A Performance Study Of InfiniBand Fourteen Data Rate (FDR)

Qian Liu, Robert D. Russell
Department of Computer Science, University of New Hampshire
Durham, New Hampshire 03824, USA
qga2@unh.edu, rdr@unh.edu

Keywords: RDMA, InfiniBand, Fourteen Data Rate, NUMA, RDMA_WRITE_WITH_IMM

Abstract

This paper evaluates the performance of Remote Direct Memory Access (RDMA) using the InfiniBand Fourteen Data Rate (FDR) Channel Adapter (CA). InfiniBand RDMA allows applications to move data directly between nodes without kernel intervention or extra data copying. While RDMA transfer is fast, this study shows that performance can be hindered by factors such as message buffer misalignment, inefficient work request signaling, cache limitations, and inefficient CPU and NUMA memory access. This paper presents recommendations for achieving optimal performance when using the RDMA_WRITE_WITH_IMM verb. We are able to improve performance by up to 40% in some cases with no extra overhead. In addition, we develop formulas to predict the best combination of some user-level RDMA operational parameters which could generate optimal throughput.

1. INTRODUCTION

Remote Direct Memory Access (RDMA) provides a reliable, message-oriented[1] transport protocol to directly transfer data between user-space virtual memory on different machines without kernel intervention and extra copying. In this paper, all access to RDMA is via the OFA verbs interface[2]. Unlike the traditional TCP/IP protocol, an RDMA implementation provides two types of transfer semantics[3]: channel semantics and memory semantics. Channel semantics are "two-sided" SEND/RECEIVE operations, which means both the sender and receiver must post a work request (WR) for data transfer. In either type of WR, buffers used to send/receive must be specified, and since those buffers don't have to be continuous, a list of Scatter Gather Elements (SGE) is used in the WR: on the sending side it is responsible for gathering buffers to send out; on the receiving side it is responsible for scattering received data into buffers.

On the other hand, memory semantics, only supported in Reliable Connection (RC), are RDMA READ/WRITE operations, which require only the "active" side to post a WR. In this case, the other side is totally "passive" during data transfer. Therefore, memory semantics are "one-sided" operations. After exchanging its local registered memory buffer information with the active side, the passive side does not have to do anything to transfer the data. It is its channel adapter (CA)'s responsibility to direct data into its local memory in response to the active side's RDMA_WRITE; or direct data out of its local memory in response to an RDMA_READ.

Because it is passive in a data transfer, there is no way for the passive side to know when a data transfer is finished. For applications that must know when transfer completes, using only the memory semantics is not suitable. However, the RDMA_WRITE_WITH_IMM operation solves this problem for such applications. This operation is an RDMA_WRITE augmented with 4 bytes of immediate data in the header of the final packet of each message, which causes the passive side's CA into whose local memory the data was written to generate a work completion, which tells the passive side that the transfer is finished. In this case, the passive side is not totally "passive", because it must post a receive WR prior to the start of the data transfer. This WR could be a "dummy" WR, it doesn't have to specify a SGE since the memory location for received data was exchanged previously, and the immediate data is stored in the work completion at the passive side. Furthermore, the value of such immediate data can be used by applications to identify the nature of the message[4].

2. EXPERIMENTAL CONFIGURATION

In order to perform an RDMA_WRITE_WITH_IMM operation, the active sender must know the passive receiving side's memory location into which it is going to write the data. This memory information could be exchanged by using SEND/RECEIVE. In addition, the passive receiving side must post a "dummy" WR for each RDMA_WRITE_WITH_IMM request before data transfer is started by the active sender. In our testing program, the passive receiver sends an explicit ACK back after it receives a message, and in turn, the active sender has the credit to send a new message out after it receives such an ACK from the receiver. The sending of RDMA_WRITE_WITH_IMMs by the sender and the sending of ACKs by the receiver are "batched", so that most of these sends are not signaled, thus reducing completion processing overhead[5]. Further investigation of this topic is in section 3.7.

Our test program uses a wait-wakeup mechanism by default: when the receiver is waiting for messages, it goes to sleep and releases the CPU. When at least one message is received, the receiver is awakened by a completion notification event generated by CA. Likewise, after posting RDMA_WRITE_WITH_IMM WRs, the sender goes to sleep and releases the CPU, and it wakes up when at least one message finishes sending. All transfers use RDMA Reliable Connection (RC) mode.

Tests are performed between two identical nodes, each consisting of a twin 4-core Intel Xeon 2.4GHz Sandy Bridge CPU and two NUMA nodes, with 64GB RAM and PCIe-3.0, running OFED 3.5[2] on Scientific Linux 6.3 with kernel version 2.6.32. Each node has a Mellanox[6] MT27500 CA with 64-byte cache line and firmware version 2.11.1250. The CA port is configured for InfiniBand[3][7] 4X FDR ConnectX-3 with 4096-byte MTU. Nodes are connected via a Mellanox SX6036 FDR-capable switch. Hyper-Threading[8] technology and CPU throttling are disabled.

FDR performance in this paper is measured at the OFA verbs interface[2]. Other papers[9] have dealt with high-level HPC and Cloud performance over FDR. We focus on two metrics: throughput and CPU time. Throughput is user payload throughput, which is computed as the total amount of user data transferred divided by the elapsed time, where elapsed time refers to the total real time between when the first transfer begins and the last transfer finishes. CPU time refers to the total time the program spends in user space and kernel space. When using the wait-wakeup mechanism, the time a program waits for work completions is not included in either CPU time, since the program releases the CPU during that time.

