Implementation and Design Analysis of a Network Messaging Module Using Virtual Interface Architecture

Gregory Amerson and Amy Apon
University of Arkansas
{gamerso,aapon}@uark.edu

Abstract
The Buffered Message Interface (BMI) of PVFSv2 is a low level network abstraction that allows PVFSv2 to operate on any protocol that has BMI support. This work presents a new BMI module that supports the VIA over an early release version of InfiniBand and also over Myrinet. The baseline bandwidth and latency of the implementation were compared to the BMI modules and were shown to achieve significantly higher performance than the TCP module, but slightly less than the GM module. Experimental results comparing a completion queue version with a notify version and using immediate versus rendezvous messages will be useful to system implementors of network messaging modules.

1. Introduction
A key component to the performance of clusters is the interconnection network and the software protocols that use this network. Protocols strive to utilize the underlying hardware more fully and to achieve “OS bypass” by not involving the host operating system in sending or receiving of messages. One example is the Virtual Interface Architecture (VIA) [4, 16], which is a software specification that offers user-level networking with direct hardware access. Many networks also provide native low-latency protocols, such as Myrinet’s GM protocol [1]. However, there are currently many choices for network software and protocols for applications to support. A network transfer layer that provides support for multiple protocols hides the details of the underlying protocols and allows portability across high performance networking software and hardware.

PVFS is a high performance parallel file system for Linux clusters [11]. PVFSv1 was designed to be a research platform for parallel I/O [13]. The PVFS designers have developed a network transfer layer called Buffered Message Interface (BMI) that is used with PVFS version 2. Multiple network protocols are supported in BMI through the use of BMI modules [2].

The two high-performance networks available for this research are Myrinet [8] and InfiniBand [3]. At the time of the study many of the standards for user-level, high performance networking for InfiniBand such as Sockets Direct Protocol (SDP) and User Direct Access Provider Library (uDAPL) were still emerging. The programming interface that was available with the InfiniBand hardware and offered the best performance was the Virtual Interface Provider Library (VIPL) provided by Intel for InfiniBand. The BMI implementation available at the time of this study had two network modules, one for TCP and GM.

The goal of this research is to design and develop a new VIA module for BMI that allows BMI to run over both InfiniBand and Myrinet. The research analyzes the software design for the BMI-VIA module including design tradeoffs of the VI Architecture such as completion queue versus the notify mechanism, the software design for a credit based flow control, and evaluates the performance of the BMI-VIA module using baseline benchmarks against other BMI modules and with varying software design parameters.

The remainder of this paper describes BMI module design issues, describes the experimental evaluation of the differences between design decisions of the BMI module as well as differences between VIA implementations, and presents conclusions based on the results.

2. BMI-VIA Module Design
The Buffered Message Interface (BMI) of PVFSv2 is a low level network abstraction that allows PVFSv2 to operate on any protocol that has BMI support [10]. Only the message semantics and strict adherence to the interface are required for the BMI module, and implementation details are left up to the module implementor. For every BMI module there are some common tasks associated with its operation regardless of what type of protocol it supports. Sup-
VIA module at the time of development and testing. BMI communication is connectionless and is based on messages. Whether the user is sending or receiving, it must first post the message buffer and then test or wait for its completion. The user may need to test for completion multiple times until the message completes [10]. In general, for each BMI message the remote address for the message destination must be supplied. However, the BMI API allows for a BMI program to receive from an unspecified host by periodically checking for an unexpected message. The BMI API also allows for immediate completion and can signal to the user so the BMI operation can complete immediately. The complete API for a BMI application is described in [2].

A BMI module must be compiled as a static module and the underlying module must support reliable, ordered delivery and provide flow control. The BMI module is not required to be thread-safe because the BMI module control layer will serialize the access to any underlying BMI module. Each module is expected to maintain a private collection of all current operations in some form by using operation queues. The GM and TCP BMI modules both use FIFO operation queues to track ongoing and queued operations. The BMI-VIA module was modeled after the GM module design so it makes use of FIFO operation queues as well. The BMI library provides several functions that support the creation, modification, and deletion of operation queues.

