Portals 3.3 on the Sandia/Cray Red Storm System

Ron Brightwell Trammell Hudson[*] Kevin Pedretti Rolf Riesen Keith D. Underwood
Sandia National Laboratories[†]
PO Box 5800
Albuquerque, NM 87185-1110

Abstract

The Portals 3.3 data movement interface was developed at Sandia National Laboratories in collaboration with the University of New Mexico over the last ten years. Portals is intended to provide the functionality necessary to scale a distributed memory parallel computing system to thousands of nodes. Previous versions of Portals ran on several large-scale machines, including a 1024-node nCUBE-2, a 1800-node Intel Paragon, and the 4500-node Intel ASCI Red machine. The latest version of Portals is the lowest-level network transport layer on the Sandia/Cray Red Storm platform. In this paper, we describe how Portals are implemented for Red Storm and discuss many of the important features and benefits that Portals provide for supporting various services in the Red Storm environment.

1. Introduction

The Portals 3.3 interface [3] is an evolution of the user-level network programming interface developed in early generations of the lightweight kernel operating systems [5, 7] that were developed for large-scale massively parallel distributed memory parallel computers. Early versions of Portals did not have functional programming interfaces, which severely hampered an implementation for intelligent or programmable networking hardware. In order to better support platforms with intelligent and/or programmable network interface hardware, the Portals 3.0 functional programming interface was developed. This interface was specifically designed to meet the requirements of a large-scale distributed memory parallel computer, such as the Sandia/Cray Red Storm machine.

The Cray SeaStar interconnect [1] was developed as part of the Cray/Sandia Red Storm massively parallel processing machine [4]. Cray has since productized this design and is selling Red Storm based machines under the product name XT3. The SeaStar interconnect was designed specifically to support a large-scale distributed memory scientific computing platform. The network performance requirements for Red Storm were ambitious when they were first proposed. The network is required to deliver 1.5 GB/s of network bandwidth per direction into each compute node and 2.0 GB/s of link bandwidth per direction. This yields an aggregate of 3.0 GB/s into each node. The one-way MPI latency requirement between nearest neighbors is 2 μs and is 5 μs between the two furthest nodes.

In this paper, we describe the implementation of Portals 3.3 for the SeaStar network interface on Red Storm and provide an initial performance evaluation using low-level micro-benchmarks. Despite the fact that the software environment is currently under active development, the initial performance results are promising. Current bandwidth performance is higher than what can be achieved using a single interface of any current commodity interconnect.

The rest of this paper is organized as follows. The next section provides an overview of the SeaStar network hardware. Section 3 discusses the software environment and the implementation of Portals. A brief description of the performance benchmarks is presented in Section 5, while performance results are shown in Section 6. Relevant conclusions of this paper are presented in Section 7.

2. Hardware

The Cray SeaStar ASIC[1] in the Red Storm system was designed and manufactured by Cray, Inc. In a single chip, it provides all of the system's networking functions as well as all of the support functions necessary to provide reliability, availability, and serviceability (RAS) and boot services. The basic block diagram can be seen in Figure 1. Independent send and receive DMA engines interact with a router that supports a 3D torus interconnect and a HyperTransport cave that provides the interface to the Opteron processor.

[*] Under contract to Sandia via OS Research, Inc.

[†] Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

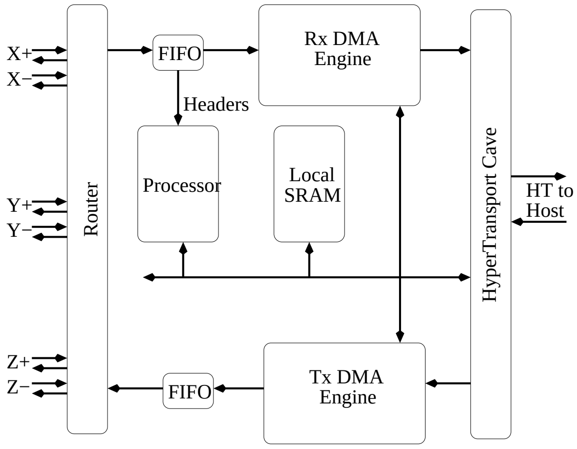

Figure 1. Basic SeaStar block diagram

An embedded PowerPC processor is also provided for protocol offload.

