Veloblock: Efficient and Scalable RDMA Fast Path for InfiniBand

Matthew J. Koop Ajay P. Sampat Dhabaleswar K. Panda
Network-Based Computing Laboratory
The Ohio State University
Columbus, OH 43210
{koop, sampat, panda}@cse.ohio-state.edu

ABSTRACT

Message Passing Interface (MPI) continues to be the dominant programming model for parallel scientific applications. As a result, the MPI library design is very important for the overall performance of the system. Additionally, InfiniBand has become a very popular interconnect for clusters. Over 25% of the Top500 supercomputers are listed as using InfiniBand. As a result, the MPI library design for InfiniBand clusters is particularly significant. To obtain the lowest latency for message passing over InfiniBand MPI libraries include message passing using Remote Direct Memory Access (RDMA) Write operations for small messages. This is often referred to as "RDMA Fast Path" or "Eager RDMA." This feature, however, can consume 512 KB of memory for each uni-directional communication channel. This requirement means that only a few of these connections can be created.

In this paper we propose Veloblock, a new RDMA Fast Path design, that uses a unique bottom-fill block-based design. Instead of using fixed-size buffers we propose methods of allowing variable-sized buffers. We implement our design and show that it uses an order of magnitude less memory and performs equal or better to the existing implementation. In particular, on a 128-core cluster we show that our design is able to outperform a non-RDMA fast path enabled design by 13% for the AMG2006 multigrid physics solver. Veloblock also outperforms a traditional RDMA fast path design by 3% on this benchmark, while using 16 times less memory for these channels.

Categories and Subject Descriptors

D.4.4 [**Operating Systems**]: Communications Management; D.4.9 [**Operating Systems**]: Systems Programs and Utilities; J.0 [**Computer Applications**]: General

Keywords

MPI, InfiniBand, Memory Scalability, RDMA Write

1. INTRODUCTION

Large-scale deployments of clusters designed from largely commodity-based components continue to be a major component of high-performance computing environments. A significant component of a high-performance cluster is the compute node interconnect. InfiniBand [7], is an interconnect of such systems that is enjoying wide success due to low latency (1.0-3.0μsec), high bandwidth and other features.

The Message Passing Interface (MPI) [14] is the dominant programming model for parallel scientific applications. As such, the MPI library design is crucial in supporting high-performance and scalable communication for applications on these large-scale clusters.

To obtain the lowest latency, MPI library implementations over InfiniBand generally include support for small message transfer using Remote Data Memory Access (RDMA) Write operations [13]. This transfer mode is referred to as "RDMA Fast Path" or "Eager RDMA," depending on the developer, however, they all follow the same implementation and design. This basic design is used in other libraries other than MPI including its usage in some GASNet [3] implementations.

Although this "Fast Path" design has been shown to improve latency, it requires a large amount of memory to be used within the implementing library. This is because the design uses fixed-sized persistent buffers. This means that the sender and the receiver must both have dedicated memory for each other. This typically means 256 KB of memory is required for both the sender and receiver. For bi-directional communication using RDMA fast path an additional 256 KB is required for each side for a total of 1 MB per bi-directional connection. For a large number of channels, the memory usage can be significant.

In this paper we propose a new design, *Veloblock*[1], for message transfer using RDMA Write operations. This novel design eliminates the need for persistent fixed-size buffers. Messages only take up as much memory as they require and the sender side no longer needs to have a set of dedicated buffers. Instead of small 32 byte message taking up a full 8 KB buffer, it can now only consume 32 bytes. This can significantly reduce the memory by a factor of 16 times from 512 KB per pair to 32 KB. We show that our design is able to outperform a non-RDMA fast path enabled design by 13% for the AMG2006 multigrid physics solver. We also outperform a traditional RDMA fast path design by 3%, while using 16 times

[1] 'Velo-' is the Latin root for 'fast', and 'block' refers to the view of memory in the design.

less memory.

The remaining parts of the paper are organized as follows: In Section 2 we provide an overview of InfiniBand. In Section 3 we discuss the protocols generally used for message transfer over InfiniBand. Section 4 presents the existing RDMA fast path design. New design options for RDMA fast path and our proposed *Veloblock* design is presented in Section 5. Evaluation and analysis of an implementation of our design is covered in Section 6. Section 7 discusses work related to our own. Finally, conclusions and future work are presented in Section 8.

2. INFINIBAND

InfiniBand is a processor and I/O interconnect based on open standards [7]. It was conceived as a high-speed, general-purpose I/O interconnect, and in recent years it has become a popular interconnect for high-performance computing to connect commodity machines in large clusters.