### 2.1. VIA Design Decisions

The messaging requirements of the BMI-VIA module are similar to those of many other messaging systems based on VIA [14, 15, 7, 12]. A discussion of those systems and their design choices is available in [5]. The BMI-VIA messaging model/system has these main tasks to accomplish: 1) guarantee that messages that are sent will have a receive descriptor that can accept them and will not be dropped, 2) avoid copying of large messages on the receiving side to save memory bandwidth, 3) limit the amount of memory allocated for receive queue, and 4) optimize the BMI-VIA module at runtime for a specific message size.

Both the VI-GM and Intel-VI provide reliable message delivery for VIA and the BMI-VIA module uses reliable message delivery to reduce the software complexity. Unlike VIA, there is no requirement of BMI that users must register message buffers. It has been shown that the only negative performance impact of registering memory is for message sizes that are less than 2K [12]. The design of the BMI-VIA module assumes for benchmarking purposes that the BMI application will use registered memory for all message buffers to improve performance. Note that this is not part of the BMI specification but rather a requirement of the BMI-VIA module at the time of development and testing.

BMI is a connectionless message system and there are special unexpected messages that allow for initiation of communication. VIA is a connection-oriented system that has two types of connection semantics: client/server and peer-to-peer. Since BMI has client/server semantics the client/server connection model for VIA is more suited to initiate communication. For a BMI server to receive unexpected messages, the BMI-VIA module must create a listen thread that uses the ConnectWait routine to wait on a VIA client to make a ConnectRequest. This should not cause an overall performance penalty because the unexpected message that initiates communication only happens once. VIA implementations that do not have the client/server connection model such as Berkeley VIA would be difficult to use for the BMI-VIA module.

The BMI-VIA implementation uses three separate VIs per BMI “connection”. The first connection is initiated with the client/server model using the ConnectWait method to create the first Virtual Interface. Two more VIs are created next, but the same ConnectWait method can not be used because any remote VI process could connect. For the remaining two VIs, the peer-to-peer model is used to ensure that the remote host can be specified during the connection request. This approach does creates complex setup and memory requirements of each BMI “connection”. But there are advantages to this design, such as the ability to send control messages on the control VI to allow them to complete during or before messages sent on the immediate VI. It also allows for easier synchronization of rendezvous messages using the control VI in conjunction with the rendezvous VI. Control messages are small and should not take up immediate message receive buffers on the immediate VI connection.

### 2.2. Credit based flow control

The BMI specification allows for any order of message posting, either receiver or sender first. However, VIA requires that a posted receive descriptor must be in place before the arrival of a send message. BMI-VIA preposts a specified number of receive buffers of a given length. The sender can be certain that messages sent of the same length or less will be received. Since these preposted receives will be exhausted, new receive descriptors must be preposted and the sender made aware of these new available receive buffers. This leads to the need for a credit-based flow control system to be used in the BMI-VIA module.

Send credits are used on both the immediate and control VIs. After each VI is created and prior to the connection request, a defined number of receiver buffers or credits are posted. Each time a sender posts a send on either the control or immediate VI the corresponding send credit limit is decremented by one. On the receiving side when a mes-
message is received the data is left in the receive buffer until the BMI user posts the matching BMI \texttt{recv()} and then the data is copied out and the buffer is marked as available to be reposted back to the receive queue.

Credit can be returned to the sender either in an explicit control message posted to the control VI, or piggybacked on other available message already being sent to sender, either immediate, control, or rendezvous. Send credits that are piggybacked instead of explicit messages will not reduce throughput, but too many explicit messages will reduce throughput. Therefore, explicit return credit messages should be avoided until the percentage of unreturned send credits is relatively high. Setting this threshold too low will cause unnecessary control messages to be sent back and forth. Delaying the return of send credits has been shown to increase bandwidth and decrease latency in other credit-based flow control systems [6]. The effect of using a high or low credit is shown in the experiments section.

### 2.3. Immediate and Rendezvous Messages

A majority of the messages sent with BMI in the PVFSv2 environment will likely be a fixed size message that will be related to other system level factors such as a stripe size of a file. The opportunity exists to optimize the message system for that communication.