The DMA engines provide support for transferring data between the network and memory while providing support for the message packetization needed by the network. They also provide hardware support for an end-to-end 32 bit CRC check. This augments the extremely high reliability provided by a 16 bit CRC check (with retries) that is performed on each of the individual links.

The physical links in the 3D topology support up to 2.5 GB/s of *data payload* in each direction. This accounts for overhead in both the 64 byte packets used by the router and the reliability protocol on the individual links. The interface to the Opteron uses 800 MHz HyperTransport, which can provide a theoretical peak of 3.2 GB/s per direction with a peak payload rate of 2.8 GB/s after protocol overheads (and a practical rate somewhat lower than that).

The PowerPC processor is designed to offload protocol processing from the host processor. It is a dual-issue 500 MHz PowerPC 440 processor with independent 32 KB instruction and data caches. It must program the DMA engines since transactions across the HyperTransport bus require too much time to allow the host processor to program these engines. On the receive side, the PowerPC is also responsible for recognizing the start of new messages and reading the new headers. Finally, the PowerPC must recognize DMA completion events. To hold local state and handle interactions with the host, the PowerPC has 384 KB of scratch memory. This memory is protected by ECC complete with scrubbing. In this context, a certain portion of the network management *must* be offloaded to the NIC, but there is an opportunity to offload the majority of network protocol processing as well.

3. Portals

The Portals [2] network programming interface was developed jointly by Sandia National Laboratories and the University of New Mexico. Portals began as an integral component of the SUNMOS [5] and Puma [7] lightweight compute node operating systems. In 1999, an operational programming interface was created for Portals so that it could be implemented for intelligent and/or programmable network interfaces outside the lightweight kernel environment [3]. Portals is based on the concept of elementary building blocks that can be combined to support a wide variety of upper-level network transport semantics.

Portals provides one-sided data movement operations, but unlike other one-sided programming interfaces, the target of a remote operation is not a virtual address. Instead, the ultimate destination of a message is determined at the receiving process by comparing contents of the incoming message header with the contents of Portals structures at the destination. The following describes these structures.

3.1. Portals Objects

Portals exposes a virtual network interface to a process. Each network interface has an associated Portal table. Each table entry can be thought of as the initial protocol switch point for incoming messages. Portal table entries are simply indexed from 0 to n-1. The current implementation of Portals for the SeaStar has 64 Portal table entries. Portal table entries are somewhat analogous to UNIX well-known port numbers or can be thought of as tagged messages, where each tag can be used to implement a different upper-level protocol.

A match entry can be attached to a Portal table entry to provide more detailed message selection capability. When a message reaches a match entry, the following information in the message header is compared to the match entry:

- Source node id
- Source process id
- Job id
- User id
- 64 match bits
- 64 ignore bits

Source node id and process id provide the ability to select only those messages from a specific node or process. Likewise, job id and user id allow for matching only those messages from a specific job or user. The job id provides a way to aggregate a group of processes, such as those launched as part of the same parallel application. All four of these can be wildcarded to allow for matching against any node, any process, any job, or any user.

The match bits can be used to encode any arbitrary tag matching scheme. The ignore bits are used to mask bits that are not needed or to wildcard specific bits to match any of the match bits.

Match entries can be linked together to form a list of match entries, called a match list. When a message arrives at a Portal index with a match entry attached, information in the message header is compared to the information in the match entry. If the entry accepts the message, the message will continue to be processed by the memory descriptor (described below) that is attached to the match entry. If the entry does not match, the message continues to the next match entry in the list. A match entry also has the option of being automatically unlinked from the match list after it has been consumed. Unlinking frees all system resources associated with the match entry.

A memory descriptor can be attached to a match entry. A memory descriptor describes a region of memory and how this memory responds to various operations. The region of memory can be a single logically contiguous region or can be a list of regions within a process' address space. There are no restrictions on the alignment of the memory region or on the length.

Each memory descriptor has a threshold value that gets decremented for each operation performed on the memory descriptor. The memory descriptor becomes inactive, or non-responsive, when its threshold reaches zero. Threshold values can be any nonnegative integer value or a memory descriptor can be assigned an infinite threshold that allows for creating a persistent memory descriptor that will respond to an unlimited number of operations.

