Advanced Flow-control Mechanisms for the Sockets Direct Protocol over InfiniBand

P. Balaji, S. Bhagvat, D. K. Panda, R. Thakur, and W. Gropp

Preprint ANL/MCS-P1422-0507

Mathematics and Computer Science Division, Argonne National Laboratory

Advanced Flow-control Mechanisms for the Sockets Direct Protocol over InfiniBand*

P. Balaji† S. Bhagvat‡ D. K. Panda§ R. Thakur† W. Gropp†

†Mathematics And Computer Science Division,
Argonne National Laboratory
{balaji, thakur, gropp}@mcs.anl.gov

‡Scalable Systems Group,
Dell Inc.
sitha_bhagvat@dell.com

§Computer Science and Engineering,
Ohio State University
panda@cse.ohio-state.edu

Abstract

The Sockets Direct Protocol (SDP) is an industry standard to allow existing TCP/IP sockets based applications to be executed on high-speed networks such as InfiniBand (IB). Like many other high-speed networks, IB requires the receiver process to inform the network interface card (NIC), before the data arrives, about buffers in which incoming data has to be placed. To ensure that the receiver process is ready to receive data, the sender process typically performs flow-control on the data transmission. Existing designs of SDP flow-control are naive and do not take advantage of several interesting features provided by IB. Specifically, features such as RDMA are only used for performing zero-copy communication, although RDMA has more capabilities such as sender-side buffer management (where a sender process can manage SDP resources for the sender as well as the receiver). Similarly, IB also provides hardware flow-control capabilities that have not been studied in previous literature. In this paper, we utilize these capabilities to improve the SDP flow-control over IB using two designs: *RDMA-based flow-control* and *NIC-assisted RDMA-based flow-control*. We evaluate the designs using micro-benchmarks and real applications. Our evaluations reveal that these designs can improve the resource usage of SDP and consequently its performance by an order-of-magnitude in some cases. Moreover we can achieve 10-20% improvement for various applications.

*This research is funded in part by DOE grants #DE-FC02-06ER25749 and #DE-FC02-06ER25755; by NSF grants #CNS-0403342 and #CNS-0509452; by an STTR subcontract from RNet Technologies; and by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy (contract DE-AC02-06CH11357).

1 Introduction

The Sockets Direct Protocol (SDP) [3] is an industry standard to allow existing TCP/IP sockets based applications to be executed on high-speed networks such as InfiniBand (IB) [1] and iWARP [6] (Figure 1). It is designed around two primary goals: (i) to allow existing applications to be directly and transparently deployed onto clusters connected with high-speed networks and (ii) to allow such deployment while maintaining most of the network performance for applications to utilize.

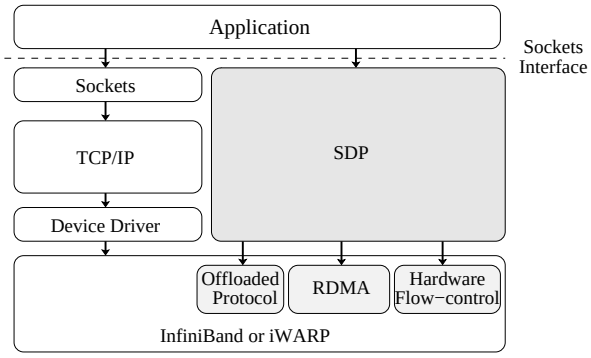


Figure 1: SDP Architecture

There have been several implementations of SDP over IB. The first implementation of SDP [9] utilized IB send-receive operations to transmit data using intermediate buffer copies. This design takes advantage of the hardware-offloaded protocol stack of IB to achieve high performance while using lesser host CPU for communication processing. Later designs of SDP [19, 18, 7] extended this to utilize IB's remote direct memory access (RDMA) capabilities and allow for zero-copy transfer of messages. Each of these designs has its pros and cons. The buffer copy approach has to deal with memory copies on both the sender and the receiver side during communication, which add overhead especially for large messages. The zero-copy approaches, on the other hand, have to deal with on-the-fly registration of buffers with the network interface card (NIC) and synchronization between the sender and receiver, which add overhead especially for small and medium-sized messages. Thus, to maximize overall performance, current SDP stacks utilize the buffer copy mechanism for communicating small and medium-sized messages (up to about 32 KB), while performing zero-copy communication with RDMA for large messages (greater than 32 KB). In this paper, we deal only with the buffer copy approach used for small and medium-sized messages.

While the existing buffer copy design takes advantage of the hardware offloaded protocol stack of IB, it is naive in aspects such as flow-control. Like many other high-speed networks, IB requires the receiver process to inform the NIC, before the data arrives, about buffers in which incoming data has to be placed. To ensure that the receiver process is ready to receive data, the sender process typically performs flow-control on the data transmission. The existing design of SDP flow-control uses send-receive-based communication, with each process managing its local flow-control buffers. With the receiver managing its local buffers, however, the sender is not aware of the receiver's exact usage status and layout. Accordingly, flow-control tends to be conservative and results in

underutilization of buffers and loss of performance.

RDMA, however, has more capabilities than just zero-copy communication. For example, it offers *sender-side buffer management*. Since RDMA is completely handled by the sender process, it allows this process to have complete control of SDP resources, such as flow-control buffers, on both the sender and receiver side. Further, IB provides other features such as hardware flow-control, whose capabilities have not been addressed so far.

Thus, in this paper, we propose two novel designs to improve the flow-control and performance of small and medium message communication in SDP over IB. In the first design, *RDMA-based flow-control*, we use RDMA to allow the sender to manage buffers on both the sender and the receiver side. This design, as we will see in the later sections, achieves a better utilization of the SDP buffer resources and consequently a better performance. However, this design assumes (from a performance standpoint) that the application will perform communication frequently enough to ensure that data is flushed out regularly from these buffers. Not doing so can result in performance penalties. In the second design, *NIC-assisted RDMA-based flow-control*, we utilize the hardware flow-control capabilities of IB to extend *RDMA-based flow-control* with communication progress (i.e., flushing out data from the SDP buffers) even when the application does not perform communication frequently enough, while not sacrificing performance.

We demonstrate the capabilities of these designs using micro-benchmarks as well as real applications. Our results show that these designs can achieve almost an order-of-magnitude improvement in the bandwidth achieved by medium sized messages. Moreover, we can achieve performance improvements of about 10% in a virtual microscope application and close to 20% in an isosurface visual rendering application.

The rest of this paper is arranged as follows. We present an overview of the relevant features of IB and SDP in Section 2. The existing flow control mechanism of SDP is discussed in Section 3. In Sections 4 and 5, we propose the RDMA-based and NIC-assisted RDMA-based flow control mechanisms. Experimental results demonstrating the performance of these designs is presented in Section 6. We discuss other existing literature related to our work in Section 7. Concluding remarks and possible future work are indicated in Section 8.

2 Overview of Relevant InfiniBand Features and the Sockets Direct Protocol

In this section, we present an overview of the features provided by IB (in Section 2.1) and SDP (in Section 2.2) that are relevant to this paper.

2.1 Overview of InfiniBand Features

In this section, we describe IB communication semantics and the hardware flow-control feature.

