A NIC-offload Implementation of Portals for Quadrics QsNet

Kevin Pedretti Ron Brightwell

Scalable Computing Systems Department
Sandia National Laboratories*
P.O. Box 5800
Albuquerque, NM 87185-1110
{ktpedre,rbbrigh}@sandia.gov

Abstract. The Portals data movement layer was specifically designed to support intelligent and/or programmable network interface cards, such as Quadrics QsNet. Portals provides elementary building blocks that can be combined to implement a variety of upper layer protocols. As such, it is general enough to support many different types of services that require data movement, such as MPI and parallel file systems. While the QsNet interface and its associated software stack were also designed to support a variety of upper layer protocols, there are significant differences in the approach taken to achieve generality. In this paper, we analyze the different capabilities offered by Portals and the QsNet network stack. We discuss the design and implementation of Portals for QsNet and present a performance comparison using micro-benchmarks. We analyze how the different approaches have impacted performance and discuss how future intelligent network interfaces may be able to overcome some of the current limitations.

1 Introduction

The Portals 3.3 interface [3] is an evolution of the user-level network programming interface developed in early generations of lightweight kernel operating systems [7,13] developed for large-scale massively parallel computers. Early versions Portals did not have functional programming interfaces, which severely hampered an implementation for commodity networking hardware. In order to better support commodity clusters and machines with intelligent and/or programmable network interface hardware, the Portals 3.0 functional interface was developed. This interface was specifically designed to meet the requirements of a large-scale parallel computer.

Portals are currently in use on the Cplant™ Linux clusters at Sandia National Laboratories. The Lustre parallel file system project is using Portals as its network transport interface [4], and Cray, Inc. is implementing Portals for the

* 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.

intelligent network interfaces on the upcoming Red Storm [1] machine. While Portals have been implemented on several different types of networks and for several different transport layers, there has not been a complete implementation that offloads all of Portals protocol processing to an intelligent commodity network interface.

We have created a user-level, NIC-offload Portals implementation for the QsNet network interface. In this paper, we analyze the different capabilities offered by Portals and the QsNet network stack. We discuss the design and implementation of Portals for QsNet and present a performance comparison using micro-benchmarks. We analyze how the different approaches have impacted performance and discuss how future intelligent network interfaces may be able to overcome some of the current limitations.

The remainder of this paper is organized as follows. The following section provides background information about Portals and QsNet, and Section 3 offers a detailed comparison of these two approaches. The implementation of Portals for QsNet is described in 4, while Section 5 presents performance results. The important conclusions of this paper are summarized in Section 6.

2 Background

2.1 The Portals 3.3 API

The Portals 3.3 API is composed of functions that create and manipulate elementary message passing building blocks. We have tried to design these building blocks and their associated operations so that they are flexible enough to support a wide variety of higher-level data movement layers. The following sections provide an overview of the basic objects and how they can be combined. For a more detailed discussion of Portals, see [3,12].

Portal Table The Portals library provides a process with access to a virtual network interface. Each network interface has an associated Portal table that contains a number of entries. Implementations may vary as to the maximum number of entries they support, but 64 is a required minimum. The table is simply indexed from 0 to n-1, and the entries in the table normally correspond to a specific high-level protocol. Portal table entries are somewhat like UNIX well-known port numbers. They provide an initial switch point to effectively separate messages intended for different protocols.

Match Entry A list of match entries can be attached to a Portal table entry to provide further message selection criteria. Messages coming to a specific Portal can be further selected based on the following criteria:

- Source node id
- Source process id
- Job id

- User id
- 64 match bits
- 64 ignore bits

The source node and process ids allow messages to be selected from a specific node or a specific process. The job id is a system-specific identifier that may be used to aggreagate processes. For example, all of the processes that are launched as part of a parallel job my be assigned the same job identifier by the system. The user id allows a process to choose messages based on the user from which they came. This can be useful for supporting utilities like a debugger, where a process may not have specific information about the process with which it wants to communicate when the communication channel is established. All of these identifiers can be individually "wildcarded" to allow for any node, any process, any job id, or any user id to match.

The match bits are 64 bits that can be used for further selection, and the ignore bits can be used to mask off bits that are not important. The match bits can be completely wildcarded by setting all the ignore 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.

Memory Descriptor A memory descriptor can be attached to a match entry. It describes a region of logically contiguous memory with an application's address space. The memory region is described by an address and a length in bytes. There are no restrictions on the alignment of the address or on the length. Each memory descriptor has a threshold value that determines how many operations can occur on the memory descriptor before it becomes inactive. A threshold value of one means that the memory descriptor can only be used for a single operation, after which it becomes inactive. Memory descriptors may also be given an infinite threshold value that allows for an unlimited number of operations to occur.

Memory descriptors have a number of options that can be enabled to increase their flexibility. Memory descriptors can be configured to only respond to put operations or get operations. Each memory descriptor has an offset value associated with it. For descriptors that are locally managed, the offset is increased by the length of each message that is deposited into the memory descriptor. In this way, a descriptor can received two messages in a row, and they will be placed in the user's memory one right after the other. The offset can also be remotely managed. This allows the initiator to determine what offset to use at the target. In this way, a sending process can explicitly indicate the desired offset for a message.