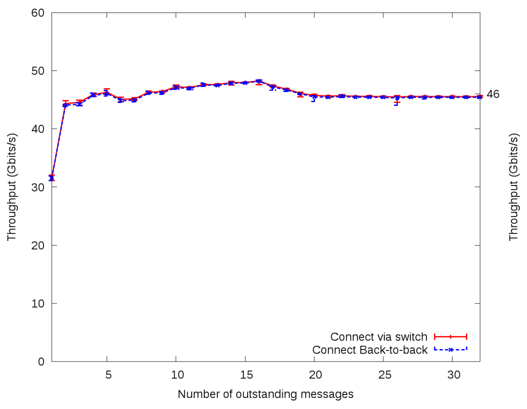
In our tests, we perform memory-to-memory data transfers using the RDMA_WRITE_WITH_IMM operation for different message sizes. In general, larger messages result in higher throughput because there is less overhead per message. For that reason, we limit our studies to message sizes of 32 kibi (Ki) bytes or more. (Messages smaller than this are preferred when minimum latency, not maximum throughput, is the desired goal.) In order to get accurate performance measurements, the sender should send enough messages to let performance settle down to a stabilized point. According to our tests, sending data for 10 seconds keeps performance at a stable level. We send 64Gi bytes of data from the sender to the receiver in all our tests.

3. PERFORMANCE RESULTS

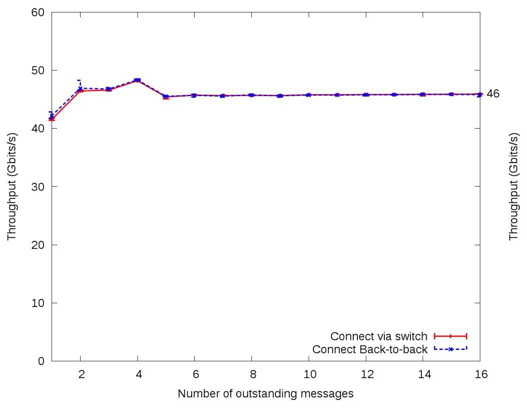
3.1. Effect Of CA Connection

In most of our tests, both CAs are connected via a switch. However, CAs can also be connected back-to-back. Compared to a back-to-back connection, obviously there should be a small amount of overhead introduced by the switch, due to the look up and forwarding according to the DLID and SL fields in incoming packets[4].

Figure 1 shows the throughput comparison of these two connection configurations, in which the solid line (connection via switch) is overlapped with the dashed line (connection back-to-back). This indicates that, regardless of message size and number of outstanding messages, there is nearly no performance difference if the switch is not heavily loaded.



(a) Message size = 256Ki.



(b) Message size = 1Mi.

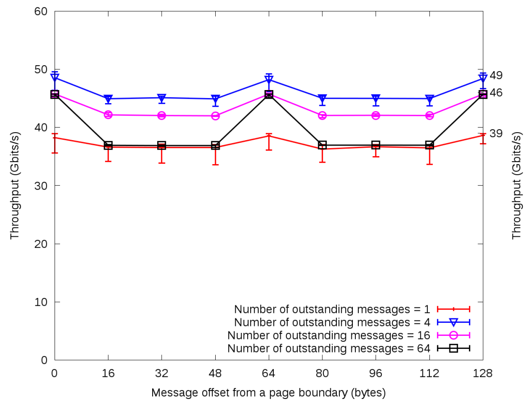
Figure 1: Throughput comparison between different CA connections

3.2. Effect Of Data Alignment

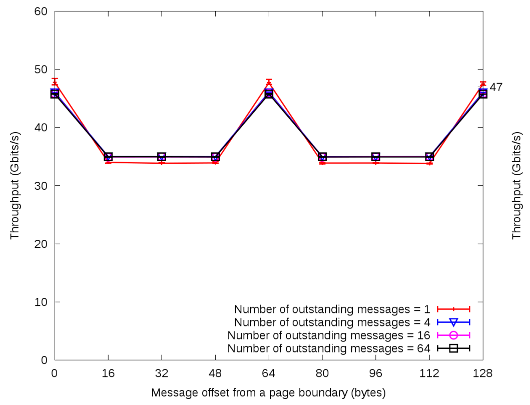
Most conventional Network Interface Controllers require allocated buffers to be aligned on specific byte boundary for efficient memory access[10]. Although there is no such requirement in the InfiniBand specification[3], we evaluated such behavior to see if it has a performance impact.

For each message, we allocated a block of memory aligned on a page boundary (4096 bytes), but the message address used in the transfer is setup at different offsets within that block. Figure 2 shows the throughput comparison among different message offsets.

For different numbers of outstanding messages, Figure 2 illustrates that throughput is always better when the message is aligned on a multiple of 64-bytes, which is the processor's cache line size in our systems. PCIe-3 allows an I/O device to read/write its data directly from/to CPU caches in order to provide stronger coupling between system cache/memory hierarchy and the I/O device[11]. This feature brings significant improvement in HPC communications. The Mellanox FDR HCA seems to be doing this, as shown in Figure 2. When message size is 512Ki, there are obvious differences among various offsets: if a message is not aligned on a multiple of the cache line size, throughput downgrades by up to 20% if the number of outstanding messages is greater than or equal to 4. However, if the number of outstanding messages is 1, the difference among offsets is not obvious as CA utilization is low in this case: it takes very little time to send data, after which the CA is idle until the next send WR is posted.