There is a number of options that indicate how a memory descriptor responds to incoming data transfer requests. Memory descriptors can be configured to respond only to get and/or put operations. Each memory descriptor also has an offset value associated with it. For memory descriptors that are locally managed, successive operations increase the offset by the length of the request. Memory descriptors may also be configured to have a remotely managed offset. In this case, the offset used in the operation is determined by the initiator of the operation.

Memory descriptors can also be configured to truncate incoming requests that are larger than the size of the memory region that the descriptor spans. By default, incoming messages that are larger than the described memory region are rejected by the memory descriptor. Allowing for truncation reduces the length of the incoming request to the only what the memory descriptor has available.

Memory descriptors with a locally managed offset also have an option, called the maximum size option, to become inactive when the amount of unused space in the memory region falls below a certain amount.

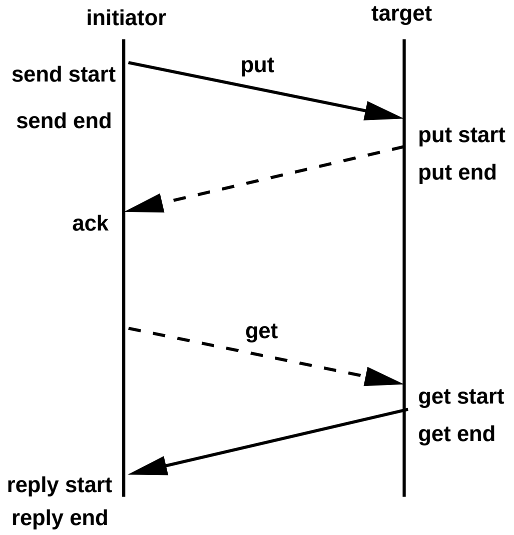

Figure 2. Portals events

By default, a memory descriptor automatically generates an acknowledgment to the originating process for every incoming successful put operation. The acknowledgment contains information about the result of the operation at the destination. In order for an acknowledgment to be generated, the initiator must request one and the target must be configured to generate one. Acknowledgments can be turned off by configuring the memory descriptor to suppress acknowledgments or by having the initiator not request one.

As with match entries, memory descriptors can also be configured to be freed when they become inactive, either via the threshold value or by the maximum size. When a memory descriptors becomes inactive, the associated match entry also becomes inactive. Memory descriptors need not always be associated with a match entry. Such free-floating memory descriptors can only be used by the source of a put operation or the target of a get operation.

A memory descriptor may have an event queue associated with it. An event queue is used to indicate the start and/or completion of an operation on a memory descriptor. An event queue can be associated with any number of event queues, but each memory descriptor may only be associated with a single event queue.

Each event queue is composed of individual events that are managed as a circular queue in a process' address space. There are several event types associated put and get operations. These events are illustrated in Figure 2. Most of these operations generate a start/end event pair that captures the state of the memory descriptor when the operation has begun and when it has completed. The memory descriptor from which a put operation has been initiated will generate a SEND event pair. The target of a put operation will generate a PUT event pair. If an acknowledgment was generated, a ACK event will also be generated at the initiating memory descriptor. For get operations, a REPLY event pair is generated at the initiator, while a GET event pair is generated at the target memory descriptor.

Split phase events are designed to preserve ordering of events, since two operations may start in order but complete out of order. The typical case is a very long message followed by a very short one. The long message will traverse the match list and memory descriptors first, so the start event associated with the long message will appear in the event queue first. However, it may take significantly longer to deliver the long message, so the end event for the short message may appear in the event queue before the end event for the long message. Split phase events can also be used to inform the user of network-related errors. An operation that has begun successfully may not be able to complete due to a catastrophic network failure. In this case, an end event can be generated that indicates that the completion of the operation was unsuccessful. In addition to the type of event, each event records the state of the memory descriptor at the time the event was generated. Individual memory descriptors can be configured so that neither start events nor end events are generated.

4. Portals Implementation

An initial reference implementation of Portals 3.3 was done by Sandia in 1999. This reference implementation was designed to be easily portable between different network and kernel architectures, with a mix of user-space, kernel-space, and NIC-space implementations as possible targets. As of this writing, there exist implementations of nearly all possible permutations of address spaces. It was not, however, designed to allow these address spaces to be mixed at runtime – the target was always a small number of applications per node all in the same address space.