Memory descriptors can specify whether an incoming message that is larger than the length of the memory region should be truncated or rejected. By default, incoming messages that are larger than the memory region (possibly including the offset) are rejected. A descriptor can be configured so that the length provided in the incoming request can be reduced to match the memory available in the region.

A memory descriptor may be configured so that it will always respond to a message up to a maximum size. If the unused portion of a memory descriptor falls below this size, then the memory descriptor becomes inactive.

Memory descriptors by default generate acknowledgements to the process that initiated the operation. For example, an incoming put operation from a remote process can request that an acknowledgement be delivered to the originating process. The acknowledgment contains information about the result of the operation at the destination process. A memory descriptor may be configured so that it does not generate an acknowledgment.

Each memory descriptor can have a user-specified value associated with it. This value is the length of a pointer, and can be used to cache information with a memory descriptor.

As with match entries, memory descriptors can also be configured to be unlinked when they become inactive. A memory descriptor becomes inactive when its threshold reaches zero or when it is unable to accept a message of a maximum specified size. Should a memory descriptor become inactive and be unlinked, the associated match entry also becomes inactive. If the match entry is also configured to unlink when it becomes inactive, it is automatically removed from the list of match entries.

Event Queue Memory descriptors may have an event queue associated with them. Event queues are used to record operations that have occured on memory descriptors. Multiple memory descriptors can share a single event queue, but a memory descriptor may have only one associated event queue.

Event queues are composed of individual events kept in a circular buffer in the application's address space. There are six types of events that represent the operations that can occur on a memory descriptor. A memory descriptor that has responded to a get request will generate a GET event. A memory descriptor that has accepted a put operation will generate a PUT event. A memory descriptor that has responded to an atomic swap operation with generate a GETPUT event. A SENT event indicates that the memory region described by a memory descriptor used in a put operation can be manipulated. A message with return data from a get operation generates a REPLY event, while a sender can receive an ACK event from a put operation that requested an acknowledgement from the target process.

A pair of events is generated for every operation except the acknowledgment. The pair consists of a *start* event that signifies the beginning of the event and a *end* event that represents the completion of the event. Split phase events are designed to preserve ordering for events that may start in order, but may not

necessarily complete in order. For example, if a long incoming message is followed by a short message, the start event of the long message will be generated before the start event of the short message, but the end event for the short message may be generated before the end event of the long message. Split phase events are also used to signify network errors. It is possible that a communication operation has started successfully, but due to catastrophic network errors, cannot complete successfully. In this case, the end event will contain a network-specific flag indicating the type of failure that has occurred. In addition to the type of event, each event records the state of the memory descriptor at the time the event was generated.

2.2 Access Control

A process can control access to its Portals using an access control list. Each entry in the access control list specifies a process id, node id, job id, user id, and a Portal table index. The access control list is actually an array of entries. Each incoming request includes an index into the access control list (i.e., a "cookie" or hint). If the information in the incoming request doesn't match the information specified in the access control list entry or the Portal table index specified in the request doesn't match the Portal table index specified in the access control list entry, the request is rejected.

2.3 Quadrics QsNet

Quadrics QsNet is made up of two hardware building blocks: the Elan network interface and the Elite eight-port switch. A QsNet switching fabric is made up of Elite switches configured in a quaternary fat-tree topology. Except for root switches, each Elite in the fabric has four down-stream ports and four up stream ports. The downstream ports of the leaf Elites are each connected to an Elan network interface. All communication links in QsNet have a peak throughput of 340 MB/s in each direction (after protocol). This network topology ensures that full bi-section bandwidth is maintained for all cluster sizes.

Elan primarily consists of a system-on-a-chip ASIC and a 64 MB memory. The ASIC contains a number of functional units including a memory management unit (MMU), a remote DMA (RDMA) engine, and a SPARC-based programmable processor.

The Elan's MMU is used to extend the host's virtual memory system to the Elan. All required virtual to physical address translations are performed transparently by the Elan's MMU. To reduce PCI bus traffic, the Elan driver maintains copies in Elan memory of the page tables of each registered host process.

The RDMA engine can be programmed by a host process to reliably transfer data to or from a remote process. Programming the RDMA engine involves initializing a RDMA descriptor and submitting it to a command port on the Elan. Once the descriptor is submitted, the operation is carried out autonomously by the Elan. An RDMA descriptor specifies:

- Virtual address of a local buffer
- Virtual address of a remote buffer
- Rank of the target remote process
- Number of bytes to transfer
- Direction of transfer (read or write)
- Virtual address of a local completion event
- Virtual address of a remote completion event

The local and remote Elans utilize their MMUs to determine the physical pages making up the local and remote buffers. When an RDMA is complete, the local and remote events specified in the originating descriptor are set to notify the respective host processes.

The Elan's programmable processor can be utilized by individual host processes to offload higher-level protocol processing, such as MPI tag matching, onto the Elan. The MMU enables each Elan program (NIC threads) to execute in the protection domain of its parent process. This means that if a NIC thread causes a fatal exception, the host process that started the thread will also fault. NIC threads also have somewhat lower-level access to the Elan than is available to host processes. Raw QsNet network transactions can be constructed and sent using special assembly instructions. Unfortunately, this capability is not publically documented.

