Fragmentation and High Performance IP

Patricia Gilfeather
Albuquerque High Performance Computing Center
University of New Mexico
pfeather@cs.unm.edu

Todd Underwood
Department of Computer Science
University of New Mexico
todd@cs.unm.edu

Abstract

Although the networking world is still dominated by IP over Ethernet, these protocol suites were not originally designed to work on modern Gigabit or Gigabyte networks. The major problem facing IP at 1 Gb/s and 10 Gb/s speeds is interrupt processing overhead and the resulting low CPU availability for applications. One solution to this problem is to implement endpoint fragmentation on a Network Interface Card (NIC), where an Ethernet driver sends large packets to the NIC and the NIC fragments the packets into frames. This approach reduces interrupt overhead and CPU utilization on the receiver.

In order to demonstrate the plausibility of this method, we modified the firmware of Alteon Acenic 2 Gigabit Ethernet cards and the standard Linux (2.4 series) driver to allow the driver to use an MTU of up to 64000 bytes without changing the 9K MTU limitation of the MAC layer. Each datagram is fragmented in the firmware on one end and reassembled in firmware at the other.

Our implementation matched peak bandwidth performance of the standard driver and firmware, while demonstrating a significant reduction in CPU utilization.

1 Introduction

1.1 The State of the Net

On commodity hardware, 100 Mb/s Ethernet adapters are capable of near line-rate TCP/IP performance (90%) at 15-20% CPU utilization. Projecting this performance out to Gigabit speeds, we can expect to see bandwidth between about 450 Mb/s and 600 Mb/s at near 100% CPU utilization. Modern systems that do better than this use one or several of a number of bandwidth-improving techniques including jumbo datagrams and interrupt coalescing. Bottlenecks for high speed performance no longer reside with the network infrastructure. With modern commodity hardware supporting memory-to-IO bandwidths of 480MB/s (3840 Mb/s), even memory performance is not the limiting factor. In fact, interrupt handling and the resulting CPU utilization is the major performance issue in efficient high-speed networks.

There have been several moderately successful attempts at addressing the new performance bottlenecks in high-performance local area network computing. The first, most-obvious choice, is to use protocols more suited to high-speed networking. There are several well-established protocols in the research community. Of particular interest are the protocols designed to bypass the operating system by exchanging buffers directly with user processes, saving a memory copy (from kernel to user address space) and an interrupt (since the OS does not need to be involved in the transfer). Scheduled Transfer developed many of the ideas involved in OS-Bypass, and the FIO/NGIO consortias' merger into Infiniband continues the investigation.

This project is an attempt to adapt some of the observations and goals of the OS-Bypass projects to IP, the commodity protocol. Since IP remains the most common protocol within LAN applications and certainly in the WAN world, IP implementations are incredibly widespread and therefore inherently appealing. Changing protocols usually means maintaining protocol stacks separately for the WAN (which remains TCP/IP) and the LAN and also requires a great deal of re-engineering. Using OS-Bypass ideas, we sought to implement a scalable and efficient way of addressing the major bottleneck in high-speed networks, while using IP.

1.2 Bottlenecks in High-Performance Communication

There are several sources of overhead that affect applications in their use of high-performance networks. The most significant bottlenecks are: interrupt processing, the resulting low availability of processors for applications, and stack overhead (the time spent tracking each incoming frame, demultiplexing it to the right application and delivering it to that application) . Memory copies are widely perceived to be significant, but are actually less significant in practice

than might be expected.

1.2.1 Memory Copy

As network speeds approach (and sometimes match) commodity IO bus bandwidth, the cost of copying network frames in memory, even once, becomes unacceptable. Most IP stacks still copy memory at least once: between user and kernel memory. This copy, added with the cost of the DMA from the NIC to host memory, is potentially devastating to full utilization of network bandwidth. Any memory copies that cannot be pipelined will have a direct impact on the communication bandwidth observed by the application.

