SAND REPORT

SAND2001-3387
Unlimited Release
Printed November 2001

Experiences with Prototype InfiniBand™ Hardware

James A. Schutt

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

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.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/ordering.htm

SAND2001-3387
Unlimited Release
Printed November 2001

Experiences with Prototype InfiniBand™ Hardware

James A. Schutt
Advanced Networking Integration Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-0806

Abstract

This report describes testing of prototype InfiniBand™ host channel adapters from Intel Corporation, using the Linux® operating system. Three generations of prototype hardware were obtained, and Linux device drivers were written which exercised the data movement capabilities of the cards. Latency and throughput results obtained were similar to other SAN technologies, but not significantly better.

Acknowledgements

I would like to thank Milt Clauser, 9227, for his role in facilitating the initial relationship with Intel, and for causing Sandia to become a member of the InfiniBand Trade Association.

I would like to thank Peter Molnar of Intel Corporation for helping to make prototype hardware available to Sandia National Laboratories.

Contents

Introduction

This report describes work funded under the ASCI Problem Solving Environment (PSE) sub-project High-Speed Interconnect (HSI).

When this work was begun in FY00, we had identified an emerging technology, the InfiniBand™ Architecture (IBA), which had the potential to address Sandia needs in massively parallel cluster computing. In particular, it seemed to have the potential of unifying the infrastructure for high performance messaging and high performance storage. Since every major computing vendor was a member of the InfiniBand Trade Association, there seemed to be some hope for InfiniBand to become a true commercial off-the-shelf (COTS) technology, with interoperability and enough competition to keep prices low.

At the same time, we had identified a need for the Advanced Networking Department to have more Linux® kernel expertise. We had observed, for example, that network throughput didn't seem to be increasing at the same rate that processing power and network bandwidth were increasing, and the kernel (i.e. device drivers, protocol stacks) is where these meet. Linux is the operating system of choice for scientific cluster computing, with a large development community outside of Sandia. Although Linux development proceeds at a furious pace, without local kernel expertise we have no choice but to wait for others to develop solutions to our problems. With local expertise, there is some hope of tracking the latest developments and adapting them to meet Sandia's needs.

Finally, we felt that the Advanced Networking Department had inadequate knowledge of Portals, the underlying messaging technology used in various versions on Sandia's Cplant™ and TFLOPS massively parallel computers.

In order to gain detailed knowledge of the InfiniBand Architecture, while addressing our need to build Linux kernel expertise and learn more about the details of Portals, we developed the following plan. We would obtain prototype InfiniBand hardware, write a minimal Linux driver that exercised the data movement capabilities of the hardware, and investigate the possibility of porting Portals over InfiniBand. This report describes what was learned in the process of doing so.

The InfiniBand Architecture

The definitive description of the InfiniBand Architecture is the InfiniBand Architecture Specification, Volumes 1 and 2 [1]. As of this writing, the current release of the specification is 1.0.a, and is available free of charge from the InfiniBand Trade Association. As the full specification comprises 1500 pages, this report cannot hope to discuss all of the features of IBA which are pertinent to this work, or to Sandia's needs. The goal of this section is to introduce enough of the concepts and terminology to make the rest of the report intelligible. Interested readers should consult the full specification.

As a predecessor to the IBA specification, the Virtual Interface Architecture (VIA) specification [2] contained many of the concepts found in IBA. It was succeeded and expanded upon by the Next Generation I/O (NGIO) specification, which to my knowledge was not publicly released. The NGIO supporters and those of a competing specification, Future I/O (also not publicly released, to my knowledge), agreed to merge their specifications around August or September 1999. The result was first called System I/O, and ultimately became the InfiniBand Architecture. The first publicly available IBA specification was released in October 2000.

The InfiniBand Architecture is a system area network technology based on a point-to-point, switched fabric. End nodes in the fabric can be host computers, or I/O devices such as disk controllers, network interfaces, graphics controllers, etc. The fabric comprises one or more IBA switches, which may be cascaded as the size of the fabric is scaled up. End nodes interface to the fabric using a channel adapter, of which there are two types. A host computer uses a host channel adapter (HCA), which must support the full functionality of the IBA. I/O devices may use a target channel adapter (TCA), which are only required to support a subset of the functionality of the IBA.

The IBA specifies both copper and optical links. The signaling rate of all links for Release 1.0 of the specification is 2.5 Gb/s. Standard 8B/10B transmission encoding is used, which results in a data rate of 250 MB/s. All links are full-duplex, and come in three widths, 1X, 4X, and 12X. A 1X link is one full-duplex link, or physical lane. A 4X link is four parallel full-duplex physical lanes, and a 12X link is twelve parallel full-duplex physical lanes. For the 4X and 12X links, data is byte-striped across the physical lanes. A 1X optical link has a short-reach (SX) option, which operates at a wavelength of 850 nm, and a long-reach (LX) option, which operates at 1300 nm. Optical 4X and 12X links have only an SX specification.

The InfiniBand Architecture specifies a transport mechanism that can support messaging semantics, remote direct memory access (RDMA) semantics, and atomic operation semantics. The transport services supported by the IBA are shown in Table 1. The IBA is designed to support both reliable transport services and unreliable transport services. For either reliable or unreliable transport, the IBA specifies both a connection-based transport service, in which two endpoints communicate exclusively with each other, and a datagram-based transport service, in which an endpoint can communicate with multiple other endpoints.

Table 1. InfiniBand Service Types

Service Type	Connection-based	Reliable	Transport
Reliable Connection	yes	yes	IBA
Unreliable Connection	yes	no	IBA
Reliable Datagram	no	yes	IBA
Unreliable Datagram	no	no	IBA
Raw Datagram	no	no	raw

In addition, the IBA supports a raw datagram service. Two types of raw datagrams are supported, IPv6 and Ethertype. Raw datagrams can be sent to IBA routers that can forward them to non-IBA destinations on a different fabric technology. In addition, the IBA also provides a multicast service, where sending to a single destination multicast address results in data arriving at multiple hosts.

The choice of the word “datagram” to designate a transport service in which an endpoint can communicate with multiple other endpoints is perhaps unfortunate. In the Internet Protocol (IP) arena, “datagram” is most often associated with the User Datagram Protocol (UDP) [3], which is an unreliable transport (however, see RFC-1151 [4] and RFC-908 [5]). In the IBA specification, however, both reliable and unreliable versions of the datagram transport service are included.

Note that many features of the IBA are required for a device to be considered IBA-compliant, but some are optional. For example, an HCA is required to support reliable connected, unreliable connected, and unreliable datagram transport services, and may optionally support reliable datagram service. On the other hand, a TCA is only required to support the unreliable datagram transport service, and may optionally support any of the others.

The IBA specification uses the phrase Channel Interface (CI) to describe the presentation of access to channel adapters, and a user of the Channel Interface is termed a Consumer. The IBA specification describes the semantic behavior of an IBA-compliant host Channel Interface through the Software Transport Verbs.

It is important to understand that the Verbs do not specify an application programming interface (API). The Verbs only describe how a consumer interacts with a Channel Interface, e.g. what sort of events can occur, and what data is needed to describe various events and actions. The API used to access a Channel Interface is determined by the supplier of the HCA and CI.

Also, the IBA specification does not specify the protocols that IBA devices use to exchange data. For example, the IBA has nothing to say regarding how a host operating system interacts with an IBA-connected RAID controller, except that it will involve some

combination of IBA transport services, such as messaging and RDMA over a reliable connection. However, other standards bodies may have efforts to standardize common applications of InfiniBand. For example, there is a draft standard in the National Committee for Information Technology Standards (NCITS) technical committee T10[1] called "SCSI RDMA Protocol" which addresses the transport of the SCSI protocol over InfiniBand.

