Experience in Offloading Protocol Processing to a Programmable NIC*

Arthur B. Maccabe, Wenbin Zhu
Computer Science Department
The University of New Mexico
Albuquerque, NM 87131

Jim Otto, Rolf Riesen
Scalable Computing Systems Department
Sandia National Laboratories
Org 9223, MS 1110
Albuquerque, NM 87185-1110

Abstract

Offloading protocol processing will become an important tool in supporting our efforts to deliver increasing bandwidth to applications. In this paper we describe our experience in offloading protocol processing to a programmable gigabit Ethernet network interface card. For our experiments, we selected a simple RTS/CTS (request to send/clear to send) protocol called RMPP (Reliable Message Passing Protocol). This protocol provides end-to-end flow control and full message retransmit in the case of a lost or corrupt packet. By carefully selecting parts of the protocol for offloading, we were able to improve the bandwidth delivered to MPI applications from approximately 280 Mb/s to approximately 700 Mb/s using standard, 1500 byte, Ethernet frames. Using "jumbo", 9000 byte, frames the bandwidth improves from approximately 425 Mb/s to 840 Mb/s. Moreover, we were able to show a significant increase in the availability of the host processor.

1 Introduction

As network transmission rates have increased, it has become increasingly difficult to deliver this increase to applications. Delivering this bandwidth to applications requires that several bottlenecks, including the speed of the I/O bus, memory copy rates, and processor capacity, be addressed. In this paper, we consider the bottleneck associated with processor capacity. In particular, we consider migrating part of the processing associated with communication protocols from the host processor to a programmable NIC (Network Interface Controller).

This offloading of protocol processing has two significant benefits. First, by removing the processor capacity bottleneck, it results in improved communication bandwidth. Second, by moving protocol processing to the NIC, we will improve the availability of the host processor for use by application programs.

While there are clear advantages associated with offloading protocol processing, we must be careful in how much work we offload to the NIC. Current NICs have fairly severe limits in both memory capacity and processing power. For example, the Alteon Acenics only have 2MB of local RAM and two 88 MHz processors.

The remainder of this paper is organized as follows. The next section explores the nature of the processor capacity bottleneck. In Section 3 we describe the protocol that provided the basis for our offloading experiments. Section 4 describes the steps that we undertook in offloading parts of this protocol. Section 5 describes the results of our offloading experiment. Section 6 discusses related work. Finally, Section 7 presents our conclusions and plans for further work.

2 The Processor Capacity Bottleneck

Given the instruction processing rates of modern processors, it may be hard to imagine that processor capacity could represent a significant bottleneck in any aspect of communication or computation. Moreover, considering the limited instruction rates of the processors available on NICs, it may seem contradictory that offloading any processing to a NIC processor will provide any significant benefit. Ultimately, the issue boils down to the costs associated with interrupts and traps on modern processors.

We are interested in performance as measured by MPI (the Message Passing Interface)[9] applications. Figure 1 shows the typical implementation of a communication protocol stack. Because the host processor is involved in every packet transmission, the strategy shown in Figure 1 implies a significant amount of overhead. In particular, the host processor will receive a large number of interrupts to handle incoming packets.

*This work was supported in part by Sandia National Laboratories under contract number AP-1739.

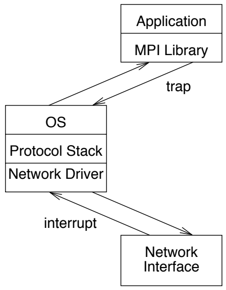

Figure 1. Typical Implementation of a Protocol Stack

If not carefully controlled, communication overhead can quickly dominate other concerns. To a large extent, communication overhead is driven by interrupt overhead, the processor time taken from the application to handle an interrupt. For current computing systems, we have measured interrupt overheads between five and ten microseconds.

To see the impact of interrupt overhead, consider the minimum inter-arrival time for 1500 byte frames for Gigabit Ethernet (ignoring the preamble, frame start delimiter, Ethernet header, checksum, and inter-frame gap):

$$1500\text{B} \times \frac{8\text{b}}{\text{B}} \times \frac{1}{1000\text{Mb/s}} = 12\mu\text{s}$$

Assuming that interrupt overheads are $8\mu s$, including packet processing, the communication overhead will be approximately 67% for Gigabit Ethernet!

