SeaStar Interconnect: Balanced Bandwidth for Scalable Performance

THE SEASTAR, A NEW ASIC FROM CRAY, IS A FULL SYSTEM-ON-CHIP DESIGN THAT INTEGRATES HIGH-SPEED SERIAL LINKS, A 3D ROUTER, AND TRADITIONAL NETWORK INTERFACE FUNCTIONALITY, INCLUDING AN EMBEDDED PROCESSOR IN A SINGLE CHIP.

Ron Brightwell
Kevin T. Pedretti
Keith D. Underwood
Sandia National Laboratories

Trammell Hudson
OS Research

Cray Inc. designed the SeaStar specifically to support Sandia National Laboratories' ASC Red Storm,[1,2] a distributed-memory parallel computing platform containing more than 11,000 network endpoints. SeaStar presented designers with several challenging goals that were commensurate with a high-performance network for a system of that scale. The primary challenge was to provide a well-balanced, highly scalable, highly reliable network. From the Red Storm perspective, a balanced network is one that maximizes network performance relative to the computational power of the network endpoints. A main challenge for SeaStar was to maximize the bytes-to-flops ratio of network bandwidth—that is, to maximize the amount of network bandwidth relative to each node's floating-point capability. Table 1 (next page) presents the peak performance per node and peak bandwidth per node of the top 10 distinct systems from the November 2005 Top500 Supercomputer Sites list (http://www.top500.org). For these purposes, a *node* is a symmetric multiprocessing unit that the network link serves, and *peak bandwidth* is the maximum nominal bandwidth of the link into that node. The SeaStar provides Red Storm with the highest (peak) bytes-to-flops ratio by more than a factor of 4.

In addition to traditional network performance requirements, SeaStar must support the challenges associated with scaling a single scientific application to a machine's full size. Red Storm's purpose is to run tightly coupled, high-fidelity modeling and simulation codes on several thousand processors for long periods of time, typically days. To maintain a high parallel efficiency as the number of nodes in the job increases, network performance must not degrade significantly at scale, and a platform must manage network-related resources appropriately.

The challenge of attaining network reliability for a platform such as Red Storm comes from providing mechanisms to ensure correct data delivery with minimal impact on performance and scalability. Reliability issues arise in a variety of circumstances and scenarios, and many of them depend on the network's fundamental properties. SeaStar has very desirable properties that greatly enhance network reliability, but its other properties present significant challenges.

Although you can view performance, scalability, and reliability independently, each

0272-1732/06/$20.00 © 2006 IEEE Published by the IEEE Computer Society

Table 1. Network bandwidth balance ratios.

Machine	Peak node speed (Gflops)	Peak node bandwidth (Gbytes/s)	Ratio (Gbytes/Gflops/s)
ASC Purple	48.0	8.0	0.17
ASC Red Storm	4.0	4.8	1.2
Blue Gene/L	5.6	0.35	0.0625
Columbia	24.0	6.4	0.27
Earth Simulator	64.0	12.3	0.192
Mach 5	16.0	0.5	0.03
MareNostrum	17.6	0.5	0.028
Thunder	22.4	1.0	0.04
Thunderbird	14.4	2.0	0.13

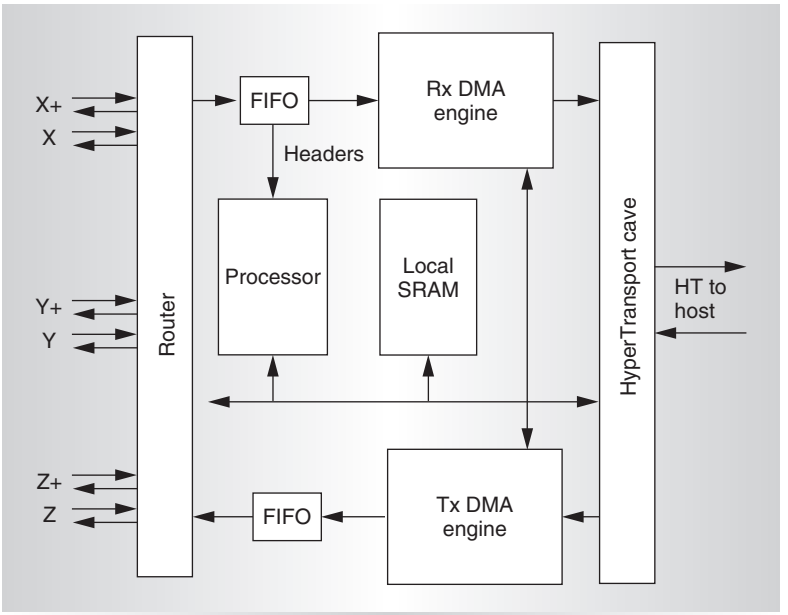

Figure 1. SeaStar block diagram.

characteristic can considerably influence the others. Design and implementation decisions made at the hardware level can have performance and scalability implications that extend through the network protocol stack all the way to the application. Likewise, higher-layer functionality requirements can have a significant impact on design and implementation choices in the layers underneath.

Red Storm

Cray has produced the Red Storm system as the Cray XT3. Tomkins[1] and Alverson[2] provide more detailed overviews. Red Storm is a joint project between Sandia and Cray that began in 2002. The machine follows the functional partition model, in which nodes run different software depending on which services they provide. Service and I/O nodes run the Linux operating system. Compute nodes run a lightweight kernel developed by Sandia and the University of New Mexico, specifically designed to support a distributed-memory, space-shared, massively parallel computing platform.[3]

Whereas Cray was ultimately responsible for delivering the final product, Sandia was responsible for the system design and architecture, and has contributed significantly to the software environment. Specifically, Sandia provided the lightweight kernel operating system and its associated runtime system.

Cray targeted the Portals network programming interface,[4] also developed jointly by Sandia and the University of New Mexico, as the lowest message-passing API for the SeaStar. Portals is a one-sided communication interface with matching semantics to efficiently support an implementation of the message-passing interface (MPI).[5] Cray initially provided the Portals implementation for the SeaStar. As the project progressed, however, the development of the Portals implementation and the SeaStar firmware evolved into a collaborative project. Sandia designed and implemented the specific network interface controller (NIC)-based firmware described here.

SeaStar hardware

Figure 1 shows a basic block diagram for the SeaStar. Independent send and receive

direct memory access (DMA) engines interact with a router (which supports a 3D torus interconnect) and a hyper transport (HT) cave (which provides an interface to the AMD Opteron processors and host memory). An embedded IBM PowerPC processor programs the DMA engines and help with other network-level processing needs.

DMA engines provide robust support for transferring data between the network and host memory by providing hardware for breaking each outgoing message into 64-byte packets and reassembling incoming messages. This packetization serves to interleave incoming packets from distinct sources' different messages. The SeaStar has a hardware mechanism to match incoming packets to their appropriate message stream, but it can process only 256 concurrent streams. This is a major scalability challenge for a network interface that potentially must handle 10,000 incoming message streams concurrently.

Besides packetization, DMA engines also provide hardware support for an end-to-end, 32-bit cyclic redundancy check. (Since the CRC is performed by the SeaStar at the receiver, it is calculated as data streams through the SeaStar, not after it has been deposited into host memory. We assume that the HT link will detect any data corruption in moving data from the network into host memory.) This augments the extremely high reliability provided by a 16-bit CRC with retries performed on each of the individual switch links.

The physical links in the network support up to 2.5 Gbytes/s of data payload in each direction, accounting for overhead in both the 64-byte packets used by the router and the reliability protocol on the individual links. The HT interface runs at 800 MHz, providing a theoretical peak of 3.2 Gbytes/s per direction with a peak payload rate of 2.8 Gbytes/s after protocol overheads (and a practical rate of approximately 2.4 Gbytes/s). The table-based router provides a fixed path between all nodes, resulting in ordered delivery of the individual packets of each message. The router supports two virtual channels, but the network interface supports only virtual channel selection on the send side. On the receive side, the router combines the two virtual channels into a single receive queue.

