Identifying the Sources of Latency in a Splintered Protocol[*]

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

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

Abstract

Communication overhead and latency are critical factors for application performance in cluster computing based on commodity hardware. We propose a general strategy, splintering, to improve communication performance. In the splintering strategy, previously centralized functionality is broken into pieces, and the pieces are distributed among the processors in a system, in such a way that ensures system integrity and improves performance.

In a previous paper we demonstrated the benefits of using splintering to reduce communication overhead. In this paper, we describe our efforts to use splintering to reduce communication latency. To date, our efforts have not resulted in the improvement that we originally anticipated. In order to identify the sources of latency, we have done a thorough instrumentation of our implementation. Based on our analysis of our measurements, we propose several modifications to the MPI library and the NIC firmware.

Keywords: Communication latency, splintering, performance measurement

1. Introduction

Protocol splintering is the process by which the functions in a protocol stack are broken into small pieces which are then redistributed among the processors in a system to improve performance while ensuring system integrity. In particular, we consider distribution among the host processor and the processor in a programmable NIC, an arrangement that is common in cluster computing systems [3]. In an earlier paper, we reported our success in using splintering to reduce communication overhead. Using splintering, we were able to reduce the host processor utilization for large messages by 80% while maintaining high bandwidth [7].

[*]Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

While communication overhead is commonly cited as the most critical factor affecting application scalability [5, 6], communication latency is also an important performance bottleneck for many applications. In this paper, we report our initial experiences in using splintering to reduce communication latency.

1.1. Splintering

Splintering allows incremental adjustment to system workload distribution. Each previously centralized function is broken into pieces, and these pieces are distributed to ensure both system integrity and improved performance.

Splintering is closely related to OS-bypass and protocol offloading. However, there are important differences between these approaches. In contrast to OS-bypass, splintering seeks to retain the OS as the central coordinator of resources. Rather than trying to bypass the OS, we are careful to engage the OS as needed. In contrast to protocol offloading, we do not attempt to break functionality at protocol boundaries. Instead, we offload bits and pieces of different protocols and re-combine them to improve performance.

While we are currently investigating splintering in the context of programmable network interface cards, splintering could be used in many more contexts. Functionality that does not endanger a system's safety nor violate a system's sharing policy can be splintered and distributed to other parts of the system. The target of a splintered activity could be another processor in a SMP system, another computer in a distributed system, or to a hardware component if the system has intelligent hardware components. The distribution of splinters can be static (e.g., during system configuration) or dynamic. The principles of the splintering strategy are:

- Distributing operating system functionality to improve performance should be selective, based on system architecture as well as system and application software characteristics.
- Improvements in performance must not compromise

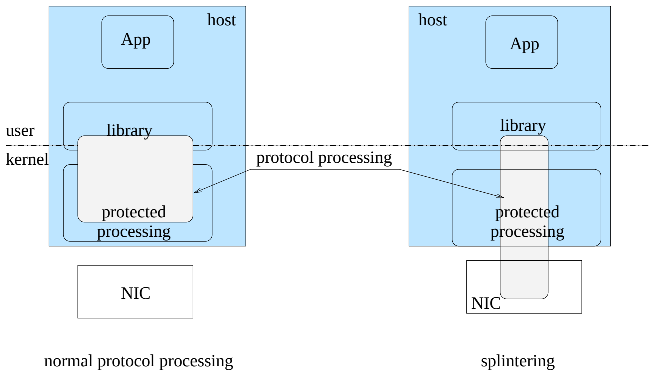

Figure 1. Applying splintering to network protocol processing

system integrity. Control of resources should remain centralized or carefully delegated (as splinters are distributed) among the processors available in a system.

- When splinters are dynamically distributed, the operating system is in control of splinter distribution. The operating system decides whether to splinter functionality based on the capabilities of the system components, and the overall performance of the system. Similarly, when splintered functionality is no longer needed in the system or splintering does not yield better performance, the operating system can gather the distributed splinters back to centralized processing again.

Figure 1 presents a graphical representation of a splintered protocol. In the initial implementation, all levels of the protocol are implemented on the host processor – parts of the implementation are split between the OS address space and the user address space. When the protocol is splintered, parts of the selected layers are implemented on the network interface card (NIC)hardware.

