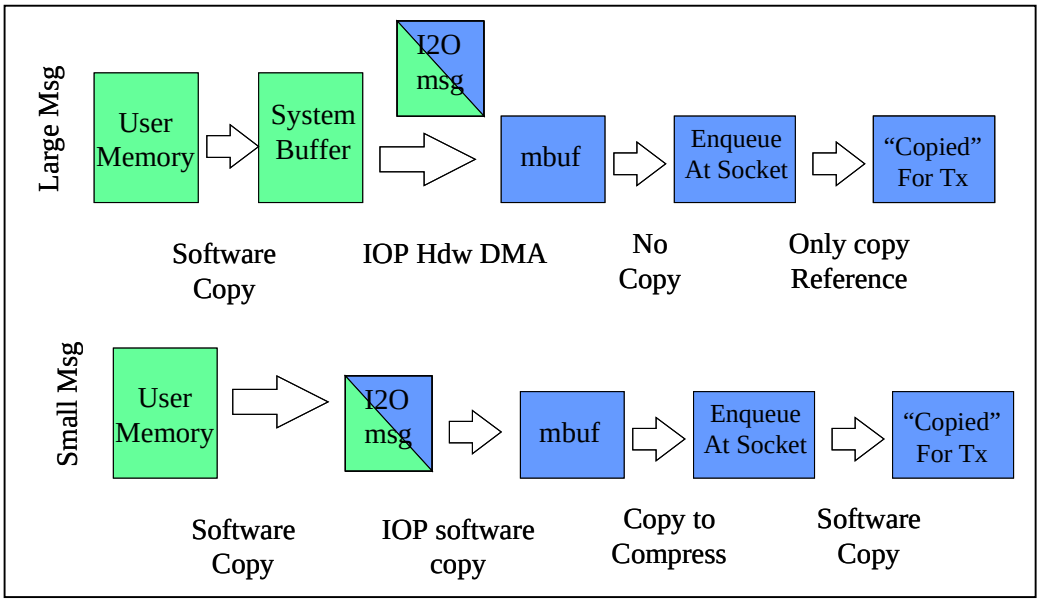
[4] This raises questions about how accurately timers are implemented on the offload design. It is quite possible that software timer "ticks" occur less frequently than intended. We did not look closely into this because it is beyond the scope of this study.

The RTX environment clearly presents challenges for addition of new code. Without a strict discipline for time-sharing the processor, every piece of code is tangled with every other piece of code when it comes to avoiding starvation and ensuring timely handling of events. Clearly, a better scheduling framework is needed, especially to support any kind of service quality provisions.

2.2.2 Networking code

The network protocol code is derived from the Reno version of BSD networking code and uses the fxp device driver. The host interfaces to the iNIC at the socket level, using I2O messaging facility as the underlying means of communication. On the iNIC side, glue code is added to splice into the protocol code at the socket level. The following two sections briefly trace the transmit and receive paths.

2.2.2.1 Transmit path



The above diagram shows the transmit paths for large and small messages. The paths are slightly different for different message sizes. The green portions (lightly shaded in non-color prints) occur on the host side while the blue portions (darkly shared in non-color prints) happen on the iNIC. Unless otherwise stated, the host in this study is an HP Kayak XU 6/3000 with a 300MHz Pentium-II processor and 64Mbyte of memory, running Windows NT 4.0 service pack 5. (The amount of memory though small by today's standards is adqeuate for TTCP runs, especially with the –s option that we used in our runs, which causes artificially generated data to be sourced at the transmit side, and incoming data to be discarded at the receive end.)

When transmitting large messages, the message data is first copied on the host side from user space into pre-pinned system buffers. Next, I2O messages are sent to the iNIC with references to the data residing in host-side main memory. On the iNIC side, servicing of the I2O messages includes setting up DMA requests. Hardware DMA engines in the PATU performs the actual data transfer from host main memory into iNIC SDRAM,

where it is place in mbuf data structures. (We will have a more detailed discussion of mbufs in Section 3.4.)

When DMA completes, iNIC code is once again invoked, to queue the transferred data with the relevent socket data structure and push it down the protocol stack. For large messages, the enqueuing simply involves linking the already allocated mbufs into a linked list. Without interrupting this thread of execution, an attempt is next made to send this message out. The main decision point is at the TCP level, where the tcp_output function will decide if any of the data should be sent at this time. This decision is based on factors such as the amount of data that is ready to go (Nagel Algorithm will wait if there is too little data) and whether there is any transmit window space left (which is deteremind by the amount of buffer space advertised by the receiver and the dynamic actions of TCP's congestion control protocol). If tcp_output decides not to send any data at this point, the thread suspends. Transmission of data will be re-invoked by other events, such as the arrival of more data from the host side, expiration of a time-out timer to stop waiting for more data, or the arrival of acknowledgements that open up transmit window.

Once tcp_output decides to send out data, a "copy" of the data is made. The new "copy" is passed down the protocol stack through the IP layer and then to the Ethernet device driver. This copy is deallocated once the packet is transmitted onto the wire. The original copy is kept as a source copy until an acknowledgement send by the receiver is received. To copy large messages, the mbuf-based BSD buffering strategy merely copies an "anchor" data structure with reference to the actual data.

Another copy may occur at the IP layer if fragmentation occurs. With proper setting of TCP segment size to match that of the underlying network's MTU, no fragmentation occurs. Execution may again be suspended at the IP layer if address lookup results in external query using ARP. The IP layer caches the result of such a query so that in most cases during a bulk transfer over a TCP connection, no suspension occurs here.

Transfer of data from iNIC SDRAM onto the wire is undertaken by DMA engines on the Ethernet devices. When transmission completes, the i960 processor is notified via interrupts.

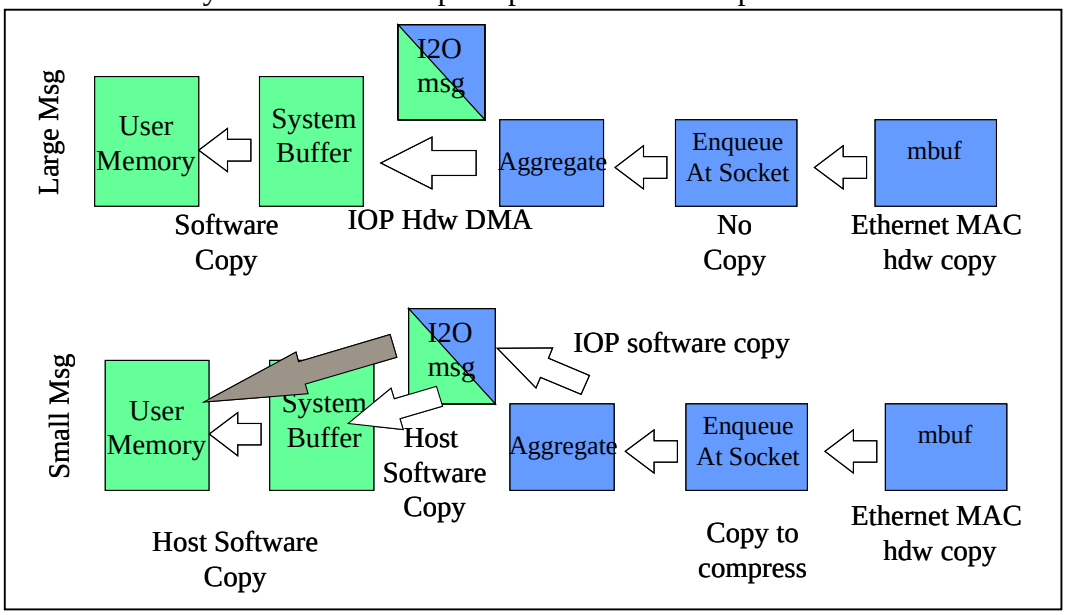
The transmit path for small messages is very similar to that for large messages with a few differences. One difference is data is passed from the host processor to the iNIC directly in the I2O messages. (The specific instruction-level mechanism on IA32 platforms is PIO operations). Thus, on the host side, data is copied from user memory directly into I2O message frames. At the iNIC side, data is now available during servicing of a transmit request I2O message. The iNIC software directly copies the data into mbufs instead of using DMA because for small messages, it is cheaper this way than using DMA. The mbuf data structure behaves differently for small messages (< 208 bytes) than for large messages. When small messages are enqueued into a socket, an attempt is made to conserve buffer usage by "compressing" data into partly used mbufs. This may result in significant software copying. Further down the protocol stack when a copy of

message data is made in the `tcp_output` function for passing further down the protocol stack, actual copy of data occurs for small messages.