The embedded processor on the SeaStar is a dual-issue, 500-MHz PowerPC 440 processor with independent 32-Kbyte instruction and data caches. The firmware running on the PowerPC is responsible for programming the DMA engines, since programming them via the host processor with accesses over the HT is prohibitively slow. On the receive side, the firmware is also responsible for recognizing the start of new messages and processing incoming message headers. The firmware must also recognize and respond to DMA completion events.

To hold the firmware and local state, and to handle interactions with the host, the SeaStar has 384 Kbytes of scratch memory. This memory is protected by error correcting code, complete with scrubbing to find and correct errors as they occur. In this context, the firmware running on the network interface must handle a certain portion of the network management. However, the firmware can also be augmented to handle other aspects of the protocol stack. Here, we present firmware implementation details for both a host-based mode, which does minimal work on the SeaStar, and a NIC-based mode, which offloads most of the Portals processing to the PowerPC.

Firmware

Sandia developed firmware to run on the SeaStar's embedded processor, using C and a standard GNU tool chain for the PowerPC 440. The firmware currently consists of just over 6,000 lines of C code and approximately 200 lines of assembly code, according to Wheeler's SLOCCount tool (www.dwheeler.com/sloccount). When compiled, this results in a 36-Kbyte binary image. We give more detailed descriptions of the development tools used and the firmware's internals in a previous paper.[6]

General architecture

At the most basic level, the firmware's job is to program the SeaStar's DMA engines to move messages between host memory and the network. The firmware must track and make progress on multiple concurrent message transmissions and receptions, and notify the host when each has completed.

At the host, several client processes require access to the high-speed network. This number varies based on the operating environ-

ment. Linux service nodes typically have 10 to 100 clients, whereas compute nodes (which run the Catamount lightweight kernel) have one management client and usually one application process per host CPU core. To keep maintenance tasks manageable, we chose to develop a single firmware image that supports both operating environments.

Host-based mode

In host-based mode, application-level requests trap into the kernel, which then forwards requests to a single firmware-level mailbox. This approach makes it possible to support many clients, because each firmware mailbox and its associated pool of message-tracking structures consume a large portion of the SeaStar's 384 Kbytes of memory.

This mode can handle transfers that are either physically contiguous or physically discontiguous. For transfers that span physically contiguous regions of memory, the kernel simply passes the starting physical address and message length in a command to the firmware. The firmware's messaging machinery then generates the DMA engine commands on the fly.

In the physically discontiguous case, which includes handling requests from Linux processes, the kernel generates the list of DMA engine commands and passes it to the firmware. When each message transmission or reception is complete, the firmware posts an event to host memory and interrupts the host processor. The operating system is then responsible for delivering the event to the appropriate client process. Brightwell, Pedretti, and Underwood give a more detailed description and performance evaluation of this mode of operation.[7]

NIC-based mode

The goal of this mode is to avoid interactions with the host processor and operating system as much as possible, similar to what other high-performance networks have done.[8,9] The benefits are reduced latency, increased message throughput (messages processed per second), and reduced host overhead related to message processing; together, these effects make more of the host CPUs available for computation. The trade-offs are that the embedded processor is far slower than the host CPUs, and the amount of SeaStar memory is more than two orders of magnitude smaller than the memory that the host could dedicate to message passing. We developed NIC-based mode to evaluate whether these obstacles were surmountable. As with supporting both Linux and Catamount, we chose to implement support for both host- and NIC-based modes in the same firmware image.

For a process to use the network, it must call the Portals library initialization routine. Using Portals' trusted mailbox, the kernel forwards initialization and shutdown requests to the firmware. For NIC-based mode, initialization involves allocating SeaStar memory for a new untrusted mailbox and related firmware structures, mapping the mailbox into the process' virtual address space, and providing the firmware with a map of this space. Once initialization has successfully completed, the process can then initiate further operations by writing directly to its mailbox in SeaStar memory.

Because this user-level mailbox is untrusted, the firmware must carefully inspect all commands received at this mailbox. The firmware uses the address map provided by the kernel to ensure that data transfers stay within the bounds of the process' address space. This is relatively straightforward for the processes running under the lightweight kernel, since Catamount provides a physically contiguous address space. However, since the SeaStar has no hardware support for virtual-to-physical address mapping, a very limited amount of scratch memory, and a relatively long access time to host memory, supporting operating systems that do not ensure a physically contiguous mapping is significantly more complex. Given this extra complexity and the additional performance costs associated with supporting physically discontiguous translations, NIC-based mode currently works only with Catamount.

In addition to protecting the source and destination of data transfers, we also needed to protect a trusted portion of the Portals header that precedes every message. This trusted portion contains information such as the source node ID and process ID, which a user-level process should not be able to modify. Protecting the trusted header from modification is complicated by the fact that the SeaStar can send mes-

sages only from host memory. If this were not the case, the NIC could simply construct the entire header in SeaStar memory and send the header in a single DMA command. Unfortunately, this limitation of the SeaStar means that the trusted portion of the header must be kept in kernel memory, and two DMA commands are necessary to send the different parts of the message header. However, the added complexity and extra DMA command incur no significant performance penalty.

Once the firmware recognizes and validates a command written into the mailbox, it performs the requested operation. It can handle some commands, such as memory registration and receive posting, immediately, and post the result back to the process. Other commands, such as *put* and *get* operations, must go to the firmware's messaging machinery. Once the data movement specified by these commands is completed, the firmware generates a completion event and writes it directly into an event queue in the process' memory. The process must eventually poll the event queue to recognize the event. Unlike the host-based implementation, posting an event does not involve raising and processing an interrupt on the host.

The NIC-based mode's most significant performance benefit is its ability to handle message reception autonomously without involving the host processor or operating system. For host-based mode, every time a new message arrives, the firmware must raise an interrupt to ask the operating system where to deliver it. From a latency perspective, this path is extremely expensive because the interrupt potentially causes a context switch into the operating system and involves several relatively slow round-trip accesses across the HT bus. In contrast, the NIC-based implementation has all the information necessary to determine where in host memory to put incoming messages, effectively bypassing or offloading this responsibility from the operating system and the application.

When a message arrives, the firmware's messaging machinery passes the header information to another routine, which parses information in the header to determine exactly where in host memory to deposit the incoming message. Once the firmware determines the message's ultimate destination, it gives the messaging machinery the destination address in host memory and the number of bytes to receive, which implicitly gives the number of trailing bytes to discard, if any. Once message reception is complete, the messaging machinery notifies another routine, responsible for writing a completion event directly into the host client's event queue in host memory.

Flow control protocol

As mentioned, a unique feature of the SeaStar is its hardware support for demultiplexing incoming packets into their appropriate message stream. However, this hardware can handle only 256 simultaneous message streams from distinct sources. When the flow reaches this capacity, the hardware cannot process long messages from new sources and discards them until space is available in the hardware resource. (Long messages are longer than 16 bytes. Short messages—less than 16 bytes—fit in one 64-byte packet along with the header and are receivable without demultiplexing hardware. Thus, the SeaStar needs an end-to-end flow control protocol to recover from lost messages in rare circumstances when the hardware resource is insufficient to handle the number of concurrent messages received.

Because the individual switch links guarantee delivery of packets uncorrupted and in order, the protocol needs only to handle the simple case of resource exhaustion. Thus, we can avoid many of the complications typical of a protocol for an unreliable network. Most notably, control messages do not require the same hardware resource as longer messages and we can assume them to always complete successfully; however, many other issues still apply. The flow control protocol must preserve pair-wise message ordering, for example. Portals and MPI mandate that message reception (and the resulting matching) must occur in order. If a message is dropped due to resource exhaustion, any subsequent in-flight messages from that source must be dropped, even if resources in the demultiplexer become available. Buffering is not feasible, because the space required on the receiver could be very large or unbounded. Our protocol drops the incoming messages instead.

