In the workshop on Communication Architecture for Clusters (CAC); held in conjunction with IPDPS, Rhodes Island, Greece, April 2006.
Also available as Ohio State University technical report OSU-CISRC-10/05-TR68.

Asynchronous Zero-copy Communication for Synchronous Sockets in the Sockets Direct Protocol (SDP) over InfiniBand [*]

P. Balaji S. Bhagvat H. -W. Jin D. K. Panda
Department of Computer Science and Engineering
Ohio State University
{balaji, bhagvat, jinhy, panda}@cse.ohio-state.edu

Abstract

Sockets Direct Protocol (SDP) is an industry standard pseudo sockets-like implementation to allow existing sockets applications to directly and transparently take advantage of the advanced features of current generation networks such as InfiniBand. The SDP standard supports two kinds of sockets semantics, viz., Synchronous sockets (e.g., used by Linux, BSD, Windows) and Asynchronous sockets (e.g., used by Windows, upcoming support in Linux). Due to the inherent benefits of asynchronous sockets, the SDP standard allows several intelligent approaches such as *source-avail and sink-avail based zero-copy* for these sockets. Unfortunately, most of these approaches are not beneficial for the synchronous sockets interface. Further, due to its portability, ease of use and support on a wider set of platforms, the synchronous sockets interface is the one used by most sockets applications today. Thus, a mechanism by which the approaches proposed for asynchronous sockets can be used for synchronous sockets is highly desirable. In this paper, we propose one such mechanism, termed as *AZ-SDP (Asynchronous Zero-Copy SDP)*, where we memory-protect application buffers and carry out communication asynchronously while maintaining the synchronous sockets semantics. We present our detailed design in this paper and evaluate the stack with an extensive set of benchmarks. The experimental results demonstrate that our approach can provide an improvement of close to 35% for medium-message uni-directional throughput and up to a factor of 2 benefit for computation-communication overlap tests and multi-connection benchmarks.

1 Introduction

Because traditional sockets over host-based TCP/IP have not been able to cope with the exponentially increasing network speeds, InfiniBand (IBA) [11] and other network technologies have driven the need for a new standard known as the Sockets Direct Protocol (SDP) [1]. SDP is a sockets-like implementation to meet two primary goals: (i) to directly and transparently allow existing sockets applications to be deployed on to clusters connected with modern networks such as IBA and (ii) to retain most of the raw performance provided by the networks.

The SDP standard supports two kinds of sockets semantics, viz., Synchronous sockets (e.g., used by Linux, BSD, Windows) and Asynchronous sockets (e.g., used by Windows, upcoming support in Linux). In the synchronous sockets interface, the application has to *block* for every data transfer operation, i.e., if an application wants to send a 1 MB message, it has to wait till either the data is transferred to the remote node or is copied to a local communication buffer and scheduled for communication. In the asynchronous sockets interface, on the other hand, the application can *initiate* a data transfer and check whether the transfer is complete at a later time; thus providing a better overlap of the communication with the other on-going computation in the application. Due to the inherent benefits of asynchronous sockets, the SDP standard allows several intelligent approaches such as *source-avail and sink-avail based zero-copy* for these sockets. However, most of these approaches that work well for the asynchronous sockets interface are not as beneficial for the synchronous sockets interface. Added to this is the fact that the synchronous sockets interface is the one used by most sockets applications today due to its portability, ease of use and support on a wider set of platforms. Thus, a mechanism by which the approaches proposed for asynchronous sockets can be used for synchronous sockets is highly desirable.

In this paper, we propose one such mechanism, termed as *AZ-SDP (Asynchronous Zero-Copy SDP)* which allows the approaches proposed for asynchronous sockets to be used for synchronous sockets while maintaining the synchronous sockets semantics. The basic idea of this mechanism is to protect application buffers from memory access during a data transfer event and carry out communication asynchronously. Once the data transfer is completed, the protection is removed and the application is allowed to touch the buffer again. It is to be noted that this entire scheme is completely transparent to the end application. We present our detailed design in this paper and evaluate the stack with an extensive set of micro-benchmarks. The experimental results demonstrate that our approach can provide an improvement of close to 35% for medium-message uni-directional throughput and up to a factor of 2 benefit for computation-communication overlap tests and multi-connection benchmarks.

2 Overview of SDP

The design of SDP is mainly based on two architectural goals: (i) to directly and transparently allow existing sockets applications to be deployed on to clusters connected with high-speed networks and (ii) to retain most of the raw performance provided by the network using features such as zero-copy RDMA operations. Figure 1 illustrates the SDP architecture.