Two approaches are commonly used to reduce communication overhead: jumbo frames and interrupt coalescing. Jumbo frames increase the frame size from 1500 bytes to 9000 bytes. This increases the minimum frame inter-arrival time from $12\mu s$ to $72\mu s$. Unfortunately, this only works for larger messages and will not be particularly helpful for 10 Gigabit Ethernet.

Interrupt coalescing holds interrupts (at the NIC) until a specified number of packets have arrived or a specified period of time has elapsed, which ever comes first. This reduces communication overhead by throttling interrupts associated with communication. When a constant stream of frames is arriving at a node, interrupt coalescing amortizes the overhead associated with an interrupt over a collection of frames. This approach will work well, with higher speed networks, but introduces a high degree of variability in the latency for small messages.

Our approach is to offload parts of the protocol processing onto the NIC. In particular, we offload message fragmentation and reassembly. We also avoid memory copies which further reduces processor overhead.

3 The RMPP Protocol

For our experimentation, we chose a simple RTS/CTS protocol called RMPP[11]. RMPP supports end-to-end flow control and message-level retransmission in the event of a dropped or corrupt packet. Figure 2 presents a graphical illustration of the packets used in this protocol. Because it is not important in the context of this paper, we will not discuss the fault tolerant aspects of this protocol. Instead, we will focus on the flow control properties.

The first step in sending a message is the transmission of an RTS (Request To Send) from the sender to the receiver. This packet includes an RMPP header (indicating that it is an RTS packet) and the initial data of the message. The packet will be filled with message data up to the MTU (Maximum Transmission Unit) of the underlying network. Importantly, the RTS packet should include headers for all upper level protocols, e.g., the header constructed by the MPI library.

When the destination receives an RTS, it can use the data in the packet to determine where the message should be delivered. For example, it could match the incoming message to a pre-posted MPI receive. Once the destination of the incoming message is known, the receiver can reply with a CTS (Clear To Send). This packet indicates the number of data packets that the sender is permitted to send before receiving the next CTS. Each CTS enables the transmission of n data packets and is used to control the flow of data from the sender. The actual value of n is at least 1 and will reflect the minimum of: the space required for 16 data packets, the space available in the application (this may reflect "pinned" pages), and the buffer space available on the NIC.

When the sender receives a CTS packet, it transmits the allowed number of data packets and, assuming the message contains more data, sends an RSM[1] Once the sender has transmitted all of the data packets, it sends an END, indicating that it is done sending the current message. The receiver acknowledges receipt of the message by sending an ACK. Upon receiving the ACK, the sender can reclaim any state related to the message transmission (the delivery is complete) and send a CLEANUP to the receiver. When it receives a CLEANUP, the receiver knows that the sender knows that the message has been delivered and the receiver can reclaim any resources associated with the message transmission.

In considering RMPP, notice that all management decisions are made by the receiver when handling an RTS or RSM. By the time the receiver has generated the CTS, it has decided: where the cleared data packets will be placed and the NIC buffer space that will be used for incoming packets. Once these decisions have been made, responses to all

[1] Like the initial RTS, each RSM is a flag in the header of the last data packet.

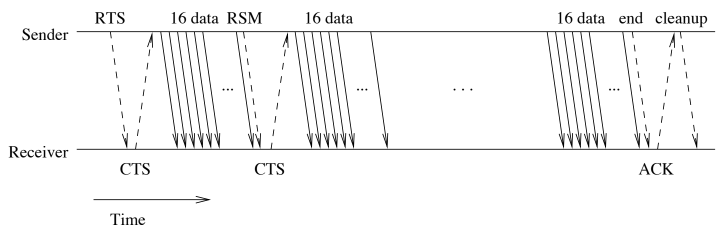

Figure 2. Message exchanges in the RMPP protocol

other packets (excluding the END, ACK, and CLEANUP) packets are straightforward. This observation lead to our approach for protocol offloading.