The protocol uses a sequence number on each message. It sends an explicit acknowledge (ACK) or negative acknowledge (NACK) in response to every message. The acknowledgment (positive or negative) explic-

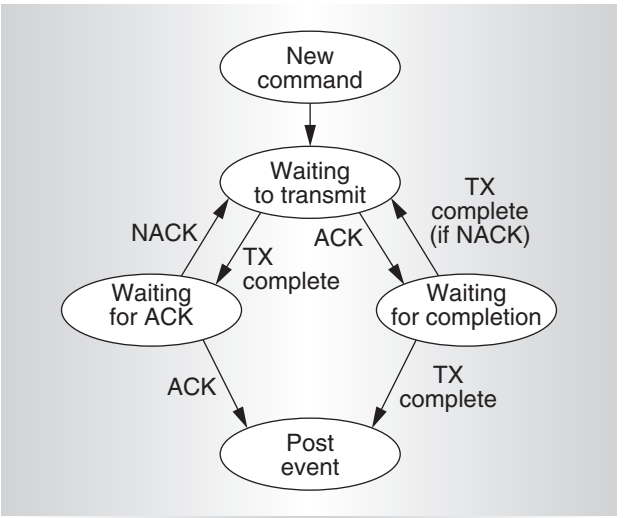

Figure 2. Life of a Tx command.

itly specifies the message to which it refers. Message-dependent deadlock[10] is not an issue, because data and control messages use independent virtual channels. The receiver tracks sequence numbers only when a message has been negatively acknowledged.

Protocol integration

The transmit side requires very little modification to support the protocol. Without the protocol, when a transmit command is issued, the firmware builds an appropriate pending structure and places it on the transmission queue; when the DMA engine signals transmit (Tx) completion, the firmware posts an event to the event queue for this message. When we add the protocol to the transmit path (as in Figure 2), the same process occurs except that a successful acknowledgment (ACK) must be received before the completion event posts to the host. Either the ACK or the completion notification from the DMA engine can occur first; thus, the protocol can mark the pending structure can as receiving an ACK while still waiting for the transmit to complete, or the message can complete and move to a hash table to await its ACK.

The other wrinkle is the receipt of a NACK. When a NACK is received, the message (through the pending structure) is marked as requiring retransmission. If it has completed or when it completes, the message will go back into the transmission queue. It is not possible to cancel a message for which the DMA engines have already been programmed, so the message must be transmitted in its entirety. A message is retransmitted each time a NACK is received until the message has a successful acknowledgment. Because this can occur only in a resource exhaustion case at the receiver, there is no limit on the number of retries, and the sender will eventually be throttled.

The changes to the receive path are technically more challenging, but conceptually as simple as those to the transmit side. In normal operation, the only change (other than the addition of tests for several exception conditions) is the addition of an ACK transmission as soon as a new message header is processed. Early ACK transmission enables the sender to free resources more quickly. It also means that an ACK (or even a NACK) could be received before the message finishes transmission.

In the presence of exception conditions, several things change. When the flow of messages exhausts the SeaStar stream management resource, the hardware must drop the message and send a NACK. When it drops the message, the firmware increments a counter to indicate that there are outstanding dropped messages, and sets a bit in a large bit mask, indicating the dropping of a message from that source. The firmware updates a corresponding entry in the receive (Rx) sequence number field to indicate the sequence number of the dropped message. The only time the firmware checks the Rx sequence number is when the counter is non-zero and the bit corresponding to the messages source is set.

Once a message has been negatively acknowledged, the next message could be out of sequence or a retransmitted message. Either can now be identified because

- a counter identifies abnormal operation,
- a bit mask indicates the affected source, and
- the first sequence number defines the expected retransmission sequence (an early test in the receive path detects abnormal operation and checks the sequence numbers only when necessary).

The firmware discontinues abnormal operation as soon as it receives the first retransmit; it then clears the bit field and decrements the

counter. Hence, this protocol expects retransmissions in order—a relatively easy guarantee to make at the sender.

Sequence number storage and synchronization

A sequence number is required for each message to let the sender know which message has been dropped and to detect the resulting out-of-sequence messages at the receiver. Connection establishment, however, is a far too heavy action for a lightweight protocol, and can lead to an application-related persistent state. Such state can be difficult to clean up quickly during abnormal termination. Furthermore, connection state must be light enough that every node can have a connection to every other. With only 384 Kbytes of SeaStar memory, this translates into having exactly one sequence number pair preallocated for every other node in the system. The sequence space can be small (16 bits), because other resources in the system limit the number of in-flight messages between a pair of nodes to fewer than 1,024. Therefore, 64 Kbytes of memory can store the sequence numbers for a 16,000-node system, which can comfortably fit in SeaStar memory.

Having node-level, persistent sequence numbers implies the need for a way to synchronize sequence numbers when a node is rebooted; however, this is not needed. Our strategy manages Rx sequence numbers loosely; normal operation has no need for expected Rx sequence numbers. The only time the Rx sequence number requires checking is after an exception.

Nominally, the protocol supports sequence number synchronization so that we could extend the protocol to have additional reliability features. To accomplish this, we reserve a special sequence number. Rebooting a node sets its Tx sequence number to the reserved sequence number for all potential pairs. Receiving the initial sequence number can serve to reset the expected Rx sequence number, if an Rx sequence number was expected. Receiving a noninitial sequence number when the initial sequence number is expected would also reset the expected Rx sequence number to whatever was received. Thus, this scheme can synchronize sequence numbers quickly, but only when necessary and without the exchange of extra messages.

Handling exception conditions

Adding the protocol to the network path brings a set of extra obligations—for example, handling the case when a node is down. Fortunately, Portals has a well-defined failure semantic so that a message to a node that is down can "fail." To handle this type of scenario, we introduced two new types of acknowledgment packets:

- a PREBOOT_ACK for nodes that are powered on, but not fully booted allows the recipient to tell the sender that this message and all previous outstanding messages to this node should be failed, and
- a FIRST_POSTBOOT_ACK ensures that the first message a node sees after boot will succeed, but all previous outstanding messages will fail.

As a final precaution, it is possible for the target node to be powered off. In these cases, the protocol must not leak resources, so we introduced a timeout mechanism. After a very long time without an ACK, the protocol will free the resource freed and fail the message. This works acceptably and a "very long time" is reasonable to define, because messages to powered off nodes are rare.

Optimization opportunities

There are two specific opportunities for optimizating this protocol that we have designed but not yet implemented for production use. The first is an implicit acknowledgment scheme that prevents the need for acknowledgment processing for extremely small messages. The second is a back-off scheme to prevent retransmission floods in worst-case scenarios.

Because we designed the protocol to handle the exhaustion of a resource to resolve packets to messages, very small messages (those that require only a single packet) do not need protocol protection. These messages can be implicitly acknowledged at the source, which reduces the protocol processing load. The design of implicit acknowledgments, however, requires some care. Because a very short message can directly follow a long message and transmit before the receiver acknowledges the long message, the very short message might require retransmission. If the hardware

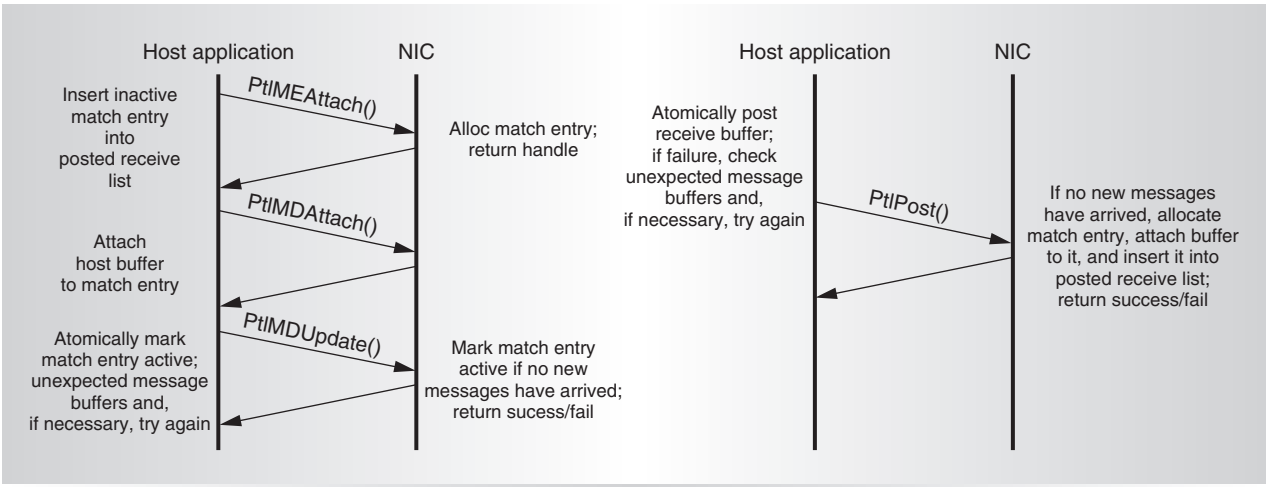

Figure 3. Posting an MPI receive using standard Portals API calls (a) versus the new PtlPost() API (b).

drops the long message, the receiver has nowhere to buffer the very short message. A prototype implementation indicates that message throughput improves by approximately 10 percent for messages with 16 bytes of payload or less; thus, it is not clear that the advantage outweighs the added complexity.