Immediate messages are those that can be sent immediately to the receiver with knowledge that it will be received. Any message that is less than the maximum immediate size, which is 16KB by default, is considered an immediate message. Immediate messages use the eager protocol because the receiver holds the message in a buffer until the matching receive is posted and then the receiver copies the messages out into the user buffer. This requires that the BMI-VIA module must manage a queue of receive buffers that can be posted to the VI receive queue to handle incoming immediate messages. There will be some limit to the amount of memory that can be kept in this receive buffer queue because it must be registered memory. If the message is less than the immediate message length, only part of the buffer will be used. The choice of the maximum immediate message size should not be set arbitrarily high to avoid misusing the registered memory that is used for send and receive buffers. Although, the experimentation shows that if larger immediate buffers are used, the amount of preposted receive buffers can be reduced to save memory without impacting performance.

Before a VI connection is established, both sides register memory that will be used for the VI communication. The credit limit parameter, which is defined at runtime, is used to control how many receive buffers are created. The receive buffers are allocated and registered in contiguous memory. This will allow one memory handle to be used for the entire receive buffer queue. The same number of receive descriptors are created and registered along with the receive buffers. Note that receive descriptors are the data structures that are posted to the VI receive queue. These descriptors contain data pointers to actual memory buffers to use for the message. A pool of send buffers is also allocated but descriptors are not posted because there is nothing to send initially. If the user decides to pre-allocate send buffers with the BMI library it can be registered and posted on the VI send descriptor queue.

Any message length that is greater than the immediate message size is handled by the Rendezvous protocol. When the rendezvous message is posted, a Request is sent from the sender to the receiver, and the Acknowledgement is not be sent back until the matching buffer is posted by the BMI user on the receiving side. The added latency is offset because the rendezvous message is a zero-copy operation since both send and receive buffers are pinned. The rendezvous protocol also allows use of the RDMA write feature by the receiver sending the remote buffer pointer with the ack.

The rendezvous protocol can use RDMA writes or the normal send/receive mechanism, since the receive buffer is guaranteed to be in memory when the sender receives the ack. In order for the receiving side to receive notification of completion, however, a receive descriptor must be posted. This cannot be posted to the immediate VI because an immediate receive buffer has already been preposted on the receiver end, which cannot be removed. Instead, the second VI connection will be used as the Rendezvous VI. The control messages used for synchronization in the rendezvous protocol will use the last VI connection, or the Control VI.

The sequence of events for a receiver are:

1. BMI user posts receive buffer
2. Post the receive buffer to the rendezvous VI
3. If rend request has arrived send the ack, if not wait for req on control VI and send ack when it arrives
4. One receive completes on rend VI signal to BMI user receive is complete

The sequence of events for a sender are:

1. BMI user posts the send buffer
2. Post a control message to the control VI making a rendezvous request
3. Wait for ack from receiver on control VI
4. Post the send buffer to the rendezvous VI
5. Signal to BMI user send complete

Note that the relative interleaving of these sequences can vary.
2.4. Completed Descriptor Handling

A complete message could be removed from the send or receive queue by checking each queue individually for completed descriptors. However since three VIs are used this would create too much polling on the six total send and receive queues for those VIs. Both a completion queue (CQueue) and a Notify approach were implemented and tested.

The CQueue approach uses a central completion queue for all three VIs used in the BMI-VIA module both send and receive queues associated with each VI. All send and receive descriptors complete asynchronously in a separate thread managed by the VIPL. When these descriptors complete they are moved to the central completion queue. One VIA method is used to check the completion queue for completed descriptors. If there is a send or receive descriptor that has completed, it is removed from the queue and a pointer to the descriptor is returned. If the completion queue is empty, the method returns, or a blocking version can be used to wait on a descriptor to complete.