2.1 Communication Model

Communication in InfiniBand is accomplished using a Queue based model. Sending and receiving end-points have to establish a Queue Pair (QP) which consists of Send Queue (SQ) and Receive Queue (RQ). Send and receive work requests (WR) are then placed onto these queues for processing by InfiniBand network stack. Completion of these operations is indicated by InfiniBand lower layers by placing completed requests in the Completion Queue (CQ). To receive a message on a QP, a receive buffer must be posted to that QP. Buffers are consumed in a FIFO ordering.

2.2 Channel and Memory Semantics

There are two types of communication semantics in InfiniBand: channel and memory semantics. Channel semantics are send and receive operations that are common in traditional interfaces, such as sockets, where both sides must be aware of communication. Memory semantics are one-sided operations where one host can access memory from a remote node without a posted receive; such operations are referred to as Remote Direct Memory Access (RDMA). Remote write and read are both supported in InfiniBand. Note that completion entries are not added to the CQ on the target for RDMA Write operations unless a special 'RDMA Write with Immediate' operation is needed.

Both communication semantics require communication memory to be registered with InfiniBand hardware and pinned in memory. The registration operation involves informing the network-interface of the virtual to physical address translation of the communication memory. The pinning operation requires the operating system to mark the pages corresponding to the communication memory as non-swappable. Thus, communication memory stays locked in physical memory, and the network-interface can access it as desired.

2.3 Transports

There are four transport modes defined by the InfiniBand specification, and one additional transport that is available in the new HCAs from Mellanox: Reliable Connection (RC), Reliable Datagram (RD), Unreliable Connection (UC), Unreliable Datagram (UD), and eXtended Reliable Connection. Of these, RC and UD are required to be supported by Host Channel Adapters (HCAs) in the InfiniBand specification. RD is not required and is not available with current hardware. All transports provide a checksum verification.

Our work is applicable to any transport that provides RDMA Write support, which are RC, XRC, RD, and UC. Support for UC would also require reliability support to be built into the upper-level protocol as well. In this paper we consider RC, however, support for other transports are minor modifications.

Note that the connection memory we consider in this paper is not that of communication contexts from creating QPs. The memory requirements we are reducing in this work is related to buffers.

3. MESSAGE PASSING PROTOCOLS

In this section we describe the two communication modes that MPI is generally implemented with.

- *Eager Protocol:* In the eager protocol, the sender task sends the entire message to the receiver without any knowledge of the receiver state. In order to achieve this, the receiver must provide sufficient buffers to handle incoming unexpected messages. This protocol has minimal startup overhead and is used to implement low latency message passing for smaller messages.
- *Rendezvous Protocol:* The rendezvous protocol negotiates buffer availability at the receiver side before the message is sent. This protocol is used for transferring large messages, when the sender wishes to verify the receiver has the buffer space to hold the entire message. Using this protocol also easily facilitates zero-copy transfers when the underlying network transport allows for RDMA operations.

In this paper we are referring only to the *Eager Protocol* portion. When mentioning RDMA in Fast Path operations, we refer to RDMA Write operations for small message transfer due to their lower overhead. The next section will describe the existing design.

4. RDMA FAST PATH AND EXISTING DESIGN

In this section we describe the existing designs for RDMA Fast Path. The first RDMA Fast Path design was described in 2003 [13] and has remained mostly unchanged since then. We first give a brief overview of what "RDMA Fast Path" means and how current designs have been implemented.

4.1 What is RDMA Fast Path?

In general RDMA fast path refers to a message passing mode where small messages are transferred using RDMA Write operations. Instead of waiting for completion queue (CQ) message, the receiver continually polls a memory location waiting for a byte change.

Waiting for a byte change leads to lower latency than waiting for a completion queue entry. Using any sort of notification negates the performance benefit. When this mode was originally proposed the difference in latency on the first-generation InfiniBand cards between RDMA fast path and the normal channel semantics of InfiniBand was 6μsec to 7.5μsec [13]. The difference on our fourth-generation ConnectX InfiniBand is much lower, but there is still a processing overhead for the card to signal receive completion.

This mode can be achieved since some adapters, such as all Mellanox InfiniBand HCAs, guarantee that messages will be written in order to the destination buffer. The last byte will always be written last.

4.2 Existing Structure

The structure of existing RDMA fast path designs is to have fixed-size chunks within a large buffer. Figure 1 shows this structure.