The second optimization relates to the potential for a single flood of messages to a node to cause an extremely high number of retries. In a typical scenario, we rarely expect most nodes to receive a flood of perfectly interleaved messages that exhaust the hardware's ability to resolve packets to messages. If this situation does happen, you might expect a retransmit or two. However, the situation in which an I/O node simultaneously receives a large write request from every compute node can still cause a performance problem. Although the design of most higher-level software can mitigate such a scenario, the potential exists, especially for I/O nodes, to address *N*-to-1 communication patterns, where *N* can be arbitrarily large. In such a scenario, our protocol will cause rapid retransmits, which will ultimately fail as they compete for scarce resources. To address this problem, the negative acknowledgment message has a field to indicate the sender's place in line. The firmware increments a simple saturating counter for each NACK sent, and decrements it for each ACK. This notifies the sender of the amount of pressure on the hardware resource and allows the sender to develop a back-off scheme before retransmitting.

Portals

We have described the Portals implementation for the SeaStar in a previous article.[11] Our initial performance evaluation of the NIC-based mode's implementation motivated optimizations. We encountered two significant opportunities for optimization—one that involved adding to the Portals specification, and one that modified the way our MPI implementation used Portals.

MPI receive posting

First, performance analysis showed that posting an MPI receive was far slower using NIC-based mode than host-based mode. This would often lead to confusing results for our standard suite of microbenchmarks. Eventually, we traced this issue to the fact that many microbenchmarks did not ensure the preposting of MPI receives. NIC-based mode's slower receive posting meant that more messages ended up being unexpected compared to host-based mode, which resulted in degraded performance.

The underlying cause of NIC-based mode's poor receive posting performance was that a round-trip to the SeaStar across the HT link is more than an order of magnitude slower than a Catamount system call (approximately 1 µs versus 65 ns). As Figure 3 shows, posting an MPI receive requires three round-trips to the SeaStar. Host-based mode can handle this sequence of three calls in the operating system, without involving the firmware, so the overhead of each is roughly equivalent to three

system calls. To reduce the number of round-trips across the HT down to one, we combined these three operations into a single function call, PtlPost().

The original Portals design focused on identifying a set of general building blocks for easy combination into a variety of upper-level protocols. In this instance, creating a specialized API call that combined several of the simpler building blocks into one aggregate operation led to significant performance advantages for a critical MPI operation. This new call is now part of the official Portals specification.

Preregistration of memory

The second significant optimization that we made to better support NIC-based mode modified the way MPI used Portals for sending messages. Like the PtlPost() optimization, we aimed this change at reducing the number of round-trips required to perform a common operation—in this case, sending a message. We also intended to reduce the number of Portals resources that MPI used for sending messages, freeing more SeaStar memory for receiving messages.

The previous implementation of MPI used three different protocols for short, medium, and long messages.[12] For medium and long messages, MPI would first create a Portals memory descriptor (MD) describing the user's buffer in one operation and then use the new MD to initiate a subsequent put operation. This would result in two HT traversals to initiate each send operation and consume an MD for each send— potentially consuming significant resources for applications that use MPI nonblocking sends.

Catamount allows an optimization to this approach. Since the regions of a Catamount process are physically as well as virtually contiguous, the MPI implementation can use only a few MDs to cover a process' entire address space. During initialization, MPI creates an MD that spans the data region, an MD that spans the stack region, and an MD that spans the heap region. This approach eliminates the need to create an MD for each individual user buffer, because one of these region MDs already covers the user buffer. This reduces the number of HT crossings to just the one for the put operation.

We expected the impact of this change on the ping-pong bandwidth performance for medium and long messages to be minimal because an HT crossing is not significant relative to the time needed for the transfer. However, we did expect this optimization to improve message rate and help free up Portals resources allocated from SeaStar memory.

Test environment

The platform used for our experiments is the 10,368 processor Red Storm machine at Sandia. This machine is a slightly specialized version of the commercial XT3 product. It differs from the XT3 in that the network is not a torus in all three directions. To support the easy switching of portions of the machine between classified and unclassified use, Red Storm has special switching cabinets. This capability and the limitation of cable lengths allow the network to be a torus only in the z direction. Each node in Red Storm has a 2.0-GHz Opteron with at least 2 Gbytes of main memory.

Benchmarks

We chose several microbenchmarks for our initial evaluation, beginning with a simple test of ping-pong latency and bandwidth. For streaming tests, we used a bandwidth benchmark developed by Ohio State University, which posts 64 messages at the receiver and then sends a stream of messages from the sender. To measure collective performance, we used the Pallas MPI benchmark suite version 2.2.1 (www.pallas.com/e/products/pmb/index.htm). Finally, we used a benchmark developed at Sandia to measure the impacts of MPI queue depths on network performance.[13]

Results

We group results into three basic categories: traditional point-to-point benchmarks, collective benchmarks, and other data. In each instance, we compare the system with Portals processing implemented on the host and the NIC. In each case, we also evaluate the impact of adding the simple protocol.

For each graph, the left axis graphs performance (either in time or bandwidth), and most graphs include the percentage advantage provided by a NIC-based implementation on the right axis.

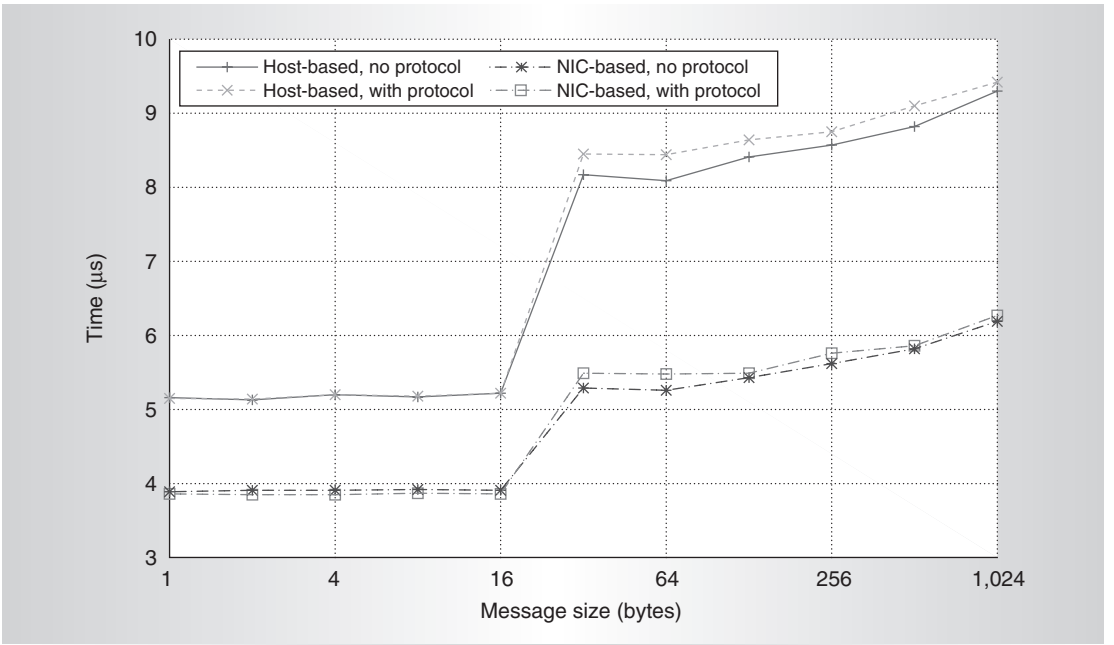

Figure 4. Small-message MPI latency.