IB Communication Semantics: IB provides two types of communication semantics: channel semantics (send-receive communication model) and memory semantics (RDMA communication model).

Channel Semantics: In channel semantics, every send request needs a corresponding receive request at the remote end. Accordingly, the receiver process has to actively participate in communication. The sender posts a send work queue entry (WQE) to the NIC informing it about the location of the buffer from which data has to be sent out. Similarly, before any data arrives, the receiver posts a receive WQE to the NIC describing the location where the incoming data has to be placed. The NIC uses these WQEs to carry out the actual data transfer, on completion of which the WQEs are placed in completion queues (CQs) and updated to reflect the amount of data transmitted (or received), as well as any errors that may have occurred during communication.

Memory Semantics: In memory semantics, RDMA operations are used. These operations are transparent at the remote end since they do not require the remote end to be involved in communication. Therefore, in an RDMA operation, the sender manages both the local and the remote buffers. This capability is referred to as *sender-side buffer management*. There are two kinds of RDMA operations: write and read. In RDMA write, the initiator directly writes data into the remote node's buffer; in RDMA read, the initiator directly reads data from the remote node's buffer. RDMA operations also have a variant known as *RDMA with immediate data*. While RDMA with immediate data operations lose receiver transparency since they require a receive WQE to be posted, they retain the sender-side buffer management capability; that is, the sender can still dictate the location to which the data is actually written. On completion of the data transfer, the receive WQEs provide information to the receiver about how much data was written and to what location.

IB Hardware Flow-control: IB provides an end-to-end (or message-level) flow-control capability for reliable connections that can be used by a receiver NIC to optimize the use of its resources. A sender NIC cannot send a message unless it has appropriate credits to do so. Each credit represents the receiver's willingness to receive one inbound message. Specifically, each credit represents one WQE posted to the receive queue. A credit, however, does not mean that enough physical memory is allocated. Even if a credit is available, the inbound message may be larger than the buffer space allotted by the receiver. Thus, the sender and receiver have to synchronize, in software, the size of the message before any transmission. More details about this capability can be found in [2].

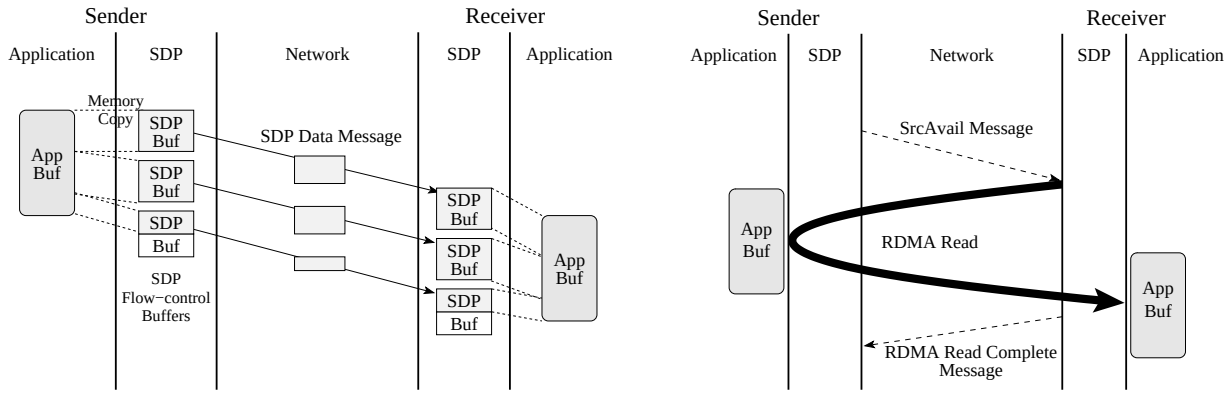


Figure 2: Existing SDP designs: (a) Buffer-copy based (b) Zero-copy based

2.2 Sockets Direct Protocol

SDP is an IB and iWARP-specific protocol standard for TCP/IP sockets that focuses on the wire protocol, finite state machine, and packet semantics. Other details can be implementation specific. SDP's upper layer protocol (ULP) interface is a byte-stream protocol. The mapping of the byte stream protocol to the underlying message-oriented network semantics is designed to enable data transfer by one of two methods: through intermediate private buffers using a buffer copy or directly between ULP buffers in a zero-copy manner.

Buffer-copy Communication: For buffer-copy communication (Figure 2(a)) [9], SDP utilizes IB send-receive operations. On a `send()` call, application data that needs to be communicated is copied into intermediate SDP flow-control buffers and transmitted to the corresponding intermediate buffers on the receiver side. On a `recv()` call, this data is copied to the final application buffer. The memory copy overhead increases with message size, making this approach beneficial only for small and medium-sized messages.

Zero-copy Communication: For zero-copy communication (Figure 2(b)) [19, 18, 7], two control messages are used, *source-avail* and *sink-avail*. When the receiver calls the `recv()` call, if the sender has not sent the data yet, the receiver sends a *sink-avail* message containing the receive buffer information to the sender. The sender uses this to directly RDMA write the data into the receive buffer. A similar approach using RDMA read and the *source-avail* control message is specified when the sender calls the `send()` operation before the receiver is ready to receive the data. The control messages associated with such zero-copy communication have two disadvantages. First, explicit synchronization is required between the sender and the receiver, i.e., until both the sender and the receiver have arrived at their respective communication calls in the application, no data can be transferred. Second, the exchange of control messages adds overhead. Thus, zero-copy communication is typically only beneficial for large messages where the benefit is greater than the overhead.

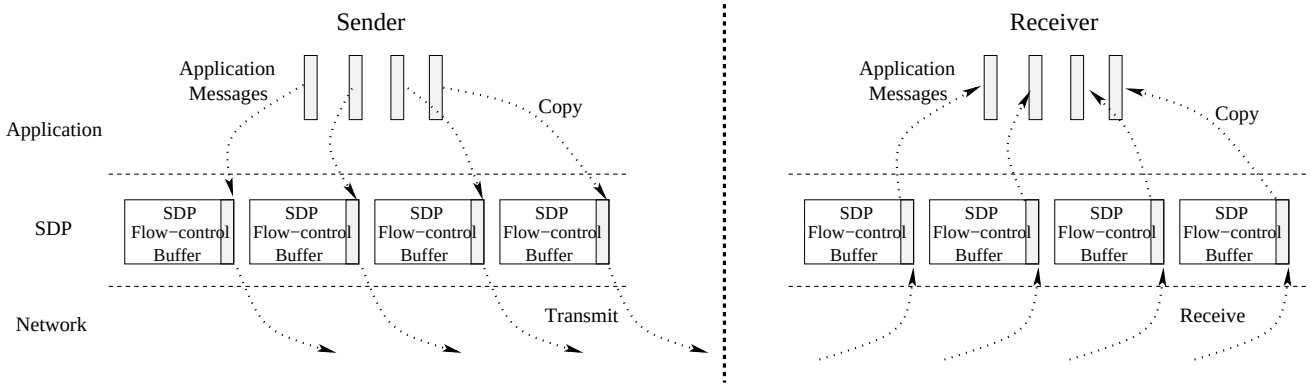


Figure 3: Credit-based Flow-control Mechanism

3 Existing Credit-based Flow-control Mechanism

As discussed earlier, several high-speed networks, including IB, require the receiver to prepost WQEs informing the NIC about receive buffers before a message arrives. These WQEs contain information about where the incoming data has to be placed. To ensure that receive WQEs are posted before any data arrives, like many other communication libraries, SDP performs flow-control of messages being sent. Current SDP implementations use a credit-based approach for achieving such flow-control. Note that this flow-control is separate from the hardware flow-control performed by IB and is a consequence of adopting existing designs of high-performance sockets on other networks [25, 21, 10] to SDP over IB. In Section 3.1, we provide an overview of the credit-based flow-control mechanism. In Section 3.2 we describe the limitations of this approach.