On the sending side, the sender selects the next available send buffer and copies the message into the buffer and performs an RDMA Write operation to the corresponding buffer on the receiver. The receiver polls on the next buffer where it is expecting a message. Upon detecting the byte change it can process the message. It can send either an explicit message to the sender to notify it that the buffer is available again or piggyback that information within message headers.

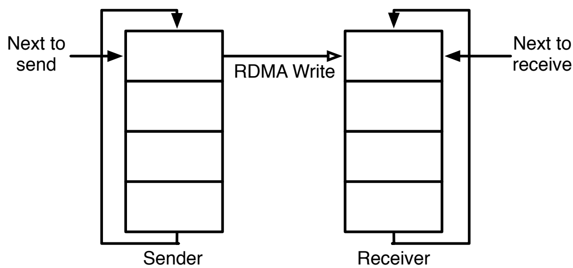


Figure 1: Basic structure of paired buffers on sender and receiver in the RDMA fast path design

4.3 Detecting Message Arrival

To detect the byte change the receiver must set the buffer into a known state prior to any data being allowed to be placed into it. Recall that to use this mode the hardware must write data in order, so the last byte must be changed to know that the entire message has been received. This traditional mode uses a head and tail flag mode. As seen in Figure 2, the head flag is first detected. If the head flag has been changed, then see if the tail flag at $base_address + size$ is equal to the head value. If it is equal, the data has arrived. To ensure that the tail flag differs from the previous value at that address the sender *must keep a copy of the data* that it previously sent. Thus the sending side must have the same amount of buffer reserved as the receiver for this channel. Since the message is filled from the beginning or "top" of the buffer we refer to this design as "top-fill."

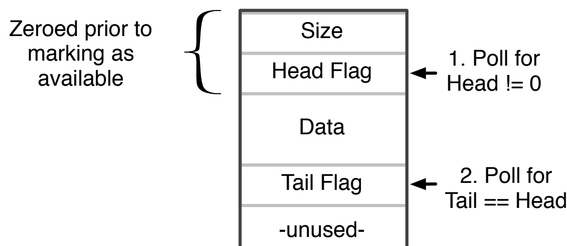


Figure 2: Message Detection Method for fixed-length segments

5. VELOBLOCK DESIGN

In this section we describe the design issues for our new RDMA fast path design, *Veloblock*. First we describe the goals of the design, followed by our solutions to achieve these goals, the various options available, and then finally the design that we select.

The broad goal for a new RDMA Fast Path design is to retain the benefits of lower latency and overhead by using RDMA Write operations for small messages, but reduce the amount of memory required. Memory usage for the RDMA fast path for each process comes from:

$$N_{buffers} \times B_{size} \times (P_{send} + P_{recv})$$

Where P_{send} and P_{recv} are the number of send and receive fast path peers, $N_{buffers}$ is the number of buffers and B_{size} is the size of each buffer. While simply reducing P, N or B can save memory, it can also reduce the performance.

Our approach to reduce the memory usage is two-fold:

- Remove the sender-side buffer. This will reduce the amount of memory required by half.
- Use memory as a "block" instead of as fixed 8 KB segments.

5.1 Remove Sender-Side Buffer

The sender-side buffer is a requirement due to the tail flag in the existing design. The head/tail flag must be selected to be something other than the current value at the tail buffer location. If the value at that position the buffer was already set to the head value it would incorrectly think the message had arrived resulting in invalid data.

To remove the sender-side buffer we propose using a "bottom-fill" approach. Using such an approach there is only a need for a tail flag instead of both head and tail flags. Addionally, there is no need to know the previous value of the tail byte – this byte can just be zeroed out at the receiver before a message arrives. This design can be seen in Figure 4(b). Note, an approach of doing a `memset` of zeros on the entire receiver buffer could also remove the need for a sender-side buffer in the top-fill design, but this also incurs a prohibitively large overhead.

5.2 Supporting Variable-Length Segments

To the best of our knowledge, all current designs of RDMA fast path use fixed-size buffers. In general these messages blocks are 8 KB or larger. However, as mentioned earlier, using fixed width buffers can be very inefficient. Clearly a variable width buffer can increase memory efficiency since a 32 byte message now only consumes 32 bytes rather than an entire 8 KB chunk.

5.2.1 Detecting Arrival

Note that a decision here has an impact on how a message can be detected. If a message no longer arrives at a pre-established location the next arrival bit will still need to be changed to zero via some method.

In existing designs the arrival bit can be zeroed out since the next arrival location is always known. Without zeroing out an entire buffer, it is not possible to always zero out the next byte on the receiver side without an additional operation.