1.2. Previous Work

Using splintering, we were able to greatly reduce the host processor utilization [7]. We use host processor availability as our measurement for the host processor utilization. The host processor availability is defined as the ratio of available host processor cycles with communication processing versus the one that without. In our step-by-step splintering implementation, we first offloaded the receive-side data movement handling to a programmable NIC. which results in a $2X$ improvement in the CPU availability. In the second step, we offloaded the send-side data movement handling, which results in another $2X$ improvement.

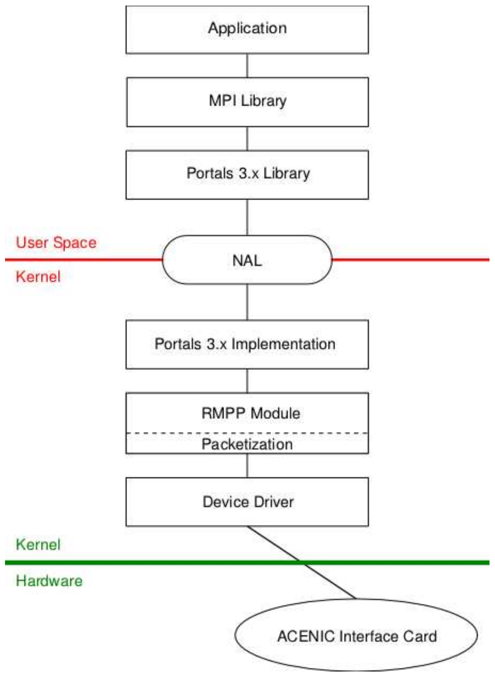

Figure 2. Conceptual view of the network protocol stack

Encouraged by our earlier implementation, we decided to use splintering to reduce the communication latency. In this paper we describe how our effort did not yield the expected result and how we have identified the obstacles in the protocol processing that have prevented us from attaining our goal.

The remainder of this paper is organized as follows. The next section explains the protocol stack we use to do the splintering implementation, and how will we use splintering to reduce communication latency in that protocol stack. Section 3 shows the initial results, and Section 4 introduces the difficulties as well as our solutions to do the measurement, in order to identify the time-consuming components. Section 5 shows the measurement results and Section 6 gives the analysis of the measurement results. In Section 7 we discuss the difference between our work and related work. Section 8 suggests the efforts we should put on to reduce the communication latency as our future work.

2. The Network Protocol Stack

In this section we describe the software layers that a message traverses on its way to and from a remote node. Figure 2 shows a graphical representation.

The application in our case is our simple micro benchmark. It is linked to the MPI library and uses basic MPI functions to send and receive short messages. The MPI library is MPICH version 1.2.0 and uses the Portals 3 API calls to transmit messages.

Portals 3 [4] is a low-level message passing mechanism, ideally suited to the implementation of message passing libraries, such as MPI, and runtime libraries. Our implementation of Portals 3 straddles the user space and kernel boundary. A thin user space library forwards the Portals 3 API calls into the Portals 3 implementation which resides inside the kernel. The Network Adaption Layer (NAL) is the mechanism our implementation uses to transport information between the two protection domains.

Most of the semantics of Portals 3 are implemented as a Linux kernel module. This module sends messages by calling the Reliable Message Passing Protocol (RMPP) [9] module below it. When new messages arrive, the RMPP module calls back into the Portals 3 module, which performs the matching and then informs the RMPP module where, in user space, to deposit the payload of the message.

The RMPP module implements a very simple protocol that is based on the assumption that network errors are rare and messages should be sent optimistically. Error detection and recovery takes a little longer, but should be rare in a good network. The RMPP module consists of two logical components. The protocol engine itself and the packetization functions that deal with the handling of individual packets.

Packets flow from the RMPP module to the Acenic kernel-level device driver and from there into the NIC itself. Incoming packet is transferred from the NIC through interrupt to the device driver, then it is passed into the RMPP module and processed by the Portals 3 module. This is not true for data packets which flow directly from and to user space and the NIC.