4 Steps in Protocol Offloading

RMPP was developed as a reliable transport layer for implementing the Portals 3.0 API[1] on Myrinet. Our first step was to make the protocol run on Gigabit Ethernet using the Alteon ACENICs. In the next two steps we offloaded the handling of data packets: first on the receiver, then on the sender. Finally, moved the RSM packets earlier in the stream of data packets to avoid bubbles in the pipeline of data packets.

4.1 Step 0: Porting RMPP to Gigabit Ethernet

Implementation of RMPP is based on two Linux kernel modules (the Portals module and the RMPP module) and the packet MCP (Myrinet Control Program). The packet MCP relays packets between the RMPP module and the Myrinet network and coalesces interrupts. The RMPP module schedules the use of the resources provided by the NIC and implements the RMPP protocol. The Portals module implements the Portals API which, in turn, provides an implementation of MPI[2]. From the perspective of the RMPP module, the Portals module provides the destination for each incoming message.

This decomposition is based on a separation between policy and mechanism. The Portals module provides the policy for message delivery while the RMPP module provides the mechanism needed to transmit messages. Similarly, the RMPP module provides the policy needed to control multiple flows, while the packet MCP provides packet transmission.

Before offloading parts of the RMPP protocol onto the Alteon ACENIC Gigabit Ethernet NIC, we decided to integrate the RMPP module into the Linux networking stack. In particular, we built a wrapper to translate between RMPP packets and *skbuf*s, the basic structure for Linux network traffic. This wrapper makes it possible to use the RMPP protocol with any of the Linux network device drivers, including the driver for the Alteon ACENIC. Figure 3 illustrates the structure of our initial implementation, including the *skbuf* wrapper.

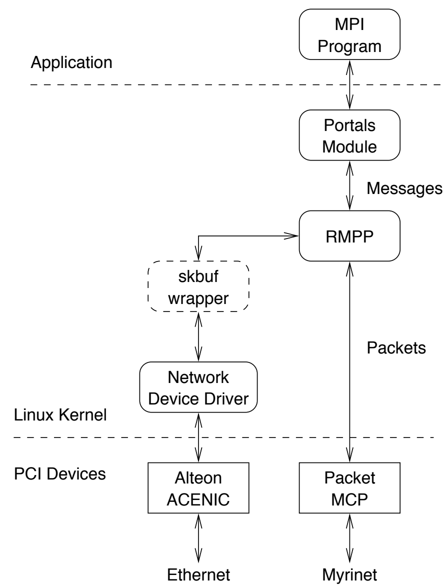

Figure 3. The Initial RMPP Implementation

4.2 Step 1: Receive Offloading

Next, we offloaded the processing of incoming data packets from the RMPP module to the NIC control program. Whenever the RMPP module generates a CTS packet, it also pushes physical addresses for the "cleared" memory region to the NIC. Now, when the cleared data packets arrive, they can be transferred directly to application memory. Figure 4 presents the resulting packet flow.

There are two advantages that should result from this modification. First, we have reduced the number of interrupts that the host processor will need to field. Second, we have reduced the number of copies for incoming data packets – data packets are delivered directly to the application with no intermediate copies. Moreover, we have not significantly increased the amount of work done by the NIC. As before, the NIC needs to schedule a DMA operation to

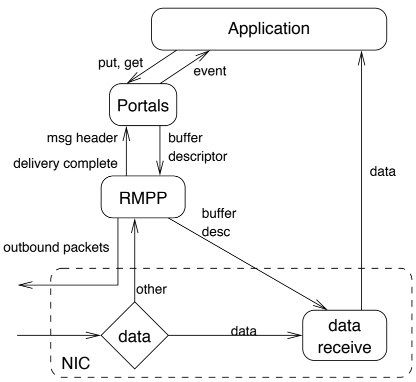

Figure 4. Receive Offloading

transfer the data packet from NIC memory to host memory. In the original implementation the target address was determined by the next entry in a queue of memory descriptors provided by the kernel. Now, the control program on the NIC can use a field in the incoming packet header to determine the host memory target address for this DMA operation. In essence, the NIC needs to be able to *demultiplex* multiple incoming data streams. In addition, the NIC needs to watch for dropped packets and notify the RMPP module whenever it detects an out of order packet. (The RMPP module uses a timeout to detect loss of the last packet in a group of cleared packets.) Neither of these activities should have a significant impact on the responsiveness of the NIC.