3.1 Overview of Credit-based Flow-control

In credit-based flow-control (Figure 3), the sender is initially given a number of credits, say N. Each process allocates N SDP send and N SDP receive flow-control buffers, each of size S bytes. The receiver posts N receive WQEs to the NIC pointing to the receive flow-control buffers; that is, the next N messages will go into these buffers. On a `send()` call, each message smaller than S bytes is copied into a send buffer and transmitted to the corresponding receive buffer. Messages larger than S bytes are segmented and transmitted in a pipelined manner. On a `recv()` call, data is copied from the receive buffer to the destination buffer, and an acknowledgment is sent to the sender informing it that the receive buffer is free to be reused. The sender loses a credit for every message sent and gains a credit for every acknowledgment received.

Previous designs [9] also use extensions to credit-based flow-control to delay acknowledgments. In other words, instead of sending an acknowledgment for every message received, the receiver can send an acknowledgment

only after half the credits have been used up. This approach reduces the amount of communication required and improves performance.

3.2 Limitations with Credit-based Flow-control

Credit-based flow-control has two primary disadvantages: buffer utilization and network utilization.

Buffer Utilization: In credit-based flow control, each message uses at least one credit irrespective of its size. For example, suppose the sender wants to send N messages each 1B, and let us say each SDP flow-control buffer is 8KB. Since the receiver has preposted N WQEs pointing to its receive buffers, each message is received in a separate receive buffer, effectively wasting the 99.98% of the SDP buffer space allotted; in other words, only 1B of each 8KB SDP buffer is utilized. This wastage of buffers also reflects on the number of messages that are transmitted; excessive underutilization of buffer space results in the sender *believing* that it has used up the receiver resources, in spite of having free buffer space available.

Network Utilization: In credit-based flow-control, on `send()` call, SDP copies the message into the send flow-control buffer, waits until it has enough credits, and immediately transmits the data to the receiver. Thus, when the application is sending out small or medium-sized messages, these messages are directly transmitted on the network. This approach results in underutilization of the network and consequently loss of performance. On the other hand, the capability to coalesce multiple small messages can allow SDP to transmit larger messages over the network and thus improve network utilization.

4 Design Overview of RDMA-based Flow-control

As described in Section 3.2, while credit-based flow-control is simple and widely accepted, it has several limitations, especially when communicating small and medium-sized messages. In this section, we describe a new flow-control approach, known as RDMA-based flow-control, that utilizes the RDMA capabilities of IB to improve the resource usage and performance of SDP flow-control.

4.1 Overview of RDMA-based Flow-control

Figure 4 illustrates RDMA-based flow-control, which differs from credit-based flow-control in two areas: improved buffer utilization and improved network utilization.

Improving Buffer Utilization: RDMA-based flow-control uses RDMA write with immediate data operations to allow the sender to manage where exactly data is buffered on the sender as well as the receiver SDP flow-control

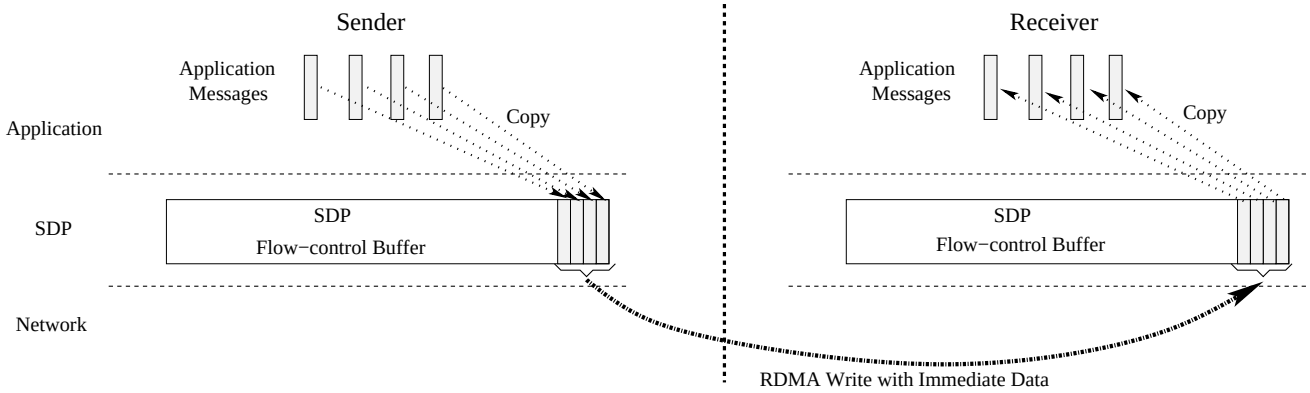


Figure 4: RDMA-based Flow-control Mechanism

buffers. This approach allows data to be better packed, thus utilizing the buffers more efficiently. In credit-based flow-control, N SDP flow-control buffers each of size S are allocated, where N is the number of credits. In RDMA-based flow-control, on the other hand, one large flow-control buffer of size $(N \times S)$ is allocated. When the first message (size P) has to be communicated, it is placed (using RDMA write with immediate data) at the start of the receive buffer. Then, when the second message of size Q has to be communicated, the sender knows the exact usage of the receive buffer; in other words, the first P bytes of the SDP buffer are used. Thus, the second message is written (again, using RDMA write with immediate data) starting at byte $(P + 1)$ of the receiver buffer. This approach allows the sender to completely utilize the available space in the sender as well as receiver SDP buffers. On a `recv()` call, once data is copied from the receiver SDP buffer to the destination buffer, the receiver sends an acknowledgment to the sender informing it about the additional available space.

Improving Network Utilization: As long as space is available in the SDP receive buffer, RDMA-based flow-control follows a similar approach as credit-based flow-control; it sends out the data as soon as a `send()` is called. Once no more space is available on the receiver side, messages are copied into SDP send buffers, and control is returned immediately to the application. This approach gives RDMA-based flow-control an opportunity to coalesce multiple small messages. Once space is freed up in the SDP receive buffer, this data is sent out as one large message instead of multiple small messages. This approach as two advantages. First, since as long as space is available in the receive buffer data is sent out immediately, latency of small messages is not hurt. In fact, when only a few small messages are transmitted, the performance should be similar to that of credit-based flow-control. Second, when a large number of small or medium messages are transmitted, though the first few messages are sent out immediately, the remaining messages are coalesced and sent out as a few large messages. This approach improves the network utilization and achieves better performance.