Clearly, many copies or pseudo-copies occur in the transmit path. In Section 3.4, we will re-examine mbuf and this issue of copying. Actual measurements of the cost involved will be presented.

2.2.2.2 Receive path

The following diagram illustrates the receive path. Broadly speaking, it is the reverse of the transmit path. (As before, the green or lightly shaded portions execute on the host side, while the blue or darkly shaded portions execute on the iNIC.) Again, we will first consider the path for large packets. The first action is taken by the Ethernet device. Its device driver pre-queues empty mbufs that are filled by the device as data packets arrive. The i960 processor is notified of the presence of new packets via interrupts, which may be invoked either by hardware interrupt dispatch or software poll.



The Ethernet device interrupt handler begins the walk up the network protocol stack. The logical processing of data packets may involve reassembly at the IP layer and dealing with out-of-order segments at the TCP layer. In practice, these are very infrequent, and the BSD TCP/IP code provides a "fast-path" optimization that reduces processing when packets arrive in order. An incoming packet's header is parsed to identify the protocol, and if TCP, the connection. This enables the packet's mbuf's to be queued into the relevent socket's sockbuf data structure. Again for large messages, this simply links mbuf's into a singly linked list.

Next, the data is handed over to the host. As an optimization, our offload implementation attempts to aggregate more packets before interrupting the host to notify it of the arrived data. The i960 processor is responsible for using hardware DMA engine in the PATU to move data from iNIC SDRAM into host memory before notifying the host about the newly arrived data through I2O messages. Data is transferred into host side system

buffers that have been pinned and handed to the iNIC ahead of time. Eventually, when software does a receive on the host side, data is copied on the host side from system buffer into user memory.

For small messages, similar differences as in the case of transmit apply. Thus, queuing into sockbuf may involve copying to compress and conserve mbuf usage. Just as in the case of transmit, small size data is passed between the iNIC and host in I2O messages. On the host side, this enables an optimization if the receiver has previously posted a receive -- data is copied by the host processor from the I2O messages (physically residing on the iNIC) into user memory directly. If no receive has been posted yet, the data is first copied into system buffer.

3 iNIC Side Behavior

This section presents the profiling statistics we collected on the iNIC side. Most measurements rely on a free running cycle counter on the i960 processor to provide timing information. Timing code is added manually to the networking and RTX source code to measure durations of interest. In most cases, this is straightforward because the code operates as interrupt handlers in a non-preemtable mode. Timing a chunk of code simply involves noting the starting and ending times and accumulating that if aggregated data of multiple invocations is being collected. The exception is the collection of statistics reported in Section 3.2 done using the i960RN processor's hardware resource usage performance registers. More will be said about this in that section.

The commonly used TTCP benchmark is used for our studies. This provides a convenient micro-benchmark for getting a detailed look of various aspects of the networking code during bulk transfer.

In the next section, we begin with some general performance statistics that show the breakdown of processing time. Our study emphasises transmit behavior as that has the bigger problem in our implementation. Furthermore, for many server applications that are the targets of the offload technology, transmit performance is more important than receive performance.

The general statistics first leads us to look for hardware bottleneck because the cost numbers look surprisingly large. During close manual inspection of certain code fragments, we also came across cases where a relatively small number of instructions, say 10's of instructions in an inner loop, take hundreds of processor cycles per iteration. Our investigation into hardware bottlenecks is reported in Section 3.2.

Next, we shift our attention to the components responsible for the largest share of processing cost, as indicated by the processor usage breakdown reported in Section 3.1. We examine tcp_output, which has some of the largest share of the processing cycles, to get a better understanding of what happens in that function. The results are reported in Section 3.3. This took us onto the track of the overall buffering strategy, a closer look of which is reported in Section 3.4. One other significant source of overhead is the host-

iNIC interface. We examine that in Section 3.5. Finally, a number of miscellaneous inefficiencies that we came across during our study are reported in Section 3.6.

3.1 General Performance Statistics

Table 1 is a summary of iNIC processor usage breakdown, roughly categorized by network protocol layer. We added a layer for the host-iNIC interface which includes the cost of moving data between the host and iNIC memories. We report the numbers in 100MHz processor clock cycles, label as "pclks". We use the term message (msg) to refer to the logical unit used by host software sending or receiving data, and the term packet (pkt) to refer to the Ethernet frame that actually goes on the wire. For all our experiements, the TCP segment size and IP MTU are set to match the Ethernet frame MTU of 1.5kbyte so that a packet is also the unit handled by the IP and TCP layers.

Table 1 contains the data for three transmit instances with representative message sizes -- 8 kbyte as a large message, 200 byte as a small but common message size, and 1byte to highlight the behavior of the protocol stack. We only report one instance of receive where TTCP is receiving with 8kbyte buffers. In all cases, the machine on the other end of TTCP is a 500MHz Pentium-III FreeBSD box equiped with a normal NIC card. This end is able to handle the work-load easily and is not a bottleneck.

The row labeled "Per msg cost" and the corresponding cost breakdown in Table 1 includes both transmit and receive costs and is the average derived by dividing the total processor usage by the number of messages. While it gives total cost that is most readily correlated to host software transmit and receive actions, it bears few direct correspondance to actions on the iNIC, except in the cases of host-iNIC interface and socket layers for transmit. For receive, even these layers' numbers have marginal correspondance to the per-message cost number because action in all network layers is driven by incoming packets.

Nevertheless, there are several interesting points about the per-message numbers. One is that for 8kbyte messages, transmit costs 66% more than receive. It was not immediately obvious why that should be the case. While transmit and receive processing is obviously different, the per-packet costs do not indicate as big a cost difference. Our investigation found that a different number of acknowledgement packets are involved in the two runs. When the iNIC transmits data, the FreeBSD box that is receiving sends a pure acknowledgement packet for every two data packets it receives. In contrast, when the iNIC is data receiver, it only sends out a pure ack packet after every six data packets. This happens because incoming packets are aggregated before being sent to the host. Only when data is sent to the host is buffering space released and acknowledged to the sender. With transmission of each pure ack packet costing about ten thousand pclks, this is a significant "saving" that improves receive performance.

The per-message cost breakdown in Table 1 highlights the layers that are most costly in terms of processor usage. For the 8kbyte message transmit (large message transmit), TCP layer is responsible for the bulk of the iNIC processor cycles. (See Section 3.3 for

further accounting.) For small message transmit, however, the host-iNIC interface accounts for the bulk of the cost. This is due to Nagel Algorithm aggregation which casuses the per-packet costs to be amortized over a number of messages, around a thousand in the case of 1-byte message. In our offload implementation, the aggregation is done on the iNIC so that host-iNIC interaction cost remains a per- message occurance. The host-iNIC interface cost dominates receive as well, though with a less dominant percentage. (See Section 3.5 for further accounting.)