4.3 Step 2: Send Offloading

In the next step we offloaded the processing associated with sending messages from the RMPP module to the NIC control program. When an application program initiates a send, the RMPP module sends the initial RTS packet and pushes a descriptor for the remainder of the message buffer to the NIC. This descriptor includes the physical addresses the to retrieve the message from host memory. Given this information, the NIC control program can to respond to incoming CTS packets by building and sending the data packets that are cleared by the CTS. Moreover, the NIC control program can also generate RSM packets. Figure 5 illustrates the information and packet flow that for this modification.

The primary advantage in this step is avoiding the memory copy from the application space to the intermediate kernel buffers. In addition, we save the occasional interrupt that would be needed to process incoming CTS packets.

4.4 Step 3: Pipeline Management

In the previous implementations, the RSM is sent with the last of the cleared data packets. This creates a bubble in

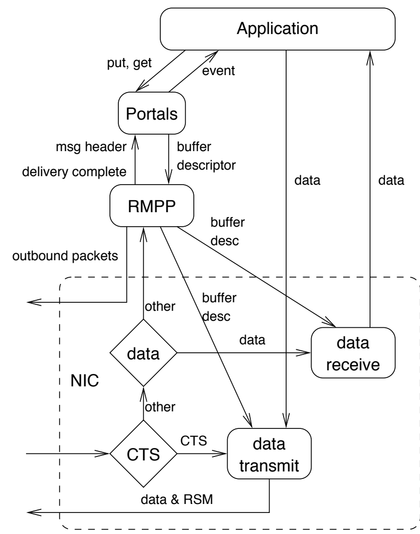

Figure 5. Sending from the NIC

the stream of data packets while the receiver processes the RSM, generates the CTS and sends it back to the sender. In the final step, we experimented with moving the RSM earlier in the stream to avoid these bubbles.

5 Results

We evaluate our modifications by measuring improvements to bandwidth and processor availability. All of the results reported in this section we obtained using two systems connected by a “crossover” Gigabit Ethernet cable. Each system has a 500 MHz Pentium III.

5.1 Bandwidth Results

We measure bandwidth using a traditional ping-pong test, using MPI for all communication. Pseudocode for this measurement is shown in Figure 6. In this test, the *ping* process sends a message to a *pong* process. The pong process replies by returning the original message. For each message size, the test reports the latency. Bandwidth is calculated by dividing the message size by the latency. The actual measurement that we use reports the minimum, maximum, and average latency.

Figure 7 presents bandwidth curves for the original implementation, an implementation that offloads receive handling, and an implementation that offloads both send and receive handling. As can be seen, each modification to the implementation results in a significant improvement in the observed bandwidth. Offloading the processing associated with incoming data packets results in a bandwidth increase from 35 MB/s to 50 MB/s, a 30% improvement. Additionally, offloading the processing associated with sending mes-

```
process ping {
  char buffer[MAXBUF];

  msize = MIN;
  while( msize ≤ MAXBUF ) {
    t1 = gettime();
    loop n times {
      send buffer;
      receive buffer;
    }
    latency = (gettime() − t1) / 2 ∗ n;
    report msize, latency;
    msize += INC;
  }
}
```

```
process pong {
  char buffer[MAXBUF];

  msize = MIN;
  while( msize ≤ MAXBUF ) {

    loop n times {
      receive buffer;
      send buffer;
    }


    msize += INC;
  }
}
```