In summary, RDMA-based flow-control avoids buffer wastage by using the sender-side buffer management capability of RDMA. Moreover, it improves network utilization and communication performance by coalescing messages.

4.2 Limitations of RDMA-based Flow-control

While RDMA-based flow-control can achieve better resource utilization and high performance, it has one disadvantage: the lack of communication progress in some cases. We describe this limitation in this section.

Let us consider an example with an SDP flow-control buffer of 64KB where the sender initiates 64 message transfers of 2KB each, for a total of 128KB. Of these, the first 32 messages (64KB) are directly transferred to the SDP buffer on the receiver side. Then, if the receiver is not actively receiving data, the sender will run out of space in the receiver buffer to write more data. Thus, the remaining 32 messages (64KB) are copied to the SDP send buffer, and control is returned to the application. At this time, suppose the application on the sender side goes into a large computation loop. The application on the receiver side, however, calls the `recv()` call, copies the 64KB it has already received, frees the SDP receive buffer, and sends an acknowledgment to the sender informing it that the SDP receive buffer is free to be reused. In this situation, though the sender has buffered data that needs to be sent and has been informed by the receiver about available space in the receive buffer, it cannot *see* this information until the application comes out of the computation loop and calls a communication function. Thus, communication progress is halted.

Note that credit-based flow-control does not face this limitation because for every `send()` call, if the sender does not have credits, it blocks until credits are received and posts the data to the network before returning control.

In Section 6.2.2, we illustrate the impact of the poor communication progress in RDMA-based flow-control using experimental evaluation.

5 Design Overview of NIC-assisted RDMA-based Flow-control

Both credit-based flow-control and RDMA-based flow-control have disadvantages. Credit-based flow-control suffers from underutilization of SDP buffer resources and the network and results in low performance. While RDMA-based flow-control improves these aspects, it suffers from limitations with respect to communication progress when a large number of small messages have to be transmitted. To deal with these issues, in this section we describe a new mechanism known as NIC-assisted RDMA-based flow-control. This mechanism extends RDMA-based flow-control by utilizing the hardware flow-control capabilities offered by IB. In other words, this

scheme provides a hybrid software-hardware approach that utilizes the capabilities of software-based schemes such as RDMA-based flow-control to coalesce data as appropriate and improve performance; at the same time it utilizes the IB NIC to ensure asynchronous (hardware-controlled) communication progress.

NIC-assisted flow control comprises of two main sub-schemes: *virtual window scheme*, which aims at utilizing the IB hardware flow-control capability while handling its shortcomings, and *asynchronous interrupt scheme*, which enhances the virtual window scheme to improve performance by coalescing data.

5.1 Virtual Window Scheme

IB's hardware flow-control is not a byte-level flow-control, but rather a message-level flow-control; it makes sure that the sender NIC sends out only as many messages as the receiver NIC is expecting. The onus of ensuring that the receiver has appropriate buffer space for each message is on the upper layers such as SDP. To handle this situation, we utilize the *virtual window (W)* scheme. The primary idea of this scheme is to ensure that each posted receive WQE has a guarantee on the amount of buffer space available. For example, if the sender wants to send a message of 8KB, the receiver has to post a receive WQE only after 8KB of space is available.

In this scheme, the receiver posts a receive WQE only when at least the necessary virtual window size space is available in the SDP receive buffer. Thus, if the SDP buffer size is S bytes, the receiver initially posts S/W receive WQEs, where W is the virtual window size. The sender, likewise, makes sure that message segments posted to the network are always smaller than or equal to W bytes, by performing appropriate segmentation. Thus, the first S/W messages can definitely be accommodated in the SDP receive buffer. If the sender has to send more messages than S/W, it posts send WQEs corresponding to the additional data. However, since all the posted receive WQEs would be used up, IB hardware flow-control ensures that this data is not sent out by the sender NIC until the receiver posts additional receive WQEs.

We note that although each receive WQE corresponds to W bytes of available buffer space, this space can be anywhere in the SDP receive buffer; that is, the mapping between the WQE and the actual location of the corresponding buffer is not performed by the receiver. The sender uses RDMA write with immediate data operations to manage the actual location of the buffer to which each receive WQE maps. This flexibility allows the receiver to manage only the logical space allocated to each WQE, instead of the actual SDP receive buffer. For example, suppose the SDP buffer is 64KB and the virtual window is 8KB. The receiver initially posts 8 receive WQEs. The virtual window allocated to each receive WQE would be bytes (1 to 8K), (8K+1 to 16K), and so forth. Now, suppose the first message is only 1KB. In this case, the virtual windows corresponding to the remaining WQEs

automatically shift by 7KB and would be bytes (1K+1 to 9K), (9K+1 to 17K), and so forth. The final 7KB is retained as free space. Since the sender is managing the actual SDP receive buffers, this shifting of the virtual windows is transparent to the receiver process. Later, if the second message that arrives is also 1KB, the virtual windows for the remaining WQEs again automatically shift and leave a total of 14KB of free space. Since this free space is more than the virtual window size (8KB), SDP can post an additional WQE, after which 6KB of free space will still be available. When the receiver applications calls a `recv()`, the data in the SDP receive buffer is copied to the destination buffer, and more free space is created.

5.2 Asynchronous Interrupt Scheme

While the virtual window scheme provides capabilities to utilize IB hardware flow-control, it does not utilize any techniques such as coalescing messages to improve performance. The asynchronous interrupt scheme thus is designed based on two primary goals: (i) to coalesce messages and improve performance and (ii) to utilize the virtual window scheme together with IB hardware interrupts to carry out asynchronous communication progress without hurting performance.

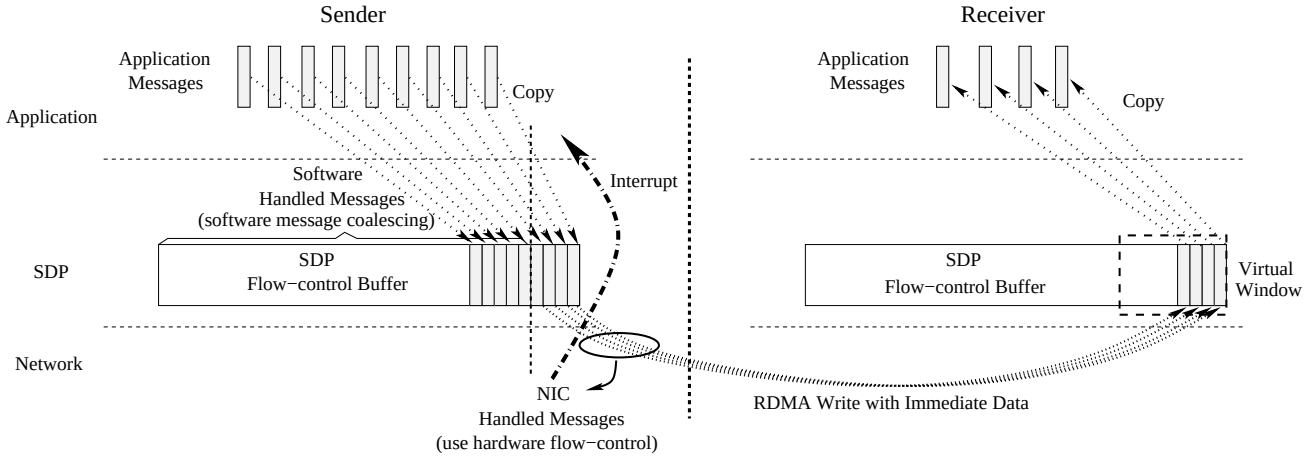


Figure 5: NIC-assisted RDMA-based Flow-control Mechanism

Message Coalescing: As shown in Figure 5, in this scheme the SDP send buffer is divided into two portions: NIC-handled buffer and software-handled buffer. The NIC-handled buffer follows a similar pattern as the virtual window scheme. That is, data is copied into the local SDP send buffer, and a corresponding send WQE is posted to the NIC. The NIC uses IB hardware flow-control to send the data only after the receiver posts a receive WQE. After the NIC-handled buffer is full, data is copied into the software-handled buffer. However, this data is not directly sent out but is held in the buffer to allow it to be coalesced with later messages.