To achieve the zeroing of the next arrival byte we propose sending an extra zero byte at either the beginning or the end of the message. Figure 3 shows how an extra byte can be sent in the variable bottom-fill design. In a top-fill design the extra zero byte is sent at

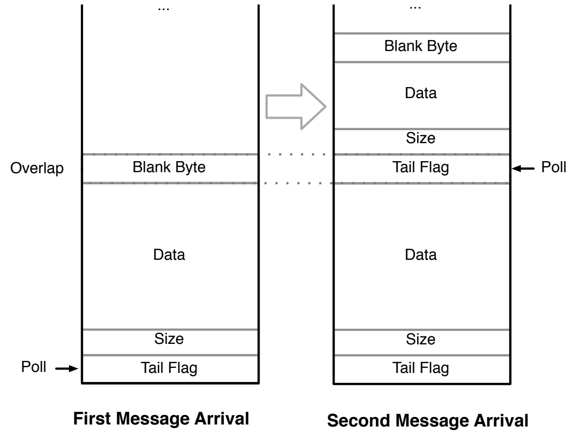


Figure 3: Message Detection Method for variable-length segments

the end of the message instead of the beginning. By sending this extra byte of data we are able to reset the bit where the next section of data is to arrive.

5.2.2 Flow Control

When using fixed-width segments flow control is generally 'credit-based,' where each buffer is a credit. So after the receiver consumes the message and the receive buffer is free the receiver can signal the sender that the buffer is available again. When using the remote memory as a block (variable length) instead fixed-segments the credit is now based on bytes. When a receiver processes a message it can tell the sender the number of additional bytes in the buffer that are now available. This can be done as a piggyback in message headers, or an explicit message. This type of notification is similar to credit control in existing designs.

5.3 Possible Designs

Using these parameters there are four possible design options that can be created with these parameters. Each of these options is shown in Figure 4. Table 1 shows the features of each design.

- *Fixed Top-fill*: In this design fixed buffers are used and filled from the top. This design requires buffers on the sender and receiver to be dedicated to each other. This is the most memory inefficient design. This is the design employed by MVAPICH [15], Open MPI [22] and others.
- *Fixed Bottom-fill*: This method uses fixed buffers, however, unlike the top-fill design it does not require dedicated buffers on the send side.
- *Variable Top-fill*: In this mode only the required amount of buffer space is used, however, it does require a dedicated sender-side buffer.
- *Variable Bottom-fill / Veloblock*: This mode is the most memory efficient. Messages only take as much memory as required and does not require a dedicated sender-side buffer.

5.4 Veloblock Design

Given these options, the highest memory efficiency will come from using the *Variable Bottom-fill* method. This is the method that we propose using and give the name *Veloblock*. Using this method messages only take as much room as they need rather than an entire block.

With this design the memory requirements can be described as the following:

$$B_{nsize} \times P_{recv}$$

Note that P_{send} is eliminated from this equation. Here only the block size and number of peers that a process is receiving from are involved. B_{nsize} here is larger than that of the original case (B_{size}), but since small messages only take up as much space as required it can be significantly less than $N_{buffers} \times B_{size}$ of the original case.

The basic MVAPICH implementation allocates 32 buffers, each of size 8 KB for each connection. This means that a receiver must allocate 32×8 KB $= 256$ KB of memory and since it is a top-fill design 256 KB on the sender side as well for a total of 512 KB. With the variable bottom-fill Veloblock design we can instead allocate one larger buffer and have messages flow in as needed. In the next section we will run Veloblock with only a 32 KB buffer and observe the performance.

6. EXPERIMENTAL EVALUATION

In this section we evaluate the design we proposed in the previous section. We first start with a description of the experimental platform and methodology. Then we evaluate the memory usage and performance on microbenchmarks and application benchmarks.

6.1 Experimental Platform

Our experimental test bed is a 128-core ConnectX InfiniBand Linux cluster. Each of the 8 compute nodes a quad socket, quad core AMD "Barcelona" processors for a total of 16 cores per node. Each node has a Mellanox ConnectX DDR HCA. InfiniBand software support is provided through the OpenFabrics/Gen2 stack [16], OFED 1.3 release. The proposed designs are integrated into the MVAPICH-Hybrid [8] communication device of MVAPICH [15]. MVAPICH is a popular open-source MPI implementation over InfiniBand based on MPICH [6] and MVICH [12].

All of the designs are implemented into the same code base and the same code flows. As a result, performance differences can be attributed to our design instead of software differences. All experiments are run using the Reliable Connection (RC) transport of InfiniBand.

