2012 IEEE 14th International Conference on High Performance Computing and Communications

A Performance Study to Guide RDMA Programming Decisions

Patrick MacArthur, Robert D. Russell
Computer Science Department
University of New Hampshire
Durham, New Hampshire 03824-3591, USA
pio3@unh.edu, rdr@unh.edu

***Abstract*—This paper describes a performance study of Remote Direct Memory Access (RDMA) programming techniques. Its goal is to use these results as a guide for making "best practice" RDMA programming decisions.**

Infiniband RDMA is widely used in scientific high performance computing (HPC) clusters as a low-latency, high-bandwidth, reliable interconnect accessed via MPI. Recently it is gaining adherents outside scientific HPC as high-speed clusters appear in other application areas for which MPI is not suitable. RDMA enables user applications to move data directly between virtual memory on different nodes without operating system intervention, so there is a need to know how to incorporate RDMA access into high-level programs. But RDMA offers more options to a programmer than traditional sockets programming, and it is not always obvious what the performance tradeoffs of these options might be. This study is intended to provide some answers.

***Keywords*-RDMA; Infiniband; OFA; OFED; HPC;**

I. Introduction

As networks grow faster, Remote Direct Memory Access (RDMA) is rapidly gaining popularity outside its traditional application area of scientific HPC. RDMA allows application programmers to directly transfer data between user-space virtual memories on different machines without kernel intervention, thereby bypassing extra copying and processing that reduce performance in conventional networking technologies. RDMA is completely message-oriented, so all application messages are sent and received as units, unlike TCP/IP, which treats network communication as a stream of bytes.

User-level RDMA programming offers many more options and is more complex than traditional socket programming, as it requires the programmer to directly manipulate functions and data structures defined by the network interface in order to directly control all aspects of RDMA message transmission. Therefore, the programmer must make many decisions which may drastically affect performance. The goal of this paper is to evaluate the performance of numerous methods of directly using the application-level RDMA features in practice.

Similar performance evaluations have been done for specific applications and tools that use RDMA, such as MPI [1], [2], FTP [3], GridFTP [4], NFS [5], AMPQ [6], PVFS [7], etc. The importance of our work is that we evaluate RDMA directly, without any particular application or environment in mind, and provide guidance on general design options faced by anyone directly using RDMA.

A. Background

Three RDMA technologies are in use today: Infiniband (IB), Internet Wide-Area RDMA Protocol (iWARP), and RDMA over Converged Ethernet (RoCE). Infiniband [8], [9] defines a completely self-contained protocol stack, utilizing its own interface adapters, switches, and cables. iWARP defines three thin protocol layers [10]–[12] on top of the existing TCP/IP protocol (i.e., standard Internet). RoCE [13] simply replaces the physical and data-link layers of the Infiniband protocol stack with Ethernet. All three technologies are packaged as self-contained interface adapters and drivers, and there are software-only versions for both iWARP [14] and RoCE [15].

The OpenFabrics Alliance (OFA) [16] publishes and maintains a common user-level Application Programming Interface (API) for all three RDMA technologies, It provides direct, efficient user-level access to all features supported by each RDMA technology. OFA also provides open access to a reference implementation of this API, along with useful utilities, called the OpenFabrics Enterprise Distribution (OFED) [17]. This API is used throughout this study.

In RDMA, actions are specified by verbs which convey requests to the network adapter. Each verb, such as `post_send`, is represented in the OFED API as a library function, `ibv_post_send`, with associated parameters and data structures. To initiate a transfer, `ibv_post_send` places a work request data structure describing the transfer onto a network adapter queue. Data transfers are all asynchronous: once a work request has been posted, control returns to the user-space application which must later use the `ibv_poll_cq` function to remove a work completion data structure from a network adapter's completion queue. This completion contains the status for the finished transfer and tells the application it can again safely access the virtual memory used in the transfer.

RDMA provides four sets of work request opcodes to describe a data transfer. The SEND/RECV set superficially resembles a normal socket transfer. A receiver first posts a RECV work request that describes a virtual memory

978-0-7695-4749-7/12 $26.00 © 2012 IEEE
DOI 10.1109/HPCC.2012.110

area into which the adapter should place a single message. The sender then posts a SEND work request describing a virtual memory area containing the message to be sent. The network adapters transfer data directly from the sender's virtual memory area to the receiver's virtual memory area without any intermediate copies. Since both sides of the transfer are required to post work requests, this is called a "two-sided" transfer.

The second set is a "one-sided" transfer in which a sender posts a RDMA WRITE request that "pushes" a message directly into a virtual memory area that the receiving side previously described to the sender. The receiving side's CPU is completely "passive" during the transfer, which is why this is called "one-sided."