The host-iNIC interface cost is important because it constraints the frequency of host-iNIC interaction, and the granularity of work that is worth offloading. The per-message host-iNIC layer cost breakdown for transmit corresponds roughly to one host-iNIC interaction. As can be seen in Table 1 (7th row from the top), this is quite high, at least three thousand pclks and goes up to seventeen thousand pclks. Further accounting and breakdown of the interface cost is provided in Section 3.5.

iNIC Role		Tx						Rx	
Msg size (byte)		8k		200		1		8k	
Per msg cost (pclks)		121,020		14,243		5,135		72,812	
Per Tx pkt cost (pclks)		16,933		107,158		4,461,191		10,843	
Per Rx pkt cost (pclks)		9,078		14,878		13,220		12,168	
Per msg cost breakdown by protocol layer:									
Layer	Host-iNIC	17,396	14%	7,001	49%	3,034	59%	26,403	36%
	Socket	5,211	4%	1,023	7%	1,666	32%	5,414	7%
	TCP	53,307	44%	4,742	33%	413	8%	13,244	18%
	IP	17,965	15%	436	3%	8	0%	12,992	18%
	IF	27,141	22%	1,040	7%	14	0%	14,758	20%
Per Tx pkt cost breakdown by protocol layer:									
Layer	Host-iNIC	3,256	19%	58,260	54%	2,646,572	59%	0	0%
	Socket	975	6%	8,509	8%	1,453,387	33%	0	0%
	TCP	6,685	39%	31,224	29%	351,673	8%	4288	40%
	IP	2,298	14%	2,554	2%	3,317	0%	3415	31%
	IF	3,719	22%	6,618	6%	6,242	0%	3139	29%
Per Rx pkt cost breakdown by protocol layer:									
Layer	Host-IOP	0	0%	0	0%	0	0%	5047	41%
	Socket	0	0%	0	0%	0	0%	1035	9%
	TCP	4,165	46%	8,692	59%	6,733	51%	1840	15%
	IP	2,156	24%	2,131	14%	2,495	19%	1932	16%
	IF	2,757	30%	4,041	27%	3,991	30%	2314	19%

Table 1: iNIC-side performance statistics overview

Table 1 also separates the cost into transmit and receive components, reported as per-Tx packet and per-Rx packet costs. In the case where the iNIC is transmitting, the received packets are pure acknowledgement packets. Conversely, the iNIC sends out pure acknowledgement packets when it is the receiver of data in a TTCP run.

The numbers in Table 1 show that the per-packet processing cost is quite high. Even for pure acknowledgements which are expected to have the lowest costs, per-packet cost, for both transmit and receive, starts around ten thousand pclks. For 8-kbyte messages, the cost of receiving a packet rises to twelve thousand pclks while the cost of sending a packet is seventeen thousand pclks. Table 1 also indicates that the per-packet transmit cost for small (< MTU size) messages can become very large. This is due to aggregating multiple small message into a larger packet, resulting in multiple interactions between host and iNIC before a packet is transmitted.

The overall high cost of protocol processing prompted us to look into systematic problems. A cause of concern is that these numbers are much higher than previously reported instruction counts [2,6]. Clearly, since we are dealing with processor cycles instead of instruction counts, an immediate question is the CPI we are getting. Unfortunately, it is not easy to measure CPI on our platform. There is some hardware support for measuring bus utilization, however, and using that, we found a bottleneck in the hardware as reported in the next section.

3.2 Hardware bottleneck

The i960RN processor has hardware performance registers for gathering utilization statistics of both the primary and secondary PCI buses, and the internal bus. In any single run, only a subset of statistics can be collected. Fortunately, our TTCP benchmark runs are highly reproducable. We therefore made multiple runs and combined the statistics into a complete picture. The picture that emerged was that neither PCI buses are used that heavily, never exceeding 12% on either PCI buses in any of our runs. In contrast, the statistics show that the internal bus is clearly a point of contention.

	iNIC's role	Tx		Rx
	Msg size (bytes)	8k	1	8k
	Total bus clocks	165M	290M	105M
Utilization	Idle	38%	48%	35%
	Data usage	10%	9%	12%
	Non-data usage	52%	44%	54%
Ownership	i960 core	55%	45%	52%
	DMA engine 0	0%	0%	4.5%
	DMA engine 1	3%	0%	0%
	PATU	1%	7.2%	2.1%
	SATU	4%	0%	6.9%
Wait/grant	i960 core	3.42	3.07	3.99
	DMA engine 0	-	-	9.57
	DMA engine 1	8.08	-	-
	PATU	5.48	3.31	6.19
	SATU	5.79	7.30	7.14
Own/grant	i960 core	6.11	5.63	5.95
	DMA engine 0	-	-	16.49
	DMA engine 1	10.58	-	-
	PATU	6.98	6.60	9.39
	SATU	19.16	17.54	11.71

Table 2: Internal bus utilization

Table 2 reports various aspects of internal bus utilization for three different TTCP runs. All percentages are relative to total bus clocks. DMA engines 0 and 1 are used for receive and transmit respectively. The overall utilization number shows the internal bus used 50-65% of the time. We further see that of the busy bus cycles, just over 80% are due to non-data usage. These are the cycles when the bus is owned but idle, such as due to wait states in memory accesses. The numbers suggest that the bus has a simple protocol that is incapable of pipelining usage of the bus. The SDRAM and its controller are unlikely to be the cause of this rather inefficient bus utilization because according to the i960RN user's manual [3], the memory controller is capable of handling multiple overlapping accesses.

The ownership data in Table 2 identifies the i960 core as the main bus user. The percentages shown are over all bus cycles and not just the non-idle cycles; as a percentage of the used bus cycles, the i960 core is responsible for 79-88% of overall usage. Another way to view the numbers in the table is that the i960 core is using the bus around half the time over the duration of each TTCP run. Of these 80% are non-data cycles. This coupled with the fact that the i960 processor core is an in-order single issue processor suggests that the i960 processor could be stalled about 40% of the time. In fact, further pipeline delay in conveying data back to the execution pipeline within the i960 core could bring that percentage even higher. We will also see next that the wait time before a grant can further lenghten the processor core stall time.

The wait/grant numbers show the average number of bus cycles that each master waits before getting a grant. Similarly, the own/grant numbers give the average number of bus cycles owned by each master once it gets a grant. One thing to note is the relatively high wait/grant numbers compared to own/grant numbers. For example, in the case of 8kbyte message transmit, the i960 processor core waits on the average 3.4 cycles and only owns the bus for 6.1 cycles (of which only 20% or a 1.3 cycles is used for actual data transfer!). Its wait time is thus more than half its usage time. It is quite likely that the wait time propagates back into the processor core as stall time. This could stretch the 40% stall time identified in the previous paragraph by another half, bringing total processor stall time to 60%.

The other bus masters also suffer similarly large wait to usage time ratios. However, their absolute bus usage is much lower and none of the other statistics we gathered indicate that any of their bus usage is a bottleneck for the overall system.

The pressing question coming out of these numbers is why the processor is using the internal bus so often. We have several speculations all related to a poorly designed i960RN memory hierarchy. It is most likely that a combination of these are responsible. Unfortunately without an accurate and modifiable simulator, we are unable to determine the relative contribution of each factor.

The i960RN has very small caches by modern standards, 8-kbytes of instruction cache and 8-kbytes of data cache. The data cache is write-through only (bad, but slighly ameliorated by write combining and two 16-byte write buffers). Its fill policy on miss is very unconventional by modern standards – on cache miss, the data cache only fetches what is requested instead of an entire cache line. This fill policy means that the data cache provides no temporal locality advantage. Perhaps to make up, the i960 supports many different size load/store instructions, up to 16-bytes.

Despite a write-through data-cache policy, the i960RN, with no snoopy cache coherence support, still suffers from cache coherency issues on read. Thus data read by the i960 core, but subsequently over-written by other devices such as DMA engines can become stale in the i960's data cache. The only mechanism for solving this problem is flushing the entire data cache, which our offload code invokes at the end of each DMA. Flushing at the cache-line granularity, a mechanism found in many modern processor families (e.g. PowerPC) should have a much less devastating impact but is unavailable on the i960RN.

Exasperated at the badly designed data-cache, we ran an experiment to find out if it is of any use at all. Table 3 compares the internal bus utilization for 8kbyte transmit when data-cache is disabled. Overall, without the data cache, execution time is 54% longer. Thus, there is value to the data cache. Table 3 also shows that the i960 core has the biggest increase in bus ownership, and that the majority of the increased cycles actually go to non-data uses.

	iNIC's role	Tx				% increase when data cache disabled
	Msg size (bytes)	8k		8k		
	Data cache	enabled		disabled		
Utilization	Total bus clocks	165M		253M		54%
	Idle	66M	38%	97M	38%	47%
	Data usage	18M	10%	22M	9%	25%
	Non-data usage	88M	52%	135M	53%	52%
Ownership	i960 core	94M	55%	145M	57%	54%
	DMA engine 0	-	-	-	-	-
	DMA engine 1	4.4M	3%	4.4M	1.7%	-1%
	PATU	1.0M	1%	0.9M	0.4%	-6%
	SATU	6.0M	4%	5.8M	2.3%	-3%
Wait/grant	i960 core	3.42		3.75		9.5%
	DMA engine 0	-		-		-
	DMA engine 1	8.08		8.37		3.6%
	PATU	5.48		5.79		5.5%
	SATU	5.79		5.31		-8.2%
Own/grant	i960 core	6.11		7.13		16.7%
	DMA engine 0	-		-		-
	DMA engine 1	10.58		10.77		1.8%
	PATU	6.98		7.28		4.3%
	SATU	19.16		20.99		9.6%

Table 3: Internal bus utilization for 8kbyte transmit.