Basic point-to-point benchmarks

Figure 4 compares the small-message latency of the host- and NIC-based Portals implementations. The difference is greater than 1 µs for extremely small messages, and nearly 3 µs for larger messages. This is the advantage of eliminating one interrupt for small messages, and two interrupts for larger messages (new message start and message completion). The transition point between 16 bytes and 32 bytes is one of the network's limitations. The small packet size forces the transition from programmed I/O (PIO) mode transfers to DMA transfers to occur at this point. Adding the protocol has measurable, but minimal, impact on ping-pong latency, because much of the protocol processing can overlap with host interactions.

The SeaStar has an impressive unidirectional MPI bandwidth, as Figure 5 shows; it is attributable to the use of the HT interface to the host. Peak bandwidth is currently greater than 1.1 Gbytes/s. We expect that Cray will deliver a second-generation SeaStar in 2006 with a bandwidth of more than 2 Gbytes/s. Although the NIC-based implementation offers very little advantage at extremely large message sizes, at small to moderate message sizes the significant reduction yields major advantages for the NIC-based implementation.

The NIC-based implementation also offers dramatic advantages in streaming bandwidth benchmarks, as Figure 6 shows. In the NIC-based implementation, the work is better partitioned between the host and NIC processor. This allows better overall message throughput than the host-based implementation; however, it also begins to introduce limitations, as more work is needed on the NIC. The protocol, for example, reduces the advantage of the NIC-based implementation because it introduces more work on the NIC. In the host-based implementation, much of this work overlaps with Portals and MPI processing occurring on the host.

Another strong advantage of moving away from bus-based interfaces to the host and toward bidirectional interfaces like HT is the ability to sustain full bidirectional bandwidth. As Figure 7 (page 52) shows, bidirectional bandwidth on Red Storm is twice the unidirectional bandwidth; however, the bandwidth curves suffer somewhat with the protocol's introduction because there is no longer anywhere to hide the extra processing overhead.

Pallas collective benchmarks

The Pallas benchmark suite includes benchmarks for many collective operations. Rather than include an excessive number of graphs,

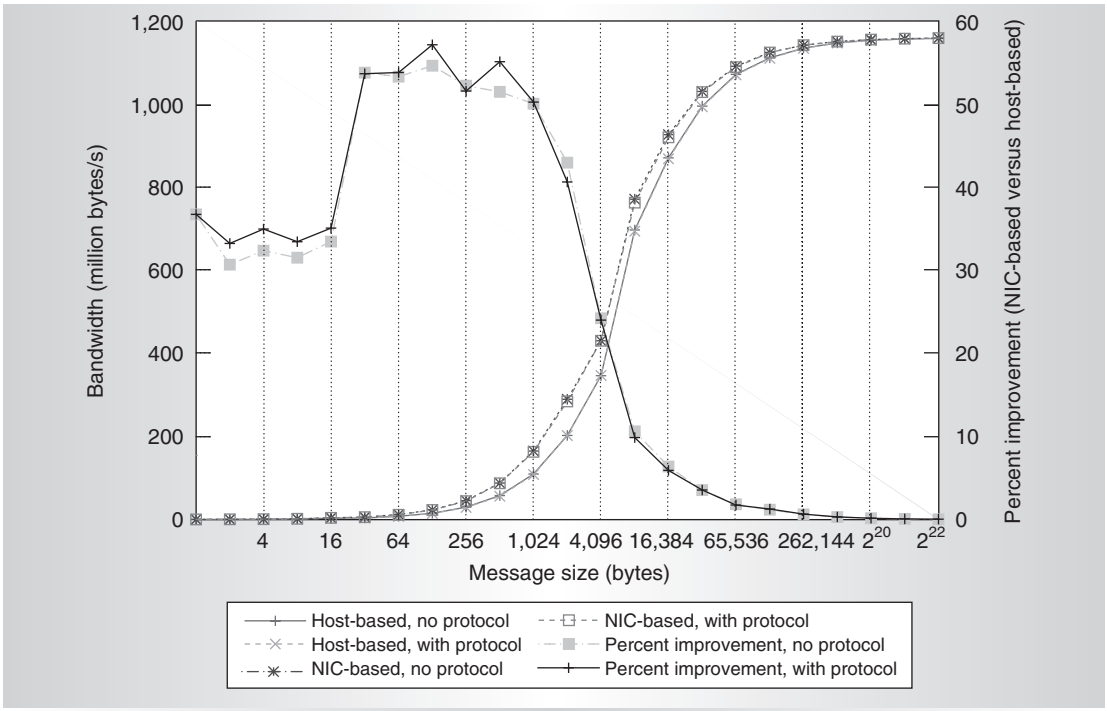

Figure 5. MPI bandwidth.

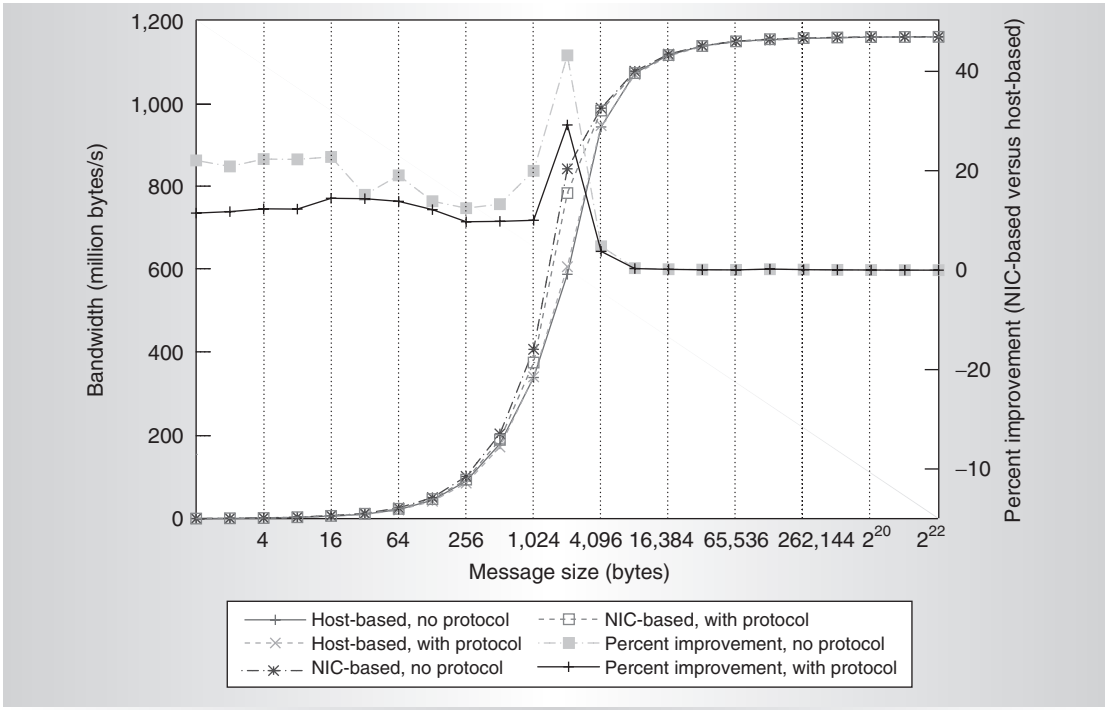

Figure 6. Ohio State University's streaming bandwidth test.

we have selected four that are relevant to many of the applications at Sandia. The first of these is MPI Barrier, a collective that is virtually never needed to write a correct MPI program; however, many application developers find it useful for debugging and timing, and so ulti-