Both the VIA and IBA specifications allow implementations to provide user-level access to a HCA without kernel intervention. The IBA specification is clearest on this point, because its Software Transport Verbs description spells out the level of access intended for consumers of each Verb. Only nine Verbs are specified to allow user-level consumer access: Create Address Handle[2], Modify Address Handle, Query Address Handle, Destroy Address Handle, Bind Memory Window[3], Post Send Request, Post Receive Request, Poll for Completion, and Request Completion Notification. All other Verbs are specified to have privileged, or kernel-level, consumer access.

A consumer moves data through a channel adapter using a send/receive work queue (WQ) pair. For example, when a consumer is employing messaging semantics, it may queue a work request on the send queue describing data to be sent, and a work request on the receive queue describing where to land expected incoming data. For RDMA semantics, both RDMA read and RDMA write requests are posted on the send queue. The channel interface transcribes information from the work request that is posted by a consumer into a work queue element that is accessible only to the channel interface and channel adapter.

Processing begins on work requests submitted to a single queue in the order that they were submitted. In general, work requests submitted to a single queue complete in the order that they were submitted, with some exceptions. Completion of RDMA and Atomic Operation requests submitted to a single send work queue are subject to more complex ordering requirements, as are reliable datagrams, and the interested reader should consult the specification. There are no ordering requirements for work requests submitted to different work queues.

A consumer can determine completion of work requests either by polling a completion queue (CQ) that has been associated with a work queue, or by requesting notification when a new completion occurs on a CQ. Completion polling is one area where VIA and IBA differ; in VIA the consumer would poll a bit in the posted work request, while in IBA the consumer polls a CQ for a new completion. When describing completion notification, the IBA specification is very clear that only one completion queue event handler shall be registered per HCA, and that registering the completion handler is a privileged operation. The specification is also very clear that once a completion notification is generated, it is the consumer's responsibility to poll the indicated CQ for completions. What is not very clear from the specification is how multiple consumers, of which some may be user-level and others may be kernel-level, receive notification from the single completion handler.

[1] See *http://www.t10.org/drafts.htm/*.
[2] Address Handles are used to specify a destination for unreliable datagrams, and so don't concern us here.
[3] Memory Windows are used to specify access rights to subsets of Memory Regions.

The standard does address how completion notifications are generated, since that is an implementation issue. However, based on my experience with the Intel prototype InfiniBand card, the process of completion notification will start with the card interrupting the processor.

The standard does state that completion notification is a one-shot event, in that once a notification has been generated, the consumer must re-enable completion notification to receive another notification. More completions can be added to a CQ after completion notification has occurred, so a consumer must dequeue all completions on a CQ after receiving a completion notification. The consumer must then enable completion notification, and check the CQ for completions that arrived while completion notification was being requested. Note that this process provides interrupt coalescing, since multiple completions can be signaled by a single interrupt.

The memory referenced in a work request must be registered with the channel interface before a work request is submitted to a work queue. Memory registration is only available to privileged consumers, and either virtual or physical addressing may be used to specify the memory region. Registering memory allows the channel interface to pin the address range into physical memory, preventing the operating system from paging it out to disk. Registering memory also allows the channel interface to provide the consumer with a local key, L_Key, and a remote key, R_Key, for the memory region. The local key allows work requests queued locally to access the memory region, and the remote key allows work requests queued remotely to access the memory region. The memory keys also serve to disambiguate memory references, since in some operating systems, e.g., Linux, each process has its own mapping for the same virtual address space.

Although no aspect of the work reported here concerned management of an InfiniBand network, it should be noted that the IBA specifies in detail the management mechanisms. Some of the highlights are presented here, but the interested reader should consult the IBA Specification, Volume 1 for details.

The unreliable datagram service is used to transport management information using Management Datagrams (MADs). In order to be IBA compliant, each port must be able to source and sink MADs. The unit of management is a subnet, and each subnet has exactly one master subnet manager, and any number of slave subnet managers. The subnet manager is responsible for topology discovery and maintenance, assignment of the Local Ids (LIDs) used for addressing within the subnet, and programming of routes into switches, among other duties. A subnet manager must be able to transfer control of a subnet to another manager of higher priority, such as might occur when two operational subnets are merged.

A slave subnet manager must present the correct SM_Key to the master subnet manager before the master will relinquish control to the slave. Every subnet manager must obtain its SM_Key via out-of-band means.

A Subnet Management Agent (SMA) is the entity on each port responsible for sourcing and sinking MADs. Each MAD contains an M_Key that the SMA uses to authenticate the MAD. The subnet manager can initialize the M_Key for a port when it initializes the port.

Note that this management model is significantly different from that of networking technologies such as TCP/IP over Ethernet. In the IBA the subnet manager has complete control over the management of all ports in a subnet, and the devices in which the ports reside have none. It should be pointed out that none of the keys specified in the IBA use any encryption mechanisms, so the implications of deploying a classified InfiniBand network must be carefully considered.

Drivers and Prototype Hardware

This section contains a narrative of experiences with three generations of prototype hardware made available by Intel Corporation under a non-disclosure agreement. The first two generations of hardware were based on the Next Generation I/O (NGIO) specification. The third-generation cards were based on the InfiniBand Architecture specification. The first-generation cards used firmware to provide all functionality, while the later generations used a combination of hardware and firmware. All generations used copper cabling, with a signaling rate of 1.25 GHz for the first generation, and 2.5 GHz for the later generations.

Several drivers were written for these prototype cards. The scope of these drivers was limited to exercising the data movement capabilities of the cards, and to serving as learning vehicles for the Linux kernel. The drivers were intended to explore how to provide the data movement functionality needed to implement message-passing systems of interest to Sandia. More specifically, in the driver used for the IBA prototype card, no attempt was made to provide the management functionality as specified in the IBA specification. Thus, that driver could not interoperate with an IBA-compliant subnet.

Since the functionality needed by these drivers does not map well onto existing system calls, all drivers provided all functionality by implementing the ioctl() system call for the device. The ioctl() system call allows arbitrary data structures to be passed from user space to a driver in kernel space, so it can be used to provide any type of desired functionality.

The drivers written for all of the prototype cards used only the reliable connected transport service of the IBA (or the analogous functionality that existed in NGIO). All drivers exposed the underlying connection-based transport by providing ioctl() methods to establish and tear down a connection with a remote process. All included ioctl() methods to register user buffers with the driver, which allows the driver to pin the memory to prevent the operating system from paging it out while a transfer is in progress, and to de-register user buffers. All of the drivers used interrupt-signaled completions. None of the drivers for these prototype cards allowed direct access to the work queues from user space; i.e. the driver interpreted data movement requests, and queued the appropriate requests to the card.

First-Generation Prototype Cards

The first pair of prototype boards from Intel, referred to as San Juan boards, arrived in late September 1999. Milt Clauser, 9227, played the key role in acquiring these boards as part of an attempt to participate in an NGIO/Merced demonstration at Supercomputing ’99. Demo Linux drivers for these boards arrived from Intel in early October, and hardware documentation arrived at the end of October. In the end, we were unable to pull the demonstration together. However, the effort did result in Sandia obtaining prototype hardware and documentation, which launched my effort to develop a driver from scratch.

Original Test Driver

The first driver version used a send/receive transfer model, where both participants in a data transfer make matching read/write calls to effect a transfer. Both blocking and non-blocking

versions of the read/write calls were implemented as ioctl() calls in the first driver. Since the non-blocking calls return before the data transfer is completed, an ioctl() method to test for completion of a pending transfer was implemented, also in blocking and non-blocking versions.

This driver did not use the memory virtual addressing support provided by the hardware, for several reasons. Since it was my first attempt to write a driver, I wanted to use the minimum functionality of the card needed to transfer data between two hosts. Also, recall that the first (and second) generation prototype cards were based on the NGIO specification, and so the cards did not have support for the L_Key and R_Key of the IBA specification. At the time, I didn't understand how to disambiguate user-space virtual addresses using the functionality provided by those cards.