In summary, our investigation found that the i960RN processor's memory system in general, and internal bus in particular are badly equiped for handling the TCP/IP protocol processing code that we are running. While the numbers suggest that the i960 processor code could be stalled a very large part of the overall execution time, there is no easy means to extract the number which will yield the CPI. One speculative guess is a CPI of about 3, based on the guess of 60% processor stall time. Even when that is factored in, the cost of TCP/IP protocol processing still seems rather large. The next section looks into the tcp_output function, which is responsible for a large fraction of processor cycles in the 8kByte transmit case, to understand where the cycles are going to.

3.3 tcp_output cost breakdown

Table 4 displays breakdown of tcp_output cost. Processor usage under tcp_output is the domiant cost for large messages but important even for 200-byte message transmit where it is responsible for a quarter of each transmitted packet's cost. It is not a dominant cost of 1-byte message transmit because host-iNIC interface is so high in that case as to completely overwhelm other costs.

INIC's role		Tx						Rx	
Msg size (bytes)		8k		200		1		Ack	
Per Tx pkt cost (pclks)		16,933		107,158		4,461,191		10,843	
Tcp_output's share of Tx cost		5,452	32%	27k	25.5%	352k	8%	4,282	40%
tcp_output component	Send? decision	16%		10%		94.6%		17%	
	mbuf alloc	8%		2%		0.1%		16%	
	Form header	15%		14%		0.6%		33%	
	Data copy	43%		70%		4.2%		0%	
	Stats & control	18%		3%		0.5%		34%	

Table 4: tcp_output processor usage breakdown.

We break tcp_output into four components, reported as percentage of `tcp_output` cost. The first is deciding whether to send a segment, reported in the row labeled "Send? Decision" in Table 4. This is the first thing that happens when execution enters tcp_output. In the case of 1-byte messages, execution ends here in most invocations of tcp_output as aggregation waits for around a thousand bytes before sending a segment. Execution may also terminate here if there is no more window space to transmit another segment under tcp's flow-control scheme.

The next component is mbuf allocation. This is the cost of allocating message buffer data structure to contain tcp and ip headers. Buffering strategy will be examined more carefully in the next section (Section 3.4).

Next comes the cost of actually formatting a header. As is evident from Table 4, this component is very significant in the case of the transmission of an acknowledgement packet.

For data packets, some form of data copy occurs in `tcp_output`. This may be just copying the anchoring data structure that contains references to the actual data, as happens for large messages such as the 8-kbyte transmit, or actual copying of data as happens for smaller messages such as the 200-byte and 1-byte messages.

Finally, tcp_output updates some statistics registers and sets up flow-control related timers. Again, this is significant for ack packets that do not have any data to copy.

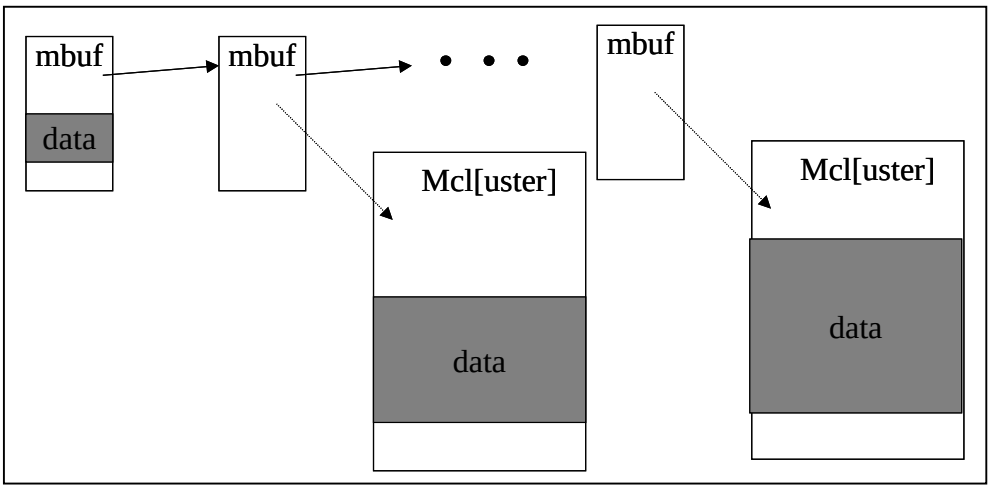
The cost of copying data for data packets stands out in Table 4. For 8-kbyte message, it accounts for 43%*32%=14% of total transmit cost, for 200-byte message, it accounts for 18%. In fact, most of the cost incurred in the socket layer, as reported in Table 1 is also related to the way buffering is done, and the socket layer accounts for a hefty third of the total cost of 1-byte message transmit. Clearly, the buffering strategy in the protocol stack warrants an examination; we report our findings of such an investigation in the next section.

3.4 Buffering… and the resulting copy

The BSD derived TCP/IP protocol stack on our offload implementation uses the venerable mbuf data structure which dates back to the VAX-11/780 era [9]. The mbuf

data structure is used throughout the BSD derived protocol stack, from buffering within a socket to holding Ethernet frames that are about to go onto the wire. In several situations, it is also used as a general memory region that is "cast" (C type cast) into some other data structure.

The mbuf data structure uses a two-tier buffering strategy. A basic mbuf is a 128-byte memory region that can contain up to about 100-bytes of data with the balance of the 128-bytes used for storing control information. Several of these can be chained together to provide larger buffering space. When allocating buffer space larger than will fit within two such basic mbuf's, larger buffers, called external mbufs, clusters or mcl's are used. On most platforms, these are two kilobytes. Each cluster still has to be anchored to a basic mbuf data structure which stores appropriate control information. The design does not permit simultaneously using the data space in the basic mbuf while using the same mbuf to point to a cluster. The following figure illustrates a possible mbuf chain.



The mbuf data structure is cleverly designed to facilitate pre or post-pending and corresponding removal of information without copying existing data -- common operations on message buffers as they move down or up network protocol stacks. Aside from the flexibility from the linked list structure, the mbuf design leaves the start and end of data in any particular mbuf data structure flexible through using a "begin of data" pointer together with information on the length of contiguous data. Copying cost is reduced in the case of clusters by allowing references to be made. For example, if a message is buffered with the use of a cluster, copying involves allocating another basic mbuf for storing control information but this new mbuf will simply point at data in the existing cluster. A reference count strategy is employed to track when cluster buffers can be reclaimed.

Since the mbuf data structure was designed at a time when memory was expensive, the designed was skewed towards conserving memory space. So for instance, when data resides in a socket, at attempt is made to compact data that are within basic mbufs to reduce fragmentation, possibly freeing up mbuf's in the process. (Compacting is not done to data in clusters.)

Function	Comment	Average cost (pclks)
MGET	Allocate a basic mbuf	290-380
MGETHDR	Allocate a basic mbuf used at head of a message	300-400
MCLGET	Allocate a cluster	400
MFREE	Free either a basic mbuf or a cluster	360

Table 5: mbuf allocation and deallocation costs.

The mbuf data structure is a cleverly designed, highly flexible data strcuture. However, it incurs considerable cost for each allocation and deallocation as listed in Table 5. Furthermore, there is a tendency in the BSD TCP/IP protocol code to allocated multiple data structures as a message goes through the network protocol layers, particularly on transmit. Table 6 shows the cost of mbuf allocation and deallocation in several TTCP runs. Clearly, it is a very significant cost component in the case of 8kbyte message transmit.

iNIC's Role	Tx			Rx
Msg size (bytes)	8k	200	1	8k
mbuf management as % of total	22%	9%	0%	1.6%
mbuf managemenet in Tx as % of Tx costs	25%	9%	0%	2.5%
mbuf managemenet in Rx as % of Rx costs	10%	2%	0%	1.5%

Table 6: mbuf management costs.