As we analyze the protocol stack, message matching looks like a good candidate to splinter to a programmable NIC. The matching information is pre-posted by the receive process, as shown in step 1 in Figure 3. Therefore it can be put into the NIC before a message arrives. In step 2, the send process calls into the Portals 3 module to create a buffer descriptor. This modules validates and pins the send buffer, returning a token for future reference. This token is used in a subsequent Portals 3 put operation shown as step 3 in Figure 3. When the NIC gets the notification from its driver, it first DMAs the buffer descriptor (bd) from the host in step 4, then it DMAs the data to be sent out in step 5. In step 6, the packet arrives at the receive NIC, which checks the packet's match information. On a successful match, the data is DMAed to the receiver's buffer, which is labeled as step 7. Then it updates the receiver's event queue in step 8 and acknowledges the sender in step 9. Finally, the packet header is sent to the operating system through an interrupt in step 10.

Using our splintering approach, we have moved some

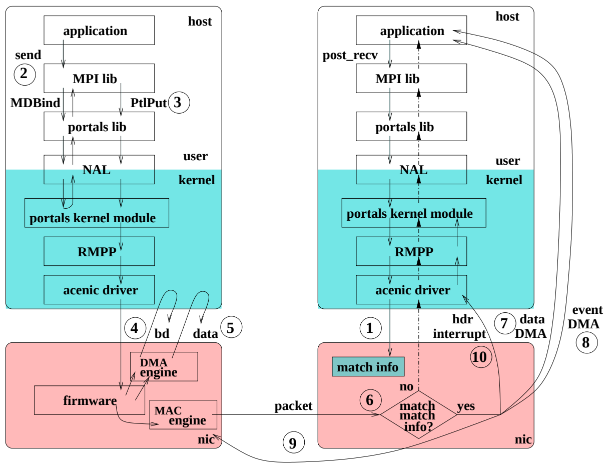

Figure 3. Splintering incoming packet matching to NIC

of the matching usually performed by the Portals 3 module into the NIC itself, so we can perform our evaluation of this approach. Acknowledgments, usually performed by the RMPP module, are also done in the NIC now.

3. Initial Results

We use 20-byte application level messages to do the measurements. Message size can be as short as zero bytes, but the measurements would not show the overhead associated with buffer management. A message of 20 bytes is long enough for the application to do some event notification between its peers, yet short enough to ignore the propagation latency. The latency of a message is composed of the protocol stack overhead, buffer management overhead and latencies that are imposed by the system.

Our splintered implementation was done in three steps. In the first step, the NIC DMAs the data into the application buffer. All other processing is still left to the host processor. The result is labeled *DMAdata* in Figure 4. In the second step, the NIC generates the acknowledgment to the sender, this result has the label *DMAdata_ackSender*. In the third step, the NIC updates the event queue to notify the receiving process. This result is labeled *DMAdata_ackSender_updateEQ*.

An interesting observation is that generating an acknowledgment on the NIC increases the latency, rather than reducing it. Our measurements show that handling send acknowledgments on the NIC takes about 8μs, while doing it on the host reduces this to approximately 3μs.

The final step of updating the event queue entry on the NIC has very good performance. It lowers the latency by about 20μs.