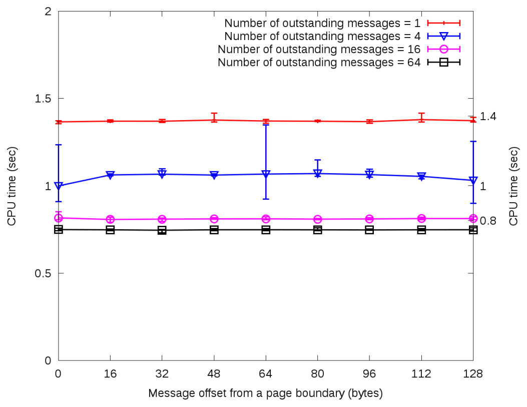
(a) Message size = 512Ki. 5% to 23% throughput improvement if message is aligned on cache line size



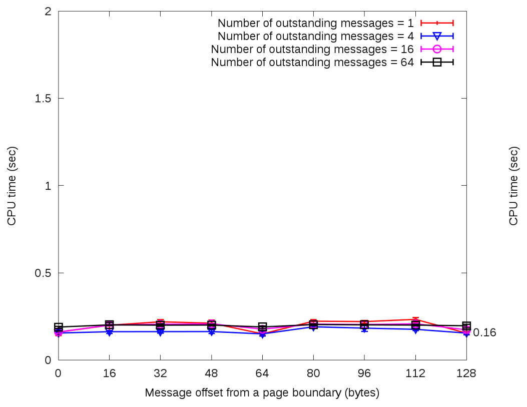
(b) Message size = 8Mi. 40% throughput improvement if message is aligned on cache line size

Figure 2: Throughput comparison among different message offsets

On the other hand, when the message size is large enough, CA is always in busy state. In other words, there are always unsent packets queued up in the channel. In this case, the CA is working much more than it is idle, so the throughput curves under various numbers of outstanding messages are overlapped, and the difference caused by various numbers of outstanding messages is much less. Figure 2b illustrates this condition for 8Mi messages, where throughput is



(a) Message size = 512Ki.



(b) Message size = 8Mi.

Figure 3: Total CPU time comparison among different message offsets

improved by 40% (from 34Gbits/s to 47.75Gbits/s) if messages are aligned on a multiple of the cache line size.

When messages are aligned on a multiple of the cache line size, throughput is much better than with other message buffer alignments, and this higher throughput does not require extra system resources. Figure 3 shows that the CPU time (total time program spent in user space and kernel space) for the two message sizes used in Figure 2 is essentially the same for all message offsets, which proves message alignment on a multiple of the cache line size doesn't add extra overhead. On the contrary, this alignment is a significant factor which will improve RDMA transfer throughput, and we used it in all other tests in this paper, including those in Figure 1.

3.3. Effect Of User-level Parameters

Basically, there are two key user-level parameters that have a significant impact on throughput:

1) The number of bytes in each message (Message Size). As mentioned before, we limited our studies to message sizes of 32Ki bytes or more in order to get maximum throughput.

2) The number of messages that are simultaneously out-

standing. In general, more outstanding messages result in higher throughput because there is better utilization of the CA and fabric resources.

Figure 4 illustrates throughput for different message sizes with various numbers of outstanding messages. Both user-level parameters have an obvious effect on throughput. When the message size is relatively small, such as 32Ki, throughput is very low if the number of outstanding messages is also small, due to the low CA utilization under such conditions. For instance, if the sender only sends one 32Ki message each time, and then waits for the receiver's ACK before sending the next message, the CA is idle until the next send WR is posted and throughput is only 10 Gigabits per second (Gbits/s). On the other hand, if the sender posts 16 or 32 32Ki messages at the same time, then throughput is much higher, more than 45Gbits/s. However, if the message is relatively large, for instance, more than 8Mi, the throughput achieved with different numbers of outstanding messages is almost identical, since such messages are large enough to always keep the CA busy. Therefore, even if the number of outstanding messages is 1 in this case, the idle time the sender waits for an ACK from the receiver is not significant.

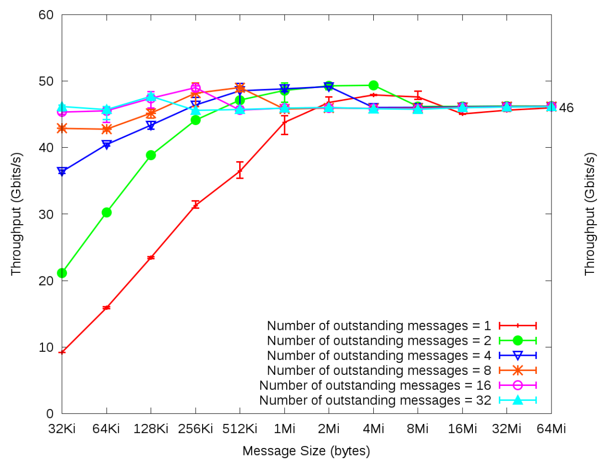


Figure 4: Throughput under different message sizes and numbers of outstanding messages

Figure 5 shows the CPU times for different message sizes with various numbers of outstanding messages. As mentioned before, in order to transfer the same amount of data from sender to receiver, more messages have to be sent if each message is small. When the message size is 32Ki, and the number of outstanding messages is 1, Figure 5 shows that the total CPU time is close to 20 seconds when sending 64Gi bytes of data. Due to the wait-wakeup mechanism, the program more frequently goes to sleep and is then awakened by a completion notification event. Therefore, it spends considerable time in user and kernel space. On the other hand, when