Nevertheless, commodity memory-to-IO bandwidth are increasing from 120MB/s speeds (33MHz PCI at 32bits is approximately 120MB/s) to 480MB/s and beyond, and performance pressure from a single memory copy is becoming less important. However, even when memory copies do not directly impact the bandwidth seen by applications, they use resources (e.g., the memory bus and the PCI IO bus) which may also impact overall application and system performance.

1.2.2 Interrupt Pressure

Interrupt pressure, however, is the most significant concern for high-speed Ethernet networks. Table 1 presents the minimum inter-arrival times for 1500 byte packets at various network speeds. The minimum inter-arrival time is calculated by assuming a constant stream of 1500 byte packets. For example, the inter-arrival time for a 10Mb/s network is calculated as follows:

$$1500\text{bytes} \times 8\text{bits/byte} \times 1\mu\text{sec}/10\text{bit} = 1200\mu\text{sec}$$

Network Speed	Inter-arrival time
10 Mb/s	1200 μsec
100 Mb/s	120 μsec
1 Gb/s	12 μsec
10 Gb/s	1.2 μsec

Table 1. Minimum Inter-arrival Times for 1500 byte Packets

If the network interface generates an interrupt for each 1500-byte packet, a processor with a 1Gb/s network interface should be capable of handling an interrupt every 12 μsec. Although estimates of interrupt handling latency vary considerably and should be measured on each system, the PCI 2.1 specification estimates typical 33MHz latencies of 10μsec to 20μsec. Expected latencies on 66MHz buses should be lower. Even an extremely efficient operating system cannot hope to manage interrupts arriving at 12μsec, and the situation becomes utterly hopeless for 10 Gb/s networks.

1.2.3 Network Processor Overhead

Computers are not merely network nodes. They are computational devices designed to run applications. CPU resources that are expended on the communications infrastructure are not available for use by the applications. This is particularly harmful for computational clusters where nodes are expected both to communicate at high speeds and to perform a useful amount of computation efficiently.

Moreover, it has been observed that many high-performance applications are more sensitive to variations in overhead than variations in message passing latency or bandwidth. Many latency-sensitive applications do exist and even though we chose to specifically concentrate on CPU overhead and utilization in bandwidth-constrained applications for this report, we make some observations about latency in the Future Work section.

2 Current High-Performance IP Solutions

The most common current strategies to easing bottleneck pressures in TCP/IP are increasing the frame size from 1500 to 9000 bytes (jumbo frames); coalescing interrupts so that each inbound back-to-back packet does not generate an interrupt; and zero-copy stack approaches designed to reduce memory copy overhead.

2.1 Jumbo frames

Standard Ethernet Frames are 1518 bytes, including the MAC header and the CRC trailer. Jumbo frames, supported by many Gigabit NIC and switch manufacturers, increase the size of an Ethernet frame to 9000 bytes. While jumbo frames do relieve both interrupt latency and stack processing pressures by allowing larger packets across the network at one time, the jumbo frames require special routers and both endpoints in a transaction must be using TCP/IP stacks which allow for jumbo frames.

Jumbo frames are inadequate primarily because they are not fully inter-operable and they are not scalable. Jumbo frames have to be fragmented when transmitted onto traditional Ethernet networks, which obviously impacts performance. Finally, and most importantly, frames of only 9000 bytes will not contribute significantly to solving comparable problems created by a 10 Gb/s network, where inter-arrival times will only increase from 1.2μs to 7.2μs.

2.2 Coalesced Interrupts

The use of algorithms to reduce (or coalesce) the number of interrupts received has been extremely successful. The NIC should only interrupt the host after a specified amount of time or a specified number of packets have arrived. This technique is scalable and inter-operable. It solves the problem of interrupt latency cleanly and elegantly (and usually entirely in firmware or hardware).

There are, however, two significant disadvantages to interrupt coalescing. The first is that small packets of a latency-bound application are not allowed to move directly through the NIC to the IP stack on the host. Instead, they are subject to being coalesced and latency-bound applications can see their performance suffer.

More significantly, interrupt coalescing does nothing to reduce the overhead that each packet incurs in the host network stack. The processor still has to process each packet and the number of packets is not reduced. So, while interrupt coalescing reduces the frequency of interrupts and therefore the interrupt processing overhead, without the help of some other method to reduce the number of frames arriving, the host processor will still have to process too many frames. This explains why jumbo frames combine so well with interrupt coalescing.