Asynchronous Communication Progress: During message coalescing, data is copied into the software-handled

SDP buffer and control returned to the application. If more messages are communicated later, they can be coalesced together with this data to form larger messages and thus improve performance. If no other messages are communicated later, however, we need to asynchronously flush this data out. To do so, we request IB hardware interrupts for the messages in the NIC-handled buffer. Thus, once the first message that is queued in the NIC-handled buffer is transmitted, an interrupt is generated that is appropriately handled to flush out the data in the software-handled buffer as well. Although hardware interrupts are typically expensive, in this design the NIC can continue to transmit other messages in the NIC-handled buffer (using IB hardware flow-control), thus parallelizing the interrupt processing with communication. This design allows us to handle the interrupt without facing any performance penalty.

6 Experimental Results

In this section, we compare the performance of RDMA-based flow-control and NIC-assisted RDMA-based flow-control, with that of credit-based flow-control. We first describe the experimental test-bed in Section 6.1. Next, we evaluate the designs based on micro-benchmarks in Section 6.2 and then on real applications in Section 6.3.

6.1 Experimental Test-bed

The experimental test-bed consists of a 16-node cluster with dual 3.6 GHz Intel Xeon EM64T processors. Each node has a 2 MB L2 cache and 512 MB of 333 MHz DDR SDRAM. The nodes are equipped with Mellanox MT25208 InfiniHost III DDR PCI-Express adapters and are connected to a Mellanox MTS-2400, 24-port fully nonblocking DDR switch. The SDP stack is an in-house implementation at the Ohio State University. This stack is similar to other SDP stacks such as that available in the OpenFabrics distribution [5] except that it is completely in user-space (OpenFabrics SDP is in kernel-space) and is built over the VAPI verbs interface provided by Mellanox Technologies (OpenFabrics SDP is build over the OpenFabrics Gen2 verbs interface).

For each experiment, ten or more runs/executions are conducted, the highest and lowest values are dropped (to discard anomalies), and the average of the remaining values is reported. For micro-benchmark evaluations, the results of each run are an average of 10,000 or more iterations.

6.2 Micro-benchmark Based Evaluation

In this section, we evaluate the flow-control designs using various micro-benchmark tests.

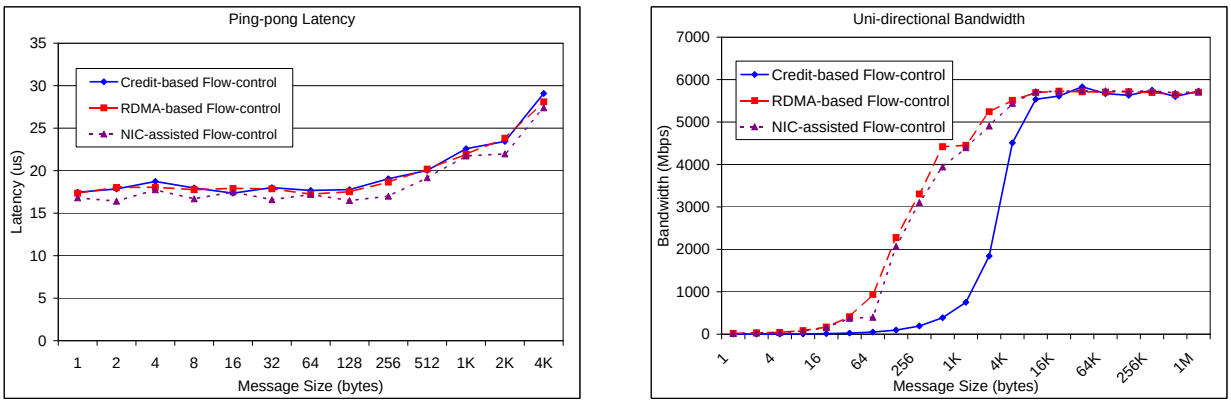


Figure 6: SDP micro-benchmark evaluation: (a) Ping-pong Latency and (b) Uni-directional Bandwidth

6.2.1 Ping-pong Latency and Uni-directional Bandwidth

Ping-pong Latency: Figure 6(a) shows the ping-pong latency of SDP with the three flow-control designs. In this experiment, the sender sends a message of size *S* to the receiver, on receiving which the receiver sends back another message of the same size to the sender. This is repeated several times and the total time averaged over the number of iterations to give the average round-trip time. The ping-pong latency reported here is one-half of the round-trip time, that is, the time taken for a message to be transferred from one node to another.

As shown in the figure, all three schemes perform identically. This result is expected because the three schemes differ only in the way they handle flow-control when there is either no remote credit available (in credit-based flow-control) or no space available in the remote SDP buffer (in RDMA-based and NIC-assisted flow-control). In the ping-pong latency test, only one message is communicated before the sender waits for a response from the remote process. Thus, there is no flow-control issue in this test and hence all schemes behave identically.

Unidirectional Bandwidth: Figure 6(b) shows the unidirectional bandwidth of the three flow-control mechanisms. In this experiment, the sender sends a single message of size *S* a number of times to the receiver. On receiving all the messages, the receiver sends back one small message to the sender indicating that it has received the messages. The sender calculates the total time, subtracts the one way latency of the message sent by the receiver, and based on the remaining time calculates the amount of data it had transmitted per unit time.

As shown in the figure, RDMA-based flow-control achieves the best performance, while credit-based flow-control achieves the worst, especially for small and medium-sized messages. For messages in the 256B to 4KB range, we notice almost an order of magnitude better performance. This behavior is expected because RDMA-based flow-control coalesces messages and thus utilizes the network more effectively resulting in a significantly better performance. In the figure, we also notice that the performance of NIC-assisted RDMA-based flow-control is

very close to that of RDMA-based flow-control. This result shows that our scheme is able to effectively hide the cost of interrupt handling by overlapping interrupt processing with data transfer time.