The implementation of Portals for the SeaStar is based on this reference implementation developed by Sandia. The Red Storm system presented a novel requirement in that it needed to support several different systems with the same code base:

- Catamount compute nodes with ”generic” app and ”generic” pct
- Catamount compute nodes with ”accelerated” app and ”generic” pct
- Linux service nodes with many ”generic” yods and kernel level Luster
- Linux compute nodes with single user level app

The NIC firmware on the SeaStar for each of these is exactly the same, so it must be generic enough to support moving data into both user-level data buffers, kernel buffers and delivering notifications to user-level event queues and to a single kernel-managed event queue. The design and implementation of the firmware is discussed in [6].

The reference implementation has a network abstraction layer (NAL) that allows all implementations to share the same Portals library code, with the NAL providing the user API to library to NIC communications paths. In practice, each of these NALs share a large amount of code for moving data between address spaces. For instance, all Linux NALs that run user-level applications with the Portals Library in kernel-space will need to use the same address validation routines and routines to copy data between user- and kernel-space.

Since each of the different cases for Red Storm differs only in the communication path between the user-level API and the Portals library code, they can share all of the library to NIC methods of the NAL. Unfortunately, the existing reference implementation was not designed with this sort of sharing in mind. To abstract the separate communication paths, Cray designed a ”bridge” layer that sits atop the NAL and overrides the methods for moving data to and from API and library-space, as well as the address validation and translation routines.

Three bridges have been implemented:

- qkbridge for Catamount compute node applications
- ukbridge for Linux user-level applications
- kbridge for Linux kernel-level applications

Every NAL for Catamount, for instance, would share the same qkbridge code. Since the bulk of the API to library NAL is in the shared routines, this design allows very rapid development of new library to NIC NALs.

The ukbridge and kbridge are very interesting because they are able to run simultaneously on a single node. Since both bridges use the same library to NIC communication paths, both kernel-level applications and user-level applications are able to cleanly share the NIC.

The latest Portals reference implementation now supports a similar interface abstraction based on the success of the bridge approach. It is hoped that this will allow more rapid development of new NALs. Thanks to the removal of much of the complexity in writing a new NAL, we hope that this ease of development will allow Portals to become more widely used on different platforms.

4.1. SeaStar NAL

There are two primary constraints to consider in an implementation of Portals for the SeaStar. The first is the limited amount of memory available in the SeaStar chip. Limiting the design to only the 384 KB of SRAM that could be provided internally helps to improve reliability and reduce cost. Unfortunately, it also makes the offload of the entire Portals functionality somewhat challenging. The second constraint is the lack of any facilities to manage the memory maps for the small pages used by Linux.

In light of these constraints, the initial design of Portals for the SeaStar places relatively little functionality on the NIC. The NIC is primarily responsible for driving the DMA engines and copying new message headers to the host. When these new headers have been copied to the host, the host is interrupted to perform the Portals processing. In response, the host pushes down commands for depositing the new message. Similarly, on the transmit side, the host pushes commands to the NIC to initiate transfers. In both cases, the PowerPC is responsible for interpreting the commands and driving the DMA engines. When a message completes, the PowerPC must again interrupt the host to allow it to post the appropriate Portals event. All of this is handled by a tight loop that checks for work on the NIC and then check for work from the host.

The commands from the host to the NIC takes different forms depending on the operating system. Under Linux, the host is responsible for pinning physical pages, finding appropriate virtual to physical mappings for each page, and pushing all of these mappings to the NIC. In contrast, the Catamount light-weight kernel running on the compute nodes maps virtually contiguous pages to physically contiguous pages. This means that a single command is sufficient to allow the NIC to feed all necessary commands to the DMA engine.

The SeaStar NAL, or SSNAL, implements all of the entry-points required by a Portals NAL, including functions for sending and receiving messages. Additionally, SSNAL provides an interrupt handler for processing asynchronous events from the SeaStar. In this way, the platform-independent Portals library code can access the SSNAL through the common NAL interface and the SeaStar firmware can access platform-independent Portals functions (e.g., Portals matching semantics) through the interrupt handler. Our measurements indicate that a NULL-trap into the Catamount kernel requires approximately 75 ns of overhead—not a significant source of overhead. Interrupts, on the other hand, are very costly, requiring at least 2 μs of overhead each. Clearly, it will be necessary to eliminate all interrupts from the data path in order to meet the performance requirements of Red Storm.