3.4.1 Buffer usage for large message transmit

Our investigation shows that roughly five mbuf-type data structure is allocated for each transmitted packet of a large message (e.g. 8-kbyte message). This seems a lot. The actual cost of using so many buffers is actually higher than indicated in Table 6 because of associated data copying cost. Why are so many instances of mbuf-type data structure used? The following list accounts for each one.

- Fraction of one mbuf, cast into data structure for storing DMA context information. One is allocated in the function `dma_send_data_from_host` for each message. Therefore, depending on the size of the message relative to packet (MTU) size, the per packet amortized cost varies. This is deallocated in the dma completion interrupt handler.
- An mbuf and a cluster, allocated in the function `dma_send_data_from_host` as buffer space to DMA data into. This is deallocated only when an acknowledgement is received from the other end of the tcp connection.
- One mbuf for tcp/ip header, allocated in `tcp_output`. This is deallocated by the Ethernet device interrupt handler once a packet is transmitted.
- One to two anchoring mbuf's, allocated in an mbuf copying function called from `tcp_output`, for "copying" cluster data by reference to send down the protocol stack. One anchoring mbuf is sufficient if the data within the packet is contained within one cluster. More often than not, the data straddles two clusters, resulting in

the need for two anchoring mbufs per transmitted packet. These are deallocated in the Ethernet device interrupt handler once a packet is transmitted.

With memory no longer as costly a resource as it used to be, the buffering strategy should be modified to minimize the number of data structure allocations and deallocations. One possible scheme allocates a large data buffer with each socket so that each packet incurs very small amortized data buffer allocation and deallocation costs. As processing passes down the protocol stack, only one anchoring mbuf-like data structure is allocated for each packet to contain its header information and identify its data by reference into the socket data buffer. Linux uses a similar scheme: a large contiguous buffer, but leaves space for header in the buffer when copying message data into it.

3.4.2 Buffer usage for small message transmit

Buffer usage behavior for small message transmit is different from that for large message transmit. Only two mbuf data structures are allocated for sub 104-byte messages, and twice that for messages larger than 104-bytes but smaller than 208-bytes. More significant, however, is the data copy that happens. The iNIC software literally copies small message data up to three times during transmision. These happen in the following situations:

- copy from I2O message frame into mbuf
- copy to compress buffer usage when queuing mbuf into sockbuf
- copy for sending down the protocol stack after the tcp layer

To get an idea of data copying cost on the iNIC, we instrumented the `memcpy` function. As shown in Table 7, this cost is very significant for 200 byte messages. Data copying is closely tied to the buffering strategy. From an efficiency point of view, it will be best if the host can directly place data into the large socket buffer that we suggested at the end of last section. By using only reference throughout the protocol stack on the iNIC, no data copy is needed.

	Memcpy cost as % of total cost	Memcpy cost per messasge (pclks)
200byte message transmit	35%	4721
8kbyte message transmit	2%	2640

Table 7: memcpy cost incurred on iNIC during TTCP runs.

3.5 Host-iNIC interface behavior

The cost of host-iNIC interface is very significant for small message transmit, as is evident from the numbers in Table 1 for transmission of 200-byte and 1-byte messages. This is partly due to Nagel Algorithm aggregation, and the fact that for the offload implementation, this aggregation happens on the iNIC. As a result, while the host-iNIC interface and socket layers are invoked for every message, the other layers are only invoked once after a number of messages accumulate enough data to trigger transmission.

Independent of the effects of aggregation, the host-iNIC cost is high on a per-instance basis. Table 1 shows that this ranges from over three thousand pclks for 1-byte transmit, to over seventeen thousand pclks for 8-kbyte transmit, to over twenty-six thousand pclks for 8-kbyte receives. To understand what accounts for these costs, we divided the host-iNIC interface into several components and measured the cost of each component. The results are summarized in Table 8.

iNIC's Role		Tx						Rx	
Msg size (bytes)		8k		200		1		8k	
Per msg cost (pclks)		121,020		14,243		5,135		72,812	
Host-iNIC interface share of per msg cost		17,396	14%	7,001	49%	3,034	59%	26,403	36%
Component	Msg unit interrupt	9976	57%	1573	22%	1742	57%	3597	14%
	Software copy from I2O buffer	0	0%	5429	78%	1292	43%	0	0%
	DMA interrupt	3703	21%	0	0%	0	0%	3781	14%
	DMA "call back" fn	3718	21%	0	0%	0	0%	0	0%
	Recv DMA	0	0%	0	0%	0	0%	19026	72%

Table 8: Host-iNIC interface cost breakdown

In Table 8, the "Msg unit interrupt" component includes the tasks of handling I2O messages sent from host to iNIC and making DMA requests to DMA hardware. Given that 200-byte and 1-byte messages do not involve any DMA, we can surmise that handling I2O message alone costs around 1.6 thousand pclks. This also implies that for 8-kbyte message transmit, setting up DMA requests, including allocating the mbuf data structures for DMA-ing data into, costs close to 8.5 thousand pclks. Our profiling shows that roughly 6 sets of mbuf-cluster pairs and DMA request chain elements are allocated for each 8-kbyte message at an average cost of roughly 1.4 thousand pclks per set. For 8-kbyte message receive, "Msg unit interrupt" results in calls to the function `sockism_release_buffer` in which host side (receive) buffer are returned to iNIC.

The "Software copy from I2O buffer" entry in Table 8 corresponds to the costs of allocating mbuf data structures and copying data from I2O buffers into these data structures. This cost is incurred only for small message transmits, such as in the cases of 200 byte and 1 byte messages. In these cases, it accounts for a very significant share of host-iNIC interface cost.

"DMA interrupt" entry in Table 8 is the initial interrupt handling at DMA completion, which may be invoked by hardware interrupt or via software polling. It includes manipulating the DMA request data structures of newly completed DMA requests, calling the relevent DMA completed call-back functions, and eventually deallocating these data structures. It also enables any pending DMA requests. This applies to both transmit and receive.

The "DMA call-back function" for transmit deallocates the I2O message frames of the messages sent by host to trigger TCP transmit. It also includes a call to invalidate the data cache, because at the point, new data has been DMA-ed into iNIC memory, data that has to be made visible to the i960 processor. This function also calls `tcp_output` to

attempt sending out tcp segments but the cost of this call is deducted from the numbers reported in Table 8.

"Recv DMA", which applies to receive only, corresponds to the function `indicate_recv_using_dma`. This is where DMA requests are allocated and queued with DMA hardware to move received data from iNIC memory to host memory. An I2O message frame is allocated; this is sent to the host by another function when DMA completes. Also allocated are message buffers on the host side used as destinations of DMA transfers.

Overall Table 8 shows that the cost of moving data between host and iNIC dominates regardless of whether this is done with DMA hardware or via embedding the data within I2O message frame, with a lower bound cost in the thousands of processor clock cycles. Clearly, the cost makes it highly inefficient for Nagel Algorithm to occur on the iNIC. To move that back to the host, however, requires an efficient way for a TCP connection's state to be shared dynamically between host and iNIC.

3.6 Miscellaneous Inefficiencies

Our evaluation found several other inefficiencies. One is the implementation of the socket buffer (sockbuf) which is based on an mbuf linked list. Many operations on socket buffer are implemented with linear traversals. For example, the socket buffer data structure keeps the mbuf list in a singly linked list with no pointer to the end of the list. Enqueuing to the end of the sockbuf (`sbappend`) ends up requiring a linear traversal through the entire list. Clearly the original implementors expected each sockbuf's buffer space to be limited to a small amount so that the linear list traversal is not a problem.

Another example is the `m_copym` function, which copies a number of bytes from a sockbuf starting at a certain offset. The BSD TCP/IP stack keeps all unacknowledge data within a sockbuf. Sending the next unsent byte typically involves sending data starting at a certain non-zero offset from the beginning of a sockbuf. The `m_copym` implementation traverses the sockbuf's linked list of mbuf from the beginning until the required offset is reached, at which point copying starts. This is not efficient in the common case where no packet is lost during transit so that the offset is the end of the regioins copied by the last call to `m_copym`. A slightly more complicated implementation that tracks this point can avoid the list traversal except when packet loss brings the offset backwards.