BMI messages take two method calls to complete, BMI_post_send/recv() and a corresponding BMI_test/wait(). Some messages will complete during the BMI_post_recv(), as when a receive message is already in a receive buffer, so the BMI_test/wait() will complete immediately. If the BMI_post_recv() is called and the message has not arrived, the BMI-VIA module will register the posted message internally. When the BMI user calls BMI_test/wait(), the BMI-VIA module begins checking the completion queue for any completed descriptors. If the BMI_test() is called, the BMI-VIA module only checks the completion queue once. All completed descriptors are processed. If no completed descriptors are found, the BMI_test() returns. However, if the BMI_wait() is called the BMI-VIA module checks the completion queue using a blocking call. If there are no descriptors to process, the completion queue method blocks for a specified timeout. Since, BMI methods can only block for a limited amount of time, a small timeout is given so the BMI user is not stalled for too long. Once the BMI user’s message arrives at the completion queue, the descriptor is removed and the data is copied into the user’s buffer. This allows the BMI_test/wait() call to return completed. Note it is common for a BMI user to repeatedly call the BMI message completion functions.

With the Notify approach, each time a send or receive buffer is posted a registered callback routine is invoked upon completion of the descriptor. Messages are completed and processed internally by the BMI-VIA module with the Notify mechanism. It still requires the BMI user to call test() or wait() before the message is copied into the message buffer. This complicates the BMI-VIA module by introducing critical sections that require mutual exclusive be-

cause the handler routine is running simultaneously with the BMI user thread. There is also additional overhead for the context switches between the interrupt handler and the notify thread. However, it is beneficial in certain benchmarks to have immediate notification of completed messages.

3. Experiments and Results

For each benchmark test, a BMI server and BMI client program were created. The client sends an unexpected message to the server with all of the parameters, calls a lightweight clock function, and both client and server enter the main loop. No dynamic memory or registering is done during this loop, and only raw throughput or latency is measured. In each iteration, either the client sends a message to the server or both send messages to each other. Once the client exits the loop, it calls the lightweight clock function and calculates the total time in the loop. The sender marks the data before sending, and the receiver checks the data for correctness. The loop and calculation are repeated for benchmarks that use varying message sizes.

The test environment uses two nodes. Each node has a single Athlon MP 2000+ processor with 512MB of RAM interconnected by a Fast Ethernet, a Myrinet switch using 64-bit LANai9 cards, and an InfiniBand fabric with host adaptors. The InfiniBand hardware supports the “Profile A” of the IBA specification which has a smaller MTU, only 128 Queue Pairs, and uses the Intel 82808XA HCA. Along with the hardware, there is an early reference Profile A software stock created by Intel that provides VIPL over the profile A hardware. This software was developed prior to completion of the IBA 1.0a specification, so there are some optional features and the software may not be fully compliant. The VI-GM implementation of VIA is used on Myrinet because it allowed co-development of this work with other research projects that were ongoing.

3.1. Baseline comparison

A baseline comparison was made that measured bandwidth for varying message sizes. The same BMI benchmark program was used to test the TCP, GM, and VIA modules. Both the BMI client and server post a send message and a receive which makes it a two-way bandwidth test. The tests were run over the Myrinet switch since GM module only runs on Myrinet. For all baseline tests the VI-GM implementation is used for the BMI-VIA module since it offers the best performance.

Figure 1 shows the throughput for each BMI module for varying message sizes from 256 bytes to 250KB which is the greatest message size allowed by the BMI-GM module. The TCP module, which uses an IP layer on top of the GM driver level, reaches it maximum bandwidth around
the 30KB message size at 118MB/s. Both the GM and VIA module reach a maximum bandwidth of near 250MB/s. The VIA module is implemented over GM so the level of abstraction decreases performance compared to the GM module.

Both VIA and GM modules have a drop in their bandwidth because of a problem that is occurring in many messaging systems. When moving from the eager protocol to rendezvous there is usually an initial performance penalty. The VIA module actually has two of these changes but only the first is noticeable. The first decrease in performance is when the VI-GM implementation switches from gm_send() (eager protocol) to gm_directed_send() (rendezvous) at the size of 4KB. So the BMI-VIA module is still using eager protocol for messages greater than 4KB but VI-GM is treating it as rendezvous. The BMI-VIA module does not actually switch to rendezvous until much higher and is not noticeable in this benchmark. The BMI-GM module switches from eager protocol to rendezvous at the 16KB message size, which is evident in Figure 1.