2.3 Zero-copy

'Zero-copy' refers to two distinct ideas. The first is eliminating copying of network data in the IP stack on the host. This is an important improvement and has significantly improved IP stack performance over the past several years. The second is the idea of eliminating all copying of network data, including the final copy from kernel memory to user memory. Although this technique has been demonstrated successfully in some cases, zero-copy to user space has yet to be proven generally useful in an operating system standard release. The primary concern is that the overhead and special-casing in the page cache necessary to manage the transition between the two address spaces may exceed the overhead of a single memory copy into a user buffer.

Even when it is possibly to eliminate all copies, it is usually not necessary. Although it is important to reduce the number of times data arriving from the network is copied, truly eliminating copies is only important when network bandwidth approaches memory bandwidth. Modern 64bit/66MHz PCI buses offer approximately 480MB/s (3840Mbps) of bandwidth between IO and memory, almost four times as fast as a 1Gb/s network.

Zero-copy does not relieve interrupt latency pressures, but it does decrease the extent to which the speed of the memory bus is introduced as a bottleneck. Zero-copy infrastructures (of either kind) do not fundamentally resolve the interrupt arrival time problem, although they do improve performance when combined with jumbo frames and interrupt coalescing.

While our results take advantage of reduced copies in the Linux IP stack, we do not currently deliver data straight to user space, and our work does not extend the results of this area of research.

2.4 No Complete Solution

Fundamentally, none of these solutions addresses the problem caused by host memory and network bandwidths approaching each other. While each solution provides a viable framework for 1 Gb/s adapters on current PCI (commodity IO) bus architectures, they fail significantly when considering 10 Gb/s network speeds. Even at 1 Gb/s speeds with combinations of all three techniques, hosts exhibit unacceptably high CPU utilization, preventing effective use of the host processor.

A different approach is needed–one that takes seriously the existing trend in network and IO architectures. It is only by taking advantage of more powerful network hardware and off-loading some processing onto it that we can hope to effectively address CPU processing pressures at higher network speeds.

3 An Alternate Solution

We propose endpoint fragmentation as a scalable solution to relieving both interrupt pressure and host processor overhead induced by high-speed networking based on Ethernet. Endpoint fragmentation involves hosts fragmenting and reassembling their own datagrams. In our case the fragmentation will be off-loaded from the host hardware and implemented on the Network Interface Card (NIC).

3.1 Fragmentation's Sordid Past

Fragmentation has been virtually nonexistent since Kent and Mogul [5] identified serious problems with intermediate fragmentation (where routers fragment packets too large for the outgoing MTU of the next link) and proposed path MTU discovery in order to avoid any fragmentation by intermediate routers between two endpoints. Endpoint fragmentation is the only kind of fragmentation allowed in IP version 6.

Endpoint fragmentation does not share most of the disadvantages of intermediate fragmentation. Endpoint fragmentation is already commonly used by applications such as NFS that benefit from large datagrams being sent and received at higher levels of the protocol stack (in the case of NFS because disk blocks are the smallest unit of currency for disks).

Moreover, fragmentation and reassembly are tasks that are extremely well-suited for implementation on a reasonably powerful NIC such as the Acenic. They are easy to separate from the rest of the IP stack. This is accomplished simply by transparently accepting packets larger than the real MTU of the link and fragmenting on the card. Almost no modification of the driver is necessary.

Other tasks, e.g. TCP's sliding window protocol, might be good candidates for implementation on the NIC; however, the limited resources available on the NIC make successful implementation of these less plausible. Fragmentation and reassembly were relatively easy to implement on the NICs but worthwhile because they benefit a wide range of applications (including UDP-only applications), and significantly offload the host processor.

We believed that this strategy has the potential to perform extremely well for large datagrams (both in terms of bandwidth and CPU utilization) without adding latency to smaller ones.