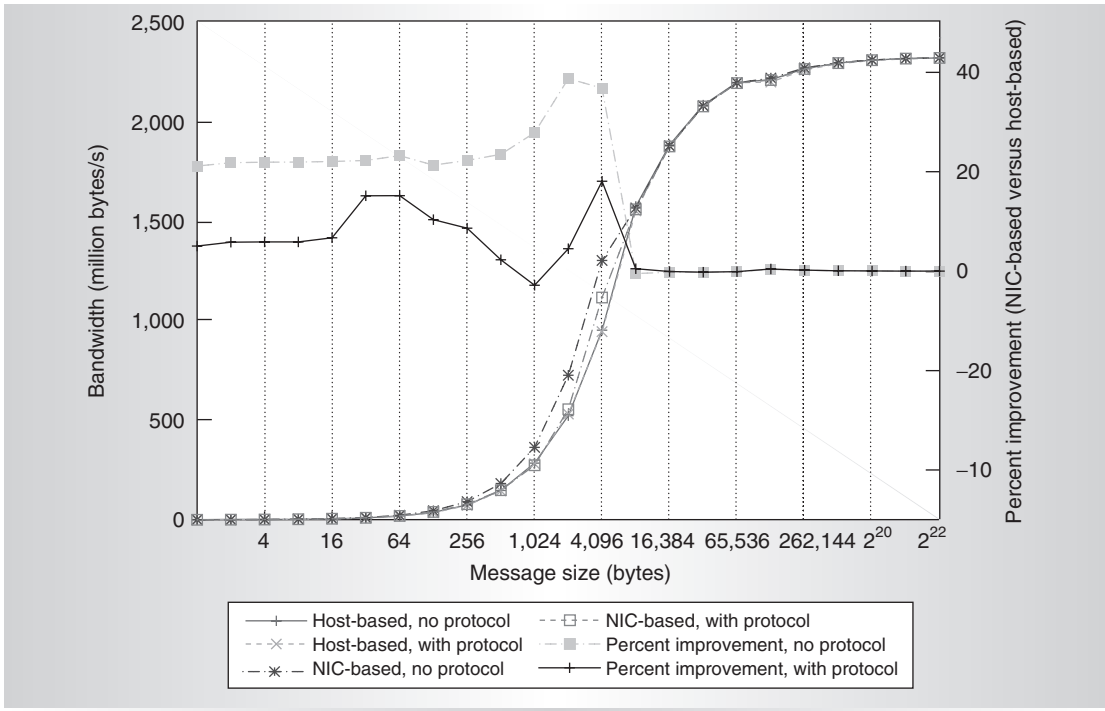

Figure 7. Ohio State University's bidirectional streaming bandwidth test.

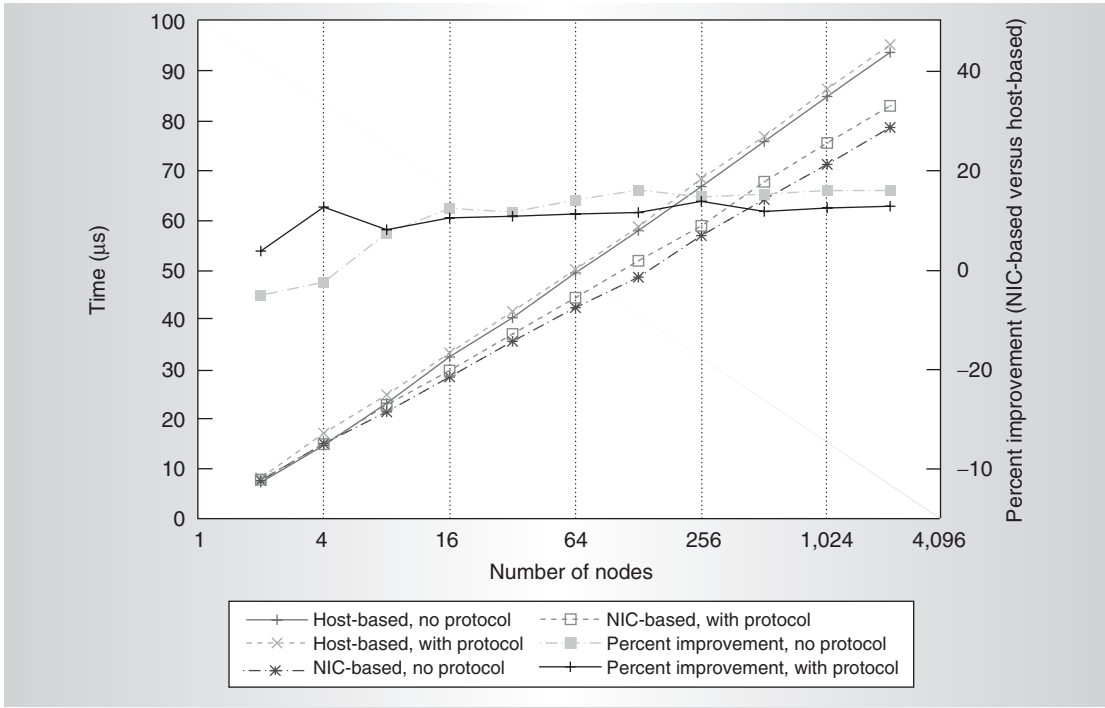

Figure 8. Barrier-scaling performance.

mately leave it in production codes. Because the barrier operation involves a significant number of small messages, it is one of the few benchmarks in which adding the protocol makes a noticeable difference in performance. Overall, the NIC-based Portals implementa-

tion has a significant advantage over the host-based implementation (as Figure 8 shows), particularly when there are more than 16 nodes.

MPI Allreduce and MPI Reduce are also common operations in many of Sandia's codes. Unlike the barrier operation, the reduction operations have data associated with them and require some computation on the host. The results presented in Figure 9 use 16-byte reductions, which are representative of the data used in Sandia operations (a double-precision number or a double-precision complex number). Moving Portals processing to the NIC has a slightly more significant advantage for MPI Allreduce than for MPI Barrier because the MPI Allreduce has work to do on the host that can overlap with the protocol processing.

MPI Reduce receives an even greater advantage thanks to its subtle differences from the MPI Allreduce operation. Whereas MPI Allreduce must reduce a single number and distribute it to all participating processors, a node in MPI Reduce can exit the call as soon as it finishes participating in the communication. This means that communications from two consecutive reductions can overlap, which leverages the offload provided by the NIC-based implementation. It also means that there is more opportunity to hide the protocol processing, and so MPI Reduce does not suffer a performance penalty from the protocol.

The final collective operation (shown in Figure 11) is MPI Allgather. This is an interesting operation in that