The BMI-TCP module illustrates the need for user-level networking and zero-copy protocols. The overhead of the protocol and extra buffer copying keeps the maximum bandwidth at 120MB/s. The BMI-TCP module offers good performance on a network where only TCP is supported.

The second baseline test shows latency calculated for round trip messages with sizes varying from 16 bytes to 64KB. First, the BMI client sends an unexpected message with the message size, and then sleeps for one second to allow the BMI receiver to post the receive message. Then the client starts the clock and sends the message. As soon as BML_wait() returns for the receiver it sends the message back to the sender who already has posted the BML_recv(). When the BML_wait() completes the client stops the clock and calculates the time. This process is repeated for each message size.

Figure 2 shows VIA and GM having much lower latency than TCP, demonstrating the need for user-level messaging. For small messages, VIA is slightly slower than GM. There is a small latency penalty to be paid for having very small messages complete asynchronously [9]. There is added overhead for a small message because the descriptor is first posted (ringing the doorbell) and then the descriptor is processed asynchronously in another thread. For a small message, the size of the descriptor in VIA may be larger than the actual data size.

### 3.2. Initial Completion Queue and Notify Results

This section compares initial results of the Completion Queue (Cqueue) and Notify mechanisms and variations of these implementations.

The bandwidth benchmark measures throughput for both the CQueue and Notify versions for VI-GM and the CQueue version for VI-IB. This benchmark is the same as in the baseline benchmark, using two-way messages for all sizes. The Control VI and Immediate VI send credit limits were chosen to provide the best level of performance. One characteristic of the two-way bandwidth benchmark is that no explicit return credit messages are necessary because messages are sent and received at each host for every iteration. New send credits that need to be returned for either VI can be piggybacked onto outgoing send messages for each host.

Figure 3 shows that either the VI-GM implementation or the Myrinet is faster than its InfiniBand counterpart. This is likely because the VI-IB implementation is not as mature as VI-GM and the “Profile A” InfiniBand fabric is not as fast as Myrinet. Although, the VI-IB version does not suffer the reduction in throughput from an internal transition from eager to rendezvous like both VI-GM versions. It could be that the VIPL does not implement either eager or rendezvous protocols. The VI-IB peaks at 130MB/s whereas
both VI-GM versions were able to reach almost 250MB/s. The VI-IB version does not suffer much when it reaches IR-threshold of the BMI-VIA module at 100KB, which must be attributed to some property of the VIPL library. The VI-GM CQueue actually performs better with a slightly smaller IR-threshold of 70KB. There is a clear drop off when VI-GM Notify reaches 100KB. The context switching contention when switching between BMI user and VIPL library threads for the Notify version slows down the effective bandwidth a small amount.

In the baseline latency test, most of the messages use the eager protocol which leads to the lowest latencies since small immediate messages are faster than rendezvous messages. This test uses the latency benchmark to measure the difference between immediate and rendezvous message latencies. Also, differences between the CQueue and Notify mechanisms are tested.

Figure 4 shows that for small messages rendezvous not only yields lower bandwidth but higher latency as well.

This graph shows why rendezvous protocol is not used for small messages. It is noteworthy that the immediate notification offered by the Notify mechanism does not improve message latency. The reason that the lines are similar before the 70 byte message size is that the smallest possible IR-threshold size is the size of an unexpected message. So the rendezvous data points actually use eager protocol for latency messages less than 70 bytes. After the 16KB message size, the two rendezvous versions have lower latency for 32KB and 64KB message sizes.

3.3. Immediate/Rendezvous Threshold

The bandwidth benchmarks showed that the IR-threshold length has a significant impact on the performance of certain message sizes. This benchmark uses the same test program as in the bandwidth test but this time it was run multiple times with a varying IR-threshold. This shows the effect that the transition from eager to rendezvous protocol has over a wide range of IR-threshold values.