Instead, this driver determined the physical address of the pages into which the user buffer was mapped, and wrote the resulting scatter-gather list into the work request submitted by the driver to the card. Since the length of the scatter-gather list supported by the card was limited, if the user buffer spanned more pages than would fit into the scatter-gather list of a single work request, the driver would submit multiple work requests.

The process of learning enough about the Linux kernel to write a driver, and the process of designing the first driver itself, began in December 1999. By June 2000 I had the driver and simple test programs written and running. In June, I began writing more complex test programs designed to stress the driver and the hardware. These test programs sent multiple messages over multiple channels through multiple ports on each card simultaneously. I found that the driver had some sort of bug that caused it to occasionally miss a completion, which caused my test programs to stall. By September 2000, I still had not found the root cause of the problem, even after extensive consultation with Intel personnel. Their testing showed no such problems, so evidently my driver was at fault. Since progress was stalled, I decided to move on.

Second-Generation Prototype Cards

I was able to obtain second-generation prototype cards in early October, 2000. These cards were also NGIO-based, but had much of their functionality implemented in hardware. The first-generation cards had been implemented completely in firmware. There were only minor programming differences between these two generations of the cards, related to the differences between firmware vs. hardware implementation, so porting the first driver to this card was completed with only minor difficulties. In the process of porting the driver to the second-generation cards, I evidently fixed the bug that had stalled my efforts with the first-generation cards.

By early December 2000, I was again running my test programs on the second-generation cards, although I quickly ran into another problem. In order to provide the reliable transport service, all of these cards transmit cells with sequence numbers that are generated on the card. At least in the first two generations of cards, there is no facility provided to allow the driver to affect the packet sequence numbers; it is all done on the card. However, when transferring messages between cards in different machines, the card would detect sequence number errors. I suspected signal integrity problems due to the 2.5 GHz signaling rate and

design problems with the card. Although I could not get direct confirmation of this from my Intel contacts, trial-and-error repositioning of the cards and cables had some beneficial effect. Eventually, I discovered that having an empty PCI slot on each side of the card, and relocating the internal Ultra-160 SCSI cable as far away as possible from the card, reduced the possibility of packet sequence errors enough to allow some testing. This suggested that whatever the problem, it was affected by electromagnetic field interference. In the end I was able to make latency measurements and minimal throughput measurements, but I was not able to repeat my earlier multiple-channel, multiple-port testing.

This driver was tested using two dual processor, 733 MHz PIII computers, each with one prototype card[1]. The second-generation prototype cards can be run in either 32-bit/33 MHz, 64-bit/33 MHz, or 64-bit/66 MHz PCI slots. However, there was some incompatibility between the cards and the motherboards (SuperMicro 370DL3) in the test systems. In that motherboard, the cards would operate in its 32-bit/33 MHz slots, but not its 64-bit/66 MHz slots.

Since each card has four ports, I typically cabled them such that there were two connections between the two test machines, and one external loop-back connection on each machine. An example of this configuration is shown schematically in Figure 1.

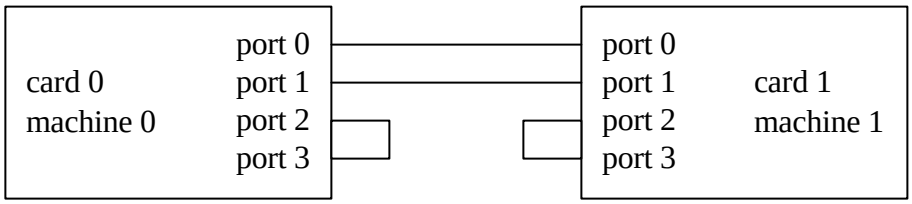

Figure 1. Example configuration for testing second-generation prototype cards.

I measured latency using a typical ping-pong test program, where a message is passed back and forth between two instances of the program. This test program used blocking sends, non-blocking receives, and blocking completion checking for the receives. Recall that all of the test drivers use user-space buffers, so all of the measurements are for user-space to user-space transfers. For this testing, I used the external loop-back, where both instances of the ping-pong test program ran on one computer, and a four-byte message making 100,000 round trips. I measured a round-trip latency of 75 µs using a uniprocessor kernel, and 70 µs using an SMP kernel with both processors.

Throughput testing was conducted using two machines, rather than an external loop-back. I measured 62 MB/s maximum throughput[2]. For this testing I also had a PCI bus analyzer connected to one of the machines, which showed that the sender had 60% bus utilization, and

[1] I could not find a record of exactly what version of the Linux kernel I was using at the time this testing was performed, but it was something in the 2.4.0-test7 through 2.4.0-test11 range.

[2] I could not find a record of what message size I used to obtain this throughput result, but it is likely that the message length was long enough that increasing it wouldn't have increased the throughput.

the receiver had 50% bus utilization. The CPUs on both sender and receiver were essentially idle, at less than 1% utilization.

This level of performance for both latency and throughput was less than I had hoped to achieve. For a data link at 2.5 GHz using 8B/10B encoding the raw data rate is 250 MB/s, so even allowing for packet header overhead, my throughput result was less than a third of the theoretical value. There didn't seem to be any driver modifications to be made that would increase throughput. For example, the driver already allocated the kernel memory needed to queue work requests when the user pinned the message buffer, so they would be reused on each send or receive call.

I attempted to characterize some of the latency sources in my driver by instrumenting it with calls to the processor time stamp counter (TSC). This is a register on Intel Pentium Pro and newer processors that is incremented once per processor clock cycle. I only used the TSC to measure latencies when using a uniprocessor kernel. Although Linux synchronizes the TSC between processors in a multiprocessor system, I don't know the degree of synchronization, and I didn't want to introduce an additional source of uncertainty.

There were two sources of latency in which I was interested. One was a result of the way Linux drivers are supposed to handle interrupt sources. Following the Linux philosophy of interrupt handling, an interrupt service routine is supposed to read just enough information from a device to allow a tasklet to be queued to process the information later. An interrupt service routine is executed with hardware interrupts disabled, while a tasklet is executed with hardware interrupts enabled, for the most part. So, this philosophy minimizes the time spent with interrupts disabled, and increases overall performance and fairness of the machine.

In the first driver, my interrupt service routine would read the completion queues to discover which work queues needed to be serviced, and then queue one tasklet for each work queue that needed servicing. So, one source of latency I could measure easily is the time between when the tasklet is queued in the interrupt handler, and when it actually begins running. Another, related, source of latency is the interrupt latency itself, i.e., the time between when the card raises its interrupt line and when the interrupt service routine actually begins execution. At the time I was working with this card/driver combination I did not know how to measure that latency. I later learned a way to do this, and will report some results in the context of a later card/driver combination.

The second source of latency I could measure easily was the time required to wake up a sleeping process. Recall that my latency tester used blocking sends and blocking completion checking for the receives. One method used in Linux to implement a blocking call is to put the current process on a wait queue, and then schedule to another process. Each time the sleeping process is scheduled, it checks for the waking event, and schedules to another process if the waking event has not occurred. Since the waking event must occur in some other thread of execution, there is a scheduling latency between when the waking event occurs, and when the process sleeping on that event is scheduled and begins execution.

Using a uniprocessor kernel and instrumented driver, I measured these two latencies to be 2000 clocks and 8400 clocks, respectively. For the 733 MHz processor in the test systems,

this turns out to be 2.8 µs and 11.5 µs, respectively. To see what portion of the round trip latency of 75 µs these two latencies can account for, consider Figure 2. That figure attempts to show the sequence and temporal correspondence of events in the master and slave processes of the latency tester. The position of top and bottom of the box enclosing each event indicates the relative timing of the start and end of the event. So, we see that the master's first non-blocking receive starts at roughly the same time as the slave's first blocking receive. Then, we see that the master's first blocking send completes at roughly the same time as the slave's first blocking receive completes.