The third set is also a "one-sided" transfer in which the receiver posts a RDMA READ request that "pulls" a message directly from the sending side's virtual memory, and the sending side's CPU is completely passive.

Because the passive side in a "one-sided" transfer does not know when that transfer completes, there is another "two-sided" opcode set in which the sender posts a RDMA WRITE WITH IMM request to "push" a message directly into the receiving side's virtual memory, as for RDMA WRITE, but the send work request also includes 4 bytes of immediate (out-of-band) data that is delivered to the receiver on completion of the transfer. The receiving side posts a RECV work request to catch these 4 bytes, and the work completion for the RECV indicates the status and amount of data transferred in the message.

B. Features Evaluated

1) Work Request Opcode Set: Several RDMA features were evaluated for this study. The most obvious feature evaluated was the work request opcode set being used for the transfer, although in practice this choice is often limited by the requirements of the application regardless of performance.

2) Message Size: The second item considered is the message size, which was arbitrarily categorized into small messages containing 512 bytes or less and large messages containing more. This size was chosen since 512 bytes is a standard disk sector; it is not part of any RDMA standard.

3) Inline Data: The API provides an optional "inline" feature that allows an interface adapter to copy the data from small messages into its own memory as part of a posted work request. This immediately frees the buffer for application reuse, and makes the transfer more efficient since the adapter has the data ready to send and does not need to retrieve it over the memory bus during the transfer.

4) Completion Detection: An asynchronous RDMA transfer starts when an application posts a work request to the interface adapter, and completes when the interface adapter enqueues a work completion in its completion queue. There are two strategies which an application can employ to determine when to pick up a work completion.

The first completion detection strategy, called "busy polling", is to repeatedly poll the completion queue until a completion becomes available. It allows immediate reaction to completions at the cost of very high CPU utilization, but requires no operating system intervention.

The second strategy, called "event notification", is to set up a completion channel that allows an application to wait until the interface adapter signals a notification on this channel, at which time the application obtains the work completion by polling. It requires the application to wait for the notification by transferring to the operating system, but reduces CPU utilization significantly.

5) Simultaneous Operations (Multiple Buffers): We set up the use of simultaneous operations per connection by posting multiple buffers in a round-robin fashion so that the interface adapter queues them.

6) Work Request Submission Lists: The functions that post work requests take a linked list of work requests as an argument. We compare the performance of creating a list of work requests and submitting them in a single posting ("multiple work requests per post") with that of posting individual work requests as single element lists ("single work request per post").

7) Completion Signaling: For all transfer opcodes except RECV, a work completion is generated only if a "signaled" flag is set in the work request. If this flag is not set, the "unsignaled" work request still consumes completion queue resources but does not generate a work completion data structure or notification event. To avoid depleting completion queue resources, applications must periodically post a signaled work request and process the generated completion.

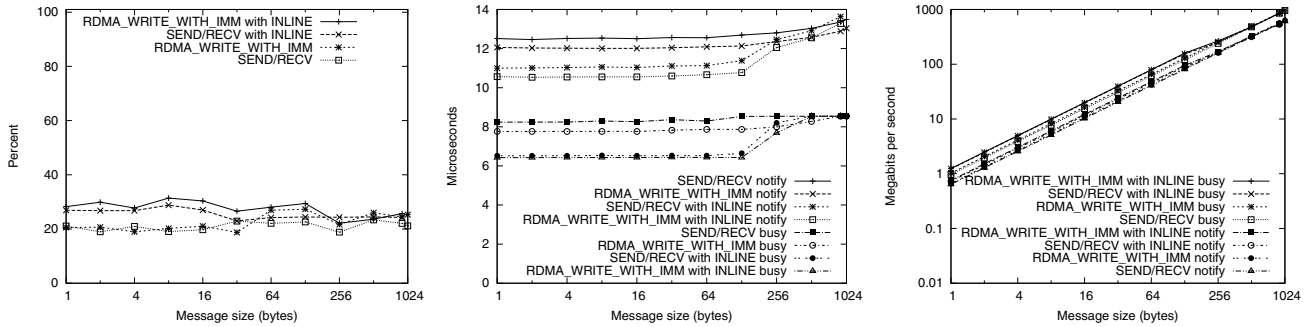
We compare sequences containing only signaled work requests ("full signaling") against sequences containing both signaled and unsignaled work requests ("periodic signaling"). SEND or RDMA WRITE WITH IMM with inline are good examples of where unsignaled work requests could be used because the data area is no longer needed by the adapter once the request is posted, allowing the application to reuse it without first receiving a work completion.

8) Infiniband Wire Speed: Infiniband hardware supports several different wire transmission speeds, and we compare the effect of these speeds on various performance measures.