In the future, a new implementation of Portals will be created to supplement the existing implementation. Much of the Portals library functionality, including matching, will be offloaded to the SeaStar firmware. This will allow arriving messages to be immediately processed, rather than waiting for the host to determine what actions to take. This implementation, referred to as *accelerated mode*, will enable user-level Portals clients to post commands directly to the firmware, without performing any system calls. Asynchronous events, such as Portals completion events, will be processed by polling when the user-level library is entered. The existing implementation, or *generic mode*, will continue to be necessary and will run side-by-side with the accelerated implementation. Limited NIC resources allow only a small number of accelerated-mode clients per node. Additionally, Linux nodes will continue to use generic-mode for the foreseeable future because accelerated mode will not support non-contiguous message buffers.

5. Benchmarks

Two different benchmarks were used to measure the performance of Portals on the SeaStar. The first benchmark was developed at Sandia as part of a set of Portals performance and correctness tests. The second benchmark is a port of the NetPIPE [8] to Portals. Both of these benchmarks provide some insight into the peak achievable performance of the SeaStar for various operations. This section describes these benchmarks in more detail.

5.1. PortalPerf Benchmark

To demonstrate the peak achievable performance of the SeaStar, we developed a Portals-level benchmark, PortalPerf, that measures latency and bandwidth for a put operation, a get operation, and bi-directional put operations.

To measure the latency and bandwidth of a put operation, we create a free-floating memory descriptor at both the initiator and target. We also create a single match entry and attach a single memory descriptor to it. All of these memory descriptors are persistent, so they are created once before the time-critical part of the benchmark is executed. These memory descriptors are used to implement a standard ping-pong performance test. Rank 0 starts a timer and initiates a put operation from the free floating memory descriptor, and rank 1 waits on an event queue until it receives a put end event. Upon receiving the event, rank 1 responds with a put operation back to rank 1. When rank 1 receives the put end event from rank 1's put operation, it stops the timer. One-way latency is calculated by dividing the total time taken by two. This ping-pong pattern is repeated 1000 times for each message size.

Performance of the get operation is measured similarly. Rank 0 creates a free-floating memory descriptor while rank

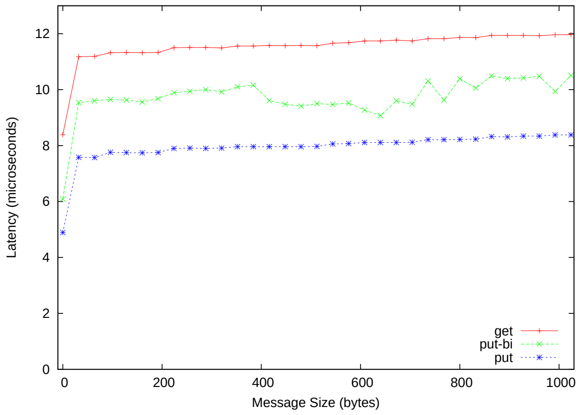

Figure 3. Latency performance

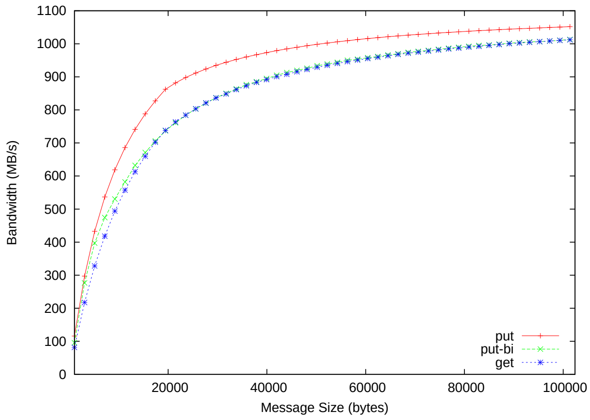

Figure 4. Bandwidth performance for medium-sized messages

1 attaches a memory descriptor to a match entry. Rank 1 starts a timer and initiates a get operation, waits for the get end event to arrive, and then stops the timer.