A suite of communication APIs accompany the QsNet hardware, including MPI, Cray SHMEM, Tports, and RDMA. The Tports interface was initially developed by Meiko for their supercomputing products circa 1994. The Meiko networking technology as well as this interface was carried forward by Quadrics into their QSNet [8] products. The Tports interface was created prior to the development of the MPI Standard, and has since been enhanced to better support an MPI implementation. Currently Tports is supported on both the QsNet (Elan-3) and QsNet II (Elan-4) products. MPI for both products is implemented on top of Tports.

In addition to Tports, Quadrics also provides a lower-level functional interface for accessing the network. The Elan-3 library provides RDMA primitives and an efficient event mechanisms that can be combined with a NIC-level thread to implement almost any type of network operation. All functional QsNet APIs, including Tports, are ultimately built using the primitives provided by the Elan-3 library.

3 Different Approaches

Portals was designed to provide a single, low-level data movement interface that meets the requirements of all networking services that need to be available on a compute node of a massively parallel machine. Our requirements are driven by the need to use a lightweight compute-node kernel that does not support a traditional IP-based networking stack. We are not able to focus only on providing support for MPI. Our requirements include functionality to support data transfer

between the components of the parallel runtime system for efficient parallel job launching, a parallel file system, remote procedure calls, and other tools, such as debuggers. As such, there are several features of Portals that are needed specifically to support this extended functionality.

Likewise, the software stack for the QsNet interconnect also is designed to support several of these different features and upper layer protocols. However, the Quadrics approach is much different from that of Portals. In this section, we compare and contrast the different ways in which general networking services are provided.

3.1 Network Protection

In order for a process to send messages over the QsNet, the process must first allocate a *capability* that grants access to the network and allows for communication with a given set of nodes. Capabilities are enforced at the hardware level. When a parallel job is started, capabilities for each of the processes are distributed to the corresponding set of nodes, and communication outside this set of processes is prohibited. This approach relies on an out-of-band mechanism to distribute capabilities.

In contrast, Portals is based on a model of doing receive-side checks and does not restrict the ability of a process to place messages onto the network. The access control list can be used to filter out unwanted messages, but it can also be used to accept requests from any other process, as is desirable for server processes.

These two approaches to network protection have different impacts. Since QsNet is able to support capabilities in hardware, the overhead of protection is minimal, but it must rely on a mechanism to handle the exchange of capabilities and revocation. The access control list approach of Portals incurs the overhead of a table lookup on each request that is received, but allows individual processes to be responsible for managing permissions.

3.2 Embedded Versus Explicit Functionality

One of the fundamental differences between the programming interfaces supplied by Quadrics and the Portals interface is the method of achieving generalized functionality. Since QsNet supports running a user-level thread on the network interface, the operations that can be performed to support data movement are essentially unlimited. Portals, on the other hand, has embedded a subset of general operations into the different Portals objects. These differing approaches are visible in several areas.

Multi-Protocol Support Portals supports multiple user-level protocols within a process via the Portal table abstraction. This way, an MPI implementation can allocate a set of entries for its use, an I/O library can allocate a different set for its use, and so forth. Match entries and memory descriptors can then be

put together to achieve the desired functionality. This building block approach supports multiple user-level protocols without being specific to any single protocol. For example, Portals can easily support offloading MPI matching semantics without being MPI-specific.

Quadrics takes a different approach to supporting multiple user-level protocols. The low-level building blocks provide only minimal functionality, but a user-level thread can be written to use these building blocks any number of ways. Specific threads can be written to handle each user-level protocol.

The ability to support running user-level threads on the network interface significantly increases the complexity of the QsNet network interface. Portals has attempted to capture a set of desirable semantics and embed this functionality into the different objects. This approach does not support the complete generality that Quadrics does, but it offers a rich set of features in a way that requires less complexity for hardware. A hardware implementation of Portals on a network interface would be much less complex than the current QsNet hardware. A drawback of the Portals approach is that a specific protocol may pay a performance penalty for functionality that it does not use.

Memory Descriptors The queued-based DMA abstraction of QsNet treats memory only as a target or source for data transfers. Any other operations associated with data transfers can be performed by the NIC-level thread. Conversely, Portals has bundled certain operations in the memory descriptor object. This approach complicates the memory descriptor object as well as the corresponding events that describe the outcome of these operations. For example, remote versus local offset management is at the discretion of the receiver and is likely to be consistent for a given protocol. A NIC-level thread can just perform the appropriate operation, but Portals must have a way to convey the desired semantics and convey the results of the operation.

Complex Matching Semantics The complex matching semantics in Portals are another example of where functionality has been embedded inside an object. The match entry list traversal semantics provide functionality that is generally appropriate for a wide range of data movement operations. However, this functionality can adversely affect performance when it is not needed. For example, traditional remote DMA semantics only specify a target address and do not require any type of message selection criteria. Even though matching criteria and match list traversal are not needed to meet RDMA semantics, the overhead of these operations will be incurred since the functionality is embedded in the object.