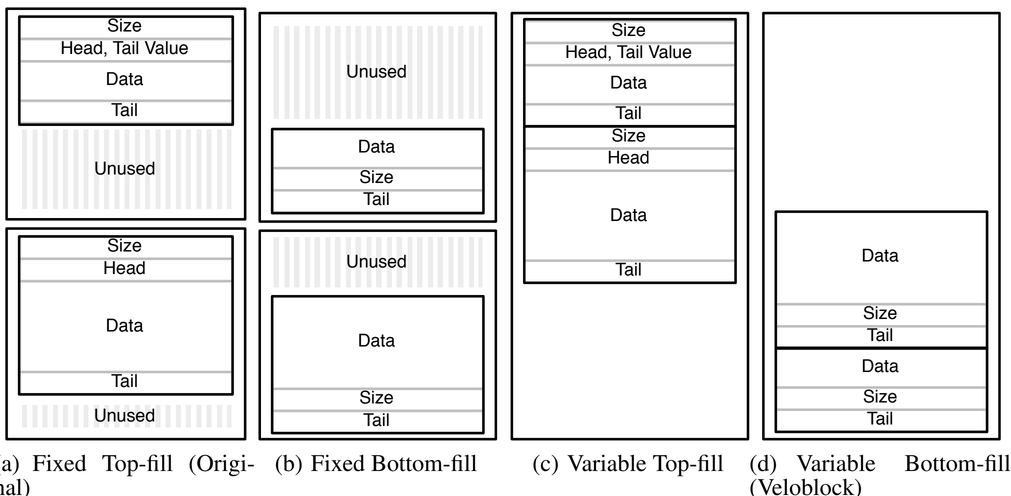
6.2 Methodology

We evaluate three different combinations:

- *Original*: The existing design described in Section 5 as Fixed Top-fill. This design consumes 512 KB of memory total per uni-directional channel (256KB on the receiver and 256KB on the sender).
- *Original-Reduced*: This is the same design as *Original*, however, we restrict the amount of memory to 32 KB to match that of our new design. This will show if our proposed design is necessary or if buffers could simply be reduced in the basic design.
- *Veloblock*: This is our new design proposed in Section 5 that uses memory as a block instead of chunks of pre-determined size. This uses 32 KB of memory total per channel.

Table 1: Comparison of RDMA Fast Path Designs

	Characteristics		Memory Usage	
	Top/Bottom Fill	Variable/Fixed	Sender Buffer	Efficiency
Fixed Top-Fill	Top Fill	Fixed	Required	Low
Fixed Bottom-Fill	Bottom Fill	Fixed	Not Required	Low
Variable Top-Fill	Top Fill	Variable	Required	High
Variable Bottom-Fill	Bottom Fill	Variable	Not Required	High



(a) Fixed Top-fill (Original) (b) Fixed Bottom-fill (c) Variable Top-fill (d) Variable Bottom-fill (Veloblock)

Figure 4: Fast Path Designs: Each figure shows one of the options available for designing a RDMA fast path. Note that all "top-fill" designs also need a mirrored sender-side buffer.

We also want to observe the effect of the number of fast path channels on application performance. In particular, we want to see if additional channels can increase performance. Due to the memory usage many implementations limit the number of fast path channels allowed – Open MPI for example allows only 8 by default. We will also track the amount of memory required for fast path communication.

6.3 Application Benchmarks

In this section we evaluate each of our configurations using two application benchmarks: AMG2006 and LAMMPS.

6.3.1 AMG2006

AMG2006 is a parallel algebraic multigrid solver for unstructured meshes. It is part of the ASC Benchmark Suite [1]. The benchmark and results we show are given using the default driver for the benchmark. It is very communication intensive and communication time can be a significant portion of the overall performance [1].

Figure 5 shows the performance of AMG2006 with varied refinement levels. Lower refinement values result in higher degrees of communication. We can see from the figure that using RDMA fast path channels clearly increases performance. For a refinement value of 3 an increase in performance of 13% for 128 *Veloblock* channels over the configuration where no RDMA fast path channels are created is observed. We also note that the *Original-Reduced* mode with 128 channels performs only 3% higher than the base case with no channels created. Additionally, the *Veloblock* design outperforms the *Original* design by 3%, while using significantly less memory.

Table 2 shows the characteristics of each of the configurations. For the refinement value of 3, the number of remote peers is on average 103.92. This average is fractional since some processes have more peers than others. When allowing up to 128 fast path connections to be created, the *Original* design will consume 52 MB of memory per process. By contrast, our new design will use only 3.25 MB/process. Note that fast path connections are created "on demand," so only if a process is communicating with a remote process will a fast path buffer be allocated.