We also came across several inefficient artifacts of the gcc960 compiler. One is each struct assignment is implemented with a `memcpy` function call, even when the struct is as small as 16 bytes. On the i960RN with its register windows, each procedure call costs between 30-40 processor clock cycles. (We measured this with a fibonacci micro-benchmark.) Contrast this with i960RN's single-instruction 16-byte load/store instructions. A pair of 16-byte load/store instructions that hit in the data cache takes only 19 cycles. This goes up to around 50 cycles when they miss in the data cache. Adding the overhead of procedure call is thus very significant for assignments of small structs, something common in the networking code. The mbuf data structure has a 16-byte

control struct at its beginning for cluster related information. Another example is the TCP/IP header. The header template that is copied into each TCP segment before further customization for the specific segment is a 40-byte chunk of data.

4 Host-side Behavior

We turn our attention to host-side behavior in this section. Our evaluation of host side behavior is at a "black box" level, focused on looking at host processor utilization and the number of host interrupts. We did not do any breakdown profiling like those we did for iNIC-side behavior. Instead, we contrast the host processor utilization under our offload implementation with that of native NT TCP/IP protocol stack, and that of an Alacritech NIC which offloads the common path processing of TCP/IP onto custom ASIC in a technology they call SLIC. All measurements reported in this section are made with the Performance Monitor tool that comes with Windows NT. The system has a 300MHz Pentium-II processor and 64Mbytes of SDRAM. When quiescent, this system has around 64 interrupts/s and negligible processor utilization.

Table 9 displays the raw bandwidth, interrupt rate, processor utilization, and privilege mode processor utilization statistics for the three platforms under comparison. The bandwidth numbers show that our offload implementation delivers worse performance than native NT implementation in most cases. We can further gather that except for the case of 1-byte message transmit, the iNIC is the bottleneck because the host processor usage rate is relatively low. The Alacritech's NIC delivers bandwdith comparable to that of native NT TCP/IP stack execpt for small messages, with the cross-over point somewhere between 2-4kbyte messages.

Performance at the extreme case of 1-byte is uniformly bad across all implementations and rather odd in the NT case where something other than processor utilization is throttling performance. In fact, if we look at processor utilization under NT's native TCP/IP, it increases as the transmitted message size decreases all the way until 400-byte message. But going down to 1-byte reverse this trend. We do not know the reason for this. Alacritech's implementation exhibits a similar anomaly at 400-byte transmit, with a dip in bandwidth and processor utilization against an otherwise similar trend.

While interesting in themselves, the raw numbers in Table 9 do not facilitate direct comparison because different platforms achieve different bandwidths. In particular, our offload implementation delivers significantly lower transmit performance, due to throughput bottlenecks on the iNIC. In an attempt to enable more intelligible direct comparison, we extrapolated the statistics from Table 9 to extract the total amount of interrupts and processor utilization over the coarse of an entire TTCP run. This is displayed in Table 10 where we also dropped the privilege mode utilization numbers because they do not tell us much more than the overall processor utilization. (The total processor usage numbers are derived by multiplying the usage percentage by the duration, and can be thought of as the total work done by the processor.)

Role		Tx								Rx
Msg size (bytes)		16k	8k	4k	2k	1460	1000	400	1	8k
Offload	Bandwidth(KB/s)	7358	6593	5519	3656	3728	2925	2491	14.5	10433
	Interrupts/s	525	900	1440	1290	1370	2030	2740	15000	1270
	Proc. usage	9.8%	12%	14.8%	18.4%	24%	31%	31%	100%	34%
	Priv. proc usage	9.3%	11.2%	13.7%	16.4%	21%	28%	28%	90%	32%
NT	Bandwidth(KB/s)	11510	11516	11373	11228	11536	11134	9970	8.6	11567
	Interrupts/s	11700	11300	11500	12150	12100	13750	10200	85	8200
	Proc. usage	62%	78%	51%	82%	85%	90%	95%	17%	72%
	Priv. proc usage	61%	76%	49%	77%	79%	85.5%	87%	15%	67%
Alacritech	Bandwidth(KB/s)	11523	11516	11523	6450	6456	5384	5.97	5.09	11561
	Interrupts/s	795	1500	2970	3280	4537	4435	101	5520	6150
	Proc. usage	30%	42%	41%	29%	24%	38%	0.3%	100%	42%
	Priv. proc usage	29%	41%	37.7%	26.5%	22%	34.5%	0.2%	95.5%	38.5%

Table 9: Host-side processor utilization and interrupts raw statistics.

Table 10 shows that both our offload and Alacritech's SLIC NIC achieve lower processor utilization than native NT's TCP/IP protocol stack for transmission of large enough messages. At messages larger than 2kbytes and again for messages smaller than 400 bytes, our offload utilization has lower procesor utilization than Alacritech's.
For small messages, however, both our offload implementation and Alacritech's NIC actually have higher processor utilization than NT's native TCP/IP stack. The cross over is around 1000 bytes for our offload implementation and a smaller size (larger than 400 bytes) for Alacritech.

Role		Tx								Rx
Msg size (bytes)		16k	8k	4k	2k	1460	1000	400	1	8k
Num messages		40960	40960	40960	160000	200000	240000	128000	640000	40960
Offload	Duration (s)	89.06	49.70	29.69	28.5	76.5	80.1	35.1	43.2	31.4
	Total interrupts	46800	44700	42800	112k	105k	163k	96k	647k	39.9k
	Total proc usage	8.7	6.0	4.4	16.1	18.4	24.8	10.9	43.2	10.7
NT	Duration (s)	56.94	28.45	14.41	28.5	24.7	26.3	5.0*	72.7	28.3
	Total interrupts	666k	322k	166k	346k	299k	362k	51k	??	232k
	Total proc usage	35.3	22.2	7.4	23.4	20.9	23.7	4.8	12.4	20.3
Alacritech	Duration (s)	56.88	28.47	14.22	49.61	44.17	43.53	8370*	122.89	28.34
	Total interrupts	45.2k	42.7k	42k	163k	200k	193k	684k	678k	174k
	Total proc usage	17.06	11.82	5.83	14.39	10.60	16.54	16.69	122.89	11.90
Interrupt ratio (offload/NT)		7%	14%	26%	33%	35%	45%	188%	10k%	17%
Interrupt ratio (Alacritech/NT)		7%	13%	25%	47%	67%	53%	1338%	11k%	75%
Proc usage ratio(offload/NT)		25%	27%	60%	69%	88%	105%	229%	349%	53%
Proc usage ratio(Alacritech/NT)		48%	54%	79%	62%	51%	70%	350%	995%	59%

Table 10: Host-side processor utilization and interrupts normalized statistics.

For our offload implementation, the problem with small messages is that Nagel algorithm aggregation is done on the iNIC, resulting in too much host-iNIC interaction overhead. It would have been cheaper to do the aggregation on the host side if connection state can be efficiently and dynamically shared between host and iNIC.

For Alacritech's SLIC NIC, the problem is not aggregating outgoing message data at all. (We used tcpdump to snoop on packets coming out of the NIC and saw that.) We looked

but found no software configuration switch for enabling/disabling Nagel algorithm for the Alacritech NIC. As a result, it actually sends out Ethernet frames that are exactly the message sizes, resuting in performance that is even worse than our offload implementation for small messages.

Overall, the host-side statistics shows that offloading can decrease processor utilization. However, it also clearly shows that the iNIC we are using is unable to achieve competitive throughput. Furthermore, handling of small messages needs improvements.

The Alacritech number also raises challenging questions for our choice of implementation technology. The Alacritech SLIC NIC uses a custom ASIC which does simple TCP/IP processing for non-exceptional cases (i.e. no out-of-order packet arrival, no resetting send window back for re-transmit, etc.). All exceptional cases are handled back at the host, and tcp connection state will have to be flushed between the NIC and host when such events occur. Such exceptional cases are rare, especially in a cluster environment. From a silicon efficiency point of view, the Alacritech approach is definitely better than our offload, since our iNIC actually has a full computer system on it. However, it does not offer the kind of control and ability for extensions to the common path that is possible in our offload implementation.

5 Related Work

This section discusses two types of related work. In Section 5.1, we discuss some previous studies of TCP/IP implementations. These, while not exactly contradict our findings, had led people to expect much lower processing costs. Then in Section 5.2, we compare our offload implementation with other similar efforts.