3.2 Implementation Overview

We implemented endpoint fragmentation in the firmware of Alteon Acenic 2 NICs. We "misconfigure" the driver to allow the use of a higher MTU than the link physically supports. The firmware on the NIC fragments the packet into real MTU sized frames. On the receiving endpoint, if the NIC has firmware for reassembly, the packet is reassembled and the driver receives a complete non-fragmented datagram (and consequently a single interrupt for a much larger amount of data). If the receiving endpoint does not have firmware-assisted reassembly, the receiving host's IP stack proceeds with fragment reassembly as usual.

Our current implementation is a quick prototype and currently works effectively up to 55000-byte datagrams with a 64000-byte "advertised MTU", which is very close to the 64K ceiling allowed by the 16-bit length field in IPv4.

Alteon distributes source code to the firmware for the Acenic 2 cards and as such provides the ideal environment to prototype and measure the endpoint fragmentation architecture. Furthermore, the cards have a well-documented API and a variety of hardware assist mechanisms (for MAC and DMA queuing) which aided prototyping and development. Finally, the cards have sufficient memory (2MB) and CPU resources (two MIPS-4000-core running at 88MHz processors) to handle fragmentation and reassembly tasks for a sample implementation.

As a prototype, our current implementation does have several limitations. It generates and assumes forward-order fragments, two endpoints (or at least a very limited number of endpoints–there is little space on the card to store the state of datagrams-in-reassembly). The current implementation also does not correctly fragment fragments. That is, it assumes that every datagram it gets from the host is a complete datagram and does not attempt to preserve its fragmentation flags and offset. Our implementation also does not calculate or validate checksums on the fragments that it creates (although it does, of course, preserve the checksum of the original datagram). The Acenic firmware already supports off-loaded checksums as part of the DMA of a packet from the host to the NIC. Adding the checksum onto each fragment this should be an easy addition that does not significantly alter CPU utilization or bandwidth.

3.3 Send

The Acenic API supports three kinds of buffers: mini (100 bytes), regular (1500 bytes plus MAC header and checksum–a total of 1518 bytes) and jumbo (9000 bytes data bytes). We modified both the firmware and the Linux driver to allow jumbo buffers of up to 64000 data bytes. The driver sends the firmware a packet by filling a buffer with data (the first 34 bytes of which are the MAC header and the IP Header) and then creating a jumbo buffer descriptor to the send ring and signalling the firmware (by updating the producer index).

The firmware DMAs the buffer descriptor as usual. When the firmware is ready to DMA the data buffer pointed to by the descriptor, it initiates DMAs of no more than 1480 bytes (1514 minus MAC header and IP header), modifies the IP header (setting the MF bit, offset and length as appropriate) and enqueues the datagram to the MAC engine. This process is repeated until the entire datagram is sent.

For simplicity, in our prototype, we do not currently handle fragments from the driver (packets greater than 64000 bytes that get fragmented in the kernel), we do not calculate nor validate checksums, and we send (and are only able to reassemble) fragments in forward order. The inability to handle fragments in the firmware can be addressed by scaling the jumbo buffer size slightly higher to 64K (in this case, the stack will never generate fragments). Checksums should be calculated at the firmware level as well. This is supported by the firmware but has simply not been integrated into our existing prototype.

We do not believe that sending fragments (or reassembling them) in reverse order will offer any advantages for our implementation. The main advantage of reverse-order fragmentation is that the first fragment received contains all of the information necessary to allocate a buffer to receive the entire datagram (frag offset + length of the the last fragment is the total length of the reassembled datagram). This is not an advantage for our implementation since the host always allocates buffers of the size of the advertised MTU (64000 bytes in our implementation) regardless of the size of the actual received datagrams.

3.3.1 Performance

The challenge for the current send implementation was to keep the MAC queue full. In order to do this, we found it necessary to use a careful balancing of DMAs (bringing the fragments over into transmit buffers) and memory copies (copying the header to each transmit buffer to be modified with the correct offset). The Tigon II chipset on the Acenic NICs is known for having a robust but fairly slow DMA engine. In particular, the DMA engine always takes at least 5μsec to start a DMA. This made it difficult for our implementation to achieve good bandwidth at low message sizes, since we had to wait for the first DMA to complete in order to get the full header.