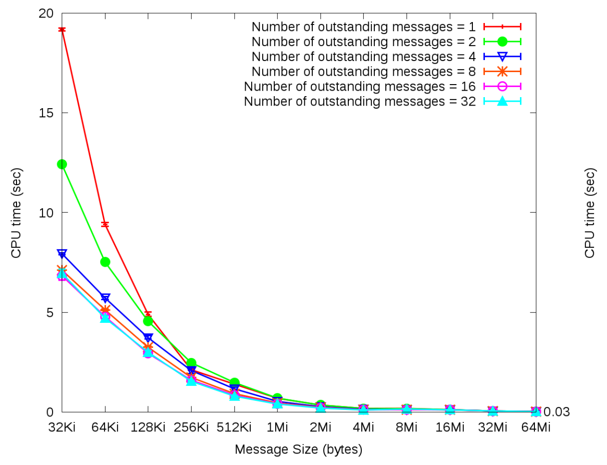


Figure 5: CPU time under different message sizes and numbers of outstanding messages

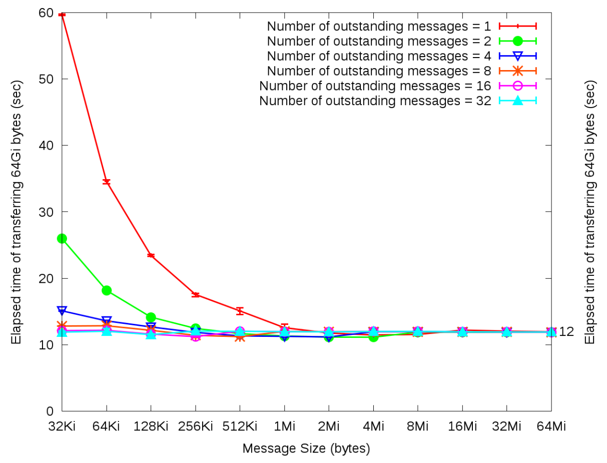


Figure 6: Total elapsed time to transfer 64Gibibytes of data

the message size is larger, fewer messages need to be sent, so the program is less frequently put to sleep and awakened. Therefore, it spends less time in user and kernel space.

Figure 6 illustrates the total elapsed time to transfer 64Gi bytes of data. In this figure, as either message size or number of outstanding messages increases, less total time is needed. This behavior is related to CA utilization. When CA utilization is very low, i.e. message size is 32Ki and the number of outstanding messages is 1, total elapsed time is close to 60 seconds. The program spends too much time in user and kernel space, and it takes more time to transfer a fixed amount of data to the receiver because of low CA utilization. On the other hand, if the number of outstanding messages is 32, as shown by the solid line with triangle points in Figure 6, although there are still more messages to be sent and the program has more chance to be awakened in this case, CA utilization is much higher because messages are queued up in the channel all the time. High CA utilization results in high throughput. Therefore, the total time is around 12 seconds.

3.4. Effect Of Caching

In Figure 1 and 4, there are a few points where the throughput drops. For instance, in Figure 1b, if the number of outstanding messages is 4, throughput is at its highest level of about 49Gbits/s. However, throughput drops to about 46Gbits/s if the number of outstanding messages increases to 5, a drop of about 6%. Throughput levels off at the lower throughput level as the number of outstanding messages increases further. Likewise, in Figure 4, for message sizes such as 2Mi, 4Mi, etc, there is always a number of outstanding messages which generates the highest throughput, after which throughput drops if that number increases. This drop is due to a limitation on the size of the caches involved in data transfer.

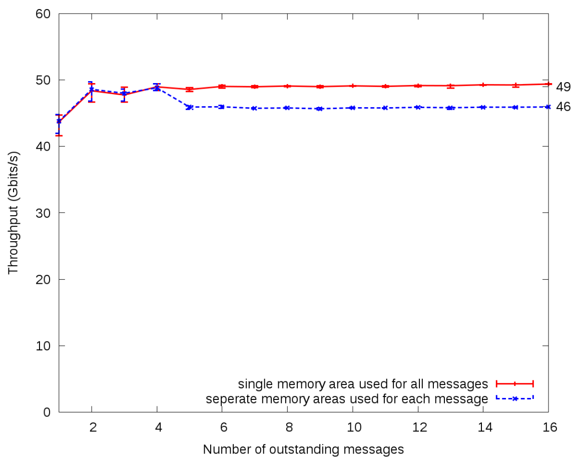
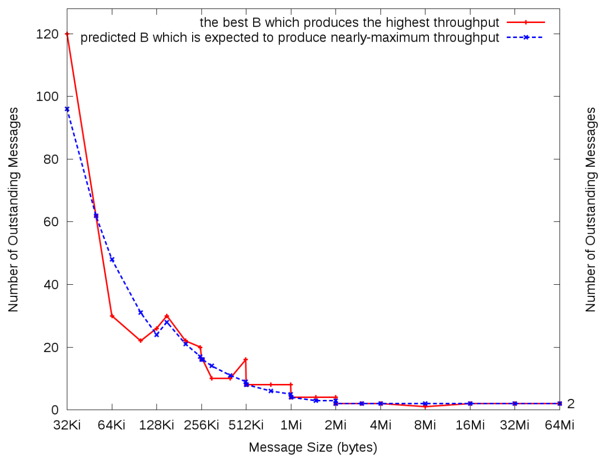