In Table 2 we also present the percentage of remote messages that are able to use the RDMA fast path channel. If there is not an available remote buffer the sender cannot use RDMA fast path. Thus, for the *Original-Reduced* configuration which has less buffers we note that only 50% of remote messages can use the RDMA fast path, where as for *Original* and *Veloblock* nearly 90% take the fast path. Note that some messages are larger than 8KB and cannot take the fast path, so 100% is often not possible.

6.3.2 NAMD

NAMD is a fully-featured, production molecular dynamics program for high performance simulation of large bimolecular systems [17]. NAMD is based on Charm++ parallel objects, which is a machine independent parallel programming system. It is known to scale to thousands of processors on high-end parallel systems. Of the standard data sets available for use with NAMD, we use the `apoa1`, `er-gre`, and `jac2000` datasets. We evaluate all data

Table 2: Communication Characteristics

Application	Dataset	Remote Peers	Configuration	Fast Path Memory (per process)	Remote Messages over Fast Path (%)
AMG2006	Refinement 3	103.92	Original	51.95 MB	89.75
			Original-Reduced	3.25 MB	49.54
			Veloblock	3.25 MB	87.22
	Refinement 4	107.16	Original	53.58 MB	90.25
			Original-Reduced	3.35 MB	43.49
			Veloblock	3.35 MB	89.23
	Refinement 5	109.12	Original	54.56 MB	89.99
			Original-Reduced	3.41 MB	49.38
			Veloblock	3.41 MB	88.36
NAMD	jac	59.13	Original	29.57 MB	83.51
			Original-Reduced	1.85 MB	52.79
			Veloblock	1.85 MB	83.53
	ergre	18.88	Original	9.44 MB	64.69
			Original-Reduced	0.59 MB	56.71
			Veloblock	0.59 MB	64.75
	apoa1	90.16	Original	45.08 MB	72.51
			Original-Reduced	2.81 MB	58.26
			Veloblock	2.81 MB	72.52

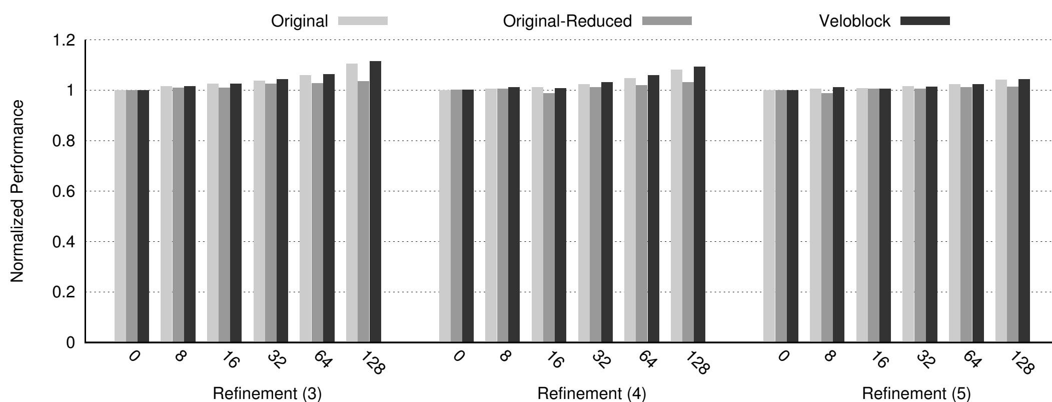


Figure 5: AMG2006 Performance (higher bars are better). The "0, 8, . . . , 128" values refer to the number of RDMA fast path connections allowed to be created per process.

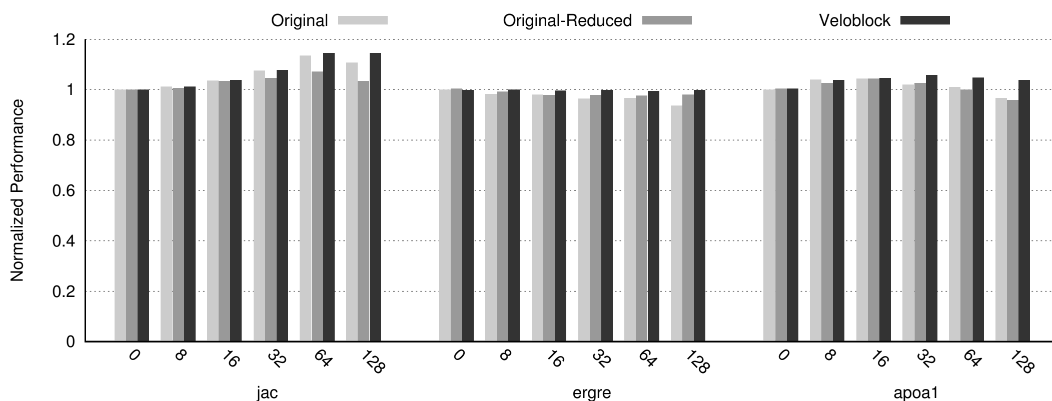


Figure 6: NAMD Performance (higher bars are better)

sets with 128 tasks.