3.4 Receive

The driver maintains a ring of jumbo receive buffers (of 64000 bytes in our prototype) and the firmware will DMA over a descriptor for a buffer as it needs one. The firmware maintains a list of packets_in_reassembly, the list contains information necessary to match new fragments to partially reassembled datagrams. Currently, demultiplexing of inbound fragments is accomplished solely via the IP ID field, which works well for two-nodes but will not generalize. Demultiplexing should be extended to a hash of source IP and the IP ID at the very least.

As fragments arrive, they are matched against the packets_in_reassembly list, and immediately transferred, via DMA, to the buffer in host memory corresponding to the appropriate buffer descriptor. Currently, the method to find a matching entry in the list is a linear search (with a cache of one– the last fragment received). For two nodes, this is not an area worth optimizing, but in order to efficiently support multiple endpoints, this should be re-implemented as a hash.

Currently, if an out-of-order or reverse-order fragment arrives, the firmware discards the packet. We feel justified by earlier research [1] of fragmentation on LANs and WANs that out-of-order fragments will not occur on a LAN (and probably not on any common WAN either). When out-of-order fragments occur, the packet will time out. Reverse-order fragmentation is likely (many OSes fragment in reverse order by default in order to attain some of the advantages previously mentioned) and reverse-order reassembly will need to be added to our firmware in the future.

4 Methodology

We used an unmodified copy of netperf 2.1pl3 obtained from http://www.netperf.org to measure unidirectional UDP streaming bandwidth. We chose netperf as a testing platform because data from netperf are frequently cited by other high-performance projects. Additionally, among network performance measurement packages, netperf is one of few to have a reasonable UDP implementation. Netperf only reports the UDP bandwidth actually received, providing a reliable means of measuring bandwidth for an otherwise unreliable protocol.

We configured two 933 MHz, Linux 2.4.0(release) servers with version 0.49 of Jes Sorenson's Acenic driver (patched only to support tracedumps). The machines were connected by a cross-over fiber cable (with no switch). To test the unmodified firmware, we configured the interfaces with MTUs of 1500 and 9000 bytes respectively. In order to test our firmware, we configured the interface with an MTU of 64000 bytes, and configured the underlying firmware to use an actual MTU first of 1.5K and then 9K.

4.1 Why UDP?

Most high-performance networking projects report TCP bandwidths rather than UDP bandwidths. By reporting UDP bandwidths, we understand that it may be difficult to compare our results to those of other projects. We intend to report TCP bandwidth and CPU utilization numbers in the future, but the TCP implementation will require modifications in order to fully take advantage of our implementation. Specifically, any TCP implementation using our methodology will have to be using large-window extensions [2], and will have to be modified to correctly handle window-scaling, back-off and PathMTU discovery to take advantage of large datagram sizes. These modifications are beyond the scope of this report but should be interesting future work.

5 Results

Using our modified firmware, we were able to measure maximum UDP receive bandwidths comparable to the maximum UDP receive bandwidth attained under unmodified firmware included with the Linux 2.4 kernel driver (see Figures 1 and 2). For 1.5K MTUs our modified firmware reached bandwidths of over 800Mb/s and remained within 10% of the bandwidth demonstrated by the unmodified firmware. For 9K MTUs both implementations demonstrate peak bandwidths of 993 Mb/s. Our firmware matched the performance of the unmodified firmware until performance declined somewhat above 45K message sizes.

More importantly, our modified firmware provided a substantial reduction in CPU utilization over the unmodified firmware (see Figures 3 and 4). CPU utilization was reduced by approximately 45% for 1.5K MTU and approximately 30% for 9K MTU. Peak receive CPU utilization was reduced from 98% to 53% for 1.5K datagrams and from 88% to 75% for 9K datagrams. The unmodified firmware hit peak CPU utilization at a message size of 9200. Our