5.1 Studies on TCP/IP implementation issues

In 1989, Clark et. al. [2] published a much cited analysis of TCP processing overhead which reported both (x86) instruction counts obtained by manually counting compiled code and latency times measured with a hardware logic analyzer. One has to be very careful in interpreing their instruction counts, which suggest that TCP and IP with a simplified (non-mbuf) buffering scheme takes under 400 instructions as summarized in Table 11. (We put together Table 11based on information extracted from their paper.)

	Data sender		Data receiver	
	Tx(data)	Rx(ack)	Tx(ack)	Rx(data)
TCP	235	191-213	235	186
IP	61	57	61	57
Buffering	40	30	?	35
TCB lookup	-	25 (est.)	-	25 (est.)
Timer operation	35	17-41	-	0-35
Total	371	320-366	296+?	303-338

Table 11: Summary of TCP/IP processing cost instruction count from Clark et. al.

In particular, one should note that the counts do not include the cost of Ethernet device driver, checksum computation and at least one if not two data copies that are present in most systems. While these are extraneous to the control portion of TCP which is the subject of their paper, the supporting infrastructure has to be there to make things work. Their reasoninng was that these costs are implementation dependent and hence left out. Due to these exclusions, which we did not separate out when profiling our offload implementation, it is not possible to directly compare our numbers with their instruction count.

Clark et. al.'s hardware measured latency numbers for a 2MIPS (20MHz) Motorola 68020 based Sun-3/60 workstation provide a more complete picture. Table 12 summarises their measurement, with the middle column taken directly from their paper and the right column derived based on the reasoning they used in their paper for extrapolating latency measurements into cycle counts. If we take the very rough view that this is a 20MHz machines, Table 12 suggests that the entire protocol stack processing takes on the order of 24 thousand clock cycles on a CISC machine. This is quite consistent with the roughly 26 thousand clock cycles we reported in Table 1 to handle an incoming and an outgoing packet under 8kbyte message transmit.

	Measured overhead (us)	Probably instruction count
User-system copy	200	400
TCP checksum	185	370
Network-memory copy	386	772
Ethernet driver	100	200
TCP+IP+ARP protocols	100	200
OS overhead	240	480
Total	1211	2422

Table 12: Summary of TCP/IP processing measured latency from Clark et. al.

The more interesting aspect of Clark et. al.'s paper is their observation that memory system and data copying, buffer layer and OS infrastructure are the main challenges to efficient TCP/IP implementation. In fact, the discussions at the end of the paper suggested that an offload approach may help overcome these problems. Unfortuantely for us, our i960RN based offload implementation failed to address the buffering layer and "special high-performance memory architecture" that Clark et. al. expects of an "outboard" (equivalent to our offload) implementation. Clark et. al. also expressed their opinion against hardwiring protocols in silicon as they believe that the protocols themselves will continue to evolve.

In a workshop talk in 1993, Van Jacobson mentioned a TCP/IP implementation that does away with mbuf's [4]. It was clear by then that mbuf's were suboptimal for the relative cost of various components in a computer system. Unfortunately, that implementation never made it into BSD networking code as far as we can tell, because today, the BSD TCP/IP networking code is still mbuf based. Van Jacobson claimed a speed improvement of one to two orders of magnitude.

In 1996, Moseberger et. al. [6] reported tcp input timing for a 1-byte message on a DEC Unix v3.2c system with an alpha 21064 processor. From IP input to TCP input, they measured 262 instructions and from TCP input to socket input, they measured 1188 instructions, with a CPI of 4.26. This suggests that from IP input to socket input takes 6177 clock cycles. This is not inconsistent with our numbers in Table 1 (consider per Rx packet cost for Rx for 8kbyte).

Moseberger et. al.'s work actually focused on compilation techniques for improving network code performance. The goal is to achieve better cache memory system behavior and take advantage of partial evaluation. They rearrange basic blocks to achieve executing along mostly contiguously addressed instructions. They also combine blocks across source-level functional and possibly module boundaries so that optimization can be done over a larger body of code. This is similar to what is done in many VLIW compilers and more recently, in Just-in-Time/Code Morphing compilers. In Moseberger's work, the partial evaluation is done at a protocol level; they did not attempt to wait till a conneciton is set up to produce a per-connection specialized code for fear of code bloat. They reported a latency improvements of about 13% for their system which has a slow Ethernet device. If the Ethernet device latency is completely removed, leaving mostly latency due to processor execution, their modifications improve latency by 40%. None of the techniques they investigated are in the gcc960 compiler used in our offload implementation.

A recent (June 2000) paper by Chase et. al. [1] studies the effect of various end-system optimizations on TCP/IP processing cost. Part of the results they reported is a breakdown of host CPU utilization on a Compaq workstation with 500 MHz Alpha 21264 processor, which uses 95% of procssor cycles when handling 370Mbits/s network traffic consisting of 1.5 kbyte (Ethernet MTU) packets. (It uses a Myrinet NIC.) Table 13 is an approximation of the breakdown percentages extracted from a graph in their paper. Other parts of the paper suggest that removing software checksum should reduce the copy and checksum cost by between a third and half. That leaves very significant copy cost, in the 15-20% range. Interrupt cost, at 20% is very significant. However, so is buffering cost, at 16%, which exceeds the 12% of TCP/IP processing, what people would consider as the "useful work ".

	Processor usage %
Copy and checksum	30%
Interrupt	20%
VM	8.5%
NIC driver	8.5%
Buffering	16%
TCP/IP	12%
Idle	5%

Table 13: Processor utilization breakdown as reported by Chase et. al. for TCP/IP over Trapez/Myrinet.

The numbers from Chase et. al.'s paper broadly agree with what we measured on our offload implementation. For example, our offload implementation drastically reduced the number of host interrupts (down by 93% for 16kbyte message transmit), removed all TCP/IP processing, checksum, NIC driver and part of buffer management from the host processor. Based on Chase et. al.'s numbers, this leaves about 25-30% of the original processor usage, which is what we saw for 8-kbyte and 16-kbyte message transmits.

5.2 Other networking protocol offload work

Over the years, many attempts at offload, or outboard implementations have been made. We focus on a few recent ones.

We examined the performance of the SLIC NIC from Alacritech back in Section 4. SLIC performs TCP/IP protocol processing on their NIC's ASIC for non-exceptional cases. One of the white papers on their web site claims that they achieve a single hardware copy between host and NIC memory (essentially what would classically be called zero-copy), interrupts reduction and reduced PCI bus traffic as there is less direct control of the NIC by the host processor. However, our measurement does not quite support that claim. Their host processor utilization is higher than our offload implementation which we know does one copy in software on the host side. It is still possible that they somehow introduce enough host-side processing to result in the observed host-side utilization, but we are unable to determine what that may be. (Our TTCP runs on the Alacritech NIC's did not encounter exceptional condition, as indicated by their own Perfmon extensions which showed that all traffic is handled by the ASIC.)

At their web site, Alacritech publishes a third party performance testing done by ZD net. These are throughput, processor usage and latency numbers measured with ZD net's NetBench benchmark. The host is a Xenon P-III 500 MHz processor with 1-Gigabytes of RAM, and most tests use multiple Alacritech's SLIC NIC's as well as multiple ports on the NIC's (we used only one port on a 4-port NIC). Since the setup and benchmark are different from ours, we cannot do a direct comparison.

Alteon's Gigabit Ethernet NIC (also OEM by HP and sold as the A4929A Gigabit NIC) offers certain features to reduce host processor utilization. These include checksum computation, interrupt coleascing under heavy traffic, and (they claim) zero-copying. Our colleagues in Cupertino reported that on a J6000 system with 440MHz PA-RISC 8600 processors, performance max-out at around 650-700Mbits/s when running Netperf.

A company called Interprophet was working on and presumably selling NIC's with TCP/IP implemented in FPGA hardware. (Our latest attempt to access their web site – on Nov 2, 2000 – failed to open the web site. The site was up about 5 weeks ago when we accessed it. It looked very much a "garage" operation.) We do not have much details or performance statistics for this NIC. Some Korean acadamic researchers reported an FPGA implementation of TCP/IP over 155 Mbit/s ATM [5]. There was little concrete performance statistics in that case too, other than the claim that they achieved wire speed. We also heard rumours that Lucent is working on silicon implementation of TCP/IP protocol, but do not have any further information.