9) RoCE: We compare the performance of RoCE to Infiniband.

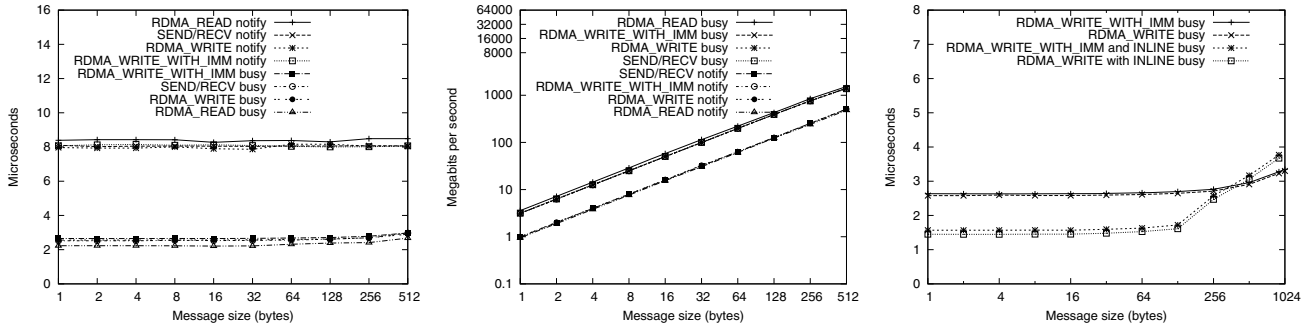
II. Test Procedures

Our tests are variations of two simple applications, ping and blast. In the ping tool, a client sends data to a server and the server sends it back. Ping has variations for SEND/RECV and RDMA WRITE WITH IMM/RECV, but not for RDMA READ or RDMA WRITE because with these opcodes a server cannot determine when a transfer



(a) CPU usage (event notification only). (b) Average one-way time. (c) Average throughput.

Figure 1. Ping with each completion detection strategy for small messages.



(a) Average one-way time. (b) Average throughput. (c) Average one-way time with and without inline, busy polling only.

Figure 2. Blast with each opcode set and completion detection strategy for small messages using one buffer.

has completed. In the blast tool, which can run with each of the 4 opcode sets, a client sends data to a server as fast as possible, but the server does not acknowledge it.

Tests are run between two identical nodes, each consisting of twin 6-core Intel Westmere-EP 2.93GHz processors with 6 GB of RAM and PCIe-2.0x8 running OFED 1.5.4 on Scientific Linux 6.1. Each node has a dual-port Mellanox MT26428 adapter with 256 byte cache line, one port configured for Infiniband 4xQDR ConnectX VPI with 4096-byte MTUs, the other for 10 Gbps Ethernet with 9000-byte jumbo frames. With these configurations, each Infiniband or RoCE frame can carry up to 4096 bytes of user data. Nodes are connected back-to-back on both ports, and all transfers use Reliable Connection (RC) transport mode, which fragments and reassembles large user messages.

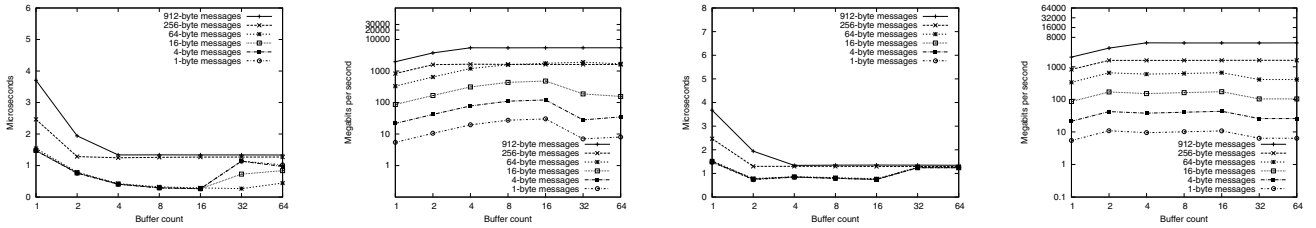
We measure 3 performance metrics, all based on elapsed time, which is measured from just before the first message transfer is posted to just after the last transfer completes. Average throughput is the number of messages times the size of each message divided by elapsed time. Average one-way time per message for blast is the elapsed time divided by the number of messages; for ping it is half this value. Average CPU utilization is the sum of user and system CPU time reported by the POSIX `getrusage` function divided by elapsed time.

III. Performance Results

A. Ping example, small messages

Ping is the application generally used to measure round trip time. It repeatedly sends a fixed-size message back and forth between client and server. In our tests, message size varies by powers of 2 from 1 to 1024 bytes. Figure 1 shows a total of 8 combinations of SEND/RECV and RDMA WRITE WITH IMM/RECV, with and without inline, and with busy polling and event notification.