Figure 7: Throughput impacted by cache hit and cache miss for 1Mebibyte messages

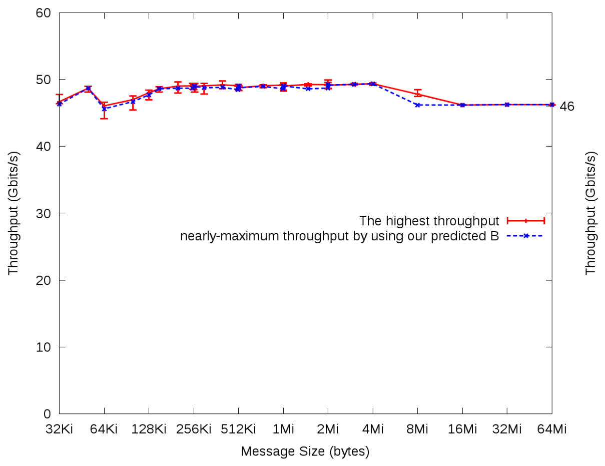
Figure 7 compares the throughput (dashed line) when each message uses a separate memory area so that the cache is never hit, with the throughput (solid line) when the cache is always hit, which means no cache item is ever replaced even though the number of outstanding messages specified is greater than the number of outstanding messages which can generate the highest throughput. The solid line results were produced by using the same memory area in each WR to guarantee that a cache item would always be hit.

The dashed line shows throughput is at its highest level if the number of outstanding messages is 4, drops if that number increases to 5, and then levels off at the lower throughput level for 6 or more outstanding messages. However, the solid line shows that if there is no cache miss when sending more than 4 messages simultaneously, throughput does not drop, but remains at its highest level.

This throughput drop could be prevented by utilizing the same memory area for all messages. However, this is not always realistic, since data for different messages may be stored in different memory locations. In order to avoid such a cache limitation and get the highest throughput, we present an approach to predict the number of outstanding messages for various message sizes to achieve their highest throughput.



(a) Number of outstanding messages (B) comparison



(b) Throughput comparison

Figure 8: The predicted “best B” and the observed “best B”

3.5. Predicting Optimal Number Of Outstanding Messages

For each message size, there is one number of outstanding messages (B) which generates the highest throughput, called “best B”. Because of the cache limitation, throughput will drop if more than B messages are simultaneously outstanding. In order to avoid this cache impact, Equation 1 is used to predict a nearly optimal “best B” that gives a nearly maximal throughput for each message size.

$$bestB = \begin{cases} min(\lceil \frac{3Mi}{MessageSize} \rceil, 128) & MessageSize \leq 128Ki \\ max(\lceil \frac{4Mi}{MessageSize} \rceil, 2) & MessageSize > 128Ki \end{cases} \quad (1)$$

For relatively larger messages, posting only one message each time could cause a time gap between messages, during which the CA is idle. In order to avoid this gap, it’s better to send at least 2 messages simultaneously for message sizes greater than 4Mi. For relatively smaller messages, the maximum number of outstanding messages shouldn’t be greater than 128 in order to avoid the cache limitation.

Figure 8a compares the observed “best B” (solid line) with the predicted “best B” (dashed line) from Equation 1. Figure 8b compares the highest observed throughput with the throughput generated by using our predicted “best B”.

The dashed line in Figure 8b shows our predicted “best B” produces the maximum throughput in most cases. For a few message sizes throughput achieved is at worst 1% lower than the maximum. The only exception is at message size 8Mi, where throughput is 3% lower than the maximum, since the predicted “best B” is 2, compared to the observed “best B” of 1.

3.6. Effect Of Completion Notification

RDMA transfer is asynchronous[3]: transmission starts when a WR is posted to the CA’s send queue, and finishes when a corresponding work completion is generated in the completion queue. An application can employ two strategies to monitor such completions. One is the wait-wakeup mechanism used in most of our testing: the program goes to sleep while the CA is sending or receiving data, and it is awakened when the CA generates a work completion. Both going to sleep and waking up require OS intervention. The other strategy is busy-polling, which means the application will not go to sleep but poll the completion queue repeatedly until a work completion generated. Under such a strategy, throughput should be better since there is no overhead caused by the OS, so the program will get completed events in real time. Figure 9 shows the highest throughput difference between the wait-wakeup (solid line) and busy-polling (dashed line) mechanisms.

Using busy-polling (dashed line), the highest throughput achieved by each message size is slightly more than 50Gbits/s in most cases. However, if the message size is greater than 8Mi, throughput drops to 47Gbits/s. This drop is caused by the cache limitation discussed previously. When using wait-wakeup (solid line), the highest throughput is slightly less than 50Gbits/s, which is never as high as the dashed line. This is entirely due to the OS intervention needed by wait-wakeup in order to put a program to sleep, deliver a completion notification to it, and then reschedule it onto a processor. Busy-polling completely bypasses the OS, since the user program directly polls the CA.