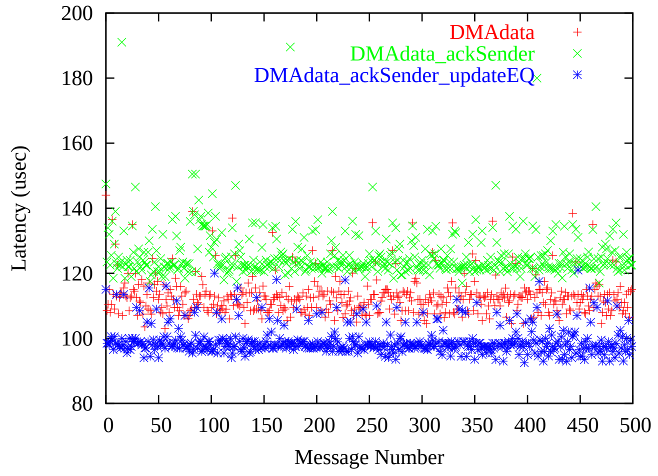

Figure 4. Latency results of three steps of splintering

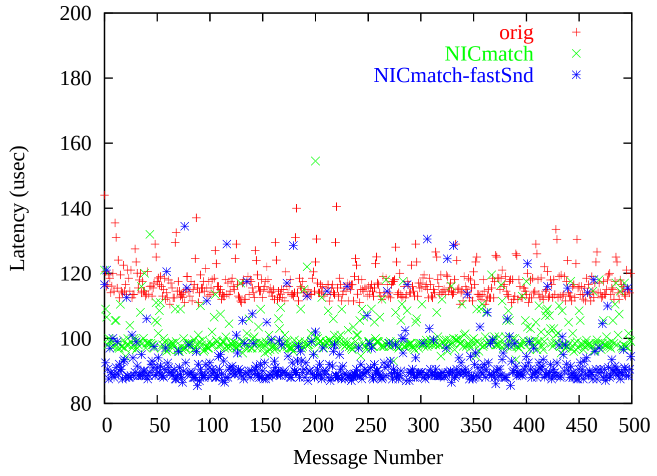

Figure 5. The latency difference between original versus splintered

At the time we finished this work, we noticed that our sends were still using the standard Linux kernel skbuffs, which are dispatched by the kernel network stack. Since our system is a dedicated system, and each node runs only one application, it is reasonable to dispatch to the device driver directly. This optimization gave us another 8μs reduction in latency.

Figure 5 shows the latency difference between the original implementation versus the final splintered implementation. *NICmatch* refers to when the NIC is doing the matching, acknowledging the sender and updating the event queue. *NICmatch-fstSnd* refers to when sending bypasses the kernel standard network stack. The original implementation has a latency of 110.5μs, and the splintered NICmatch implementation has a latency of 94.5μs. The final NICmatch with fast sending has a latency of 86μs. Splintering of the protocol results in a 22% reduction in latency.

The resulting 86μs is still significantly higher than our expectation of 50μs. We suspect that the protocol has some large overhead components. In order to identify these, we did a thorough measurement of the protocol processing using short messages.

4. Measurement Techniques

Doing the measurement is not quite easy. First, it involves many components, crossing multiple address spaces. Second, the whole protocol stack crosses different hosts, and different processors within a node. As a result, some of the intervals we are interested in measuring span multiple clocks. In this section we present our measurement goals as well as the techniques we used.

4.1. Timing goals

Following are the components in the protocol stack that we planned to measure the time spent on:

- The MPI library.
- The reference library (LIB).
- The Network Abstraction Layer (NAL).
- The device driver for the Alteon Acenic.
- The PCI bus.
- The firmware running on the NIC.
- The hardware latency, including the latencies of the DMA engines, the MAC engines, and the physical wire.
- Interrupt overhead.

Although the firmware controls the DMA engines to read and write to the host memory, we count its latency as hardware latency. The reason is that DMA engine start time and transfer time are totally determined by the hardware. Even though coordinating DMA and MAC engine to work well to obtain good performance is controlled by the software, we wish to isolate the DMA and MAC latency so as to be able to trade off the hardware and software cost. The higher the DMA latency, the more software effort should be put on to merge multiple DMAs.

4.2. Four clocks

The hardest part of doing the measurement is that it involves four clocks. One clock on the send host processor, one on the send NIC processor, and symmetrically, two clocks on the receive host and NIC. The difficulty comes when trying to measure a time interval that starts on one clock and ends on another. In order to eliminate the consideration for synchronizing different clocks, we always start and stop the timer on the same clock. By doing the measurements on selected intervals, we still be able to get accurate or approximate results on all time intervals.

We used two general approaches to address the problems posted by multiple clocks: ping-pong and deduction from multiple measurements.

4.2.1 Using Ping-Pong

Using ping-pong test, we always start and stop the timer on the same clock. Using this technique, we obtained PCI bus latency, read DMA latency on the NIC, write DMA latency on the NIC, one-way latency of a message going through two hosts, and one-way latency of the MAC engine and physical wire.

As an example, to measure the PCI bus latency, we did a ping-pong test between the device driver and the NIC. The timer starts when the driver updates a PCI register to signal the NIC. When the NIC receives the signal, it updates another shared PCI register immediately. The timer stops when the driver notices the change on that PCI register. Note that we assume PCI read and PCI write are symmetric.