For a bidirectional put, both rank 0 and rank 1 initiate put operations and wait for both the send start/end event pair to arrive as well as the put start/end event pair that signifies the arrival of the incoming put message. The processes are first synchronized so that they are guaranteed to be sending data at the same time.

5.2. NetPIPE Benchmark

We developed a Portals-level module for NetPIPE version 3.6.2. As with the PortalPerf benchmark, this module creates a memory descriptor for receiving messages on a Portal with a single match entry attached. The memory descriptor is created once for each round of messages that are exchanged, so the setup overhead for creating and attaching a memory descriptor to a Portal table entry is not included in the measurement. Compared to PortalPerf, NetPIPE is a little more sophisticated in how it determines message size and number of exchanges for each test it conducts. Rather than choosing a fixed message size interval and fixed number of iterations for each test, it varies the message size interval and number of iterations of each test to cover a more disparate set of features, such as buffer alignment. NetPIPE also provides a performance test for streaming messages as well as the traditional ping-pong message pattern. The Portals module that was developed for NetPIPE allows for testing put operations and get operations for both uni-directional and bi-directional tests and for uni-directional streaming tests for gets and puts.

6. Results

Figure 3 shows latency performance for the put, get, and bi-directional put operations. Zero-length latency is 4.89 μs, 6.09 μs, and 8.39 μs respectively. A significant amount of the current latency is due to interrupt processing by the host processor. We expect that latency performance will improve as more of the message processing duties are offloaded to the SeaStar.

Figure 4 shows bandwidth performance for put, get, and bi-directional put operations. The bandwidth for a unidirectional put tops out at 1051.43 MB/s for a 100 KB message. The bandwidth curves are fairly steep, with half the bandwidth for a unidirectional put being achieved at a message of around 6 KB. The bandwidth for a get operation is only slightly less, 1012.17 MB/s for 100 KB message. We can also see that the impact of the bidirectional put on bandwidth is not that significant, since the link bandwidth is higher than the actual bandwidth into each node.

Figure 5 shows the bandwidth performance for very long message sizes, from 100 KB up to 2 MB. Bandwidth does not fall off as message size continues to increase. The asymptotic bandwidth achieved by the unidirectional put is about 1108 MB/s. Bandwidth performance in excess of 1500 MB/s has been measured on the system, but this performance currently is not consistent. We are currently working with Cray to investigate the source of this inconsistency.

Figure 6 presents latency results from the NetPIPE benchmark. Results for a standard put are included as well as the latency for streaming puts, which measures effective throughput. The streaming numbers are currently dominated by the interrupt overhead needed to process a message. At 12 bytes for the unidirectional tests and 24 bytes