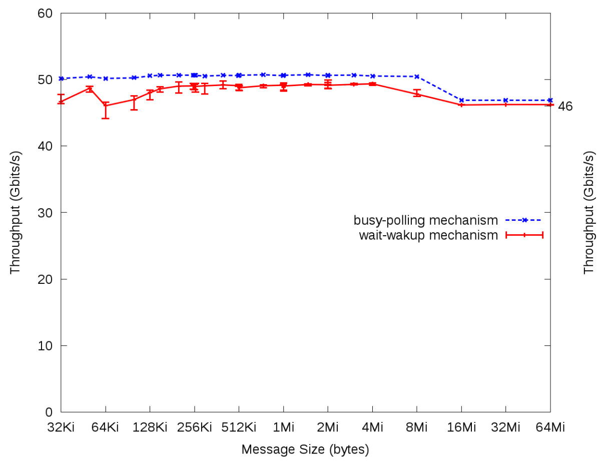


Figure 9: The highest observed throughput comparison between busy-polling and wait-wakeup mechanism

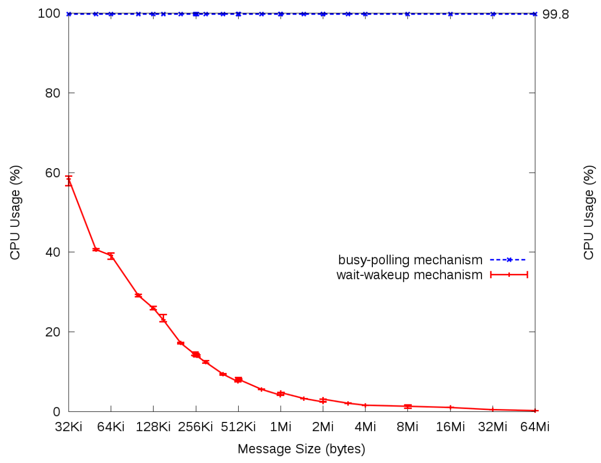


Figure 10: Comparison of corresponding CPU usage for Figure 9

Although it can achieve higher throughput, busy-polling spends too much CPU time repeatedly polling the completion queue during which it never releases the CPU. For those points where the highest throughput is achieved in busy-polling and wait-wakeup, as shown in Figure 9, Figure 10 compares their corresponding CPU usages. The trade-off is obvious. Busy-polling (dashed line) always takes 100% of the available CPU cycles, while wait-wakeup (solid line) uses fewer CPU cycles as the message size increases, since there are fewer messages that have to be sent for the same total amount of data when the message size is large. CPU time is close to 0% for large enough messages. Users can choose between these two options depending on the relative importance of increased throughput or increased CPU usage.

3.7. Effect Of Completion Signaling

[5] compares full signaling (signal every send WR) and periodic signaling (only signal a send WR for every B/2 send WRs, in which B is the number of outstanding messages) for RDMA_READ and RDMA_WRITE with inline data. In this paper, we investigate an approach to improve throughput by periodically signaling a send WR for bulk data transmission using RDMA_WRITE_WITH_IMM. Note that receive WRs are always signaled, the application has no choice.

Whether a RDMA_WRITE_WITH_IMM WR is signaled or not has an effect on performance, because a signaled send WR generates a work completion which will be caught and processed by the OS if wait-wakeup is used. Accordingly, overhead is added by this OS intervention. An unsignaled send WR, on the other hand, will not generate a work completion or extra OS overhead. The ideal situation would be if all send WRs were unsignaled, because then no extra overhead would be added. However, if that were done, the send queue (SQ) and/or the completion queue (CQ) would eventually overflow, since unsignaled send WRs take resources in these queues. In order to avoid such queue resource depletion, a signaled WR must be sent periodically, once every S WRs. The completion of this signaled WR will "confirm" that previous unsignaled WRs posted for the same SQ and CQ actually completed, and it will also release the queue resources taken by those unsignaled WRs.

In order to improve performance, S should be large enough to guarantee that the added overhead be as low as possible. On the other hand, S should be small enough to guarantee that completion of the signaled WR would release the resources taken by unsignaled WRs in time to prevent overflow in the CA's queues. We use the following formula to determine S.

$$S = \begin{cases} min(\frac{B}{2}, 1) & 16Ki < MessageSize < 128Ki \\ min(\frac{SQ\_DEPTH}{2}, \text{SQ\_DEPTH - B}) & otherSizes \end{cases} \qquad (2)$$

In Equation 2, SQ depth is the actual send queue depth set by the CA. On some CAs, the actual SQ/RQ/CQ depths are not always identical to the values given by the user with the ibv_create_qp verb. The CA may increase those values based on considerations specific to its implementation.

According to our findings, some small messages will gain a small performance improvement if every send WR is signaled, but for most message sizes, especially larger messages, only sending a signaled WR periodically according to our formula helps to fully utilize the CA and improve performance.

Figure 11 compares the throughput when using wait-wakeup for various message sizes sent using our predicted "best B". Three mechanisms for how often to send a signaled WR are compared: signal every WR sent (solid line with circle points); send a signaled WR for every B/2 WRs (solid line); and send a signaled WR for every S WRs (solid line with triangle points), where S is determined by Equation 2.