4.2.2 Using Multiple Measurements

Some time intervals do not have an explicit start point or end point defined by a single clock. Therefore it is impossible to do the measurement directly on that interval by a single clock. The measurements of the read DMA latency, the write DMA latency, and the interrupt overhead fall into this category.

Interrupt overhead is the CPU cycles taken away by a single interrupt. The interrupt is initiated on the NIC, but is handled on the host. On the NIC, we do not know the end time, and on the host, we cannot get the start time. We measured the interrupt overhead by doing multiple measurements: MPI receive handling without interrupt, MPI send handling without interrupt, and MPI receive and send handling with interrupt.

We measured the time intervals of MPI library send handling as (12μs), receive handling as (2μs). We also measured the interval from the start of a receive until a new send starts in the MPI library on the pong node. It turned out to be 23μs instead of $12(sending) + 2(receiving) = 14\mu s$. The 9$\mu$s difference is the interrupt handling overhead, which

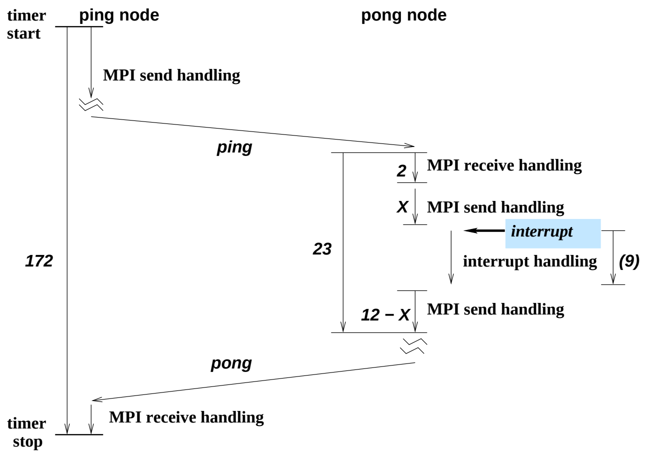

Figure 6. The interrupt overhead (time unit μs)

is counted into the whole message latency. The interrupt comes in sometime before the host finished the receive or send processing, so it is interrupted. As we measure the individual latency of the receive and send handling in the MPI library, we always take the shortest number, which does not include the interrupt overhead. This is shown in Figure 6.

Another example is the measurement of read and write DMA on the NIC. These time intervals also span multiple clocks, with start point and end point on different clocks, host processor clock and NIC clock in this case. The read DMA latency is measured using two ping-pong tests between two NICs. In one ping-pong test, each NIC just receives and echoes back a local packet. The measured interval includes the latencies of the MAC engine, the physical wire and firmware latency of send and receive processing. In the other ping-pong test, each NIC does a read DMA and sends that data to the other NIC. The time difference of the two ping-pong tests is the latency of two read DMAs.

Write DMA is measured in the similar way, it uses two ping-pong tests between the host and the NIC. It is always the host that does the timing. In one ping-pong test, the host signals the NIC, and the NIC simply interrupts the host. The time interval includes the PCI write latency, the interrupt overhead and the firmware latency to do a processing on the PCI signal. In the other test, the host signals the NIC, the NIC does a write DMA to the host memory, then interrupts the host. The time difference is the latency of one write DMA.

An unsolved problem is the overlap that happens in the protocol processing. We know that the DMA engines and the MAC engines on the NIC do work during the NIC processor is doing other work. Our individual measurement only shows how much latency for each stand-alone component, we have not found a good technique to analyze the overlap.

4.3. The timing tools

We chose the Intel 64-bit cycle counter as our timing tool on the host side. The cycle counter is incremented every clock cycle, and the measurement overhead is 70 cycles on a 500 MHz machine, which is 0.14μs. Therefore, it is a reasonably accurate timing tool. Another reason we choose the cycle counter is that our measurement goes across all address spaces, including application, different kernel modules, and the core kernel. The cycle counter is address space neutral, so the results obtained from various parts of the system would be fair to compare.

On the NIC, we use the timer register on the NIC, which is updated every microsecond, so each measurement on the NIC has an error of $\pm 1\mu$s. We reduce the error by doing multiple measurements.