Figure 1a shows slight differences between the CPU usage by opcode sets, with inline requiring more cycles for messages less than 32 bytes due to the extra data copy involved. Only event notification cases are graphed, as busy polling always has 100% CPU usage. Figure 1b shows clearly that busy polling produces lower one-way times than event notification, and transfers with inline perform better than those without it (6.2 microseconds for messages smaller than the 256 byte cache line). Figure 1c shows that for each



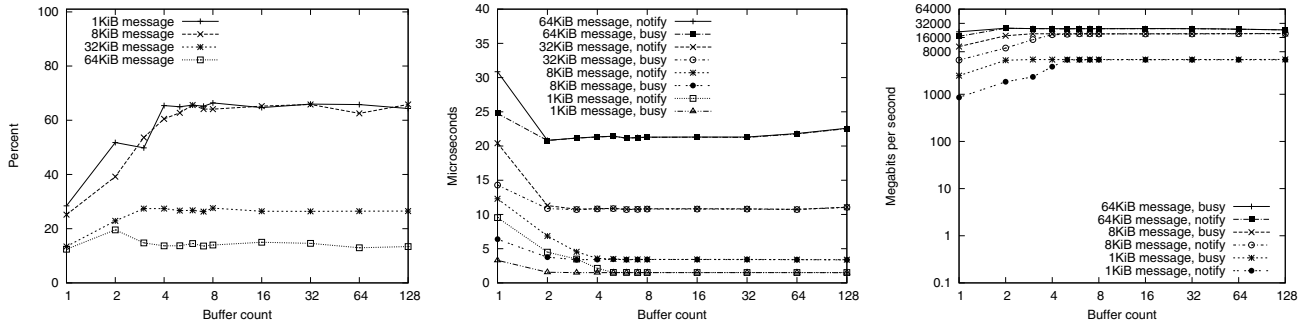
(a) Average one-way time with single work request per posting.

(b) Average throughput with single work request per posting.

(c) Average one-way time with multiple work requests per posting.

(d) Average throughput with multiple work requests per posting.

Figure 3. Blast with RDMA WRITE with busy polling for small messages with inline using multiple buffers.



(a) Active side CPU usage (event notification only).

(b) Average one-way time.

(c) Average throughput.

Figure 4. Blast with RDMA WRITE for each completion detection strategy for large messages using multiple buffers and multiple work requests per posting.

opcode set throughput increases proportionally with message size and is slightly better for busy polling at any given size. There is little difference in throughput between opcode sets, with low total throughput for them all, since small messages cannot maximize throughput. Using either SEND/RECV or RDMA WRITE WITH IMM/RECV with inline and busy polling gives the best one-way time and marginally better throughput for small messages, but suffers from 100% CPU utilization.

B. Blast example, small messages, single buffer

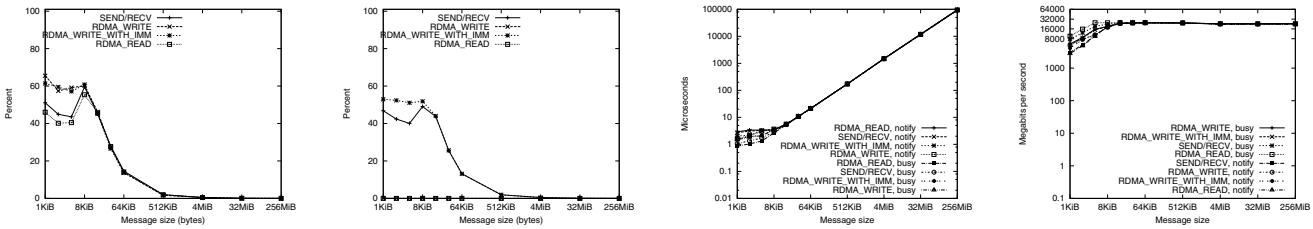
Next is a blast study using small messages. It compares all transfer opcode sets for both busy polling and event notification. As with ping, busy polling cases have lower one-way time and higher throughput, as shown in Figure 2a and Figure 2b respectively. CPU usage for event notification cases is not significantly different between the opcode sets. As expected, RDMA READ with event notification performs poorly. However, RDMA READ with busy polling gives slightly better performance than other opcodes, which is odd, as RDMA READ is expected to perform worse because data must flow from the responder back to the requester, which requires a full round-trip in order to deliver the first bit.

Figure 2c examines the use of inline in WRITE operations for small message blast. This is only done for busy polling as it performs much better than event notification for small messages. Figure 2c shows that one-way time is lowest for both opcodes when using inline and messages smaller than the 256 byte cache line, although our adapters accepted up to 912 bytes of inline data.