6.2.2 Communication Progress Benchmark

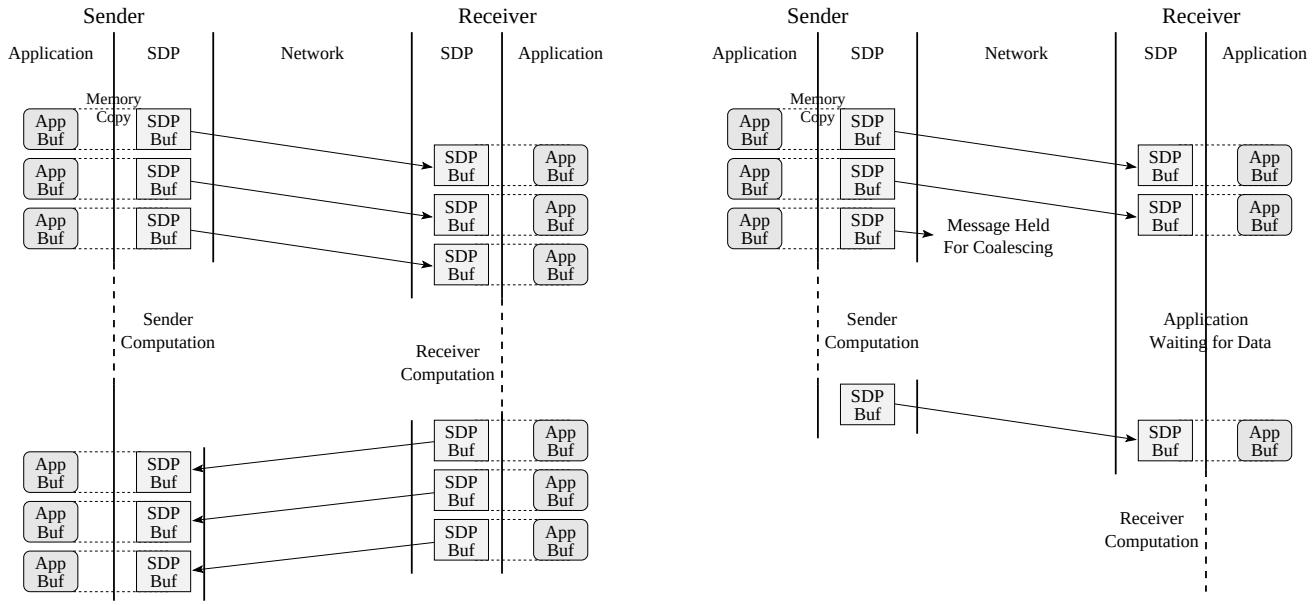
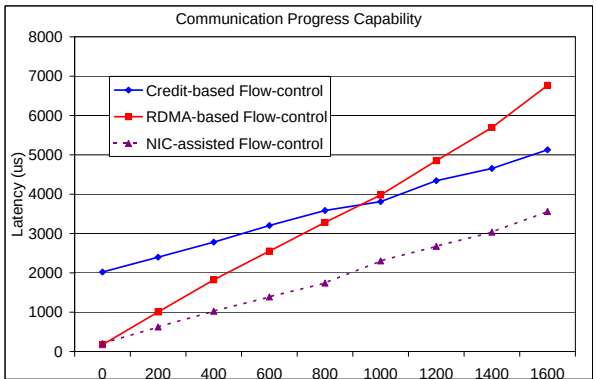


Figure 7: Communication Progress Test: (a) An example of good communication progress – sender and receiver computations are nearly parallelized and (b) An example of bad communication progress – sender and receiver side computations are serialized.

The communication progress test (Figure 7) is similar to a ping-pong latency test but with two changes. First, instead of one message being sent in each direction, a burst of 100 messages is used. Second, after each burst, an additional computation is added. If the flow-control scheme can achieve good communication progress (Figure 7(a)), it can send out data even when the application is performing other computation. Thus, the receiver can receive the data immediately, and the computation on both the sender and the receiver is parallelized to some extent. However, if the flow-control scheme buffers data in its send buffer without performing good communication progress (Figure 7(b)), the transmission of data is delayed until the computation is completed; that is, the receiver would be waiting to receive more data, which is available in the sender's SDP buffer but has not been transmitted. Only after the sender's computation is complete, when it tries to receive data, is this data flushed out. Thus, in this case, the computation on the sender and receiver is completely serialized resulting in poor performance.

In Figure 8, we report the performance of the three flow-control schemes for a message size of 4KB with varying amounts of computation. In the figure, we notice that when there is no or minimal computation, RDMA-based flow-



control and NIC-assisted RDMA-based flow-control take the least amount of time. Credit-based flow-control, on the other hand, takes the most time. As the amount of computation increases, however, we see that credit-based flow-control and NIC-assisted RDMA-based flow-control scale well, while RDMA-based flow-control deteriorates rapidly. In fact, for computation amounts greater than 1000μs, it is outperformed even by credit-based flow-control.

This test shows that credit-based flow-control and NIC-assisted flow-control are able to achieve good communication progress even when the application performs interleaving computation. For credit-based flow-control, when no remote credits are available, the scheme just blocks, waiting for the credits. Thus, the `send()` call does not return until the data is actually sent out. Consequently, the communication progress is good. For NIC-assisted RDMA-based flow-control, although data is buffered in the SDP send buffer without being immediately transmitted, the NIC interrupt ensures that the data is flushed out even when the application is busy with its computation. Thus, again the communication progress is good. RDMA-based flow-control, on the other hand, is not able to achieve good communication progress because this scheme buffers data hoping to coalesce it with later messages. Without communicating more messages, however, when the application starts doing additional computation, the buffered data has to wait without being flushed out.

6.2.3 Buffer Utilization Test

The buffer utilization test demonstrates the amount of SDP buffer space that is utilized by the different schemes. In this benchmark, we profile the SDP library to periodically monitor the amount of buffer space in which data is already copied and is not free to be used. The average percentage usage of the buffer space is measured and shown in Figures 9(a) (for 64KB SDP buffer size) and 9(b) (for 256KB SDP buffer size). We note two important aspects in these figures:

1. The buffer utilization of the RDMA-based flow-control and NIC-assisted RDMA-based flow-control is much higher than that of credit-based flow-control. This is attributed to the sender-side buffer management capability of RDMA, which allows data messages to be placed more compactly, thus allowing for improved buffer usage. In credit-based flow-control, when each SDP buffer is 8KB (Figure 9(a)), the scheme is able to reach 100% utilization only for message sizes of 8KB or higher. When each SDP buffer is 32KB (Figure 9(b)), the scheme achieves a maximum of 25% buffer utilization.