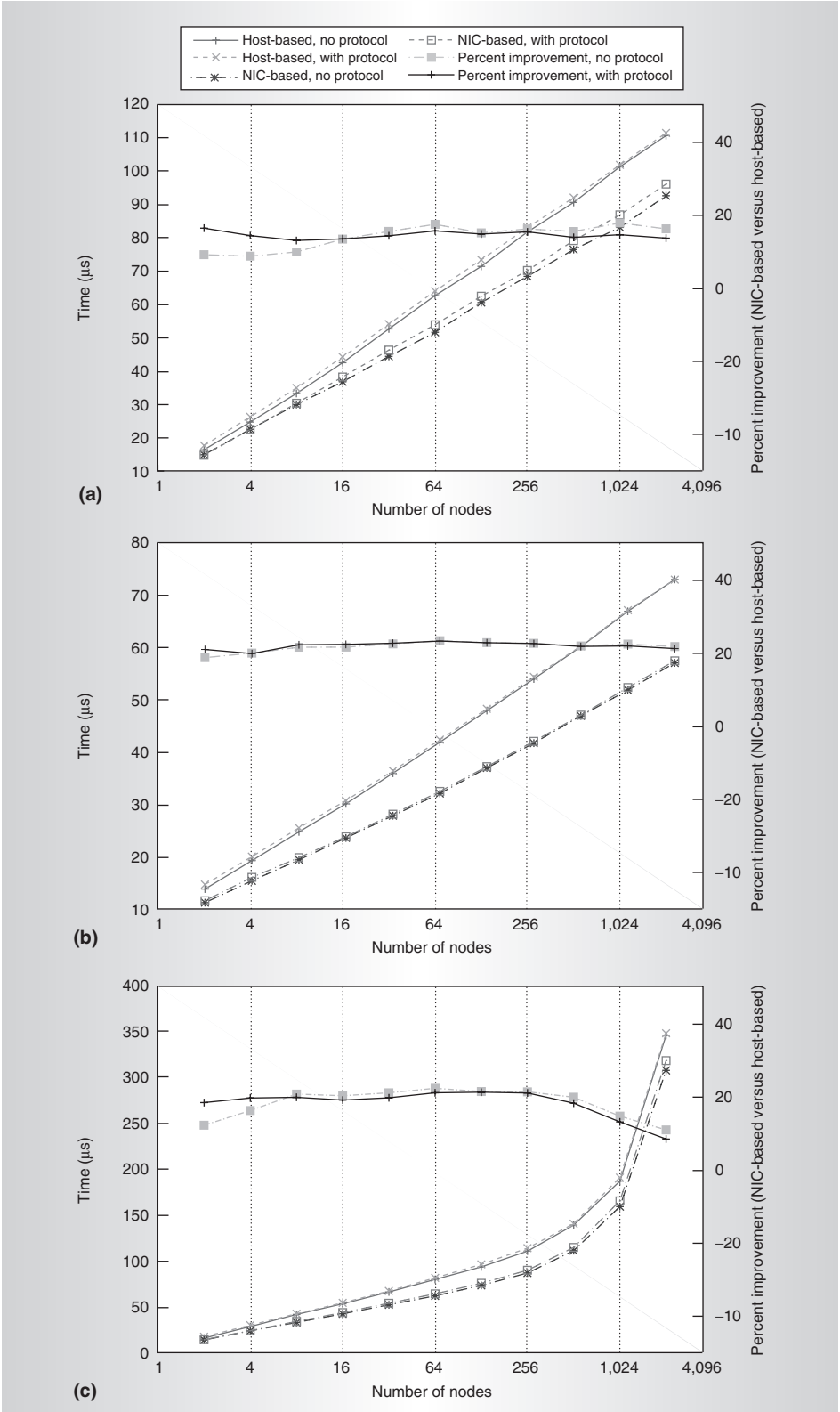

Figure 9. Scaling performance for MPI Allreduce (a), MPI Reduce (b), and MPI Allgather (c). Message scaling is 16 bytes in all cases.

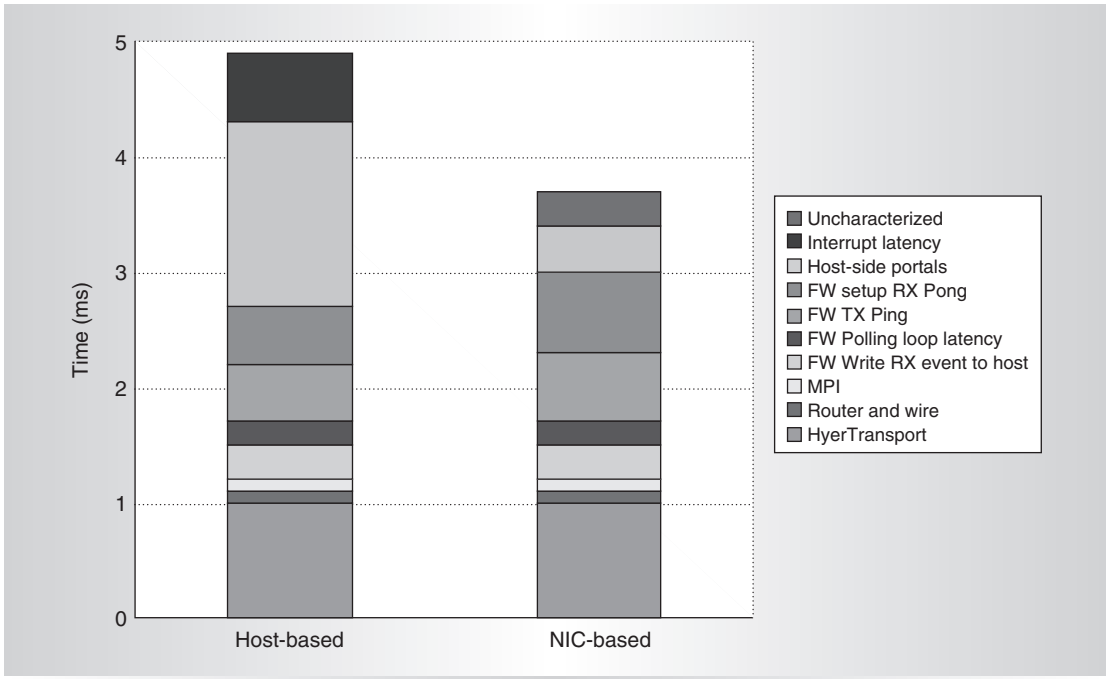

Figure 10. Profiles of one-way ping-pong time.

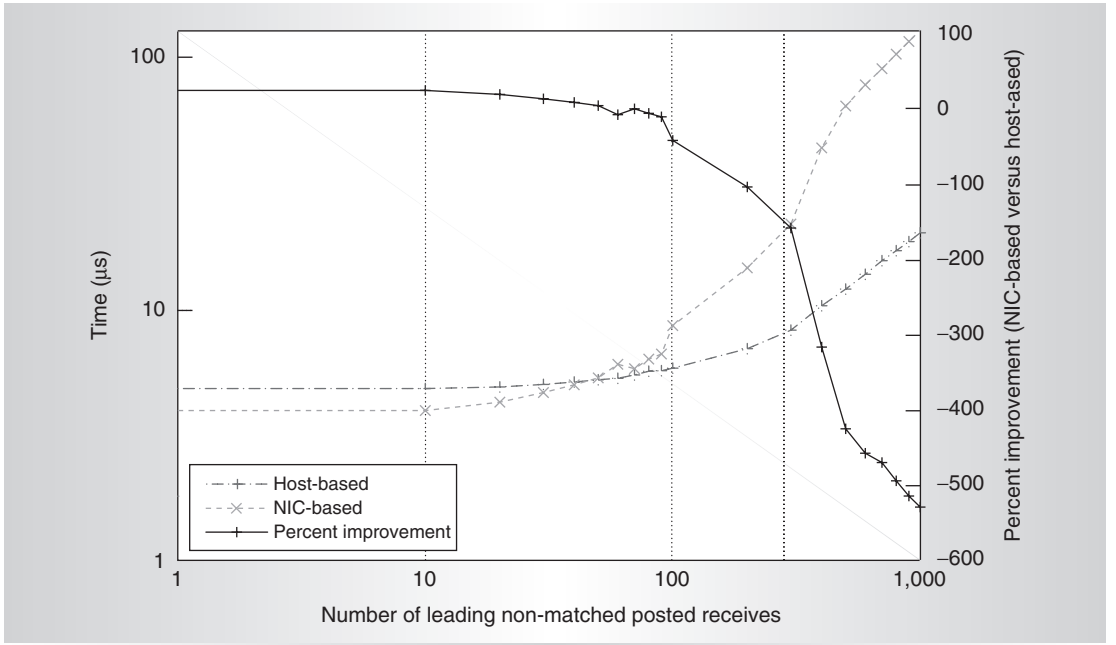

Figure 11. Effective network latency for various receive-queue traversal lengths.

the number of messages sent and the size of the message scale with the number of nodes. Thus, the time grows quadratically with the number of nodes, and the time for the host- and NIC-based implementations begin to converge slightly at larger message numbers.

Profiling results

So, where does the time go? Figure 10 presents a one-way profile of the ping-pong operations for both the host- and NIC-based Portals implementations. One striking thing to note is that the HT latency is a major con-

tributor to overall latency. Whereas HT generally has a far lower latency than other bus interfaces, the architecture of the SeaStar requires that an embedded processor communicate with the host processor through a shared RAM resource. The time for the processors to recognize that a new item has been written is a dominant factor in the HT latency.