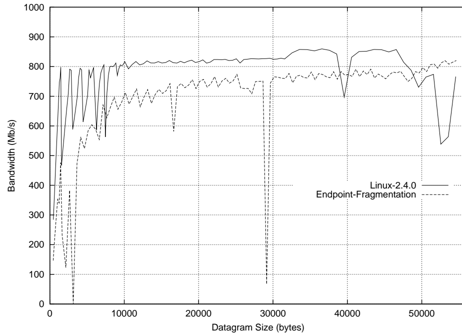

Figure 1. Bandwidth for UDP Receive with 1500 byte MTU

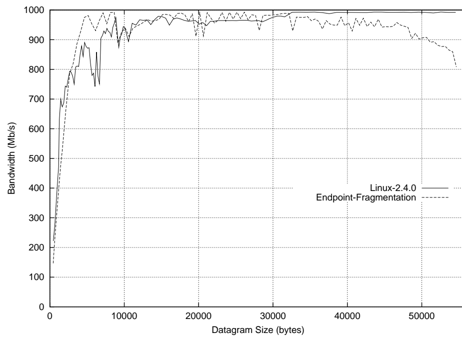

Figure 2. Bandwidth for UDP Receive with 9000 byte MTU

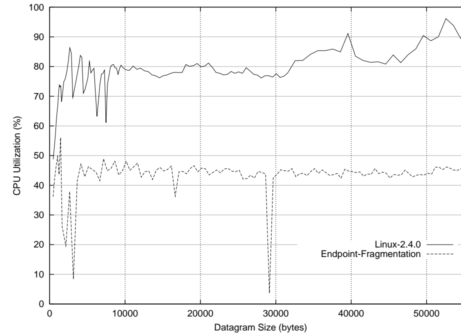

Figure 3. CPU Utilization for UDP Receive with 1500 byte MTU

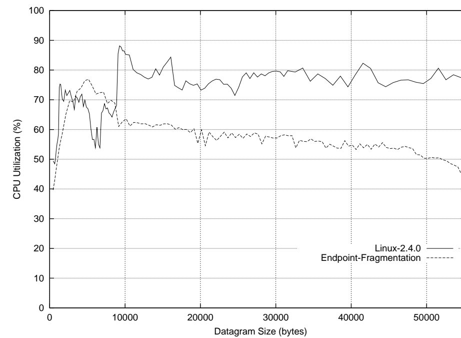

Figure 4. CPU Utilization for UDP Receive with 9000 byte MTU

firmware hit peak CPU utilization at a message size of approximately 5000. CPU utilization for our firmware declined to below 50% as message sizes increased. CPU utilization for the unmodified firmware remained between 70% and 80% as message sizes increased.

6 Analysis

6.1 Bandwidth

In examining Figure 1 there are two distinct inverted spikes in the endpoint fragmentation bandwidth results. We believe that these are caused by a bug in our send implementation. Notice that there is a corresponding inverted spike on the CPU utilization shown in Figure 4, which indicates that the receiver is idle at this point. These anomalies have been bounded within a 10-byte range and would be fixed in a robust implementation.

The bandwidth bottleneck in our implementation of the firmware is due to the wait for the completion of the first DMA. The first fragment contains the header for the entire datagram, which is used as the basis for the header for each of the following fragments. Both implementations of the firmware exhibit relatively unstable bandwidths until reaching message sizes of at least 9K. We believe that the difference between our endpoint fragmentation bandwidth and the unmodified firmware is due to our relative inexperience with the firmware and API. We noticed that very small changes in the firmware resulted in significant changes in performance. We will continue to consider ways to match performance for small message sizes without sacrificing scalability at larger message sizes.

As with the bandwidth for 1.5K MTUs, the bandwidth for 9K MTUs is unstable below 9K message sizes (see Figure 2). The only other interesting feature of the bandwidth curve for our endpoint fragmentation is the loss of performance when message sizes exceed 45K. This is probably the result of a resource limitation on the NIC or in the driver configuration, and might be avoided with either more memory available for receive buffers or more careful management of memory in our implementation. We are continuing to investigate this phenomenon.

6.2 Processor Overhead

The substantial decrease in CPU utilization provided by our implementation is very encouraging. For message sizes larger than 5K bytes, the CPU utilization for 1.5K MTUs is consistently below 50% (see Figure 3). Notice that the CPU utilization for the unmodified firmware continues increasing over the same range.