Figure 6. Pseudocode for the Ping-Pong Test

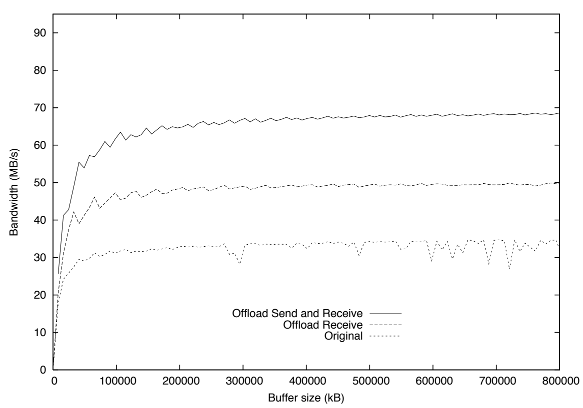

Figure 7. Bandwidth Improvement

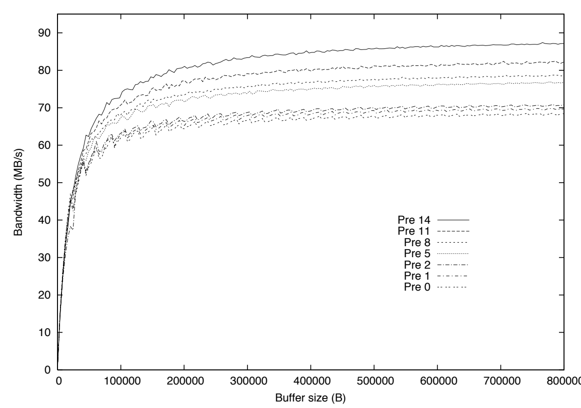

Figure 8. Early RSM

sages increases the bandwidth to 69 MB/s, an addition 28% improvement.

To gain insight into the improvement observed in Figure 7, we calculated the per packet latency. For the original implementation this value stabilizes at approximately $40\mu s$. When we offload receive processing, this values stabilizes at approximately $27.5\mu s$, an improvement of $12.5\mu s$ per packet. When we also offload the send processing, this value stabilizes at approximately $20\mu s$, an additional savings of $7.5\mu s$ per packet.

Because the only significant improvement associated with offloading send processing is the avoidance of a memory copy, we conclude that the cost of copying a packet from application space to kernel space is close to $7.5\mu s$. Moreover, we conclude that the additional savings ($5\mu s$) observed in offloading receive processing is due to the elimination of the per packet interrupt. That is, the elimination of the processor capacity bottleneck.

After offloading the processing associated with sending messages and receiving data packets, we measured the performance improvements when moving the RSM earlier in the stream. Figure 8 presents the bandwidth improvements associated with moving the RSM earlier in the stream of data packets. The "Pre 0" reflects the default case of transmitting the RSM in the last cleared data packet. The other graphs represent experiments where the RSM was sent 1, 2, 5, 8, 11, or 14 data packets before the end of set of cleared packets. Sending the RSM 15 packets before the last cleared packet increases the bandwidth from 69 MB/s to 88 MB/s, an improvement of nearly 22%.

If we assume that the "Pre 15" graph represents the best we can do by pre-sending the RSM, we can calculate the length of the bubble in the stream of data packets. As we noted earlier, offloading both the reception of data packets and the sending of messages results in a per packet latency of $20\mu s$. By sending the RSM 14 packets before the end of the cleared data packets reduces this to 15.5μs, a savings of 4.5μs per packet. Because we generate an RSM/CTS pair for every 16 packets, we calculate the bubble size as $16 * 4.5 = 72$ microseconds.

5.2 Processor Availability Results

Netperf [5] is commonly used to measure processor availability during communication. Netperf uses a two step process to measure processor availability. First, netperf measures the time taken to execute a delay loop while the node is quiescent. Then, it measures the time taken for the same delay loop while the node is involved in communication. The ratio between the first and second measurement provides the availability of the host processor during communication. In netperf, the code for the delay loop and the code used to drive the communication are run in two separate processes.

Netperf was developed to measure the performance of TCP/IP and UDP/IP and it works well in this environment. However, there are two problems with the netperf approach when applied to MPI programs. First, MPI environments typically assume that there will be a single process running on a node. As such, we should measure processor availability for a single MPI task while communication is progressing in the background (using non-blocking sends and receives). Second, and perhaps more important, the netperf approach assumes that the process driving the communication relinquishes the processor when it waits for an incoming message. In the case of netperf, this is accomplished using a *select* call. Unfortunately, many MPI implementations use OS-bypass. In these implementations, waiting is typically implemented using a polling loop. (This is reasonable, given the previous assumption that there is only one process running on the node.)