5. Measurement Results

In this section, we summerize our measurement results using the techniques we described in Section 4. We split the result into two categories: host-side and NIC-side. The host-side latency refers to latencies of the host side processing, as well as the PCI bus latency and the interrupt overhead. The NIC-side latency includes firmware latency, DMA latencies, and transfer latency.

5.1. Host-side Latency Measurement

Figure 7 shows the host-side latency. The measured result of the round trip time of a ping-pong test is 172μs. Therefore our one-way latency is 86μs. The numbers followed by RTT is the round trip time, because those time intervals go across different clocks.

MPI library uses 12μs for send, and 2μs for receive. The Portals 3 library consumes 2μs for send processing. RMPP uses 10μs to do send processing. And the device driver uses 2μs for building the buffer descriptor and translating a physical address to a bus address. Interrupt overhead, as mentioned before, is 9μs. PCI bus latency is 1.5μs.

The total host-side latency is 38.5μs, therefore the NIC-side latency can be counted up to 47.5μs.

5.1.1 NIC-side Latency Measurement

Figure 8 shows the latency measurement on the NIC-side. The numbers in parentheses are derived numbers, using techniques described in Section 4.

There are two read DMAs on the send path, one for the buffer descriptor, the other one is for data. Each individual read DMA takes 3.5μs, so we count total read DMA latency as 7μs. Firmware latency on send is 10.5μs. On receive, the firmware latency is 20μs. The total latency for the send MAC engine to start putting data on the wire, till the data goes across the wire, and received by the receive MAC engine is 2μs. We have 8μs left, which is counted as write DMA latency on the receive side.

6. Analysis of the Measurement Results

Figure 9 shows the summary of the measurement, the numbers in parentheses are derived numbers. Counting the latency as three components: host processing latency, hardware latency, and NIC processing latency, the results can be broken to:

- Host processing latency, 28μs. This includes time used by the MPI library, Portals 3 library, RMPP module and device driver. We can do optimization in this processing path to reduce this latency.

- Hardware latency, 27.5μs. This includes interrupt overhead, read DMA, write DMA, MAC engine, wire, and PCI bus latency. We cannot change each individual latency, but we can eliminate some of them to lower the total latency. For example, we can remove the interrupt, and one of the two read DMAs.

- NIC processing latency, 30.5μs. This involves send and receive processing of the firmware running on the NIC. To reduce this latency, we should simplify the firmware processing, especially for receiving.

The only dubious place is the calculation only leaves 8μs for the three write DMAs on the receiving processing on the NIC. Our measurement shows that one write DMA consumes 10μs. The inconsistency comes from several places:

- The timer on the NIC is updated every microsecond, this gives us an error rate of $\pm 1\mu$s for each measurement. Even though we did multiple measurements for each timing interval, we still cannot guarantee up to one microsecond accuracy.

- There are possible overlaps between sending MAC and read DMA, and between write DMA and interrupt handling. Therefore, send-side read DMAs may be lowered less than 7μs, so receive-side write DMAs can be accounted for longer time.

The shaded areas are the large time consumers. The first one is the MPI library. In the current implementation, it takes 12μs to send and 2μs to notify the receiver. The other one is the firmware. It takes 10.5μs to process a send and 20μs to process a receive. The sending processing involves two DMAs from the host memory, which is merged to one in the EMP [10] implementation. The receiving processing incurs three write DMAs to the host memory.