Figure 6 shows the performance of each of these three datasets with increasing numbers of allowed fast path connections. The first of the benchmarks, `jac`, is the Joint-Amber Charmm Benchmark and is very communication intensive. For this benchmark the *Veloblock* design again performs the best with a 14% improvement over the case with no RDMA fast path. The original configuration is a 12% improvement, but requires a significant amount of memory. The *Original-Reduced* configuration, which uses the same amount of memory as our new design, shows only a 4% improvement.

From Table 2 we can see that for the `jac` benchmark nearly 84% of messages are able to use the fast path for both *Original* and *Veloblock* and only 53% for *Original-Reduced*. This explains the performance gap in the results. The new *Veloblock* design additionally consumes 42 MB of memory less per process than the *Original* mode.

For the `apoa1` benchmark the *Veloblock* continues to show benefits with up to a 6% improvement over the base case with no RDMA fast path connections. We do notice that after 32 connections performance is gets slightly worse. We attribute this to the additional time to poll for message arrival when additional connections are added. If not enough messages are transferred over this path then the polling overhead can exceed the benefit. For example, with `apoa1`, increasing from 32 fast path connections to 64 the number of messages taking the fast path only increases from 36.40% to 47.78%.

The `ergre` benchmark shows a similar trend where the polling for the *Original* designs seems to lower performance. To avoid these types of problems RDMA fast path implementations can tear down connections that do not meet a pre-configured message rate.

7. RELATED WORK

Many other works have proposed methods of increasing the efficiency of communication buffers. The work most similiar to ours is Portals [5], in which buffers are allocated from a block as they arrive, so space is the limiting factor rather than the number of messages. This work, however, was not done with the InfiniBand and had additional NIC features to allow this flow. This style of support was also suggested for VIA, a predecessor of InfiniBand, but was never adopted into hardware [4].

Others have also tried to increase communication buffer efficiency for MPI over InfiniBand. The Shared Receive Queue (SRQ) feature of InfiniBand allows a single pool of buffers for multiple peers instead of separate pools per peer [21, 18]. Noticing that the efficiency of the buffers as was low, as these buffers are fixed size, Shipman proposed using multiple SRQs, each with a different size [19]. Each of these SRQ-based enhancements, however, are for the channel semantics of InfiniBand and loses the performance features of the RDMA fast path.

Additional research has investigated to lower connection memory in InfiniBand. This has been done with respect to the new eXtended Reliable Connection transport of InfiniBand [9, 20] and hybrid designs to use multiple InfiniBand transports to reduce connection memory [10, 11, 8]. This research is complementary to ours and involves reducing connection context memory. Our research, by contrast, focuses on the memory used for message buffers.

Balaji also explored the idea of a block-based flow control for InfiniBand for the Sockets Direct Protocol (SDP) [2]. In this work RDMA Write with Immediate operations were used, which eliminates the benefit of lower latency. It also does not have to address the issues to clearing out the arrival bytes since it uses the Completion Queue (CQ) entry.

8. CONCLUSION AND FUTURE WORK

Both MPI and InfiniBand are important components in many high-performance computing environments. Existing MPI implementations for InfiniBand are able to provide the best performance when they use a "RDMA Fast Path" mode, but this mode can consume significant amount of memory when used. 512 KB of memory per communicating peer can be used when this mode is used. Due to this reason MPIs generally have to severely restrict the number of these connections allowed to be setup.

In this paper we have designed *Veloblock*, a new and more efficient and scalable RDMA Fast Path design. Memory usage is an order of magnitude less than current designs used in other MPI libraries over InfiniBand. We performed a performance evaluation using various application benchmarks and showed performance improvement using RDMA Fast Path techniques. In particular, on a 128-core cluster we show we show that our design is able to outperform a non-RDMA fast path enabled design by 13% for the AMG2006 multigrid physics solver. With this result we show RDMA Fast Path does increase performance. Further, Veloblock also outperform a traditional RDMA fast path design by 3% on this benchmark, all while using 16 times less memory for these channels.