To avoid these problems, we modify the body of the outer loop for the *ping* process used to measure bandwidth. The new loop body is presented in Figure 9. On each iteration, the *ping* process posts an nonblocking send and a nonblocking receive. It then enters a loop in which it polls for completion of the receive. Inside of this loop, the *ping* process simulates work by delaying for an interval specified by the polling interval, *p*. Notice that the total amount of work to be done drives the measurement. To determine processor availability, we compare the time it takes to complete this code when the *pong* process sends reply messages versus the time when the *pong* process consumes the first message.

Figure 10 illustrates the relationship between bandwidth and processor availability for different message sizes. In examining the graphs in this figure, we first note that when we offload both send and receive the bandwidth and processor availability are both relatively high, 61MB/s[2] and 92%, respectively. When we only offload the receive processing, the bandwidth is significantly higher than it is for the original implementation (38 MB/s versus 12 MB/s), but the host processor availability is significantly lower (72% versus 58%). In this case, it is difficult to know if we are effectively trading bandwidth for availability.

```
t1 = gettime();
repeat {
   isend buffer;
   ireceive buffer;
   repeat
      loop p times work−−;
   until receive done or work ≤ 0;
   n++;
until work ≤ 0;
duration = gettime() − t1;
report duration, n;
```

Figure 9. Measuring Processor Availability

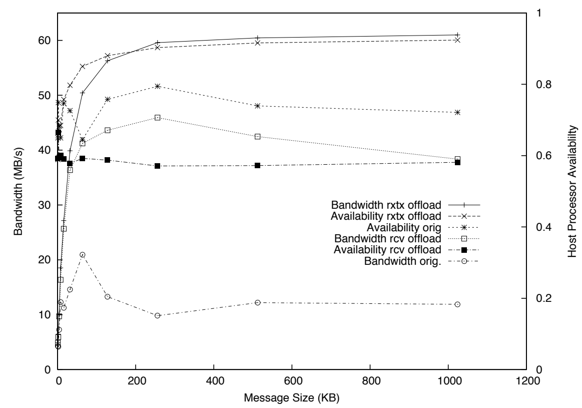

Figure 10. Bandwidth and Availability

Given the tradeoff between processor availability and bandwidth, it is often difficult to characterize performance improvements. It is frequently helpful to consider the *effective bandwidth*, the product of the availability and the bandwidth. Figure 11 presents effective bandwidth graphs. Here, the improvement becomes much more apparent. When we offload receive processing, the effective bandwidth improves from 8.64 MB/s to 22.3 MB/s, a factor of 2.6. When we offload both send and receive processing, the effective bandwidth goes to 56 MB/s, a factor of 6.5 improvement.

As a final experiment, we decided to compare offloading of send and receive processing with coalesced interrupts and jumbo frames. The results of this comparison are shown in Figure 12. In this case, we no improvement from interrupt coalescing, some improvement for jumbo frames, and a sig-

[2] The slightly lower bandwidth is due to the work loop.