C. Blast example, small messages, multiple buffers

Next consider the use of multiple outstanding buffers. We initially post an RDMA WRITE for every buffer, then repost each buffer as soon as we get the completion of its previous transfer. This way, the interface adapter processes posted work queue entries in parallel with the application code processing completions.

In Figure 3a, we vary the buffer count for several message sizes using RDMA WRITE with inline and busy polling. One-way time for messages less than or equal to 64 bytes is only about 300 nanoseconds when using 8 or 16 buffers, and is less than 1 microsecond when using 2 or 4 buffers. For larger buffer counts, one-way time for messages smaller than 64 bytes increases, but remains around 300 nanoseconds for 64 byte messages. Throughput, shown in Figure 3b, increases proportionally with message size, except that throughput of 64 byte messages slightly exceeds that of 256 byte messages for 8 or more buffers, while throughput for smaller messages drops noticeably for 32 or 64 buffers.



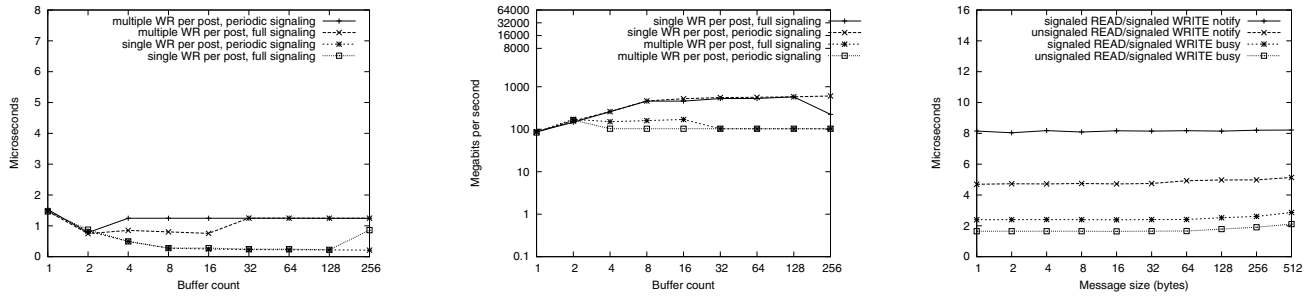
(a) Active side CPU usage (event notification only).

(b) Passive side CPU usage (event notification only).

(c) Average one-way time.

(d) Average throughput.

Figure 5. Blast with each opcode set and each completion detection strategy for large messages using four buffers with multiple work requests per posting.



(a) Average one-way time for blast with RDMA WRITE and busy polling for 16 byte messages, with each work request submission strategy using multiple buffers with multiple work requests per posting.

(b) Average throughput for blast with RDMA WRITE and busy polling for 16 byte messages, with each work request submission strategy using multiple buffers with multiple work requests per posting.

(c) Average one-way time for ping with client issuing RDMA WRITE and RDMA READ, with each completion detection strategy for small messages.

Figure 6. Comparing completion signaling strategies.

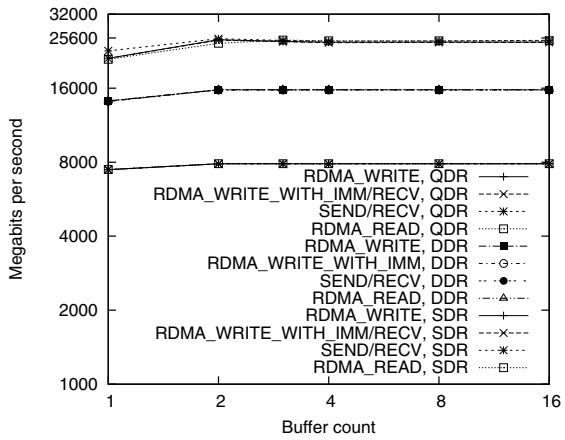
D. Blast example, small messages, multiple buffers, multiple work requests per posting

The next study is identical to the previous, except instead of posting each work request as we process its previous completion, we place it into a list and post that list after processing all available completions. Comparing one-way times in Figure 3c with those in Figure 3a shows that times for 256 and 912 byte messages are unchanged, but for 64 byte and smaller messages they increase when using more than 2 buffers, and for more than 16 buffers they increase to that of 256-byte messages. For messages of 64 bytes or less with 4, 8 or 16 buffers Figure 3d does not show the increase in throughput seen in Figure 3b. Perhaps the time needed to process a large number of completions before posting a single list of new work requests causes the adapter's work queue to empty. In all cases, posting multiple work requests produces less dependence on the number of buffers than does single posting of work requests.