In examining Figure 4, notice that the CPU utilization for the fragmentation firmware at 9K MTUs declines steadily from over 60% at 10K message sizes to under 50% at 55K message sizes. This reduction corresponds to the slight decline in bandwidth that our implementation demonstrates over the same range. This appears to be a clear case of trading bandwidth for CPU capacity and does not represent and independent trend of reduced CPU utilization.

7 Recommendations

In order to seriously pursue further off-loading network processing from the host CPU, more powerful NICs are needed. In particular, substantially more RAM, faster processors and more sophisticated DMA engines. In our implementation, additional RAM could be used to to store a larger list of datagrams in the process of reassembly, as would be required for a robust implementation. More generally, significantly more RAM would be needed to consider storing the state required by TCP or any other connection-oriented protocol. (Note that TCP and other connection-oriented protocols still benefit from off-loading fragmentation and reassembly, checksumming and many other tasks to the NIC).

One of the most difficult aspects of developing efficient firmware implementation for the Acenic is ensuring that the MAC queue is always full and is never waiting on NIC processor computations. This task would be considerably easier with faster and more powerful processors. The work that needs to be done by the NIC CPU to implement fragmentation and reassembly is minimal. Nevertheless, computation had to be implemented with care in order to avoid starving the MAC queue. This problem would become untenable for any protocol requiring more processing.

Finally, as we off-load more sophisticated tasks onto NICs, we will need a richer environment for controlling transfers between host and NIC memory. Our implementation would have greatly benefitted from greater control of the DMA channel, including: control over start-up of the channel, completion notification, threshold control, and finer grained priorities for requests. Something similar to SCSI's tagged queuing would have been helpful in our implementation.

8 Future Work

8.1 Checksum

Our current implementation does not compute or verify checksums on the generated fragments (although it does preserve checksums on the original datagram). To be more specific, in all of our results, the operating system calculated checksums on each IP datagram, but in our firmware, we do not re-calculate checksums for each fragment or verify the correct checksum on the receiver. While this approach is reasonable for testing a prototype on a reliable network, calculating and validating checksums will be required for interoperability. Given the fact that our results indicate that our performance is sender-constrained, we expect it will be challenging to maintain the reported performance while computing checksums on all outgoing fragments.

8.2 Latency

Our implementation should be able to deliver highly predictable, lower latencies for small datagrams when compared to the interrupt coalescing strategy used in the unmodified firmware. We are measuring latencies to verify that our implementation meets these expectations.

8.3 TCP

The most important, but most challenging, area of future work involves TCP. We need to identify an API between the driver and the firmware to communicate window scaling and congestion avoidance behavior. Getting high TCP bandwidth with low CPU utilization would provide substantial validation of this approach to high-performance IP.

9 Acknowledgments

The authors wish to thank Dr. Arthur B. Maccabe for his guidance and Wenbin Zhu for her tremendous help in understanding the Acenic driver and forward-porting tracedump support. We also wish to thank the Albuquerque High Performance Computing Center for support and resources and the NCSA for partial support of this research under Grant No. 8-90521.

References

[1] P. Gilfeather and T. Underwood. Fragmentation made friendly. Web: 'http://www.cs.unm.edu/˜maccabe/SSL/Fragmentation/', Dec. 2000.

[2] V. Jacobson, R. Braden, and D. Borman. RFC 1323: TCP extensions for high performance, May 1992. Obsoletes RFC1072, RFC1185 [3, 4]. Status: PROPOSED STANDARD.

[3] V. Jacobson and R. T. Braden. RFC 1072: TCP extensions for long-delay paths, Oct. 1988. Obsoleted by RFC1323 [2]. Status: UNKNOWN.

[4] V. Jacobson, R. T. Braden, and L. Zhang. RFC 1185: TCP extension for high-speed paths, Oct. 1990. Obsoleted by RFC1323 [2]. Status: EXPERIMENTAL.

[5] C. A. Kent and J. C. Mogul. Fragmentation considered harmful. *WRL Technical Report 87/3*, Dec. 1987.