From Figure 2 we see that both the master and slave processes execute the same basic loop: post non-blocking receive, post blocking send, post blocking receive notification. The non-blocking receive is always posted before the send to ensure that a message is never sent before the corresponding receive is posted. However, the user has to start the slave process first to ensure this. The round-trip latency reported here is the time it takes either the master or the slave to execute the loop, divided by the number of times through the loop. The messaging latency is one-half of the round trip latency.

Now, consider a blocking send call. Recall that in this driver it is implemented as an ioctl() call, which is a system call. So, the process traps into kernel mode, and a send work request is constructed and queued on a send work queue. The process puts itself on a wait queue, and schedules away to some other process.

Some time later, the send work request completes and generates an interrupt. The currently running process is interrupted, and the interrupt service routine queues a tasklet to service the completion on the send work queue. When the interrupt service routine completes, the kernel checks to see if there is any other work to do before resuming the interrupted process. It finds the queued tasklet and runs that. The tasklet finds the wait queue associated with the send request process, and wakes any processes sleeping on the queue. Waking the process just means setting the process's state to run-able, so that it is a candidate to be selected by the

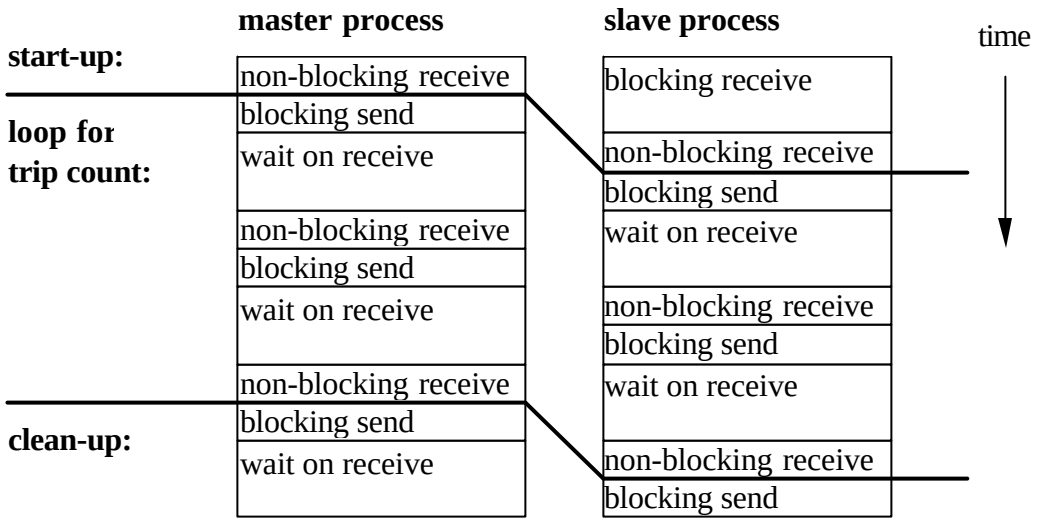

Figure 2. Sequence of events in a latency test program for first test driver.

scheduler the next time it runs.

The tasklet then returns, and the kernel runs any other tasklets that may be queued. Finally, when there is no more kernel-mode work to do, the kernel schedules a new process. I am not sure of the details here, but at some point the scheduler will notice that the process that called the blocking send is now run-able, and schedules it. That process has now been "awakened", and makes its next call, in this case to the ioctl() for blocking receive notification. The whole process now repeats itself, except the work queue in question is the receive queue specified in the earlier non-blocking receive call.

So, we see that the tasklet startup latency and the process wakeup latency each occur twice in each loop iteration of the latency test program, for both master and slave. Since the driver was implemented using a reliable transport service, a send request and its matching receive request will complete at roughly the same time. Thus, the tasklet scheduling and process wake-up latencies for the sender and receiver occur roughly in parallel, rather than serially. Depending on the amount of overlap, we should expect tasklet startup and process wakeup latencies to account for at least 28.6 µs of the 75 µs roundtrip latency exhibited by this driver.

In February, 2001 I took delivery of several new dual processor, 1 GHz PIII computers, based on the Intel SBT2 motherboard, to use as test hosts. These hosts were acquired because they were certified by Intel as compatible with the upcoming third-generation, IBA-based prototype cards. The third-generation cards would only be compatible with either PCI-X or 64-bit/66 MHz PCI slots. Since I had experienced compatibility problems with the second-generation cards and my original test hosts, I felt it was advisable to procure hosts known to be compatible with the new cards.

I installed the second-generation cards in the new hosts in 66 MHz slots, and found that they worked perfectly, at least with respect to PCI compatibility. The packet sequence number error problem was still present, confirming that it was a problem with the cards and not the host, or host/card combination. As I write this report I was chagrinned to find that the only test results with the 1 GHz hosts I had recorded at the time I received them was a 120 MB/s throughput result.

However, since the 1 GHz hosts and cards were still available, for the sake of completeness I reran my tests[1]. Results are presented in Table 2, where the round trip latency was measured based on the elapsed time for 10,000 message round trips. The "error" entries for the throughput results between two hosts indicate the sequence number problem discussed earlier. Although I didn't record it in the table of results, I got sequence errors for throughput testing between two hosts using 16 KB and 32 KB message buffers also. I cannot explain why the problem does not occur at larger message buffer sizes.

Several trends can be identified in Table 2. The fact that results with a UP kernel are generally better than results with an SMP kernel is most likely indicative of the extra locking needed throughout the kernel in the latter case. Results for testing between two hosts, where a single instance of the test program is running on each, are generally better than results from

[1] This testing was performed using version 2.4.7 of the Linux kernel.

Table 2. Performance of first test driver with second generation prototype cards in 1 GHz hosts.

	external loopback		between two hosts	
	UP	SMP	UP	SMP
round-trip latency, µs 4 byte msg.	58	58	44	49
tasklet latency, µs (clocks)	2.1 (2100)	n/a	1.8 (1800)	n/a
wakeup latency, µs (clocks)	7.7 (7700)	n/a	4.1 (4100)	n/a
throughput, 10^6 B/s 4096 byte msg.	62	59	67	64
throughput, 10^6 B/s 8192 byte msg.	83	81	88	86
throughput, 10^6 B/s 65,536 byte msg.	121	120	error	error
throughput, 10^6 B/s 131,072 byte msg.	112	112	120	119
throughput, 10^6 B/s 262,144 byte msg.	113	113	121	121

testing with external loopback, where the two instances of the test program run on one host. This could be due to the extra cost of scheduling two processes in the latter case, as well as contention for resources on the card, or possibly some contention for the PCI bus. Note that for the external loopback case, throughput on the PCI bus is twice the measured throughput, since each message is both read and written across the same PCI bus.

It is also interesting to compare the limited results available from the testing on the 733 MHz hosts to the results in Table 2. Although the processors in the test hosts are 36% faster, the round trip latency using external loopback decreased by only 23% and 17%, respectively, for UP and SMP kernels. This is expected if a non-negligible part of the latency is attributable to the cards themselves. Note that the 733 MHz host testing results showed a lower latency on an SMP kernel (70 µs) vs. a UP kernel (75 µs), while the 1 GHz host testing results showed the same 58 µs latency. The lack of a difference between SMP and UP results in the latter case is likely due to differences in the scheduler in the different versions of the kernel used for the testing.

Also interesting is that the maximum throughput increased by a factor of two. Recall that the PCI slots used in the 1 GHz hosts (64 bit/66 MHz) have a throughput larger by a factor of four than those used in the 733 MHz hosts (32 bit/33 MHz). The fact that the maximum throughput increased in direct proportion to the PCI bus frequency suggests that PCI transaction handling in the card had a role in limiting the throughput achieved by the card.

This driver was a valuable learning experience, but it had several shortcomings that I wanted to address. First, its performance was less than expected, and although it was not altogether clear whether the reason was due to software or hardware issues, I had identified latency sources that were a direct result of implementation decisions. Second, it used only a limited subset of the capabilities of the hardware. Third, the send/receive model of transfer was limiting because of the requirements it places on data ordering.