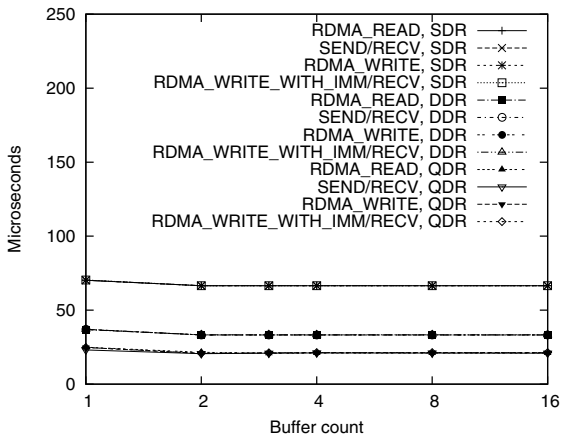
E. Blast example, large messages, multiple buffers

We next examine the effects of the buffer count and message size on large messages, using RDMA WRITE without inline (since inline can be used only with small messages). We vary the buffer count from 1 to 128 for 1, 8, 32, and 64 kibibyte messages, and post multiple work requests per list. Figure 4a shows that 64 kibibyte messages have the lowest CPU utilization when using event notification, and, for all message sizes examined, using more than 4 buffers has little or no effect on CPU utilization. The one-way time, shown in Figure 4b, and throughput, shown in Figure 4c, both increase as message size increases. Also, for both one-way time and throughput, busy polling and event notification results converge given enough buffers (ranging from 2 or more buffers for 64KiB to 5 or more for 1KiB).

Next we study the effect of each opcode set for large message transfers. We vary the message size from 1 kibibyte to 256 mibibytes and use only 4 buffers, as it was just shown that using more buffers produces no performance gains. The CPU usage for the active and passive side of each transfer is shown in Figure 5a and Figure 5b, respectively. Active side CPU utilization generally decreases with message size, although there is a bump around 8KiB. Passive side CPU utilization is always 0 for RDMA WRITE and RDMA READ, but is similar to the active side for SEND/RECV and RDMA WRITE WITH IMM/RECV. One-way time, shown in Figure 5c, and throughput, shown in Figure 5d, both

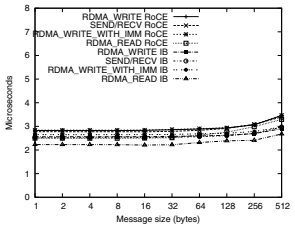


(a) Average throughput.

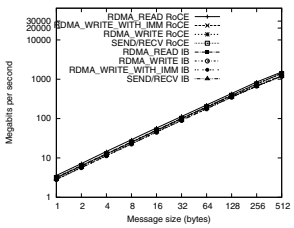


(b) Average one-way time.

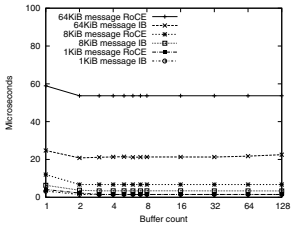
Figure 7. Blast with each opcode set and each Infiniband speed with busy polling for 64KiB messages using multiple buffers with multiple work requests per posting.



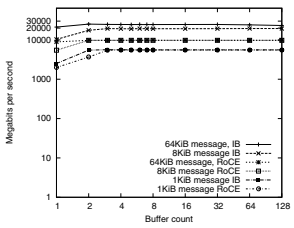
(a) Average one-way time with each opcode set for small messages using one buffer.



(b) Average throughput with each opcode set for small messages using one buffer.



(c) Average one-way time with RDMA WRITE for large messages using multiple buffers with multiple work requests per posting.



(d) Average throughput with RDMA WRITE for large messages using multiple buffers with multiple work requests per posting.

Figure 8. Comparison of QDR Infiniband and RoCE for blast with busy polling.

perform best for busy polling up to about 16 kibibytes, at which point there is no difference between busy polling and event notification. At 32 kibibytes and above, the transfer operation also has no effect on performance.

F. Completion signaling

All tests so far used full signaling. In this test we use blast with RDMA WRITE and 16 byte messages to compare full signaling, where every request is signaled, against periodic signaling, where only one work request out of every $nbuffers/2$ is signaled. Effects are visible only when using more than 2 buffers, since otherwise we signal every buffer. Both one-way time in Figure 6a and throughput in Figure 6b show much better performance when using one work request per post than when using multiple requests per post. But there is no performance difference between full and periodic signaling with one work request per post, and with multiple work requests per post the only effect of periodic signaling is to decrease performance when there are 4, 8 or 16 buffers. We believe this is due to the fact that in the blast example all buffers need processing after they are transferred, so not signaling for a completion just delays that processing until a signaled completion occurs, at which point all buffers transferred up to that time are processed in a big batch, breaking the flow of new transfer postings.