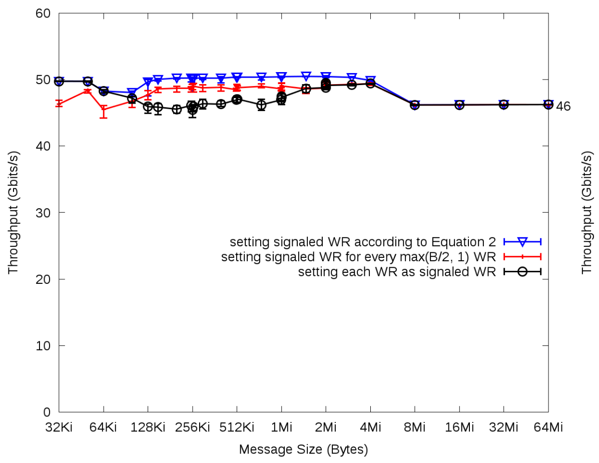


Figure 11: Throughput comparison of various WR signaling mechanisms. Predicted "best B" is used for the number of outstanding messages

The solid line with triangle points in Figure 11 shows that by using our Equation 2 to decide when to send a signaled WR, throughput can be maximized over all message sizes. For most message sizes, throughput improves by 5% to 10% when compared with other WR signaling strategies. For message sizes greater than 128Ki, throughput is greater than 50Gbits/s, which was shown in section 3.6 to be the highest throughput achieved under busy-polling. This demonstrates that periodically sending a signaled WR according to Equation 2 under wait-wakeup not only produces the best throughput (50Gbits/s) achieved under busy-polling, but saves many CPU cycles compared to busy-polling. For message sizes greater than or equal to 8Mi, there is almost no difference between these 3 signaling mechanisms, since the impact of the cache limitation is greater than the reduced overhead contributed by not processing the additional completions.

3.8. Effect Of NUMA Affinity

Multi-core nodes with Non-Uniform Memory Access (NUMA) are common for HPC platforms. With NUMA, memory is physically distributed into different memory banks, and the times required for a processor to access various memory banks are different. Thus, on systems with more than one NUMA node, performance would be better when a program uses the local NUMA node to which the CA is connected[12][13].

Therefore, another element which could improve performance is CPU and memory affinity, because the overhead introduced by physical memory access could be minimized. On the machines used for our tests, there are two NUMA nodes, and our CAs are connected to NUMA node 0 according to the following detection command given in [13].

$$cat\ /sys/class/infiniband/[interface]/device/numa\_node \quad (3)$$

Figure 12 compares four affinity combinations: (i) both receiver and sender running without any affinity (the way all previous tests were run); (ii) only receiver running with CPU and memory affinity; (iii) only sender running with CPU and memory affinity; (iv) both receiver and sender running with CPU and memory affinity. The "best B" is used for each message size to achieve the best throughput.

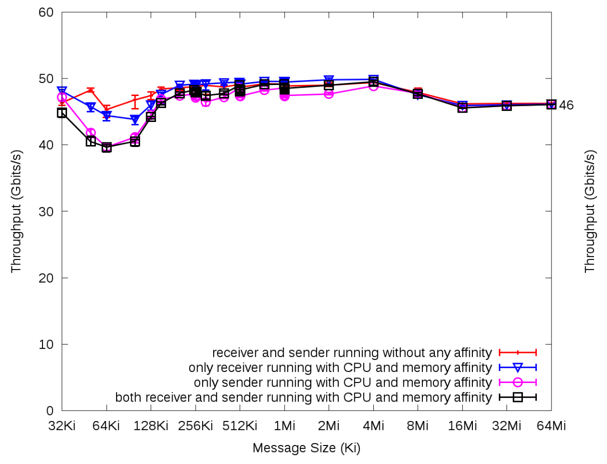


Figure 12: Throughput comparison when using NUMA affinity, the observed "best B" is used for the number of outstanding messages

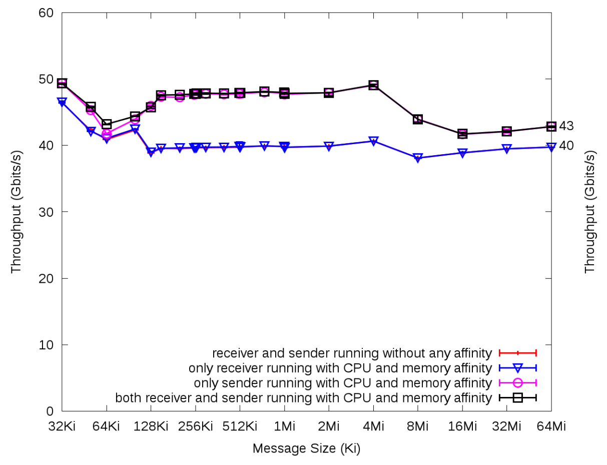


Figure 13: Throughput comparison when using NUMA affinity, the observed "best B" is used for the number of outstanding message

From Figure 12, if only the receiver is running with CPU and memory affinity (solid line with triangle points), throughput is better than our original tests which ran without any affinity (solid line) at message sizes 32Ki, 256Ki, 512Ki, 1Mi, 2Mi, 4Mi and 8Mi. If only the sender is running with CPU and memory affinity (solid line with circle points), throughput is lower than that of our original test in most cases. When both sender and receiver are running with CPU and memory affinity (solid line with box points), performance does not improve. It seems that for RDMA_WRITE_WITH_IMM, only running the receiver side with CPU and memory affinity improves performance, and then only for some message sizes.

However, on two other nearly identical platforms, which are also connected via the same Mellanox SX6036 switch, throughput shows different behavior when using NUMA affinity. Figure 13 illustrates this affinity comparison, in which the "best B" is used for each message size to achieve the best throughput. Compared to running both sender and receiver without affinity (solid line), running both sender and receiver with CPU and memory affinity (solid line with box points) will improve throughput significantly by up to 20%.