This last shortcoming is a result of the design of the InfiniBand Architecture. There is no mechanism for an incoming message to match up with a specific entry in the receive work queue. The next available entry in a receive work queue is used to accept the next incoming message. If the incoming message is longer than the buffer space provided by the next entry in the receive queue, an error is generated.

In my test programs this was not a problem, because they were designed to enforce message ordering. However, suppose the sender needed to send two different messages, say A and B, and queued the send requests in that order. If the receiver queued the receive requests in the order B, A, it would end up with the messages switched, assuming the lengths of the receive work requests were large enough.

Revised Test Driver

By the beginning of March, 2001, I had a design for a second test driver. This revised driver was intended to be a proof of concept for an implementation of Portals 3.0 over InfiniBand [6,7]. As such, I wanted to model the “matching put” and “matching get” operations of Portals. I also wanted to attempt to minimize latency by minimizing interrupts, and by eliminating the process wake-up latency inherent in blocking calls.

However, in the interest of simplicity I did not implement the full matching semantics of Portals. In Portals a portal address comprises a process id, memory buffer id, offset, and a set of match bits. My revised test driver design exposes to users only a memory buffer id, which is obtained when a buffer is pinned. As mentioned earlier, the connection-oriented nature of the underlying transport is exposed to users, so each end of the connection is directly associated with a single process. Since the driver can support many processes/connections, the process id is used internally by the driver to look up pinned buffers.

As discussed earlier, this driver supports ioctl() methods to register and de-register user buffers, and to establish and tear down a connection. In addition, it supports ioctl() methods to submit put and get requests to move data, and to request notification of data request completion on a registered buffer.

Buffer registration has two effects: assignment of a tag to a buffer, and pinning the buffer in memory. Put/get requests reference a local buffer tag and a remote buffer tag. If a remote buffer has not been registered with a tag that matches the remote tag in the put/get request, no data is transferred. Put/get requests are non-blocking in this driver, meaning that the request returns before data transfer is completed.

In order to determine that a buffer is ready to be reused, or that new data has arrived, an I/O completion notification may be registered on a data buffer. This takes the form of a user data

structure that is associated with a data buffer when a completion notification is registered. Either incoming notification or outgoing notification may be registered on a local buffer, but not both simultaneously. When the next appropriate I/O operation, either locally or remotely initiated, completes on the associated local data buffer, the notification structure is updated with the status and the association is broken between the notification structure and the data buffer. Thus, completion notification is a one-shot event. The user process can poll this notification structure to check for a change in the completion status.

Outgoing I/O completion notification occurs when data has left the local buffer. Receiving an outgoing completion notification means that the local buffer is free for reuse, but it does not signify that the data has safely arrived in the remote buffer. Incoming I/O completion notification occurs when new data has arrived in the local buffer, and is ready for use. Note that an incoming notification can occur as a result of either a locally initiated get request, or a remotely initiated put request, and the converse is true for outgoing notification. As an example, if an incoming notification is registered on a buffer, followed by a put request, the notification will not occur until a remote process performs a put operation into that local buffer.

Completion notification structures may be reused, i.e., re-associated with the same or a different message buffer. The memory used for the notification structures must be registered with the driver, in the same way as data buffers. Also, a single larger buffer may be registered, and then broken up into several notification structures.

The initiator of a put/get request may optionally request acknowledgement that the target has responded to the request. For a put request, a successful acknowledgement means that the target has safely stored the data in the remote buffer. For a get request, a successful acknowledgement means that the data has arrived in the local buffer. An acknowledged request is the only way for an initiator to be informed of error conditions on the target.

Put and get requests are implemented in this driver using a simple request/response mechanism. The request initiator sends a simple protocol message containing information describing the request, e.g, the tags for the local (initiator) and remote (target) buffers. The request target responds by performing the data transfer operation.

Put and get requests have a long message implementation and a short message implementation. In the long message implementation, data is transferred using an InfiniBand RDMA request. In the short message implementation, data is embedded in a protocol message, which has a fixed header size and a variable payload. Protocol messages are transferred using the send/receive model. The maximum length of a protocol message, which determines the crossover point between short and long messages, is a driver compile-time option. Although protocol message buffers are allocated at this size, the number of bytes transferred in a protocol message is just the size of a protocol header plus the message size.

The protocol messages and RDMA requests needed to implement the four types of data transfer requests are shown in Figure 3. In every case, a data movement request starts with the initiator sending a protocol message with the request to the target. In the short message put request case, the user data is copied into this protocol message.

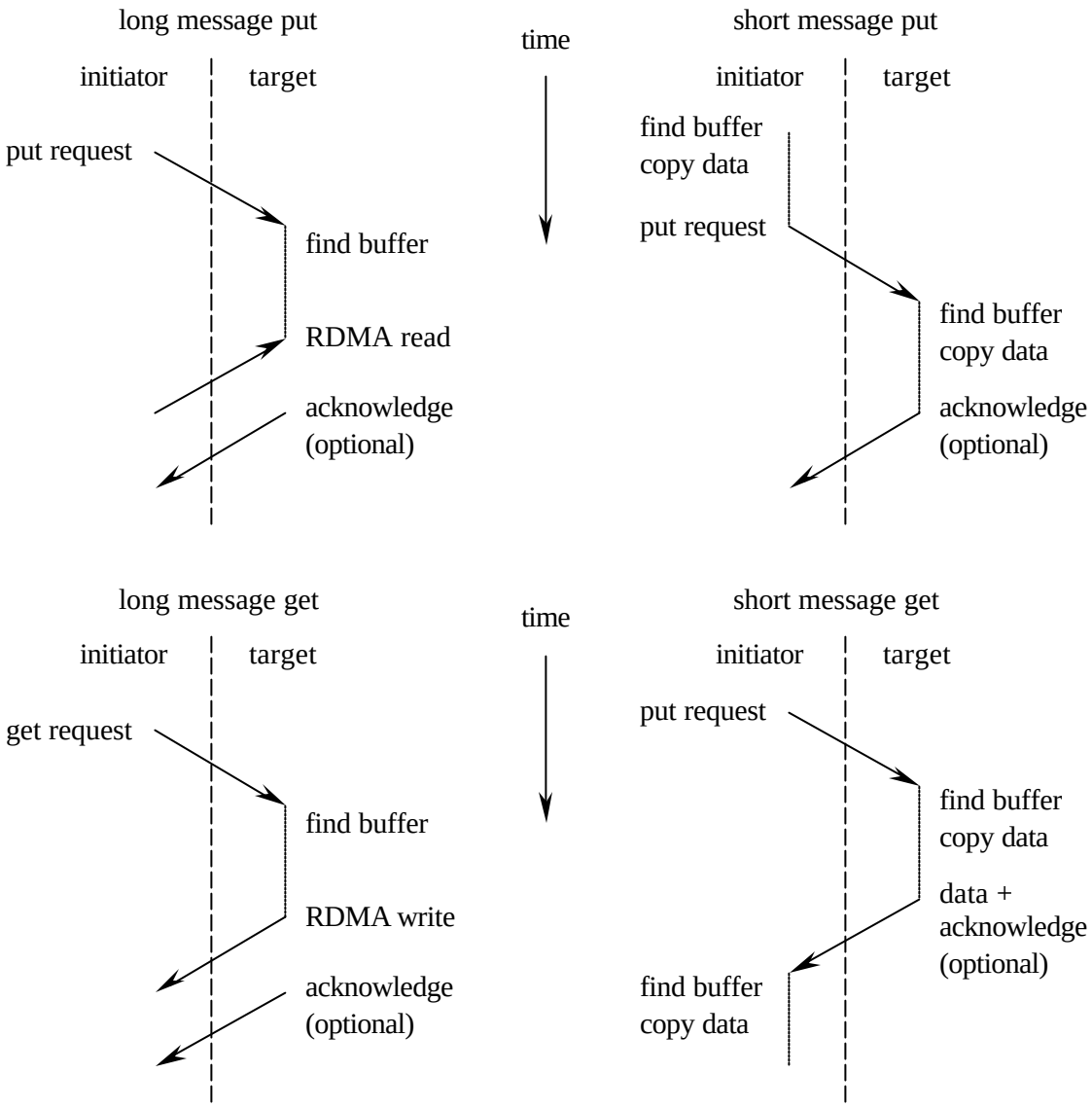

Figure 3. Protocol messages and RDMA requests for second test driver.