Some tests show that eager protocol, especially over InfiniBand, maintains its advantage over rendezvous even with large message sizes. One possible explanation is that the memory bandwidth of the nodes using the VI-IB implementation is not at full capacity even with the extra copies associated with eager messages. It could be that if the nodes were involved in a parallel system each node would have greater contention for its memory bandwidth and the advantages of the rendezvous protocol would reassert themselves.
representation of the transition from eager to rendezvous. Notice that in each of these graphs there is a distinct top and bottom line that is retraced by many test runs. The top line represents the eager protocol offering higher throughput than the rendezvous protocol which is represented by the bottom line on the composite graph. When each individual test makes the transition from eager to rendezvous it moves from the top line to the bottom line. Note that for an individual run, the rendezvous protocol does offer better performance than if the eager protocol was continued past the IR-threshold. In Figure 6 the advantage of the eager protocol is more defined even at the higher message sizes where extra memory copies should give an advantage to the rendezvous protocol. But it shows on both figures that there is a penalty incurred for a few message sizes after the transition. However, it does not take long for the rendezvous protocol to maintain the same performance as shown by the convergence of the eager/rendezvous lines in Figure 5. Note that in Figure 5 both eager and rendezvous protocol lines show a performance penalty as the VI-GM implementation internally switches from eager to rendezvous protocol at the 4KB message size. These figures use logscale on the y-axis to provide clearer graphs.

Figure 7 sets the IR-threshold at 16KB for the VI-GM CQueue, VI-GM Notify, and the VI-IB CQueue implementations. The graphs are not shown in logscale. The CQueue for VI-GM has the best throughput performance. The performance gap that increases at higher messages sizes between VI-GM and VI-IB are attributed to the Profile A hardware and software of the InfiniBand fabric. All three versions see a decline in throughput just after the threshold value. At 4KB message size the VI-IB implementation performs just as well as the VI-GM.

It has been shown that there is always an initial penalty when switching from eager to rendezvous protocol. If an application knows the size of message it most frequently uses, it can set the IR-threshold somewhere near the message size. There is a window after each threshold where a rendezvous message will perform worse than an immediate message; although, this window closes quickly, and rendezvous messages soon perform better and have lower memory bandwidth usage.

An alternative to paying this penalty for messages slightly bigger than the IR-threshold would be to send multiple immediate messages in place of a single rendezvous message. For instance, if the IR-threshold is 4KB, an 8KB message could be sent using two 4KB immediate messages. Also, a 10KB message could be sent using three 4KB messages. Sending multiple eager protocol messages could have disadvantages, however, such as exhausting the send credit limit and causing the sender to stall and wait for return of send credits. So the limit of how many multi-immediate messages that can be sent should be kept low.

The same bandwidth tests in previous sections were used to measure any performance improvement of multi-immediate messages. Only the CQueue version of the BMI-VIA module had the modifications added to use multi-immediate messages. At runtime, a multi-immediate limit was specified so that if a message that was greater than the immediate message size it would sent as a multi-immediate message or rendezvous. Results showed that multi-immediate messages did not affect performance much for very small and very large IR-threshold values, so only the medium threshold sizes are included in the results.

Figure 8 shows the effect of using multi-immediate messages. The performance drop-off is mostly avoided in Figure 8 for messages greater than 4KB but less than 12KB. Once the multi-immediate line reaches the “Without Multi Immediate” rendezvous line it is no longer more efficient to use. The overhead of posting and sending multiple messages quickly becomes too costly to be effective. The technique was also tried with for messages larger than 8KB. Re-
results, not shown due to space reasons, show some improvement but not as much as with the 4KB size.

A test was created to benchmark the latencies associated with these multi-immediate messages with the rendezvous counterparts. Messages sizes were chosen for both 4KB and 8KB IR-thresholds that would be converted into multi-immediate messages. The test was run once with multi-immediate messages activated for 4KB and 8KB and once with them disabled. Results of the latency test show expected results for 4KB immediate messages but not 8KB. The 4KB test shows that the multi-immediate message just greater than 4KB has a slightly lower latency. But a multi-immediate message over 8KB has higher latency than the rendezvous version. It was shown that multi-immediate messages for messages slightly higher than 8KB result in greater throughput. However, the overhead of sending two immediate messages instead of one rendezvous has increased the latency by almost a factor of two.