An example without this batching effect is ping with an active client using RDMA WRITE to push messages and RDMA READ to pull them back from a passive server. Every RDMA WRITE can be unsignaled because its buffer does not need any processing after the transfer. Figure 6c shows that one-way time is always lower when the RDMA WRITE is unsignaled, more so with event notification, less so with busy polling. This figure also shows remarkably little variation with message size.

G. Infiniband speed comparison

All previous tests were done at QDR speed, the maximum supported by our adapter. However, Infiniband adapters can be configured to run at several speeds, as shown in Gigabits per second (Gbps) in Table I. Usable Gbps is 20% lower than

raw Gbps due to 8b/10b encoding on serial lines. Figure 7a shows that throughput doubles from near the SDR maximum of 8 usable Gbps to near the DDR maximum of 16, but it does not double again from DDR to QDR. The observed 25.6 Gbps for QDR is about 20% lower than the expected 32 Gbps due to the overhead of the PCIe-2 bus, an impact also noted in [2] and [18]. PCIe-2 overhead also shows up in Figure 7b, where the observed DDR one-way time of 33 microseconds is half the 66 observed for SDR, but the observed QDR time of 21 microseconds is 20% greater than the expected 16.5.

Designation	raw Gbps	usable Gbps	Gbps over PCIe-2
IB 4X SDR	10	8	8
RoCE	12.5	10	10
IB 4X DDR	20	16	16
IB 4X QDR	40	32	25.6

Table I
INFINIBAND AND ROCE SPEEDS.

H. Infiniband and RoCE comparison

We next compare 10 Gbps RoCE with 25.6 Gbps QDR Infiniband. Looking first at small messages for each opcode set, Figure 8a shows one-way times for Infiniband are less than a microsecond lower than those for RoCE, and Figure 8b shows that although QDR Infiniband has much greater maximum throughput than RoCE, there is little observable difference for small messages.

Examining larger messages transferred with RDMA WRITE, the differences between Infiniband and RoCE are greater. One-way time and throughput for 1, 8, and 64 kibibyte messages are shown in Figure 8c and Figure 8d. For all large message sizes, one-way time is roughly proportional to message size and is essentially independent of the buffer count for both technologies, but as message size increases, one-way time for RoCE increases faster than for Infiniband. When using one buffer, the throughput increases with message size for both technologies, but as the buffer count increases, all of the RoCE curves converge near its maximum 10 Gbps, whereas for Infiniband, the 8 and 64 kibibyte curves converge near the higher QDR maximum of 25.6 Gbps and only the 1 kibibyte curve converges at 10 Gbps.

IV. CONCLUSIONS

In all situations performance is much more sensitive to the choice of RDMA options when using small messages than when using large messages.

For all 4 opcode sets with small messages up to 4 kibibytes, much lower one-way time and much higher throughput are achieved by using busy polling rather than event notification to wait for completions. For messages of 16 kibibytes and larger both busy polling and event notification produce the same one-way time and throughput. But busy polling also causes 100% CPU utilization for all message sizes, compared to about 20% with event notification for small messages up to 512 bytes and 0% for messages of 4 mibibytes or more.

Regardless of whether busy polling or event notification is employed, messages smaller than the cache line size give noticeably better one-way times for opcode sets that use inline, with a slight improvement in throughput but slightly higher CPU utilization with event notification. Although the amount of inline data allowed depends on the implementation, it always makes sense to use inline whenever adapters support it.

An application must use the opcode best for its needs. In situations such as ping, where both sides need to know when data arrives, the choice is limited to SEND/RECV and RDMA WRITE WITH IMM/RECV, both of which perform essentially equally. For each of these, busy polling and inline always give better one-way times and throughput, but higher CPU utilization.

For all message sizes the choices for completion detection and inline are more significant factors in determining performance than the opcode, which may be limited by application demands. For example, RDMA READ and RDMA WRITE result in no CPU utilization on the passive side, which may be important for passive side scalability. More often, both sides of a transfer need to know when it completes, in which case SEND/RECV or RDMA WRITE WITH IMM/RECV is best, and if messages are small enough to allow inline then the SEND or RDMA WRITE WITH IMM could also be unsignaled.

It is best to have between 3 and 8 transfers simultaneously queued on the adapter. With small messages this number should be closer to 8; for large messages closer to 3. Using more buffers gives no performance increase, so staying within these limits avoids consuming extra adapter resources. Our studies on buffer numbers do not consider the additional delay introduced when communicating nodes are separated by any significant distance. Clearly a longer communications channel can store more buffers in transit, so that more simultaneously queued buffers would be necessary to keep the channel full, especially when small messages are being transmitted.