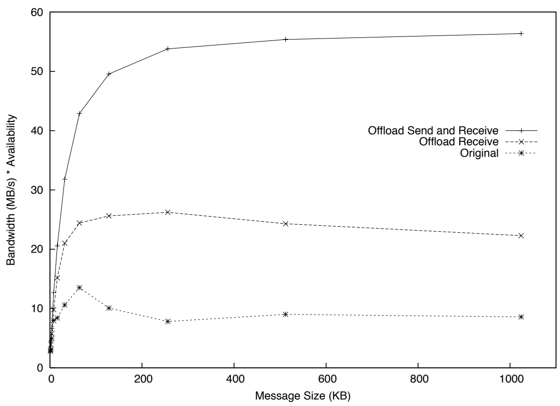

Figure 11. Effective Bandwidth

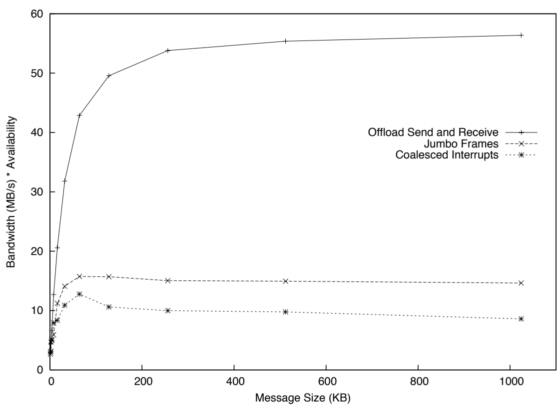

Figure 12. Comparing Effective Bandwidths

nificant improvement for offloading.

6 Related Work

Earlier work in the Scalable Systems Lab at UNM [4, 14] showed that offloading fragmentation and defragmentation for IP (Internet Protocol) packets could result in a significant improvement in communication bandwidth and host processor availability. Like the work reported in this paper, IP fragmentation studies were based on the Alteon Acenic.

To avoid the problems associated with memory copies, communication latency, and communication overhead, several groups have proposed protocols that support "OS bypass" [3, 13, 7, 10]. In concept, each process gets its own virtual network interface which it can use directly (bypassing the need to go through the operating system). Needless to say, these proposals do not entirely bypass the OS. They rely on interaction with the OS to enforce resource protection (usually limited to address translation) or to ensure that resources (page frames) are available. In the OS bypass strategy, memory copies are avoided by using buffers in the application memory rather than buffers in the operating system (as such, the application is in control of when copies need to be made). Communication latency is reduced by avoiding the need to trap into OS for sends and interrupt the OS for receives. Communication overhead is also reduced by the elimination of interrupts.

While we borrow many ideas from OS bypass, we use the OS to control the use of network resources and to match incoming messages with their final destination. When MPI applications try to overlap communication with computation, having a mechanism to match incoming messages with pre-posted receives offers significant performance advantages [8]. Providing this mechanism requires some intervention by the OS, either as we have done in processing the RTS or in providing scheduling for multiple processes or threads.

The EMP [12] project at the Ohio Supercomputer Center takes offloading one step further and provides matching for MPI messages on the NIC. Using similar hardware (Alteon Acenic NICs and relatively slow Pentium processors), they report impressive performance numbers: 880 Mb/s bandwidth and 23 μs latency. In contrast to the EMP approach, we have started from the perspective that the OS should manage the resources provided by the NIC. This includes buffer space, information about messages in transit, and flow control. EMP starts with all resource management on the NIC or in the application. We are currently in the process of migrating more of this management into the NIC and the EMP project is looking into providing some NIC management in the OS.

7 Conclusions and Future Work

Through our experiments, we have shown that protocol offloading is a viable approach to reducing the processor capacity bottleneck and improving communication performance. So far, we have only offloaded the handling associated with data packets. All control packets (with the exception of CTS packets) are handled by the OS and, in particular, the RMPP and Portals modules. When we started this work, we expected to offload all of RMPP and Portals to the NIC. And, while we plan to continue selectively offloading more of the protocol processing, this represents an interesting milestone in our efforts.

While programmable NICs are a great asset in this kind of study, they are never price-competitive with NICs that do not support a programming interface. Because the processing that we have offloaded is very straightforward and the resource management is done by the OS, one could easily imagine the development of NICs that incorporate only this level of processing. In this context, it is worth noting that the Portals module simply provides an answer to the question of where an incoming message should be placed. While we have focused on MPI-based message passing, it is interesting to consider other ways that this question might be answered. One obvious possibility is to have the OS match the incoming data with the application buffer provided in a socket read operation. Additionally increasing the MTU to a reasonable size would provide a simple, true zero-copy TCP capability that has been so elusive [6].

We will also examine other parts of the protocol processing that could be offloaded to the NIC. While it might be natural to assume that we would next offload the retransmit logic to the NIC (as is done in EMP), we plan to leave this in part of the protocol processing in the OS. First, because we expect that dropped packets will be infrequent, we would prefer to keep this part of the protocol processing out of the fast path whenever as possible. Second, this part of the protocol processing is most closely related to resource management and naturally belongs in the OS.