We believe that differences in machine architecture cause the difference in performance when using NUMA affinity. One of these two nearly identical nodes has a twin 4-core Genuine Intel 2.3GHz Sandy Bridge CPU and two NUMA nodes, the other has a twin 8-core Genuine Intel 2.6GHz Sandy Bridge CPU and two NUMA nodes. Both nodes have 64GB RAM and PCIe-3.0 running OFED 3.5[2] on Scientific Linux 6.3 with kernel version 2.6.32. Both nodes have a Mellanox[6] MT_27500 Family CA with 64-byte cache line and firmware version 2.11.500. The CA port is also configured for InfiniBand[3][7] 4X FDR ConnectX-3 with 4096-byte MTU. The interaction between NUMA affinity and machine architecture deserves further investigation.

4. CONCLUSION

According to our findings, connecting CAs through a switch versus back-to-back does not cause any significant difference in RDMA performance, whereas different completion notification methods do have a significant impact. However, the relatively higher throughput achieved under busy-polling is obtained at the cost of 100% CPU time.

In all cases performance is much more sensitive to message alignment, with better throughput if the starting address of a message is aligned on a multiple of the cache line size. When using wait-wakeup, the highest throughput we can get is about 50Gbits/s, lower than the expected 54.54Gbits/s, the theoretical maximum throughput for FDR 4X. This is due to protocol and other overheads such as the PCIe-3 bus overhead, an impact also noted in [14].

Cache limitation has a negative impact on RDMA performance. By using our Equation 1, this limitation can be avoided, and the nearly optimal number of outstanding messages (B) can be predicted for each message size. The throughput achieved by using the "predict B" is always close to or equal to the best achievable throughput.

Periodically sending a signaled WR not only prevents overflow in the CA's queues, but reduces overhead caused by completion processing on the sending side of a transfer, which in turn improves the performance. By using our Equation 2 to predict the signaling frequency, throughput can be maximized over all message sizes. Compared to other WR signaling strategies, throughput can be improved by 5% to 10% by using our Equation 2 to periodically send a signaled WR. Under wait-wakeup, sending a signaled WR periodically achieves the best throughput (50Gbits/s) produced under busy-polling. Unlike busy-polling, this best throughput is achieved with fewer CPU cycles.

NUMA affinity also has effect on RDMA performance. Due to the large amount of data usually transferred in industry and scientific applications, memory accessing time becomes a critical factor which impacts performance. Specifying NUMA affinity may provide better performance, although it doesn't always improve performance. This effect could be caused by differences in machine architectures, which needs future investigation.

5. ACKNOWLEDGMENT

The authors would like to thank our funding agency. This research is supported in part by National Science Foundation grant OCI-1127228.

REFERENCES

[1] Recio, Renato, Bernard Metzler, Paul Culley, Jeff Hilland, and Dave Garcia. 2007. "A Remote Direct Memory Access Protocol Specification". RFC 5040. Internet Engineering Task Force. `http://www.ietf.org/rfc/rfc5040.txt`

[2] OpenFabrics Alliance. `http://www.openfabrics.org`

[3] Infiniband Trade Association. 2007. *Infiniband Architecture Specification Volume 1, Release 1.2.1.* `http://www.infinibandta.org/`

[4] Shanley, Tom. 2002. *Infiniband Network Architecture.* Addison Wesley, Reading, MA.

[5] Macarthur, Patrick, and Robert D. Russell. 2012. "A Performance Study to Guide RDMA Programming Decisions." 2012 IEEE 14th International Conference on High Performance Computing and Communications, HPCC '12, (Liverpool, UK, June 25-27). IEEE, New York, NY, 778-785.

[6] Mellanox Technologies. `http://www.mellanox.com`

[7] Grun, Paul. 2010. "Introduction to InfiniBand for End Users." InfiniBand Trade Association. `https://cw.infinibandta.org/document/dl/7268`

[8] Intel. "Intel Hyper-Threading Technology." `http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html`

[9] Vienne et al. 2012. "Performance Analysis and Evaluation of InfiniBand FDR and 40GigE RoCE on HPC and Cloud Computing Systems." In High-Performance Interconnects, 20th Annual Symposium on (Santa Clara, CA, 22-24 Aug). IEEE, New York, NY, 48-55.

[10] Bryant, Randal, David R. O'Hallaron. 2011. *Computer Systems: A Programmer's Perspective (2nd Edition).* Prentice-Hall, Boston, MA.

[11] Ajanovic, Jasmin. 2009. "PCI Express 3.0 Overview." `http://www.hotchips.org`

[12] Majo, Zoltan, and Thomas Gross. 2011. "Memory System Performance in a NUMA Multicore Multiprocessor." In Proceedings of the 4th Annual International Conference on Systems and Storage, SYSTOR '11, (Haifa, Israel, May 30-June 1). ACM, New York, NY, 127-136.

[13] Mellanox Technologies. 2013. "Performance Tuning Guidelines for Mellanox Network Adapters." `http://www.mellanox.com/related-docs/prod_software/Performance_Tuning_Guide_for_Mellanox_Network_Adapters.pdf`

[14] National Instruments. 2009. "PCI Express – An Overview of the PCI Express Standard." Published August 13. `http://zone.ni.com/devzone/cda/tut/p/id/3767`

Biography

Qian Liu received the BS and MS degree in Computer Science from Southwest Jiaotong University, China. He is currently a PhD student in Computer Science at the University of New Hampshire.

Robert D. Russell (Ph.D. '72) is an Associate Professor in the Computer Science Department and InterOperability Laboratory at the University of New Hampshire. His interests are in Operating Systems, High Performance Networking, and Storage. He is also the principal developer and instructor for the OpenFabrics Alliance (OFA) training courses entitled "Writing Application Programs for RDMA Using OFA Software."