3.3 Network Failures

None of the low-level user interfaces for data movement on QsNet expose network failures. Instead, separate system processes monitor the state of the network and take appropriate action when errors are recognized [10]. In general, there is no

way for a user-level process using QsNet to recognize or recover from catastrophic network failures. Portals uses the split-phase event notification mechanism to allow the underlying transport layer to expose failures. Failure notification is probably unnecessary for most user-level protocols, such as MPI, but is necessary to support system-level services, such as file systems and parallel runtime systems. The failure semantics of Portals add to the overhead of generating events.

3.4 Portability

One of the main reasons that Portals encapsulates functionality into objects is to support portability and allow implementations of Portals for different high-speed networks. While Quadrics is able to depend on hardware support, Portals is designed to be able to leverage a certain amount of general hardware capabilities, but the design stops short of mandating any specific advanced hardware capabilities.

There are specific mechanisms in Portals that are intended to allow for optimizations, but, in some cases, these hints may actually degrade performance where no optimization is possible. For example, the memory descriptor threshold value was intended to provide a hint to the underlying transport layer as to the persistence of the memory being used. For networks such as QsNet that do not require memory regions to be explicitly registered and pinned, managing the threshold value for a memory descriptor simply adds overhead.

4 Portals Implementation

In spite of these different approaches, QsNet is still an appealing platform for implementing Portals. The programmable network interface and standard development environment make it easy to compile and run a thread on the QsNet NIC, called the Elan. This enables a user-level, NIC-offload Portals implementation, where Portals protocol processing is performed by the Elan instead of the host. The vendor's Tports API takes advantage of this capability to obtain excellent performance [5,9].

Despite these advantages, there are aspects of QsNet that complicate a Portals implementation as well. First, while the drivers and user-level libraries are open-source, the NIC firmware is not. This limits the level at which the hardware can be programmed. Second, user-level processes execute inside of a sandbox established by the runtime system when a parallel application is launched. The processes in a QsNet parallel application may only communicate with one another. This prevents a general-purpose Portals implementation where any process can communicate with any other process. Third, it is not possible to have an Elan NIC thread service multiple user-level host processes without going through the kernel (this would violate virtual memory protections). Therefore, each Portals process must start its own NIC thread, which competes with other threads for NIC processor time.

For a prototype QsNet Portals implementation, these complications are not significant roadblocks. In fact, using one NIC thread per-process actually simplifies development, since there is no need to worry about crashing the NIC, the host, or both (the NIC thread executes in the parent process's protection domain). A production quality QsNet Portals implementation would need to find a way to break out of the communication sandbox to be useful for global services such as filesystems and job launchers. A possible work-around for this problem is to use so-called "hand-rolled capabilities" to manually setup which QsNet processes can communicate.

A number of experiments were conducted to determine the best way to utilize the QsNet NIC for a NIC-offload Portals implementation. The two major choices were to offload all of Portals onto the NIC or to only offload a portion of it. Offloading all of Portals would theoretically result in the lowest host overhead, however experiments showed that the round trip time to send commands over the PCI bus to the NIC and process them there was too high (the NIC processor is slow relative to the host CPU). Instead, a partial-offload approach was chosen where only receive processing is performed on the NIC. This approach has the potential for lower latency because many Portals operations can be performed entirely on the host, allowing the NIC thread to focus solely on receive processing.

4.1 Design

The Portals API defines two basic data movement operations which must be implemented: PUT and GET. These both have asynchronous semantics, meaning that an application may continue to perform computation after initiating an operation. The Portals Library notifies the application when an operation is complete by posting an END EVENT into a Portals Event Queue (EQ).

The Put operation is implemented on QsNet by two protocols. The first, is a short message protocol that is optimized for transferring small messages with very low latency. The second is a long message protocol that uses a zero-copy rendezvous protocol optimized for transferring large messages with high bandwidth and low host processor overhead. The long message protocol is also used for implementing the Portals GET operation. The following sections describe these protocols in more detail.

Both of these protocols utilize the remote DMA (RDMA) primitives provided by the Elan-3 library [11] for communication. The Elan-3 library provides the lowest user-level access to the QsNet NIC and therefore has the potential for the highest performance. In general, programming to the Elan-3 library API should be avoided since it will not be carried forward to future Quadrics' products. However, in our case, it is necessary to use the Elan-3 library because it is the only functional interface that provides direct access to the Elan's RDMA engine.

In additional to traditional RDMA read and write, Elan provides a queued RDMA write operation for efficiently sending small messages to a queue on a remote NIC. This capability distinguishes QsNet from strictly RDMA based interconnects, such as Infiniband, which must resort to host-based, round-robin polling techniques to process small messages efficiently [6].