When the target receives the request, it processes the request based on the request type. For all request types, if acknowledgement was not requested, and the request cannot be fulfilled, it is ignored. For example, the requested buffer may not have been registered on the target. If acknowledgement was requested and the request cannot be fulfilled, a protocol message containing the reason is constructed and sent as a response.

For a long message request, the target constructs and queues the appropriate RDMA work request, if possible. If the request included an acknowledgement request, the target also sends a response protocol message containing the acknowledgement. For the short message put request case, the target copies the data out of the request message into the appropriate target buffer, if it can; otherwise the data is discarded. In the short message get request case, the target prepares a response protocol message by copying the target buffer into the response message, and including acknowledgement information if requested.

As mentioned above, one of the goals for this test driver was to minimize latency, in part by minimizing the number of interrupts. From Figure 3 we can see that to minimize latency we want to begin processing a protocol message as soon as possible after it arrives. Thus, we want to use interrupt-signaled completions for receive work queue entries that handle protocol messages. Polling for completion of protocol messages on a receive work queue would only reduce latency if the polling interval was shorter than the interrupt latency.

However, completions for send work queue entries that handle protocol messages are not time critical, and we only need to process completions fast enough to ensure that there are always send work queue entries available for new protocol messages. Thus, we can poll for completions when convenient on the send work queue that handles protocol messages.

Another potential source of latency is the blocking of a protocol request by an RDMA request that moves a large amount of data, since both types of requests are processed on a send work queue. This can be accomplished by segregating protocol messages and RDMA requests on separate send work queues.

The second test driver addresses both of these issues by using two queue pairs, one for interrupt-signaled completions and the other for polled completions, on each of the target and initiator. This is shown schematically in Figure 4. In the common case that both nodes in a communication pair are both message initiators and message targets, the protocol messages will still be separate from the RDMA requests. This is shown schematically in Figure 5.

As noted earlier, for this design put and get requests are non-blocking. Thus, they return as soon as the appropriate work request is queued. Also, all of the protocol messages in this driver are handled in the interrupt service routine. Thus, when a message is short enough to be copied to/from a protocol message, this copying takes place in the interrupt service

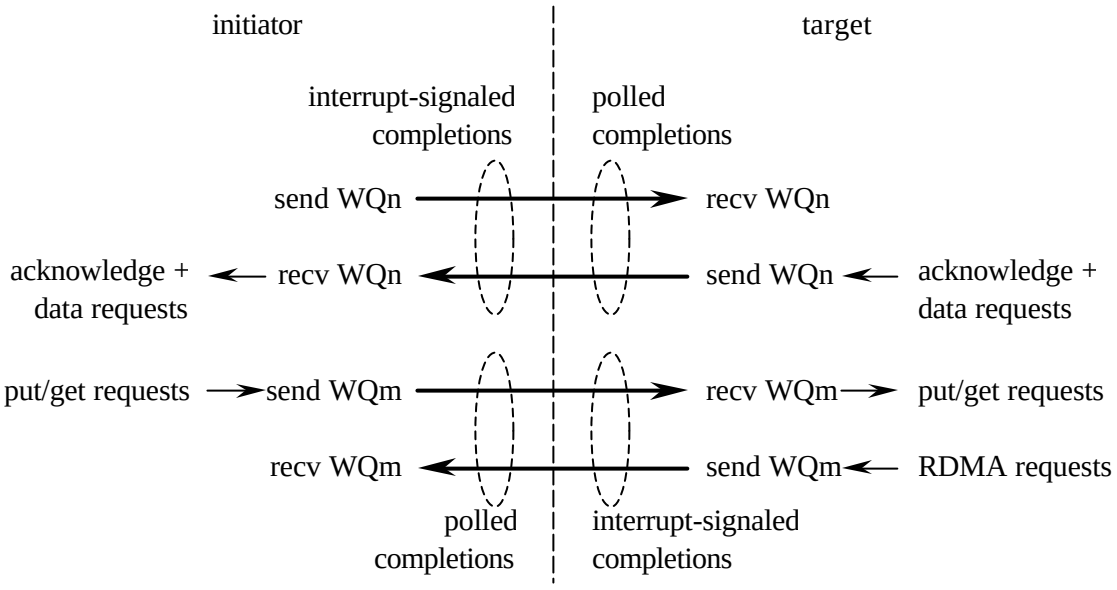

Figure 4. Use of polled-completion and interrupt-signaled completion queue pairs by message initiator and target in second test driver.

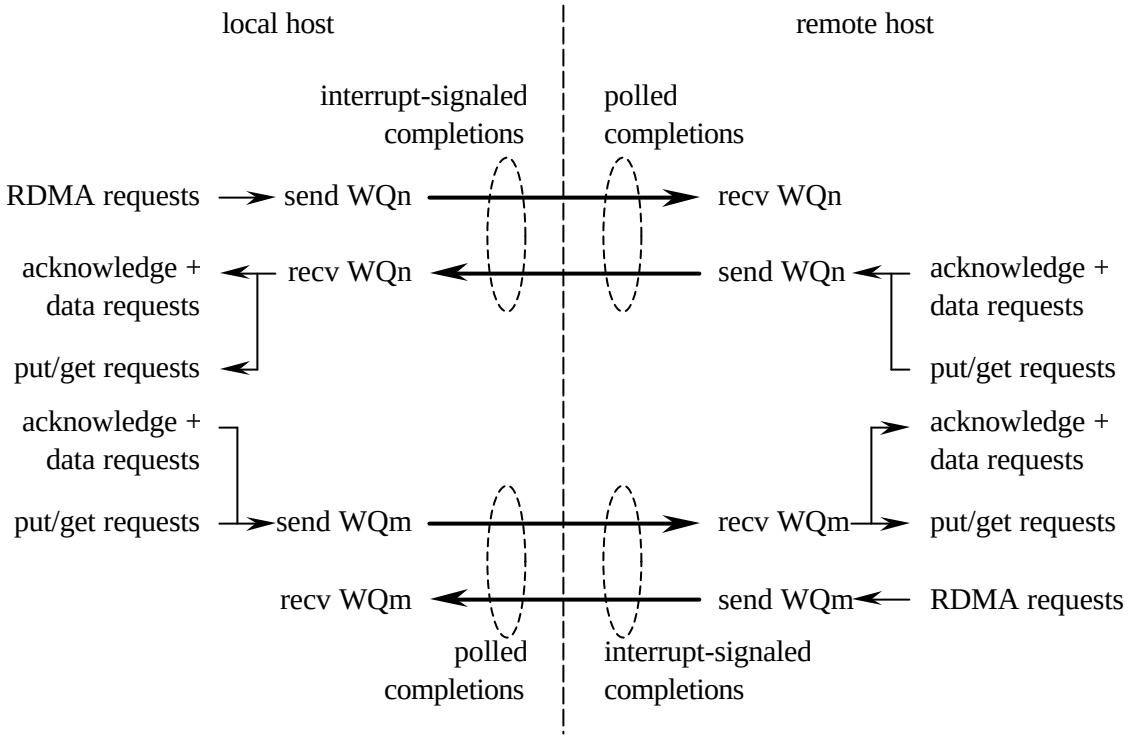

Figure 5. Use of polled-completion and interrupt-signaled completion queue pairs by local and remote hosts in second test driver.

routine. This is directly counter to the Linux design philosophy for interrupt service routines, because of the impact on other system components. However, it was done in this case to achieve the minimum latency for this driver, regardless of the impact on the rest of the system, and cannot be recommended for general practice.