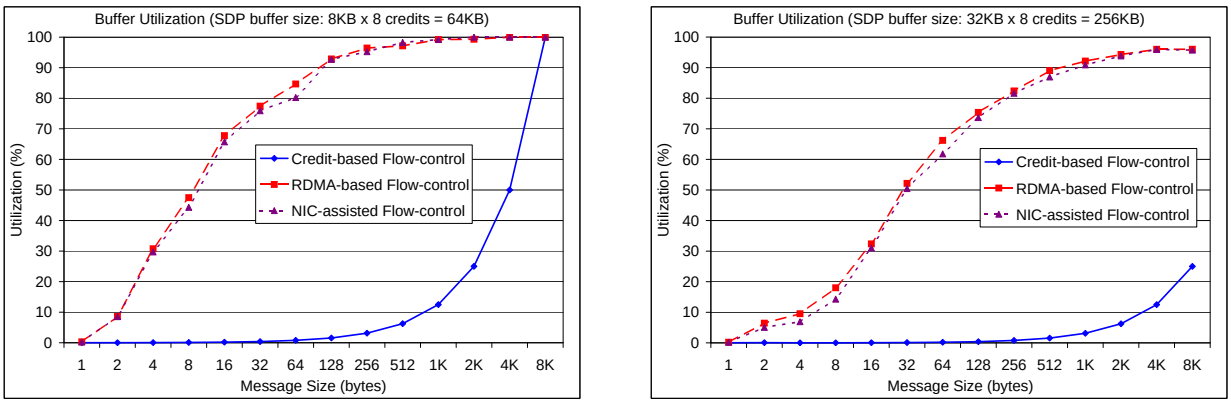


Figure 9: Buffer Utilization with SDP buffer size of : (a) 8KB x 8 credits = 64KB and (b) 32KB x 8 credits = 256KB

2. Although the overall trend of these results is similar to the bandwidth test (Figure 6(b)), we notice that the buffer utilization peaks a lot more rapidly; that is, for a SDP buffer size of 64KB, peak buffer utilization is achieved at a message size of 512B itself. This indicates that the sender is able to pack data into the send buffers and is ready to transmit it, but the receiver is not able to receive data as fast, resulting in more data being accumulated in the SDP buffers and consequently a high buffer utilization.

6.3 Application-based Evaluation

In this section, we evaluate the three flow-control designs based on two different applications, virtual microscope [15] and iso-surface visual rendering [14], that have been developed using the data-cutter library [12]. We first give an overview of the data-cutter library and the two applications, and then demonstrate their performance using the different flow-control designs.

Overview of the Data-cutter Library: Data-Cutter is a component-based framework [13, 16, 23, 24] developed by University of Maryland. It provides a framework, called filter-stream programming, for developing data-intensive applications. In this framework, the application processing structure is implemented as a set of components, called *filters*. Data exchange between filters is performed through a *stream* abstraction that denotes a unidirectional data flow from one filter to another. The overall processing structure of an application is realized by a *filter group*, which is a set of filters connected through logical streams. An application query is handled as a *unit of work* (UOW) by the filter group. The size of the UOW also represents the granularity in which data segments are distributed in the system and the granularity in which data processing is pipelined. Several data-intensive applications have been designed and developed by using the data-cutter run-time framework such as the virtual microscope application and the iso-surface visual rendering application.

Virtual Microscope: Virtual microscope [15] is a digitized microscopy application. The software support required to store, retrieve, and process digitized slides to provide interactive response times for the standard behavior of a physical microscope is a challenging issue [4, 15]. The main difficulty stems from the handling of large volumes of image data, which can range from a few hundreds of megabytes to several gigabytes. At a basic level, the software system should emulate the use of a physical microscope, including continuously moving the stage and changing magnification. The processing of client queries requires projecting high-resolution data onto a grid of suitable resolution and appropriately composing pixels mapping onto a single grid point.

Iso-surface Visual Rendering: Iso-surface rendering [17] is widely used technique used in many application areas, including environmental simulations, biomedical images, and oil reservoir simulators, for extracting and simplifying the visualization of large datasets within a 3D volume. In this paper, we utilize a component-based implementation of iso-surface rendering [14].

Evaluation of the Data-cutter Applications: Figure 10 shows the performance of the virtual microscope and iso-surface visual rendering applications for the different flow-control designs. Both applications have been executed with a UOW of 1KB. The complete dataset is about 1GB, which is hosted on a *RAM disk* in order to avoid disk fetch overheads in the experiment. The virtual microscope application used five filters: *read data*, *decompress*, *clip*, *zoom*, and *view*. For this application, five instances of the filter group (total 25 filters) were placed on 13 dual-processor nodes. The iso-surface visual rendering application used four filters: *read dataset*, *iso-surface extraction*, *shade and rasterize*, and *merge/view*. For this application, six instances of the filter group (total 24 filters) were placed on 12 dual-processor nodes. Each filter performs some computation and communicates the processed data to the next filter. Once the communication is initiated, the filter starts computation on the next UOW, thus attempting to overlap communication with computation.

As shown in the figure, credit-based flow-control shows poor performance for both applications with all dataset sizes, while RDMA-based flow-control and NIC-assisted RDMA-based flow-control achieve significantly better performance. In these applications, since multiple UOWs are processed and communicated to the next filter, the coalescing capability of these designs allows them to utilize the network more effectively and hence achieve better performance. Our designs outperform credit-based flow-control by around 10% for the virtual microscope application and close to 20% for the iso-surface visual rendering application.

We also notice no major difference in the performance of RDMA-based flow-control and NIC-assisted RDMA-based flow-control. This result shows that the enhanced communication progress is not very beneficial since the

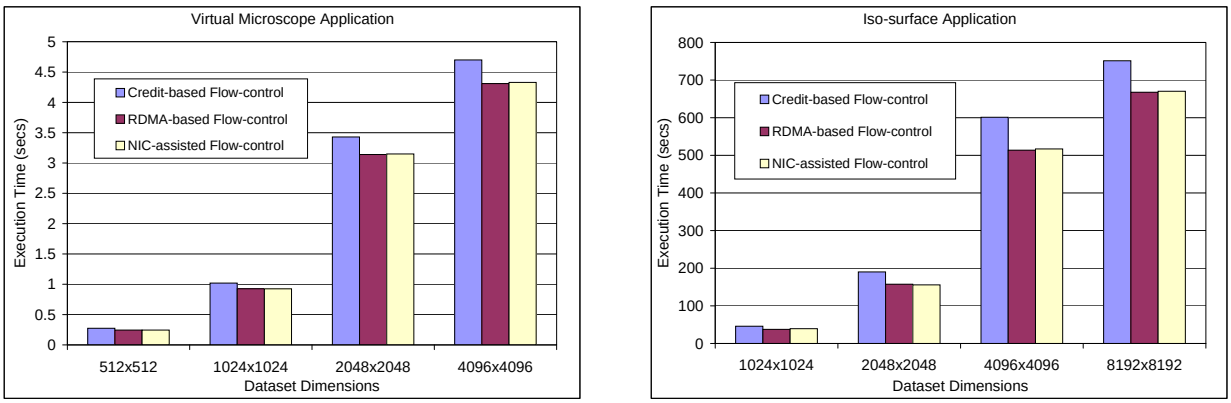


Figure 10: Evaluation with the Data-cutter Library: (a) Virtual Microscope Application and (b) Iso-surface Visual Rendering Application

applications themselves frequently make communication calls to ensure such progress.

7 Related Work

The concept of high-performance sockets, such as SDP, has been around for some time over different networks. Shah, et al., from Intel, were the first to demonstrate such an implementation over the GigaNet cLAN network [25]. This was followed by other implementations of high-performance sockets [21, 20, 11, 10]. After such high-performance sockets were standardized by the industry to form SDP, different implementations and evaluations of this standard have evolved [9, 19, 18, 8]. Our paper extends these previous designs to provide efficient flow-control and strongly complements this previous research.