4.2 Short Message Protocol

The short message protocol is used for PUTs of up to 288 bytes, which is a hard limit imposed by the maximum size of a queued RDMA on QsNet (320 bytes) and the size of the Portals control header (32 bytes). A high-level overview of the protocol is shown in Figure 1. The initiator begins by issuing a queued RDMA to transfer the message data and a control header to a queue in the target NIC's memory (1). Once the RDMA is complete, the NIC firmware sets an event on the initiator and wakes up the Portals NIC thread that was sleeping on the queue (2). If the initiator did not request an explicit ACK [1], the operation is complete from the initiator's point of view. At the target, the NIC thread attempts to match the new message to a buffer that was previously setup by the target (3). The matching operation involves sequentially comparing the tag specified in the message header to each element in a linked list of previously posted buffers. If no matching buffer is found, the NIC thread simply discards the message and, if the initiator requested an explicit ACK, sets a NAK event on the initiator to indicate that the message has been dropped and that the operation is complete (5). If a match is found, the message data is copied across the PCI bus into the next slot of the Portals EQ associated with the matched buffer (4). The NIC thread also advances the EQ's tail pointer to post the Portals END EVENT. The next time the host application checks the EQ, the host-side Portals library will copy the message data from the event queue slot to its final destination (i.e., the matched buffer). The intermediate copy through the EQ slot is required to meet alignment requirements which allow the message to be copied across the PCI bus in an efficient burst operation. Finally, if the initiator requested an explicit ACK, the NIC thread sets an ACK event on the initiator to indicate that the operation is complete (5).

4.3 Long Message Protocol

The most significant difference between the long and short message protocols is that data is transfered autonomously by the QsNet's RDMA engine instead of in-line with the control header. Using the RDMA engine is intended to lower host CPU overhead and increase bandwidth because it moves data directly from source to sink without any intermediate buffer copies (i.e., it is a zero-copy operation) and without any intervention from the host CPU. The RDMA engine is not used for data transfer in the small message protocol because it requires considerable overhead to set up.

A high-level overview of the long message protocol PUT is shown in Figure 2. The initiator begins by issuing a queued RDMA to transfer the control header to a queue in the target NIC's memory (1). Once this completes, the NIC firmware sets an event on the initiator and wakes up the Portals NIC thread that was sleeping on the queue (2). Since queued RDMA operations may be reordered

[1] This is the common case. MPI over Portals implements small sends by a Portals PUT with no ACK.

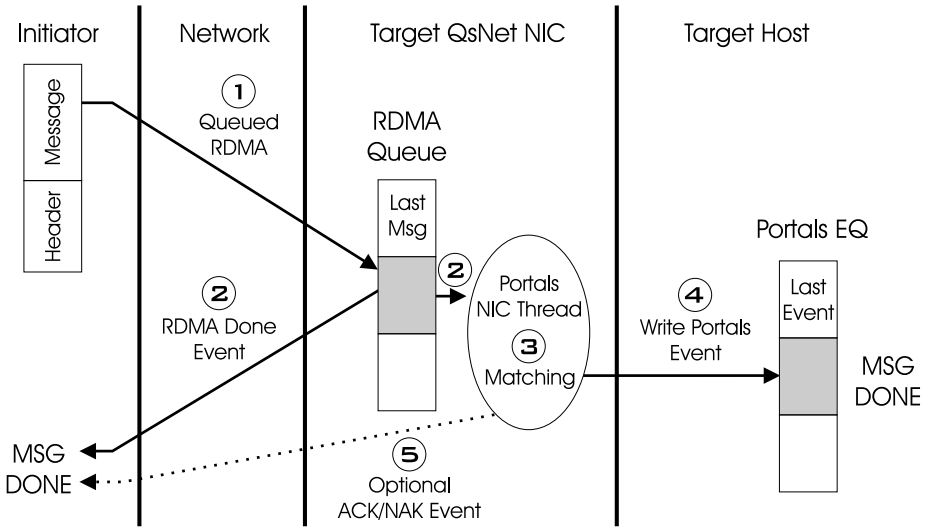

Fig. 1. Short Message Protocol

and Portals requires in-order delivery of messages, the initiator must be careful to not start a new operation until after the previous operation's header (if any) has been queued. At the target, the NIC thread performs matching in exactly the same way as in the small message protocol (3). If no matching buffer is found, the NIC thread discards the header and sets a NAK event on the initiator to indicate that the operation is complete (not shown). If match is found, the NIC thread sets up the Portals END EVENT in the next open slot of the EQ associated with the matched buffer (4). The NIC thread only advances its local copy of the EQ's tail pointer, leaving the host's copy unchanged. This effectively reserves the slot, blocking the EQ until the RDMA engine has finished moving the message data. Next, the NIC thread programs the local RDMA engine to move data from the source buffer to the sink buffer and to set events on the initiator and target when the transfer is finished (5). When the specified target event is set the Portals END EVENT setup by the NIC thread is posted (6), the Portals EQ becomes unblocked, and the PUT operation is complete from the target's point of view. Similarly, when the specified initiator event is set (6), the PUT operation is complete from the initiator's point of view.

The Portals Get operation is implemented in the same way as shown in Figure 2, except the RDMA engine moves data in the opposite direction.