During the course of developing this second test driver, I learned how to use a PCI bus analyzer to measure the interrupt startup latency. I define this to be the time between when the card asserts its interrupt line, and when the appropriate interrupt service routine begins servicing the card. The timing of the first event can be determined by using the bus analyzer to monitor the card's interrupt line. The timing of the second event can be determined by using the bus analyzer to monitor accesses of the card's interrupt status register, since normally the first action taken by an interrupt service routine is to read that register. With the second-generation prototype card in a 66 MHz PCI slot of a 1 GHz host, running version 2.4.1 of the Linux kernel, I found this interrupt startup latency to be 4 µs.

Note that this second test driver design overcomes another of the shortcomings of the first driver design mentioned earlier, in that it uses more of the features of the hardware. In addition to using the send/receive capabilities, it also uses the RDMA capabilities provided by the prototype cards. Also, although it is not directly relevant to the design issues discussed above, the second driver used the virtual addressing capabilities provided by the hardware, rather than relying on physical addressing as did the first test driver.

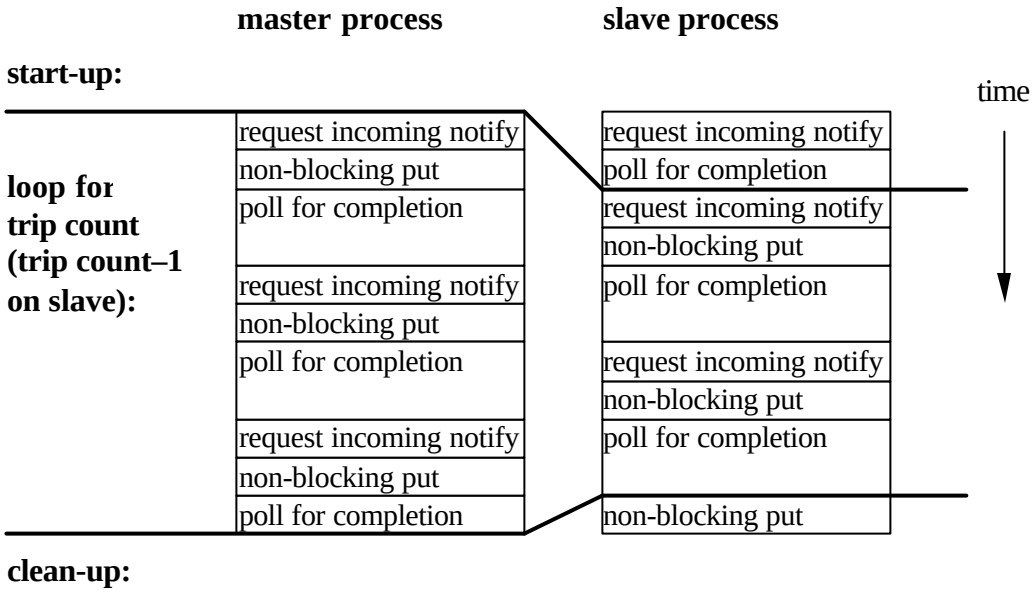

Figure 6. Sequence of events in a latency test program for second test driver.

Figure 6 shows attempts to show the sequence and temporal correspondence of events in the master and slave processes of the latency test program written for the second test driver. Again the master and slave execute the same basic loop: request incoming notification on a message buffer, call the (non-blocking) message put function, and then poll the notification structure until it is updated to indicate that a message has been received. The slave loops one iteration less than the master, so that the number of puts and completions match up. Message acknowledgements were not used in this test program. Instead, each process uses the fact that it receives a message to know that the message it previously sent arrived. Note that each of the master and slave processes only have a single message buffer, and the incoming notification request and the message put calls apply to that same buffer.

In this test program, the completion notification polling loop contains two operations. The first is to check the value of the completion member of the notification structure, and the second is an invocation of the system call sched_yield(). The latter is required when running the test program with external loopback on a uniprocessor kernel, where the master and slave processes must share the processor. In that case, without the call to sched_yield(), it is possible for one process to consume its time slice polling, even though the message it is waiting on cannot be sent because the other process hasn’t been scheduled, so it can’t poll to discover a message has arrived. Thus, latency is inflated due to process scheduling effects.

The latency tester was written in this fashion to minimize the number of interrupts generated. If we compare the sequence of events in Figure 6 with the message diagrams of Figure 3 and the work queue usage of Figure 5, we see that we will incur one interrupt each on the master and slave for one message round trip, in the case where the message is short enough to be copied into the request protocol message. This interrupt occurs when the request protocol message arrives. In the long message case, where RDMA is used to transfer the data, the test

Table 3. Performance of second test driver with second generation prototype cards in 1 GHz hosts.

	round-trip latency, µs between two hosts	
message length, bytes	UP	SMP
4	51	47-51
32	52	47-51
256	61*	58*
2048	107	111
4060	156	162-166
4096	error	error

* For this message length, the test program would hang if more than 200 round trips were attempted.

program will incur two interrupts each on the master and the slave. The first interrupt occurs when the request protocol message arrives, and the second occurs when the RDMA transfer completes.

The implementation of this second driver design for the second-generation prototype cards was completed by the beginning of May, 2001, along with the latency test program. Testing was interrupted by the arrival of the third-generation cards in mid-May, so the results presented here were generated at the time of the writing of this report, for the sake of completeness.

One of the goals of testing this driver was to determine the appropriate value of the crossover point between short and long messages. Unfortunately, the sequence number problem of the second-generation prototype card seems to manifest itself in all RDMA transfers, i.e., I could not get a single RDMA transfer to complete without a sequence number error occurring. Consequently, all testing of this driver on second-generation cards used a 4096-byte protocol message, which has a maximum payload of 4060 bytes. This sequence number problem also precluded any throughput testing with this hardware/driver combination.

The results of this testing[1] is shown in Table 3, where again the round trip latency was measured based on the elapsed time for 10,000 message round trips. Comparing these results with the latency results from the first driver, from Table 2, we see that the anticipated drop in latency did not materialize. In fact, the round trip latency between two hosts increased slightly. Evidently, the cost of preparing and interpreting the protocol packets in the second driver offsets the cost of waking up a sleeping process in the first driver. We also see that under SMP, this second driver exhibits more variability in its timings.

[1] This testing was performed with version 2.4.7 of the Linux kernel.

Third-Generation Prototype Cards

As mentioned earlier, I received the third-generation prototype cards in mid-May, 2001. These cards were the first prototypes based on the IBA specification. Porting of the second driver to these cards was straightforward. The differences between this version of the hardware and the previous version were driven by changes needed to adhere to the IBA specification, and were not extensive. They included changes in the number and functionality of control registers, changes in the format of the virtual addressing support, changes in completion notification, and changes in port initialization. Porting of the driver was completed by the beginning of August 2001.

This third generation of the hardware exhibited none of the difficulties of the earlier generations. Testing was conducted with the cards installed in the dual processor 1 GHz hosts. The third-generation cards support either PCI-X or 64-bit/66 MHz PCI, and were thus installed in the 66 MHz slots in the test hosts. Testing was conducted between two SMP hosts.

The latency test program for the second test driver was used to measure both latency and throughput. Throughput was obtained by dividing the test message length by one-half of the round trip latency. Latency and bandwidth results results are presented in Table 4, and latency results are plotted in Figure 7. Again, the round trip latency was measured based on the elapsed time for 10,000 message round trips.