Rather than offloading the retransmission logic, we will explore ways in which the matching of incoming messages to pre-posted receives can be offloaded to the NIC. The goal is to identify a general purpose matching strategy that could be used for a variety of upper level protocols and implement this on the NIC. If we can provide the NIC with enough information to match an incoming message with a pre-posted receive, the NIC will be able to: 1) notify the local OS that it has initiated a receive, 2) generate and send a CTS to enable the flow of more data packets, and 3) transfer the initial and subsequent data packets directly to the application. This should result in a significant reduction in latency for small messages.

Acknowledgements

This work would not have been possible without support from a variety of places. Much of the UNM effort was funded by a contract from Sandia National Laboratories. The Alteon Acenic NICs were borrowed from the Albuquerque High Performance Computing Center and were acquired as part of an IBM SUR (Sponsored University Research) grant.

We also had a great deal of support from our colleagues in the Scalable Systems Lab at UNM. Patricia Gilfeather and Todd Underwood were very helpful in our initial efforts to get code running on the Acenics. Bill Lawry and Riley Wilson provided the benchmarking tool used to obtain the processor availability results.

Finally, Pete Wyckoff from the Ohio Supercomputer Center was very helpful in providing debugging support and guidance in subtle programming issues related to the Acenics. Pete also read an early draft of this paper and offered many helpful suggestions

References

[1] R. Brightwell, T. Hudson, R. Riesen, and A. B. Maccabe. The Portals 3.0 message passing interface. Technical Report SAND99-2959, Sandia National Laboratories, December 1999.

[2] R. Brightwell, A. B. Maccabe, and R. Riesen. Design and implementation of MPI on ortals 3.0. In *Lecture Notes in Computer Science Proceedings of the Euro-PVM/MPI Conference*, September 2002.

[3] Compaq, Microsoft, and Intel. Virtual interface architecture specification version 1.0. Technical report, Compaq, Microsoft, and Intel, December 1997.

[4] P. Gilfeather and T. Underwood. Fragmentation and high performance IP. In *CAC Workshop*, April 2001.

[5] R. Jones. The network performance home page. http://www.netperf.org/netperf/NetperfPage.html.

[6] C. Kurmann, M. Müller, F. Rauch, and T. M. Stricker. Speculative defragmentation–a technique to improve the communication in software efficiency for gigabit ethernet. In *Proceedings of the 9th International Symposium on High Performance Distributed Computing (HPDC)*, August 2000.

[7] M. Lauria, S. Pakin, and A. Chien. Efficient layering for high speed communication: Fast messages 2.x. In *Proceedings of the IEEE International Symposium on High Performance Distributed Computing*, 1998.

[8] A. B. Maccabe, W. Lawry, C. Wilson, and R. Riesen. Distributing application and OS functionality to improve application performance. Technical Report TR-CS-2002-11, Computer Science Department, The University of New Mexico, April 2002.

[9] Message Passing Interface Forum. MPI: A message-passing interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8, 1994.

[10] Myricom, Inc. The GM message passing system. Technical report, Myricom, Inc., 1997.

[11] R. Riesen. *Message-Based, Error-Correcting Protocols for Scalable High-Performance Networks*. PhD thesis, The University of New Mexico, Computer Science Department, Albuquerque, NM 87131, 2002.

[12] P. Shivam, P. Wyckoff, and D. Panda. EMP: Zero-copy OS-bypass NIC-driven gigabit Ethernet message passing. In *Supercomputing*, November 2001.

[13] Task Group of Technical Committee T11. Information technology - scheduled transfer protocol - working draft 2.0. Technical report, Accredited Standards Committee NCITS, July 1998.

[14] T. Underwood. Fragmentation as a strategy for high-speed IP networking. Master's thesis, The University of New Mexico, Computer Science Department, Albuquerque, NM 87131, 2001.

[15] W. Zhu. OS bypass investigation and experimentation. Master's thesis, The University of New Mexico, Computer Science Department, Albuquerque, NM 87131, 2002.