5 Performance

In this section, we compare the latency and bandwidth of Portals, Tports, and the lowest-level Elan-3 put and Elan-3 get interface for QsNet. While not measured, MPI-level performance for Portals and Tports should be similar to the

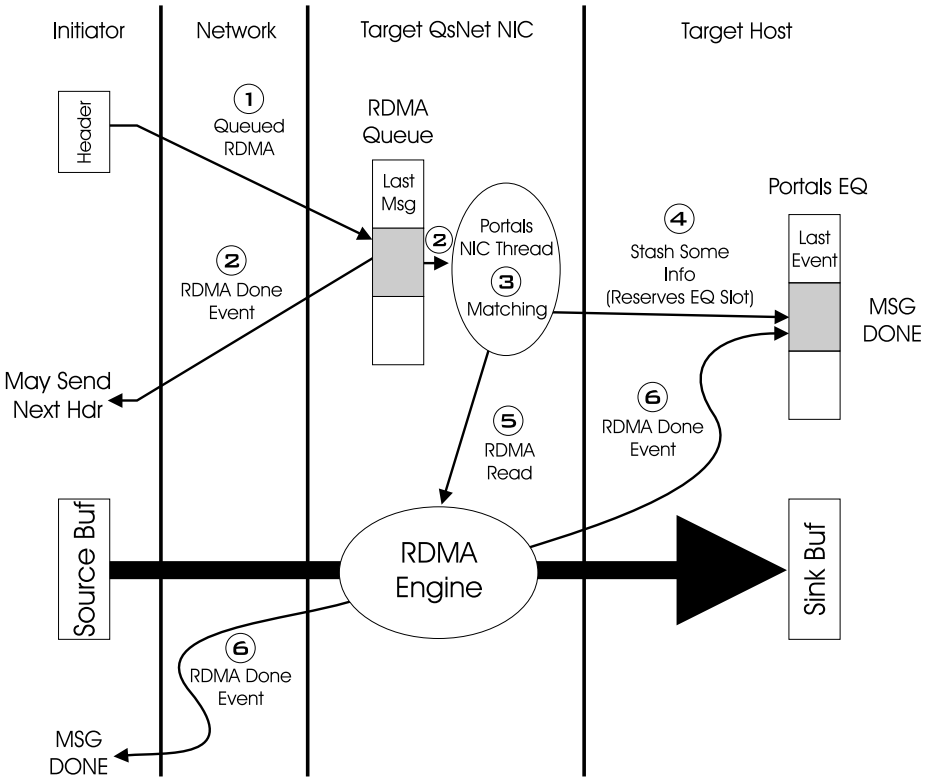

Fig. 2. Long Message Protocol

results presented here since both of these APIs have very "thin" MPI implementations built on top of them.

Performance was evaluated using a simple ping-pong micro-benchmark. Each iteration begins by an initiator submitting an operation (Put or Get) targeting the target. When the target receives notification that the initiator's operation has completed, it submits an operation of the same type and size targeting the initiator. The iteration is complete when the initiator is notified that the target's operation has completed. One-way latency is obtained by dividing the measured iteration time by two. Similarly, one-way bandwidth is measured by dividing the operation size by the one-way latency. It is important to note that setup time (e.g., registering memory, posting a receive, etc.) is included in the iteration time. Additionally, the ping-pong is designed to ensure that no unexpected messages are received.

The ping-pong results for Portals, Tports, and put/get were obtained using two nodes of a four node development cluster at Sandia National Laboratories. This cluster is made up of Dell 1550 nodes (Serverworks HE-SL chip-set) each containing two 1 GHz Pentium III processors, 2 GB of main memory, and a QsNet adapter. The QsNet adapter is attached via a 64-bit, 66 MHz PCI slot having a theoretical maximum bandwidth of 532 MB/s. QsNet has a theoretical maximum bandwidth of 340 MB/s.

5.1 Latency

Small message PUT latency is shown in Figure 3. QsNet Portals has approximately 15 μs higher latency than Tports for all message sizes measured. The jump in Portals PUT latency after the 32 byte message size is a result of an optimization in the short message protocol for very small messages. This optimization treats the Portals header and up to 32 bytes of data as a single 64 byte object. This simplifies housekeeping and allows efficient 64 byte block copies to be used across the PCI bus and the QsNet network. Tports employs a similar optimization. The jump in Portals PUT latency after 288 byte messages represents the transition from the short to long message protocol. Tports, which uses the same threshold, exhibits a similar discontinuity.

To get a better understanding of where the increased Portals latency was coming from, both QsNet Portals and Tports were instrumented. From the initiator's perspective, each iteration of the ping-pong consists of the operations shown in Table 1. Similar operations are performed by the target except in a different order.

As can be seen from Table 1, each operation requires considerable more time using Portals. There are a few explanations for the differences.