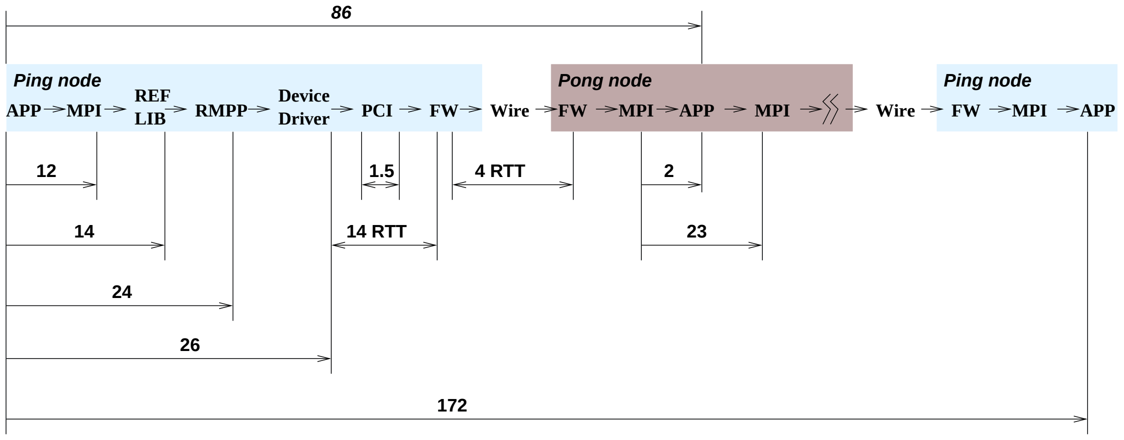

Figure 7. The host-side latency (time unit μs)

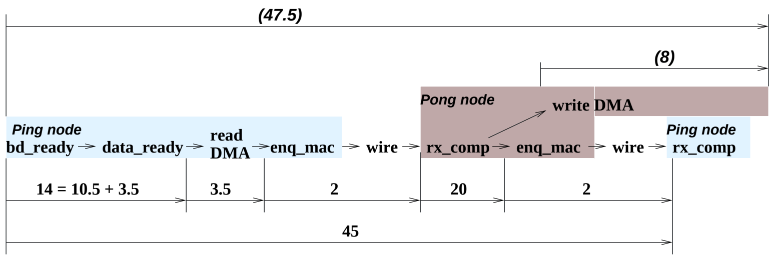

Figure 8. The NIC-side latency (time unit μs)

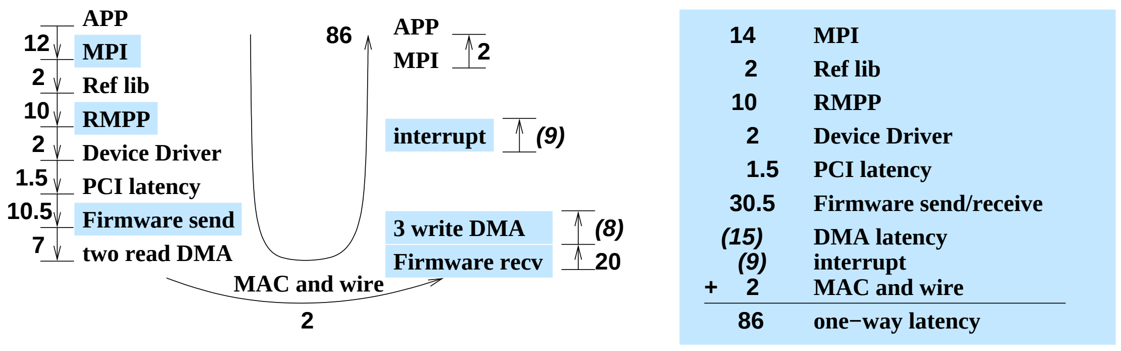

Figure 9. One-way latency layout (time unit μs)

7. Related Work

There have been a great deal of research in message passing performance measurement. Some have been done at the message level, where the message level latency and bandwidth are measured. The methodology is ping-pong test in most cases [2, 11, 13, 12].

Anderson et al., from DEC did a good job of detailed profiling of the whole system [1]. Pratt from NASA developed an infrastructure, GODIVA [8], for instrumentation of the source code, which is similar to our approach. However, both systems use linear measurement, where instrumention code is inserted into the execution path. It is not possible to obtain information of host-NIC interaction, or host-host interaction.

The EMP [10] uses the same programmable NIC as we do. Our measurement showed that EMP has a 28μs latency on our machines with MPI message length of 20 bytes.

In contrast to previous research, the novelty of our approach is the instrumentation on both the host and the NIC to obtain information of host-host and host-NIC interactions. This information includes: PCI bus latency, interrupt overhead, firmware latency on the NIC, and MAC engine and physical wire latency.