6 Conclusion

We conclude this report by first listing in Section 6.1 the limitations of this study, which are possible subjects of future work. In spite of these limitations, this study has shed light on our network protocol offload work. Section 6.2 recapitulates these results. Next, we consider the implications of these results, particularly with respect to the whole protocol offload and iNIC idea in Section 6.3 . Finally, Section 6.4 wraps up the entire report with some suggestions for future work.

6.1 Limitations of this Study

A number of important issues were not addressed in this study. Firstly, receive behavior has not been studied in any detail. Secondly, connection setup and tear down performance has not been evaluated. Thirdly, we did not study bulk transfer when a large number of connections are outstanding. Fourthly, it will be good to move beyond micro-benchmark like TTCP to an application level benchmark to get a better understanding of overall performance.

On the iNIC side, it is unfortunate that we are unable to determine the CPI and cache miss rate on the i960RN processor. Although we have a good approximate picture of what happens on the iNIC, a quantitatively more precise picture will be very useful. The i960RN's unusual data cache design also raises questions about how a more modern, superscalar processor and memory hierarchy would perform. All these questions can be better answered in a new iNIC that we are jointly developing with Cyclone. In addition to supporting Gigabit Ethernet, this new iNIC has a PowerPC 750 processor with performance registers for determining CPI and cache miss rates and a modern cache memory hierarchy and modern processor core. In the long run, a flexible, accurate simulator is probably the best tool for investigating implementation isssues.

On the host side, running the Windows NT OS makes for a bad comparison because NT is well known to have much higher TCP/IP networking cost than any of the many Unix flavors. It will be interesting to use our offload iNIC with a Unix host and measure the host processor utilization. Again, the new iNIC we are developing together with Cyclone will be an interesting candidate for study because its first target host OS is HPUX.

6.2 Summary of Results from this Evalution

Despite the limitations listed in the previous section, we learned a number of useful things from this study.

Firstly, this study quantified the cost of TCP/IP protocol processing on our i960RN based iNIC offload implementation. It shows that the cost is much higher than was originally expected, running into tens of thousands of processor clock cycles for each packet.

Next, several causes for the high processing cost were identified. Hardware limitations is shown to be a big cause of the bad performance. While the internal bus is the immediate

hardware bottleneck, the problem extends to the data cache and memory hierarchy design.

Another major problem is bad buffering strategy which is a software decision. The bad design probably aggravates the weak memory hierarchy design problem. Of course even if the hardware were better, there is no point doing more work than necessary so the buffering strategy should be improved. Our study clearly identifies more attractive alternate buffering strategies to try out.[5]

Handshaking across the host-iNIC is found to be quite costly, and is responsible for the bad performance of our offload implementation for small message transmits. Furthermore, regardless of message size, there seems to be no cheap way to move data between host memory and iNIC. Both hardware DMA and direct embedding data in I2O message frames incur costs that have lower bounds in many thousands of cycles. From a pure performance point of view, avoiding this cost altogether with some unified memory would be best. Such an approach will have implications on how much control the iNIC retains apart from the host, a consideration that is important for some proposed usage of our offload approach.

Finally, our study validated that host processor utilization is lower with our offload implementation but only for large messages. Data copying, followed by system calls, are the next overheads to target.

6.3 Implications of our Findings

This work clearly calls into question the original hypothesis that offloading TCP/IP protocol processing to a specialized networking software environment coupled with cheap embedded processors is a cost effective way of improving system performance. At least in the case of our offload implementation on the i960RN, neither was the performance adequate nor the system cheap.

One could argue that the serious deficiencies in hardware and networking software we found makes it impossible to draw definitive conclusions. Surely, another better attempt should be made, in which better buffering strategies are tried, and a better processor used on the iNIC. Perhaps that will demonstrate much better or at least adequate performance.

We believe that this approach is fundamentally flawed. At the low bandwidth end, say 100 Mbit/s, the cost of such an iNIC is simply too high for it to be a viable approach. At the high bandwidth end, say 10 Gigabit/s, it is highly doubtful that any general purpose processor available in the next three years will be adequate to the task. Fundamentally, the issue is with implementation approach, not the idea of offloading itself.

One important question for us as we continue to explore iNIC implementations is the degree to which the offloaded functions need to be modified. If modifications are rare, than a hardwired approach such as that taken by Alacritech is probably the most cost

[5] We learned just before this report was completed that the offload implementation team had plans for better buffering strategy but didn't get around to implementing it.

effective solution. If changes are frequent, than one needs to examine alternate micro-architectures, such as multi-threaded processors common in the new generation of network processors. There is yet room for research in the network processor arena, where the specific architectures are far from "standardized" or well understood.

6.4 Recommendation for Future Work

We recommend three areas for future work.

One is a follow-up of this evaluate to arrive at a more complete picture, particularly to address the limitations of this study and the short comings found in this study. The following are some things to try out:

- Use a different buffering strategy on the iNIC. In particular, try replacing the BSD derived TCP/IP protocol stack with Linux's.
- Perform Nagel algorithm aggregation on the host side.
- Eliminate host side copying; since TCP source buffering is done on the iNIC, it seems eminently possible to replace the host-side user space to system buffer copy with a hardware DMA from host-side user space into iNIC memory.
- Use our new PowerPC 750 based iNIC to measure processing cost, CPI, and cache miss rate. If necessary, use a simulator to get a better picture of the memory system behavior.
- Evaluate the connection set-up and tear-down aspects of the implementation. Also try out bi-directional transfers, and having multiple connections active at the same time. This will exercise the protocol stack more broadly.

The other area of work is to look at innovative micro-architectures that are better equiped for network protocol processing, for instance, a better way of integrating the network interface into the host system so that the host-iNIC interface cost is lower than in our current implementation. Another aspect of this work is finding cost effective ways to exploit a high degree of processing parallelism. Interesting avenues include some form of multi-threaded processing, specialized hardware, and field programmable hardware. We need to clearly understand the kinds of parallelism that can be efficiently exploited and possible hardware structures that may be needed to make it efficient.

Lastly, this networking work should be put in a concrete context so that attention can be directed to the dominating networking characteristics of the intended applications. Work under this category includes having a clear grasp of the degree of separation of iNIC control from host, the degree of programmability and the level of computation power needed on an iNIC. Members of our research team had proposed putting things like web caches and fire-wall filters on an iNIC. The implications of such ideas, such as on memory requirements -- web caches are memory intensive -- and access to message data -- fire-wall filters that want to examine all the message data will put demands on the iNIC's memory system -- should be taken into account.

Bibliography

[1] Jeff Chase, Andrew Gallatin and Ken Yocum. *End-System Optimizations for High-Speed TCP*. June 2000. (submitted for publication but already available from web site at: http://www.cs.duke.edu/ari/publications/publications.html.)

[2] David D. Clark, Van Jacobson, John Romkey and Howard Salwen. *An Analysis of TCP Processing Overhead.* IEEE Communications Magazine, 27(6):23-29, June 1989.

[3] Intel Corporation. *i960 RM/RN I/O Processor Developer's Manual*. July 1998. Order Number:273158-001.

[4] Van Jacobson. *Some Design Issues for High-Speed Networks*. Networkshop'93, Melbourne, Australia, November 1993.

[5] Killyeon Kim, Kuhwan Kim, Kyohong Jin and Jungtae Lee. *A Design and Implementation of High Speed TCP/IP Hardware.* Proceedings of APCC '97, 3rd Asia Pacific Conference on Communications, pp. 212-216, vol 1, 1997.

[6] David Mosberger, Larry L. Peterson and Sean O'Malley. Analysis of Techniques to Improve Protocol Processing Latency. Proceedings of Conference on applications, technologies, architectures, and protocols for computer communications (SIGCOMM'96), Palo Alto, CA, pp. 73-64, August 28 - 30, 1996.

[7] C Partridge. *Jacobson on TCP in 30 Instructions*. <Message-ID 1993Sep8.213239.28992@sics.se> Usenet, comp.protocols.tcp-ip Newsgroup (Sept.).

[8] Lance W. Russell and Bert Munoz. Distributed Services Utility System Software Architectural Specifications. March, 2000. (http://web.hpl.hp.com/org/labs/csl/pss/dsu/docs/dsu-sa.pdf)

[9] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated, Volume 2 – The Implementation*. Addison-Wesley, 1995.