First, the 'Post receive for pong' operation requires two Portals API calls (one to attach a 'match entry' to the receive queue, and one to attach a buffer to the match entry) that are currently each implemented in a way that requires multiple PCI bus accesses. Tports only requires one API call for this operation, resulting in roughly half the number of PCI bus accesses. It should be possible to optimize the QsNet Portals implementation by delaying the PCI accesses

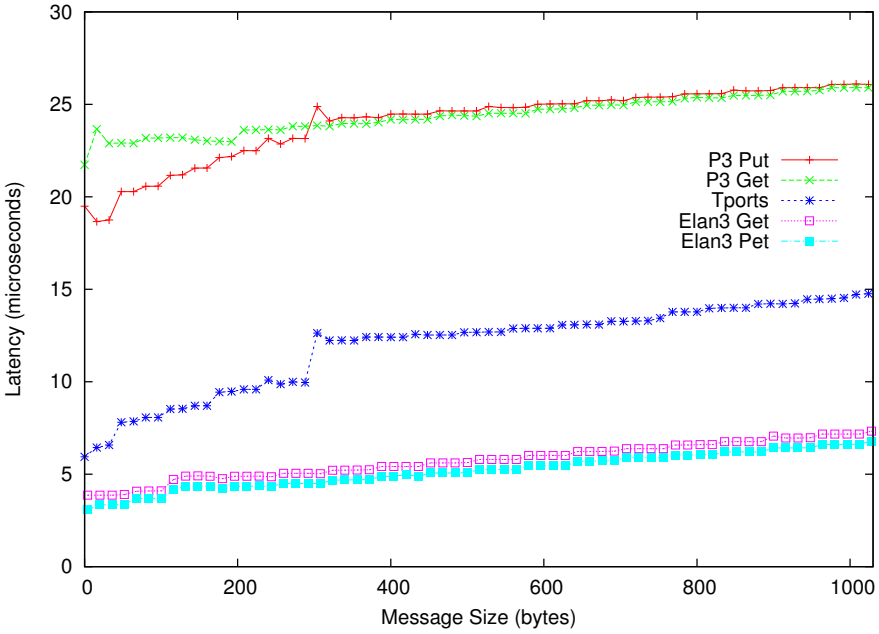

Fig. 3. Latency

	Portals	Tports
Post receive for pong	6.4	1.7
Send ping	2.2	1.0
Wait for send to complete	2.8	0.1
Wait for pong to arrive	24.7	8.5
Total:	36.1	11.3

Table 1. Pingpong Timing for Initiator (μs)

	Portals	Tports
Take lock	0.2	0.1
Inspect header	0.4	0.1
Match	1.4	0.3
Housekeeping	0.7	0.2
Write completion event	0.7	0.5
Unlink matched buffer	1.3	0.1
Drop lock	0.1	0.1
Total:	4.9	1.5

Table 2. NIC Thread Timing (μs)

until all information needed to post a receive is available. This optimization is expected to roughly halve the time needed to post a receive using QsNet Portals.

Second, the 'Wait for send to complete' operation is considerably more complex for Portals because of the large Portals event structure (128 bytes) and the Portals API requirement that two events be posted to a queue (one indicating the start of the send and another indicating the completion of the send). Tports, by contrast, must only perform a single word copy to indicate that the send has completed. These results suggest that it would be worthwhile to examine ways to make the Portals API event mechanism more lightweight. A recent version of the Portals API specification does provide a way to turn off 'start' events, meaning that only one event would need to be posted to indicate the completion of the send. Early experiments show that this roughly cuts in half the time needed for the 'Wait for send to complete' operation.

Lastly, the processing performed by the Elan NIC thread is more involved for Portals compared to Tports. Table 2 presents the results of instrumenting the Portals and Tports NIC threads. The timing is for the critical code path taken when a 0 length message arrives. At a high level, both NIC threads must lock the pre-posted buffer list, inspect the arriving message's header, search a list of pre-posted buffers for a matching buffer, set a completion event, and drop the pre-posted buffer list lock. The results indicate that Portals requires 1.1 μs more to match a pre-posted receive than Tports. This can be attributed to the Portals API's more complex matching semantics which contains more match bits than Tports and threshold, offset, and permission checks that Tports does not have. Additionally, the extra housekeeping that Portals requires after finding a match to update threshold counts and increment buffer offsets requires additional processing. The other significant difference is in the time needed to unlink a matched buffer. Tports can make certain optimizations because a matched buffer is always unlinked from the pre-posted buffer list. Portals does not have this requirement. The current QsNet Portals implementation performs a PCI write to notify the host when a buffer is unlinked. Tports does not need to perform any PCI accesses to unlink a buffer, and is therefore considerably faster.

For completeness, the latency for small GET operations is also shown in Figure 3. Since Tports is a message passing interface, it does not have a GET operation, so no direct comparison can be made.

5.2 Bandwidth

Bandwidth for PUT operations is shown in Figure 4. Generally, Tports has higher bandwidth than Portals for all message sizes until an asymptote is reached at approximately 307 MB/s. The difference is a result of the higher per-operation overhead of Portals compared to Tports.