The next significant characteristic to note is that the router time and the MPI time are both small slivers of the overall time. The router is highly optimized for high-performance computing, unlike current InfiniBand routers, for example, which are an order of magnitude slower (http://www.infinibandta.org). And most of the work for MPI (such as context and tag matching) is included in Portals.

Moving up the bar graphs, the time to notify the host of an Rx event is comparable for both host- and NIC-based implementations, as is the time spent in the *polling loop*. This is the main loop on the NIC that looks for work to do. It is responsible for polling hardware to check for new events and for polling a RAM region used for communicating with the host. The initial firmware implementation used roughly 500 ns per cycle through this polling loop; we have optimized this to less than 130 ns through profiling and analysis of the code paths.

Both transmit and receive operations push more work to the NIC for the NIC-based implementation. Thus, NIC-based implementations require more time to set up transmit and receive operations. However, the host-side Portals work drops dramatically, and moving most of the Portals work to the NIC eliminates the interrupt latencies. The NIC-based implementation still has a small amount of time that is not well characterized. Some of this time might be due to the timing granularity on the NIC and some to the overhead added by the protocol.

Other issues

Moving processing to the network interface can have significant ramifications. One limitation of traditional benchmarks is that they do not consider "real" scenarios. For example, benchmarks tend to consider only one receive at a time, while applications tend to have more than one posted receive. Host-based implementations typically keep posted receives in a linked list that the receiver checks or "walks" every time a new message arrives. When that processing moves from a fast host processor to a slower NIC processor, a long posted-receive queue can translate into much higher effective network latency. Figure 11 illustrates the difference in latencies as the length of the posted-receive queue increases. The NIC pays a penalty of approximately 30 ns for each item traversed (for the fastest NIC in the industry[13]); thus, after traversing a list of 50 items, the NIC-based approach actually loses to the host-based approach.

There are two important caveats attached to this result. Foremost, the benchmark's nature is inherently friendlier to the more advanced processor in the host. The two major issues are that the host processor has a much larger cache, and that, unlike the NIC processor, the host processor has a hardware prefetcher. In a benchmark world with a strictly ordered list, the combination of these two means that the host processor is faster than it should be on a per-message basis by at least a factor of 5 (the host processor should pay a cache miss for every list item). The second caveat is that, although some applications use extremely long lists, many have typical list lengths of 30 items or fewer.[14]

The Cray XT3 system has opportunities for improvement that we expect to implement in the near future. For example, we are currently implementing a back-off scheme to better manage a flood of traffic to a single node. In addition, we are considering performance optimizations (such as offloaded collective operations) that leverage the 500-MHz, embedded PowerPC. We are also considering a rendezvous protocol on the PowerPC for long messages. The current MPI implementation uses eager sends for all messages, which occasionally forces the receiver to drop long messages if the receiver does not find a matching posted receive. Implementing a rendezvous protocol in the NIC will allow support for true independent progress in MPI while eliminating the risk of potentially retransmitting large messages.

We are also planning an in-depth analysis of the impact of these additions and changes on real applications at scale. Several of these enhancements might significantly increase application performance and scalability.

Acknowledgments

We gratefully acknowledge the work of the members of the Scalable Computing Systems and Scalable Systems Integration departments at Sandia, especially Jim Laros and Sue Kelly.

References

1. W.J. Camp and J.L. Tomkins, "Thor's Hammer: The First Version of the Red Storm MPP Architecture," *Proc. Supercomputing 2002 Conf. High Performance Networking and Computing,* Nov. 2002; http://gaston.sandia.gov/cfupload/ccim_pubs_prod/camp.pdf.
2. R. Alverson, "Red Storm," in invited talk, Hot Chips 15, Aug. 2003; http://www.hotchips.org/archives/hc15/2_Mon/1.cray.pdf.
3. S.M. Kelly and R. Brightwell, "Software Architecture of the Light Weight Kernel, Catamount," *Proc. 2005 Cray User Group Ann. Tech. Conf.,* 2005, www.cs.sandia.gov/~smkelly/SAND2005-2780C-CUG2005-CatamountArchitecture.pdf.
4. R. Brightwell et al., "Portals 3.0: Protocol Building Blocks for Low Overhead Communication," *Proc. Workshop Communication Architecture for Clusters,* IEEE Press, 2002, pp. 164-173.
5. "MPI: A Message-Passing Interface Standard," *Proc. Int'l J. Supercomputer Applications and High Performance Computing,* MIT Press, 1994, pp. 159-416.
6. K.T. Pedretti and T. Hudson, "Developing Custom Firmware for the Red Storm SeaStar Network Interface," *Proc. 47th Cray User Group Ann. Tech. Conf.,* 2005, http://www.cug.org/5-members_only/1-attendees/proceedings_attendee_lists/2005CD/S05_Proceedings/pages/Authors/Pedretti/Pedretti_slides.pdf.
7. R. Brightwell, K. Pedretti, and K. Underwood, "Initial Performance Evaluation of the Cray SeaStar Interconnect," *Proc. 13th IEEE Symp. High-Performance Interconnects* (Hotl 05), IEEE CS Press, 2005, pp. 51-57.
8. F. Petrini et al., "The Quadrics Network: High-Performance Clustering Technology," *IEEE Micro*, vol. 22, no. 1, Jan.-Feb. 2002, pp. 46-57.
9. "Myrinet Express (MX): A High Performance, Low-Level, Message-Passing Interface for Myrinet," Myricom Inc., July 2003, http://www.myri.com/scs/MX/doc/mx.pdf.
10. Y.H. Song and T.M. Pinkston, "A Progressive Approach to Handling Message-Dependent Deadlock in Parallel Computer Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 14, no. 3, Mar. 2003, pp. 259-275.
11. R. Brightwell et al., "Implementation and Performance of Portals 3.3 on the Cray XT3," to be published in *Proc. 2005 IEEE Int'l Conf. Cluster Computing* (Cluster 2005), IEEE Press, 2005.
12. R. Brightwell, "A Comparison of Three MPI Implementations for Red Storm," *Lecture Notes in Computer Science,* LCNS 3666, B.D. Martino, D. Kranzlmuller, and J. Dongarra, eds., Springer Berlin/Heidelberg, 2005, pp. 425-432.
13. K.D. Underwood and R. Brightwell, "The Impact of MPI Queue Usage on Message Latency," *Proc. Int'l Conf. Parallel Processing* (ICPP 04), IEEE CS Press, 2004, pp. 152-160.
14. R. Brightwell and K.D. Underwood, "An Analysis of NIC Resource Usage for Offloading MPI," *Proc. 18th Int'l Parallel and Distributed Processing Symp.* (IPDPS 04), IEEE CS Press, 2004, pp. 183-190.

Ron Brightwell is a principal member of the technical staff at Sandia National Laboratories. His research interests include high-performance scalable communication interfaces and protocols for system-area networks and operating systems for massively parallel processing. Brightwell has a BS in mathematics and an MS in computer science from Mississippi State University. He is a member of the IEEE Computer Society and the ACM.

Kevin T. Pedretti is a senior member of the technical staff at Sandia National Laboratories. His research interests include low-latency, high-throughput hardware for MPI, and lightweight compute node operating systems. Pedretti has a BS and an MS in electrical and computer engineering from the University of Iowa. He is a member of the IEEE and the IEEE Computer Society.

Keith D. Underwood is a senior member of the technical staff at Sandia National Laboratories. His research interests include reconfigurable computing, the role of FPGAs in high-performance computing, and network (and network interface) architectures for high-performance computing. Underwood has a BS and a PhD in computer engineering from Clemson University.

Trammell Hudson is a senior partner at OS Research, a high-performance and embedded-system firm. His primary research interests are low latency, OS bypass message passing systems, and embedded Linux systems. Hudson has a BS in computer science from Tulane University. He is a member of AUVSI.

Direct questions and comments about this article to Ron Brightwell, Sandia National Laboratories, PO Box 5800, Albuquerque, NM, 87185-1110; rbbrigh@sandia.gov.