8. Future Work

Looking at the numbers of latencies and overhead in our protocol, several places appear to be good candidates for improvement. The firmware running on the NIC takes too much time. The DMA latency is big, 3.5μs for a read DMA and 10μs for a write DMA, we should definitely reduce the number of DMAs in the protocol processing. The MPI library and RMPP each also takes a big share. We propose the following strategies to reduce the latency in our future work:

- **Firmware** The largest time consumer is the firmware, which takes 30.5 μs out of the total of 86μs. Out of the 30.5μs, receive processing takes 20μs. By combining the two read DMAs to one, the send processing on the NIC can be cut in half. By simplifying matching for receives, we should be able to cut the receive processing in half too. This can reduce the firmware latency from 30.5μs to 15μs.

- **DMA latency** Three write DMAs on receiving is too much. The header DMA is only there to ask the operating system to release the receive buffer registration. Therefore it can be done in the background, and we can remove that DMA it from the critical path. If we can change the MPI library so it reserves a small data buffer in each event entry, then data DMA and event DMA can be merged into one for short messages. On the send side we can reduce latency by combining the two read DMAs. Overall, we should be able to reduce the DMA latency by half; from 15μs to 8μs.

- **Interrupt** We will remove the 9μs interrupt overhead from the critical path. Interrupts will be queued and handled by the operating system when it is idle. This requires the operating system to provide a mechanism to process queued interrupts.

- **MPI library** Improvements in the MPI library by tuning it should help short message latency.

- **NAL** The prototype network abstraction layer has too much overhead and we expect it can be reduced by half to about 5μs.

Adding up all these savings, we should be able to reach a latency of 49.5μs, which was our original goal for this work. The thorough measurement explained in this paper, make it possible to target the problem areas very specifically.

Acknowledgments

We wish to thank Pete Wyckoff of OSC for making EMP available and answering all our questions about it. Patricia Crowley of UNM has helped us with the code that runs on the Acenic, which is greatly appreciated. Patrick Bridges of UNM made many good suggestions on what to measure. Finally, we want to thank the other members of the Scalable Systems Lab (SSL) at UNM.

References

[1] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandervoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profi l-ing: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4), Nov. 1997.

[2] S. Araki, A. Bilas, C. Dubnicki, J. Edler, K. Konishi, and J. Philbin. User-space communication: A quantitative study. In *Proceedings of Supercomputing'98 (CD-ROM)*, Orlando, FL, Nov. 1998. ACM SIGARCH and IEEE.

[3] R. Brightwell, L. A. Fisk, D. S. Greenberg, T. Hudson, M. Levenhagen, A. B. Maccabe, and R. Riesen. Massively parallel computing using commodity components. *Parallel Computing*, 26(2–3):243–266, Feb. 2000.

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

[5] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and v. Thorsten. LogP: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 1–12, May 1993.

[6] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd Symposium on Operating Systans Design and Implementation (OSDI-99)*, pages 87–100, Berkeley, CA, Feb. 22–25 1999. Usenix Association.

[7] A. B. Maccabe, W. Zhu, J. Otto, and R. Riesen. Experience in offbading protocol processing to a programmable NIC. In *Proceedings of IEEE 2002 international conference on cluster computing*, Chicago, Illinois, Sept. 23–26 2002.

[8] T. W. Pratt. How to ”quantify” an application code to create a benchmark. http://ct.gsfc.nasa.gov/eval/Round2/quantify.html, June 30 1999.

[9] 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, July 2002.

[10] P. Shivam, P. Wyckoff, and D. Panda. EMP: Zero-copy OS-bypass NIC-driven gigabit ethernet message passing. In *Proceedings of Supercomputing'01 (CD-ROM)*, Denver, Nov. 2001. ACM SIGARCH/IEEE.

[11] D. Turner and X. Chen. Protocol-dependent message-passing performance on linux clusters. In *Proceedings of IEEE 2002 international conference on cluster computing*, Chicago, Illinois, Sept. 23–26 2002.

[12] J. Vetter. Performance analysis of distributed applications using automatic classifi cation of communication ineffi ciencies. In *Conference Proceedings of the 2000 International Conference on Supercomputing*, pages 245–254, Santa Fe, New Mexico, May 8–11, 2000. ACM SIGARCH.

[13] J. Vetter. Dynamic statistical profi ling of communication activity in distributed applications. In S. T. Leutenegger, editor, *Proceedings of the 2002 International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS-02)*, volume 30, 1 of *SIGMETRICS Performance Evaluation Review*, pages 240–251, New York, June 15–19 2002. ACM Press.