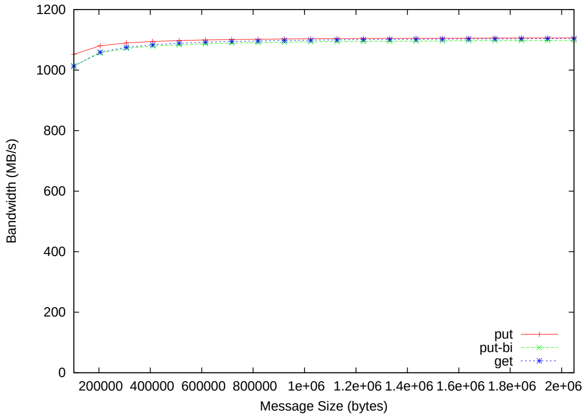

Figure 5. Bandwidth performance for long messages

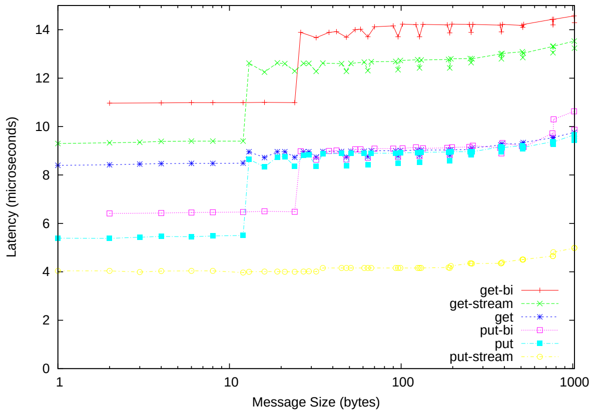

Figure 6. NetPIPE latency performance

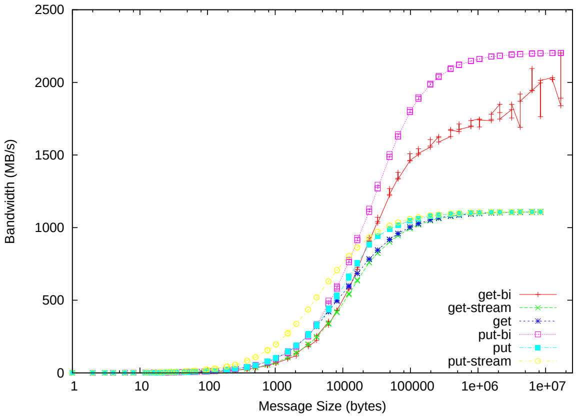

Figure 7. NetPIPE bandwidth performance

for the bidirectional tests, we see the results of a small message optimization currently in the firmware. Because 12 bytes of user data will fit in the 64 byte header packet, these 12 bytes can be copied to the host along with the header. This allows the new message and message completion notification to be delivered simultaneously and saves an interrupt. The cross-over occurs at 24 bytes for bidirectional tests because total data transferred is counted. For longer messages, the combination of the independent progress semantics of Portals with the processing of the message headers on the host requires that two interrupts be used — one to have the header processed and one to post a completion notification to the application. In the fully offloaded implementation, both interrupts will be eliminated as the NIC will process headers and will write completion notifications directly into process space.

Much as with the PortalPerf benchmark, Figure 7 indicates a steep growth in bandwidth. The bidirectional bandwidth delivers twice the unidirectional bandwidth with a peak at over 2200 MB/s of data payload. Half of the bandwidth is achieved at a respectable 8 KB in the unidirectional case, but that rises to nearly 24 KB in the bidirectional case. In both cases, we expect a dramatic decrease in the point at which half bandwidth is achieved as processing is offloaded from the host and the costly interrupt latency is eliminated.

7. Conclusion

This paper has described the implementation of Portals for the Cray SeaStar, the custom interconnect developed by Cray for the Red Storm and XT3 machines. The current implementation of Portals for the SeaStar does a limited amount of processing on the network interface and relies on the host to perform many of the message processing duties. This initial implementation has demonstrated respectable performance, with a zero-length half round trip latency of 4.89 μs and a peak bandwidth of over 1.1 GB/s. The software stack for both the generic mode (using the host CPU) and the accelerated mode (using the NIC CPU) are currently under active development. We expect both latency and bandwidth performance to increase for each mode over the next several months.

8. Acknowledgments

The authors 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] R. Alverson. Red Storm. In *Invited Talk, Hot Chips 15*, August 2003.

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

[3] R. Brightwell, W. Lawry, A. B. Maccabe, and R. Riesen. Portals 3.0: Protocol building blocks for low overhead communication. In *Proceedings of the 2002 Workshop on Communication Architecture for Clusters*, April 2002.

[4] W. J. Camp and J. L. Tomkins. Thor's hammer: The first version of the Red Storm MPP architecture. In *In Proceedings of the SC 2002 Conference on High Performance Networking and Computing*, Baltimore, MD, November 2002.

[5] A. B. Maccabe, K. S. McCurley, R. Riesen, and S. R. Wheat. SUNMOS for the Intel Paragon: A Brief User's Guide. In *Proceedings of the Intel Supercomputer Users' Group. 1994 Annual North America Users' Conference*, pages 245–251, June 1994.

[6] K. T. Pedretti and T. Hudson. Developing custom firmware for the Red Storm SeaStar network interface. In *Cray User Group Annual Technical Conference*, May 2005.

[7] L. Shuler, C. Jong, R. Riesen, D. van Dresser, A. B. Maccabe, L. A. Fisk, and T. M. Stallcup. The Puma operating system for massively parallel computers. In *Proceeding of the 1995 Intel Supercomputer User's Group Conference*. Intel Supercomputer User's Group, 1995.

[8] Q. O. Snell, A. Mikler, and J. L. Gustafson. NetPIPE: A network protocol independent performance evaluator. In *Proceedings of the IASTED International Conference on Intelligent Information Management and Systems*, June 1996.