3.4. Send Credit Limit

In all of the throughput benchmarks before there has been no explicit return of send credits because the test program used two-way messages so all credits would be piggybacked. In this section, the tests examine the effect on throughput by using a high and low credit limit. All of the send credit throughput tests used one-way messages that guaranteed that the sender would continually exhaust send credits and would have to wait for new credits to be returned in order to continue. The send credit tests are primarily concerned with eager protocol messages because rendezvous messages are not affected as much by credit limits since no credit system is used by the rendezvous protocol. However, request and acknowledge control messages used in the rendezvous protocol do have send credit limits on the control VI. Nevertheless, this proved to have no effect on the throughput of the rendezvous messages by varying the send credit limit on the control VI, so the tests concentrate on immediate messages. During all credit limit tests, the IR-threshold was set to 100KB because only immediate messages are affected.

Figure 9 shows the results using VI-GM with a IR-threshold of 100KB. The credit limit has a negligible effect on throughput for message sizes less than 16KB. Any message sizes greater than 16KB gain a gradual performance increase until the IR-threshold is reached. But the BMI application would have to decide if the increased throughput can justify an increase in the memory requirements of a higher credit limit. If the IR-threshold is less than 16KB, then a higher send credit limit does not contribute to a noticeable increase in throughput.

Figure 10 shows the same test as Figure 9 but over the
VI-IB platform. As noted earlier, the InfiniBand implementation always performs better using the eager protocol instead of rendezvous. Therefore, having a higher send credit limit for immediate messages will show a broader increase in throughput over VI-IB. As shown in Figure 10 the higher credit limit improves throughput for all immediate message sizes until the IR-threshold is reached.

3.5. Computation During Rendezvous

The benchmarks used in other throughput measurements all use the BMI\_post\_send/recv() and then immediately call BMI\_test/wait(). Test programs may post multiple messages before waiting on them but no significant computation or procedures are performed between the post and the test for completion. A test program was created that added the possibility of computation in message loop. This computation could be anything that takes time or delays the calling of BMI\_wait(). The test procedure is described below.

<table>
<thead>
<tr>
<th>BMI Receiver</th>
<th>BMI Sender</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin loop …</td>
<td>Begin loop …</td>
</tr>
<tr>
<td>BMI_post_recv()</td>
<td>BMI_post_send()</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>Some Computation</td>
<td>Some Computation</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>BMI_wait()</td>
<td>BMI_wait()</td>
</tr>
<tr>
<td>End loop …</td>
<td>End loop …</td>
</tr>
</tbody>
</table>

This presents a problem for rendezvous protocol if there is computation between the post() and the wait() API when using the CQueue version. When a rendezvous message is posted on the sending side, a request control message is posted and the BMI call returns. As soon as the request message is received by the BMI receiver, the completed descriptor is moved to the completion queue, and the BMI receiver will not be aware of its completion until the BMI\_wait() call, which is where the CQueue descriptor processing is done. During the BMI\_wait() on the receiving side, the acknowledgement message is sent to the sender. Again, it is received by the BMI sender and the descriptor is moved to the completion queue and is noticed once the sender calls BMI\_wait(). At this time the actual data message is sent. Performance is not affected if both BMI applications immediately call the BMI\_wait() after the posting operation. Figure 11 shows a scenario where the CQueue version increases the latency of rendezvous messages.

In all of the benchmarks shown to this point, the Notify mechanism has performed more poorly than the Completion queue mechanism. However, by processing the completion of descriptors immediately through the registered handlers, the Notify mechanism allows the acknowledgement message to be sent as soon as the request is received. Figure 12 shows that all messages and the actual data can be transferred while both BMI sender and receiver are in computation. This allows the Notify mechanism to deliver lower latency and higher throughput than the CQueue version during this type of benchmark.

The normal benchmark for throughput was modified to include a parameter to determine if some artificial computation would be added into the server or client loops. The BMI client calculates the time taken to completed the loop iterations and generates the output for the graph.