Because the driver has short message and long message implementations, it is desirable to know the optimal message length at which to switch from the short to the long message implementation. If the short message implementation is used for messages that are too long, extra latency will be incurred due to the cost of copying a long message into and out of the protocol message. Conversely, if the long message implementation is used for messages that are too short, extra latency will be incurred due to the cost of the extra protocol message needed to make the transfer. When the optimal protocol message length is used, messages of every length are transferred with minimum latency.

The optimal crossover point between short and long messages is that protocol message length where the round trip latency for short and long messages is the same. To determine the optimal crossover for this driver, I generated test results for 4096-byte protocol messages (4060 byte maximum payload) and 64-byte protocol messages (28 byte maximum payload). These results indicated an optimal protocol message length of 2048 bytes (2012 byte maximum payload). As reported in Table 4, for 2048 byte protocol messages there are no sudden increases or decreases in latency as the message length changes, indicating that every message is transferred with minimum latency.

Note that the round trip latency for four byte messages on the third-generation prototype cards, at ~36 µs, is much better than the ~50 µs recorded using the second-generation hardware. Because of the differences in the programming interfaces to the cards, particularly for completion notification, it is difficult to assign the improvement in latency to hardware versus software changes. However, the drivers are essentially the same, so I would have to attribute most of the improvement to differences in the cards.

Table 4. Performance of second test driver with third generation, InfiniBand-based prototype cards in 1 GHz hosts.

	two SMP hosts, 64 byte protocol msg.		two SMP hosts, 2048 byte protocol msg.		two SMP hosts, 4096 byte protocol msg.	
message length, bytes	round trip latency µs	throughput 10^6 bytes/sec	round trip latency µs	throughput 10^6 bytes/sec	round trip latency µs	throughput 10^6 bytes/sec
4	35.0	0.23	36.0	0.22	36.5	0.22
28	35.9	1.6				
32	67	0.96	37.7	1.7	38.2	1.7
256	76	6.7	47.3	5.4	47.8	11
2012			94	43		
2048	97	42	98	42	98	42
4060					151	54
4096	123	67	122	67	122	67
8192	173	95	173	95	173	95
16,384	272	120	272	120	272	120
32,768	473	139	474	138	474	138
1,048,576	15900	132	15900	132	15900	132

Maximum throughput for the third-generation cards is still somewhat disappointing, at 138 MB/s. Consider the results for the driver using 64-byte protocol messages, and assume that the round trip latency for 32-byte messages (the shortest message transferred using RDMA) is representative of the software overhead of the driver and test program for long messages. Under that assumption, 406 µs of the 473 µs round trip latency for 32,768 byte messages could be attributed to hardware, resulting in a maximum throughput of 161 MB/s. This suggests there is still some room for improvement in the hardware performance.

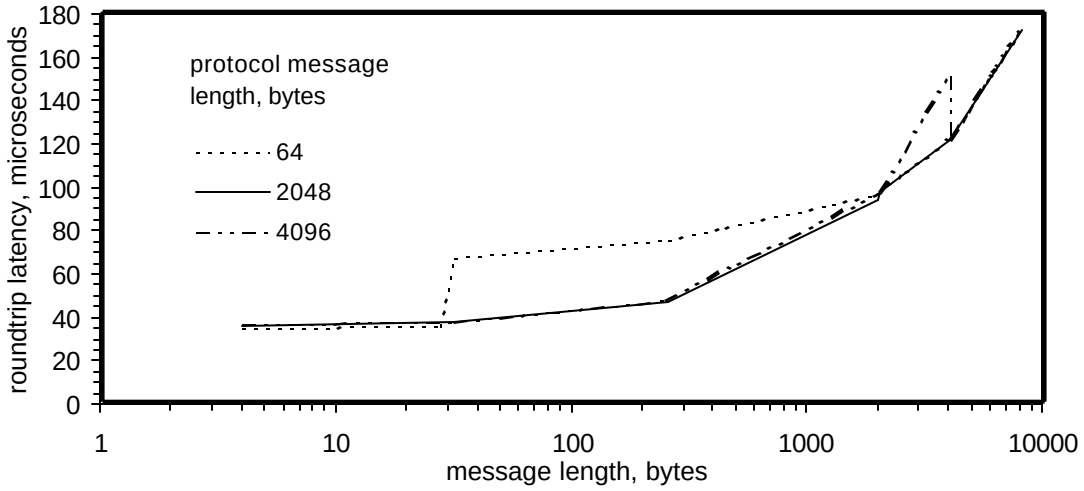

Figure 7. Round trip latency results for various protocol message lengths, for second test driver and InfiniBand-based prototype cards.

Summary and Recommendations

This report describes work undertaken to explore InfiniBand, an emerging system-area networking technology, and develop Linux kernel programming expertise. Three generations of prototype hardware were obtained, and Linux device drivers were written which exercised the data movement capabilities of the cards. Latency and throughput results obtained for a 1X InfiniBand link were similar to other SAN technologies, but not significantly better.

Due to the scope of the InfiniBand specification, a great deal of software is needed to implement an InfiniBand-compliant network. It is probably not appropriate for Sandia to undertake the implementation of the software needed to provide an InfiniBand-compliant Channel Interface, but rather should obtain this from a vendor if we deploy an InfiniBand-based solution. Any software Sandia develops should probably be written to a vendor's Channel Interface API.

Unfortunately, because the InfiniBand specification does not specify any APIs, different vendors may develop different Channel Interface APIs, complicating our task of writing platform-independent software. In fact, this may be one of the greatest issues facing widespread deployment of InfiniBand. Although a great deal of effort has been put into hardware interoperability, software is intended to be an opportunity for vendor "value-add", which may complicate true multi-vendor interoperability.

At the time this report was written, InfiniBand products were still not commercially available, and will probably not begin to appear until mid-2002.

References

1. InfiniBand Trade Association, *InfiniBand Architecture Specification, Release 1.0.a, Volume 1 and Volume2,* InfiniBand Trade Association, June, 2001. Available from http://www.infinibandta.org/estore.html.

2. Compaq Computer Corp., Intel Corp., and Microsoft Corp., *Virtual Interface Architecture Specification Version 1.0*, Compaq Computer Corp., Intel Corp., and Microsoft Corp., December 1997. Available from http://www.viarch.org.

3. J. Postel, *RFC-768, User Datagram Protocol.* August 1980. Available from http://www.faqs.org/rfcs/rfc768.html.

4. C. Partridge and R. Hinden, *RFC-1151, Version 2 of the Reliable Datagram Protocol (RDP).* April 1990. Available from http://www.faqs.org/rfcs/rfc1151.html.

5. D. Velten, R. Hinden, and J. Sax, *RFC-908, Reliable Data Protocol.* July 1984. Available from http://www.faqs.org/rfcs/rfc908.html.

6. R. Brightwell, T. Hudson, A. B. Maccabe, and R. Riesen. *The Portals 3.0 Message Passing Interface Revision 1.0, SAND99-2959.* Sandia National Laboratories, November 1999.

7. R. Brightwell and A. B. Maccabe, "Scalability Limitations of VIA-Based Technologies in Supporting MPI," conference paper, Fourth MPI Developer's and User's Conference, Cornell University, Ithaca, NY, March 20-23, 2000. Available from http://www.cs.sandia.gov/~bright/main.html.

Distribution

1	MS	0139	M. O. Vahle, 9900
1		0318	P. D. Heermann, 9227
1		0801	A. L. Hale, 9300
1		0801	M. R. Sjulin, 9330
1		0806	S. A. Gossage, 9336
5		0806	J. A. Schutt, 9336
1		0806	L. Stans, 9336
1		0807	J. P. Noe, 9338
1		0812	M. J. Benson, 9334
1		0847	M. J. Clauser, 9227
1		0986	R. S. Berg, 2662
1		9003	P. W. Dean, 8903
1		9011	H. Y. Chen, 8915
1		9018	Central Technical Files, 8945-1
2		0899	Technical Library, 9616
1		0612	Review and Approval Desk, 9612 for DOE/OSTI