In general, rather than collecting work requests into lists it is better to post them individually as soon as possible and let the adapter queue them. This ensures that the connection is kept busy. A list might be used if several work requests must be created and sent together.

Completion signaling has a small performance impact in specialized circumstances. Full signaling should be used if there is any need to process a transfer's completion, but in a situation such as ping with RDMA WRITE followed by RDMA READ, performance improves if the RDMA WRITE is not signaled.

For small messages there is little performance difference between RoCE and Infiniband. For larger messages,

QDR Infiniband's 25.6 Gbps outperforms RoCE's 10 Gbps. Therefore, if an application only transfers small messages or network equipment cost is a significant factor, then RoCE may be appropriate, as it runs over Ethernet wires and switches that may already be installed. Since the API is identical across technologies, an application could be written and tested on RoCE, then migrated to Infiniband. This means that RoCE may be good for initial RDMA programming or development, or in applications where high throughput is not necessary but the other benefits of RDMA are still desired.

Non-RDMA factors are also important. Platforms with PCIe-2 cannot fully utilize an Infiniband 4X QDR link, although fabric switches should be able to handle full traffic volume, even if each endpoint has limited throughput.

ACKNOWLEDGMENT

This research is supported in part by National Science Foundation grant OCI-1127228.

REFERENCES

[1] M. Koop, T. Jones, and D. Panda, "Reducing Connection Memory Requirements of MPI for InfiniBand Clusters: A Message Coalescing Approach," in *Seventh IEEE International Symposium on Cluster Computing and the Grid*, 2007.

[2] M. Koop, W. Huang, K. Gopalakrishan, and D. Panda, "Performance Analysis and Evaluation of PCI 2.0 and Quad-Data Rate InfiniBand," in *Sixteenth IEEE Symposium on High Performance Interconnects*, 2008.

[3] P. Lai, H. Subramoni, S. Narravula, A. Mamidala, and D. Panda, "Designing Efficient FTP Mechanisms for High Performance Data-Transfer over InfiniBand," in *International Conference on Parallel Processing*, 2009.

[4] H. Subramoni, P. Lai, R. Kettimuthu, and D. Panda, "High Performance Data Transfer in Grid Environment Using GridFTP over InfiniBand," in *International Symposium on CLuster Computing and the Grid*, 2010.

[5] B. Li, P. Zhang, Z. Huo, and D. Meng, "Early Experiences with Write-Write Design of NFS over RDMA," in *IEEE International Conference on Networking, Architecture, and Storage*, 2009.

[6] H. Subramoni, G. Marsh, S. Narravula, P. Lai, and D. Panda, "Design and Evaluation of Benchmarks for Financial Applications using Advanced Message Queueing Protocol (AMPQ) over InfiniBand," in *Workshop on High Performance Computational Finance*, 2008.

[7] J. Wu, P. Wyckoff, and D. Panda, "PVFS over InfiniBAnd: Design and Performance Evaluation," Ohio Supercomputer Center, Tech. Rep. OSU-CISRC-5/03-TR25, 2003.

[8] Infiniband Trade Association, "Infiniband Architecture Specification Volume 1, Release 1.2.1," Nov. 2007.

[9] P. Grun, *Introduction to InfiniBand for End Users*. InfiniBand Trade Association, 2010.

[10] P. Culley, U. Elzur, R. Recio, S. Bailey, and J. Carrier, "Marker PDU Aligned Framing for TCP Specification," RFC 5044, Oct. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc5044.txt

[11] H. Shah, J. Pinkerton, R. Recio, and P. Culley, "Direct Data Placement over Reliable Transports," RFC 5041, Oct. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc5041.txt

[12] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia, "A Remote Direct Memory Access Protocol Specification," RFC 5040, Oct. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc5040.txt

[13] Infiniband Trade Association, "Supplement to Infiniband Architecture Specification Volume 1, Release 1.2.1: Annex A16: RDMA over Converged Ethernet (RoCE)," Apr. 2010.

[14] B. Metzler, F. Neeser, and P. Frey, "Softiwarp: A Software iWARP Driver for OpenFabrics," www.openfabrics.org/archives/spring2009sonoma/monday/softiwrp.pdf, 2009.

[15] System Fabric Works, "Soft RoCE," www.systemfabricworks.com/downloads/roce, 2011.

[16] OpenFabrics Alliance, "http://www.openfabrics.org."

[17] OpenFabrics Enterprise Distribution, "www.mellanox.com/pdf/products/software/OFED_PB_1.pdf," 2008.

[18] National Instruments, "PCI Express – An Overview of the PCI Express Standard," http://zone.ni.com/devzone/cda/tut/p/id/3767, 2009.