Figure 13 shows that the overlapping of program computation and asynchronous handlers in the Notify mechanism allow it to outperform the CQueue mechanism. When both sender and receiver are performing computation, the client’s computation is manufactured to be larger than the server in order to create the worst case scenario – the receivers acknowledgement returning during the sender’s computation. The difference is greater when both sender and receiver perform the computation rather than just the receiver.
3.6. Temp Buffer Design Alternative

The BMI-VIA module’s default operation is to leave received messages in the receiver buffer queue until the BMI user posts the receive, and then the message is then copied out. This frees up the receive buffer which allows it to be reposted. An alternative would be to create a temporary buffer to store the received messages. When the message arrives, if the BMI user has not posted the receive, a temporary buffer is allocated and the message copied out of the receive buffer into the temporary buffer. This frees the receive buffer and allows it to be reposted more quickly which frees up more send credits to eventually return. Then when the BMI user posts the receive, the message is copied out of the temporary buffer and into the user buffer. Then the temporary buffer is deallocated. This was initially avoided in the design of the BMI-VIA module to reduce the number of copies, but is used in this test.

This section shows a benchmark that illustrates if the BMI receiver is under load and is slowing down the sender, using the temporary buffer design alternative can improve performance. When the BMI receiver is under load and cannot process the messages at the rate they are arriving, the receive buffer queue fills up. This quickly exhausts the sender’s credits and makes it pause sending. However, if the alternative approach is used, those messages are copied to a temporary buffer and then the receive buffer is reposted so it slows down the sender as little as possible.

The test includes four different scenarios: with or without the new temporary buffer alternative, and with or without the artificial receiver load. The artificial load was generated by another program running on the BMI receiver side that makes heavy use of memory bandwidth, disk I/O, and the CPU. Figure 14 shows the four tests when the credit limit was high. The “W/O temp buffer no load” plot is similar to the previous throughput graphs and shows the best performance on this graph. Adding the temporary buffer alternative without additional load yields a lower throughput than the first plot. It maintains performance until messages get larger than 16KB. Then the added overhead of memory allocation and copying decreases performance.

When artificial load is added to the receiver, the baseline bandwidth drops drastically to less than half of its previous throughput as shown in Figure 14. However, the temporary buffer alternative is able to achieve close to the same throughput as in the unloaded tests. The same scenario occurs in Figure 15 but to a lesser degree because the total number in the receiver buffer queue is less.

These benchmarks show that if BMI servers knew they were going to be under heavy load, they could activate a runtime option to use of the temporary buffer and increase
4. Conclusions

BMI was designed to give PVFSv2 the needed network abstraction and messaging layer. This work presents a new BMI module that supports VIA. The BMI-VIA module successfully operates on both Myrinet and InfiniBand and implements a credit based flow control mechanism.

The baseline bandwidth and latency of the BMI-VIA module was compared to the other BMI modules. It was shown to achieve significantly higher performance than the TCP module, but slightly less than the GM module. The CQueue version of the BMI-VIA module offered the best overall performance when running over the VI-GM implementation. Test results show that the InfiniBand Verbs used to implement VI-IB are not as mature as the GM protocol for Myrinet used in VI-GM. As with the InfiniBand hardware, the VIPL from Intel was meant as an early reference design and may not be fully optimized.

If each BMI application that uses the BMI-VIA module knows the most prevalent size of messages that will be sent, it should configure the BMI-VIA module at runtime to use the eager protocol for this message size. In general, immediate messages offer better performance than rendezvous messages especially if the VIPL is implemented using another low-level protocol such as GM. If the Immediate/rendezvous threshold is going to a large message size, the send credit limit should be also configured at runtime to a smaller number to decrease memory requirements of BMI-VIA module. It was also shown that a low send credit limit combined with a large IR-threshold has a negligible impact on performance.

The Notify mechanism was only able to out-perform the CQueue version in one benchmark. This was an unexpected result, because having routines handle the incoming message immediately would seem to offer better throughput. However, the context switch time and the overhead of multiple threads decreased throughput.

The BMI-VMI module is not complete or robust enough to work in a full BMI or PVFSv2 environment, but is complete enough to thoroughly test in single client/server environments. The BMI-VIA module design focuses on the actual message transfer system and not the setup and tear down operations, which leaves some room for much optimization. The performance impact of these VI connections is an area of research [17]. Future work is also possible in several other areas, including the handling of VIPL API errors, combining the completion queue used for the immediate VI and the rendezvous VI in a single implementation, improving the efficiency of the rendezvous protocol and multi-immediate messages.

References