6 Conclusion

The design and performance of a NIC-offload Portals API implementation for QsNet has been presented. QsNet has been shown to be a good match for Por-

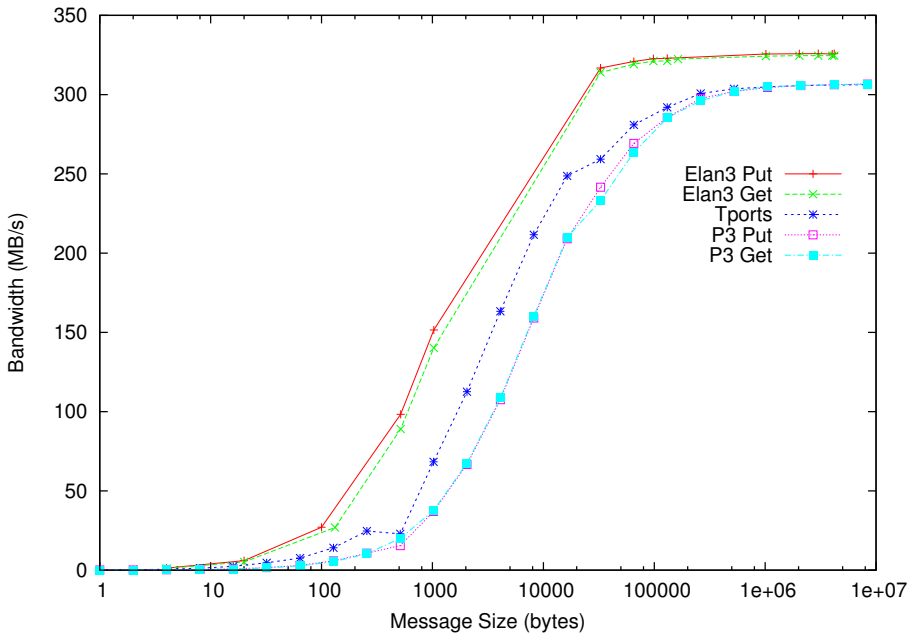

Fig. 4. Bandwidth

tals. On the other hand, Portals is not able to achieve as low of latency as the vendor's Tports API. Careful tuning of Portals, similar to what the vendor has presumably spent a great deal of effort performing for Tports, would likely result in reduced latency; perhaps by a factor of two. However, the increased capabilities and semantics of Portals compared to Tports means that Portals will always have somewhat higher latency. The next generation of QsNet, QsNetII, provides additional capabilities which are expected to help mitigate the difference [2]:

- Higher Performance PCI-X Interface
  Portals copies more control information across the PCI bus than Tports. Any reduction in PCI latency should benefit Portals more than Tports.
- Faster NIC Processor
  Portals matching is much more complex than Tports matching. The programmable processor on the QsNetII NIC is at least twice as fast as QsNet's.
- Richer Event Processing
  The QsNetII DMA engine is able to write completion events of up to 2 kb (vs. 64 bytes currently), big enough to handle an entire Portals event (> 128 bytes). The DMA engine may also be able to write events into a queue, rather than to a fixed address. This may allow a more direct mapping of QsNet completion events to Portals event queues.

We hope to explore Portals on QsNetII as soon as development hardware is available.

References

1. Bob Alverson. Red Storm: A 10,000 Node System with Reliable High Bandwidth, Low Latency, Interconnect. In *Fifteenth Symposium on High-Performance Chips*, August 2003.
2. Jon Beecroft, David Addison, Fabrizio Petrini, and Moray McLaren. QsnetII: An interconnect for supercomputing applications. In *Hot Chips 15*, Stanford, CA, August 2003.
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. Cluster File Systems, Inc. *Lustre: A Scalable, High-Performance File System*, November 2002. `http://www.lustre.org/docs/whitepaper.pdf`.
5. Jiuxing Liu, Balasubramanian Chandrasekaran, Jiesheng Wu, Weihang Jiang, Sushmitha Kini, Weikuan Yu, Darius Buntinas, Pete Wyckoff, and Dhabaleswar K. Panda. Performance comparison of mpi implementations over infiniband, myrinet and quadrics. In *Supercomputing 2003 Conference*, Phoenix, AZ, November 2003.
6. Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Pete Wyckoff, and Dhabaleswar K. Panda. High performance RDMA-based MPI implementation over InfiniBand. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 295–304, 2003.
7. Arthur B. Maccabe, Kevin S. McCurley, Rolf. E. Riesen, and Stephen. R. Wheat. SUNMOS for the Intel Paragon: A Brief User's Guide. In *Proceedings of the Intel Supercomputer Users' Group Annual North America Users' Conference*, pages 245–251, June 1994.
8. Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The Quadrics network: High-performance clustering technology. *IEEE Micro*, 22(1):46–57, January/February 2002.
9. Fabrizio Petrini, Salvador Coll, Eitan Frachtenberg, and Adolfy Hoisie. Performance evaluation of the quadrics interconnection network. *Journal of Cluster Computing*, 6(2):125–142, April 2002.
10. Quadrics. *RMS Reference Manual Version 1.1*, November 2003. `http://web1.quadrics.com/onlinedocs/Linux/Eagle/html/RMSRefMan/index.html`.
11. Quadrics Ltd., Bristol, England, UK. *Elan Programming Manual*, 4.1 edition, May 2002.
12. Sandia National Laboratories, Albuquerque, NM, USA. *The Portals 3.3 Message Passing Interface, Revision 1.0*, 1 edition, May 2003.
13. P. Lance Shuler, Chu Jong, Rolf E. Riesen, David van Dresser, Arthur B. Maccabe, Lee Ann Fisk, and T. Mack Stallcup. The Puma Operating System for Massively Parallel Computers. In *Proceedings of the 1995 Intel Supercomputer User's Group Conference*. Intel Supercomputer User's Group, 1995.