Liu and Panda had previously studied the capabilities of IB hardware flow-control with respect to the Message Passing Interface (MPI) [22]. However, their study did not combine the capabilities of IB hardware flow-control with any software techniques and thus did not achieve any significant performance gains. In our paper, instead of directly using hardware flow-control, we combine it with software techniques to coalesce messages and thus achieve good performance. In summary, our paper provides a novel and interesting contribution for high-performance programming layers such as SDP.

8 Concluding Remarks and Future Work

Like many other high-speed networks, IB requires the receiver process to inform the NIC, before the data arrives, about buffers in which incoming data has to be placed. This requirement mandates the need for flow-control to ensure that the sender does not transmit messages before the receiver is ready to receive them. In this paper, we discussed the limitations with the existing flow-control mechanism, credit-based flow-control, in the Sockets

Direct Protocol (SDP) over IB. Specifically, we pointed out that SDP currently does not take advantage of various features provided by IB. For example, RDMA communication is used only for zero-copy data transfer, and its other capabilities such as *sender-side buffer management* are unutilized. Similarly, IB provides other features, such as hardware flow-control, that have not been harnessed so far. Thus, in this paper, we proposed two new flow-control mechanisms, known as RDMA-based flow-control and NIC-assisted RDMA-based flow-control, to handle these limitations and improve the resource usage and performance of SDP. We presented a detailed overview of the two designs and evaluated them using micro-benchmarks as well as applications. Our results show that these schemes can achieve nearly an order-of-magnitude improvement in the bandwidth achieved by SDP. Application-level evaluation reveals that these schemes can provide around 10% improvement in performance for a virtual microscope application and close to 20% improvement in performance for an iso-surface visual rendering application.

As future work, we would like to study similar flow-control designs in other programming models, such as the Message Passing Interface, which currently utilize credit-based flow-control mechanisms. Also, this work was done with our in-house SDP implementation over the VAPI verbs implementation provided by Mellanox Technologies. We would like to port our changes to other stacks such as OpenFabrics SDP as well.

References

[1] InfiniBand Trade Association. http://www.infinibandta.com.

[2] InfiniBand Trade Association, InfiniBand Architecture Specification, Volume 1, Release 1.0. http://www.infinibandta.com.

[3] SDP Specification. http://www.rdmaconsortium.org/home.

[4] A. Afework, M. D. Beynon, F. Bustamante, A. Demarzo, R. Ferreira, R. Miller, M. Silberman, J. Saltz, A. Sussman, and H. Tsang. Digital Dynamic Telepathology - The Virtual Microscope. In *Proceedings of the 1998 AMIA Annual Fall Symposium*. American Medical Informatics Association, November 1998.

[5] OpenFabrics Alliance. http://www.openib.org.

[6] S. Bailey and T. Talpey. Remote Direct Data Placement (RDDP), April 2005.

[7] P. Balaji, S. Bhagvat, H.-W. Jin, and D. K. Panda. Asynchronous Zero-copy Communication for Synchronous Sockets in the Sockets Direct Protocol (SDP) over InfiniBand. In *the Workshop on Communication Architecture for Clusters (CAC); held in conjunction with the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rhodes Island, Greece, Apr 2006.

[8] P. Balaji, W. Feng, Q. Gao, R. Noronha, W. Yu, and D. K. Panda. Head-to-TOE Evaluation of High Performance Sockets over Protocol Offload Engines. In *Proceedings of the IEEE International Conference on Cluster Computing*, Boston, MA, Sep 27-30 2005.

[9] P. Balaji, S. Narravula, K. Vaidyanathan, S. Krishnamoorthy, J. Wu, and D. K. Panda. Sockets Direct Protocol over InfiniBand in Clusters: Is it Beneficial? In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2004.

[10] P. Balaji, P. Shivam, P. Wyckoff, and D.K. Panda. High Performance User Level Sockets over Gigabit Ethernet. In *Proceedings of the IEEE International Conference on Cluster Computing*, September 2002.

[11] P. Balaji, J. Wu, T. Kurc, U. Catalyurek, D. K. Panda, and J. Saltz. Impact of High Performance Sockets on Data Intensive Applications. In *Proceedings of the IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 2003.

[12] M. Beynon, T. Kurc, A. Sussman, and J. Saltz. Design of a Framework for Data-Intensive Wide-Area Applications. In *Proceedings of the 9th Heterogeneous Computing Workshop (HCW2000)*, pages 116–130. IEEE Computer Society Press, May 2000.

[13] M. D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz. Distributed Processing of Very Large Datasets with DataCutter. *Parallel Computing*, October 2001.

[14] M. D. Beynon, T. Kurc, U. Catalyurek, and J. Saltz. A Component-based Implementation of Iso-surface Rendering for Visualizing Large Datasets. *Report CS-TR-4249 and UMIACS-TR-2001-34, University of Maryland, Department of Computer Science and UMIACS*, 2001.

[15] U. Catalyurek, M. D. Beynon, C. Chang, T. Kurc, A. Sussman, and J. Saltz. The Virtual Microscope. *IEEE Transactions on Information Technology in Biomedicine*, 2002.

[16] Common Component Architecture Forum. *http://www.cca-forum.org*.

[17] J. Gao and H. Shen. Parallel View Dependent Isosurface Extraction using Multi-Pass Occlusion Culling. In *Proceedings ACM/IEEE Symposium on Parallel and Large Data Visualization and Graphics*. ACM SIGGRAPH, 2001.

[18] D. Goldenberg, M. Kagan, R. Ravid, and M. Tsirkin. Transparently Achieving Superior Socket Performance using Zero Copy Socket Direct Protocol over 20 Gb/s InfiniBand Links. In *Workshop on Remote Direct Memory Access (RDMA): Applications Implementations, and Technologies (RAIT)*, 2005.

[19] D. Goldenberg, M. Kagan, R. Ravid, and M. Tsirkin. Zero Copy Sockets Direct Protocol over InfiniBand - Preliminary Implementation and Performance Analysis. In *IEEE Hot Interconnects: A Symposium on High Performance Interconnects*, 2005.

[20] J. S. Kim, K. Kim, and S. I. Jung. Building a High-Performance Communication Layer over Virtual Interface Architecture on Linux Clusters. In *Proceedings of the IEEE International Conference on Supercomputing (ICS)*, 2001.

[21] J. S. Kim, K. Kim, and S. I. Jung. SOVIA: A User-level Sockets Layer Over Virtual Interface Architecture. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2001.

[22] J. Liu and D. K. Panda. Implementing Efficient and Scalable Flow Control Schemes in MPI over InfiniBand. In *Workshop on Communication Architecture for Clusters (CAC)*, April 2004.

[23] R. Oldfield and D. Kotz. Armada: A Parallel File System for Computational Grids. In *Proceedings of CCGrid2001*, May 2001.

[24] B. Plale and K. Schwan. dQUOB: Managing Large Data Flows Using Dynamic Embedded Queries. In *IEEE International Conference on High Performance Distributed Computing (HPDC)*, August 2000.

[25] H. V. Shah, C. Pu, and R. S. Madukkarumukumana. High Performance Sockets and RPC over Virtual Interface (VI) Architecture. In *International Workshop on Communication and Architectural Support for Network-Based Parallel Computing (CANPC)*, 1999.