In the future we plan to evaluate our designs on larger-scale. We also plan to evaluate the possibility of variable-sized chunks. Currently Veloblock allocates a 32 KB block for each channel, but some applications may benefit from a larger chunk. In particular, channels between some pairs of processes may be allocated less (e.g. 16 KB), while others be granted more (e.g. 48 KB) if they are exchanging more messages.

Acknowledgment

This research is supported in part by U.S. Department of Energy grants #DE-FC02-06ER25749 and #DE-FC02-06ER25755; National Science Foundation grants #CNS-0403342, #CCF-0702675, and #CCF-0833169; grant from Wright Center for Innovation #WCI04-010-OSU-0. ; grants from Intel, Mellanox, Cisco, and Sun Microsystems; Equipment donations from Intel, Mellanox, AMD, Advanced Clustering, IBM, Appro, QLogic, and Sun Microsystems.

9. REFERENCES

[1] ASC. ASC Sequoia Benchmarks. https://asc.llnl.gov/sequoia/benchmarks/.

[2] P. Balaji, S. Bhagvat, D. K. Panda, R. Thakur, and W. Gropp. Advanced Flow-control Mechanisms for the Sockets Direct Protocol over InfiniBand. In *ICPP '07: Proceedings of the 2007 International Conference on Parallel Processing*, page 73, 2007.

[3] D. Bonachea. Gasnet specification, v1.1. Technical report, Berkeley, CA, USA, 2002.

[4] R. Brightwell and A. B. Maccabe. Scalability Limitations of VIA-Based Technologies in Supporting MPI. In *Fourth MPI Developer's and User's Conference*, 2000.

[5] R. Brightwell, A. B. MacCabe, and R. Riesen. Design and Implementation of MPI on Portals 3.0. In *Proceedings of the*

9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, pages 331–340. Springer-Verlag, 2002.

[6] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI, Message Passing Interface Standard. Technical report, Argonne National Laboratory and Mississippi State University.

[7] InfiniBand Trade Association. InfiniBand Architecture Specification. http://www.infinibandta.com.

[8] M. Koop, T. Jones, and D. K. Panda. MVAPICH-Aptus: Scalable High-Performance Multi-Transport MPI over InfiniBand. In *IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS 2008)*, April 2008.

[9] M. Koop, J. Sridhar, and D. K. Panda. Scalable MPI Design over InfiniBand using eXtended Reliable Connection. In *IEEE Int'l Conference on Cluster Computing (Cluster 2008)*, September 2008.

[10] M. Koop, S. Sur, Q. Gao, and D. K. Panda. High Performance MPI Design using Unreliable Datagram for Ultra-Scale InfiniBand Clusters. In *21st ACM International Conference on Supercomputing (ICS07)*, Seattle, WA, June 2007.

[11] M. Koop, S. Sur, and D. K. Panda. Zero-Copy Protocol for MPI using InfiniBand Unreliable Datagram. In *IEEE Int'l Conference on Cluster Computing (Cluster 2007)*, September 2007.

[12] Lawrence Berkeley National Laboratory. MVICH: MPI for Virtual Interface Architecture. http://www.nersc.gov/research/FTG/mvich/ index.html, August 2001.

[13] J. Liu, J. Wu, and D. K. Panda. High Performance RDMA-based MPI implementation over InfiniBand. *Int. J. Parallel Program.*, 32(3):167–198, 2004.

[14] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, Mar 1994.

[15] Network-Based Computing Laboratory. MVAPICH: MPI over InfiniBand and iWARP. http://mvapich.cse.ohio-state.edu.

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

[17] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kale. NAMD: Biomolecular Simulation on Thousands of Processors. In *Supercomputing*, 2002.

[18] G. Shipman, T. Woodall, R. Graham, and A. Maccabe. Infiniband Scalability in Open MPI. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[19] G. M. Shipman, R. Brightwell, B. Barrett, J. M. Squyres, and G. Bloch. Investigations on infiniband: Efficient network buffer utilization at scale. In *Proceedings, Euro PVM/MPI*, Paris, France, October 2007.

[20] G. M. Shipman, S. Poole, P. Shamis, and I. Rabinovitz. X-SRQ - Improving Scalability and Performance of Multi-core InfiniBand Clusters. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 33–42, 2008.

[21] S. Sur, L. Chai, H.-W. Jin, and D. K. Panda. Shared Receive Queue Based Scalable MPI Design for InfiniBand Clusters. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[22] The Open MPI Team. Open MPI. http://www.open-mpi.org/.