SDP's Upper Layer Protocol (ULP) interface is a byte-stream protocol that is layered on top of IBA's message-oriented transfer model. This mapping was designed so as to enable ULP

[*] This research is supported in part by Department of Energy's grant #CNS-0403342 and #CNS-0509452; grants from Intel, Mellanox, Cisco Systems, Sun Microsystems and Linux Network; and equipment donations from Intel, Mellanox, AMD, Apple, Microway, PathScale and Silverstorm.

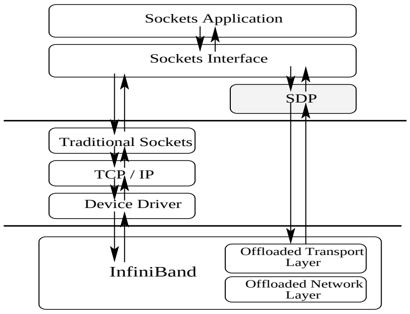

Figure 1: Sockets Direct Protocol

data to be transfered by one of two methods: through intermediate private buffers (using a buffer copy) or directly between ULP buffers (zero copy). A mix of IBA Send-Recv and RDMA mechanisms are used to transfer ULP data. The SDP specification also suggests two additional control messages known as *buffer availability notification* messages, viz., *source-avail* and *sink-avail* messages for performing zero-copy data transfer.

Sink-avail Message: If the data sink has already posted a receive buffer and the data source has not sent the data message yet, the data sink performs two steps: (i) registers the receive user-buffer and (ii) sends a *sink-avail* message containing the receive buffer handle to the source. The data source on a data transmit call, uses this receive buffer handle to RDMA write data into the remote buffer.

Source-avail Message: If the data source has already posted a send buffer and no *sink-avail* message has arrived, it performs two steps: (i) registers the transmit user-buffer and (ii) sends a *source-avail* message containing the transmit buffer handle to the data sink. The data sink on a data receive call, uses this transmit buffer handle to RDMA read data from the remote buffer.

These control messages, however, are most beneficial only for the asynchronous sockets interface due to its capability of exposing multiple source or sink buffers simultaneously to the remote node. Accordingly, most current implementations for synchronous sockets do not implement these and use only the buffer copy based scheme. Recently, Goldenberg et. al., have suggested a zero-copy SDP scheme [10, 9] where they utilize a restricted version of the *source-avail* based zero-copy communication model for synchronous sockets. Due to the semantics of the synchronous sockets, however, the restrictions affect the performance achieved by zero-copy communication significantly. In this paper, we attempt to relieve the communication scheme of such restrictions and carry out zero-copy communication in a truly asynchronous manner, thus achieving high performance.

3 Related Work

The concept of high performance sockets (such as SDP) has existed for quite some time. Several researchers, including ourselves, have performed significant amount of research on such implementations over various networks. Shah, et. al., from Intel, were the first to demonstrate such an implementation for VIA over the GigaNet cLAN network [16]. This was soon followed by other implementations over VIA [13, 6] as well as other networks such as Myrinet [15] and Gigabit Ethernet [5].

There has also been some amount of previous research for the high performance sockets implementations over IBA, i.e., SDP. Balaji et. al., were the first to show the benefits of SDP over IBA in [4] using a buffer copy based implementation of SDP. As mentioned earlier, Goldenberg et. al., recently proposed a zero-copy implementation of SDP using a restricted version of the *source-avail* scheme [10, 9]. In particular, the scheme allows zero-copy communication by restricting the number of outstanding data communication requests on the network to just one. This, however, significantly affects the performance achieved by the zero-copy communication. Our design, on the other hand, carries out zero-copy communication while not being restricted to just one communication request, thus allowing for a significant improvement in the performance.

To optimize the TCP/IP and UDP/IP protocol stacks itself, many researchers have suggested several zero-copy schemes [12, 18, 7, 8]. However, most of these approaches are for asynchronous sockets and all of them require modifications to the kernel and even the NIC firmware in some cases. In addition, these approaches still suffer from the heavy packet processing overheads of TCP/IP and UDP/IP. On the other hand, our work benefits the more widely used synchronous sockets interface, it does not require any kernel or firmware modifications at all and can achieve very low packet processing overhead (due to the thin native protocol layers of the high-speed interconnects).

In summary, AZ-SDP is a novel and unique design for high performance sockets over IBA.

4 Design and Implementation Issues

As described in Section 2, to achieve zero-copy communication, *buffer availability notification* messages need to be implemented. In this paper, we focus on a design that uses *source-avail* messages to implement zero-copy communication. In this section, we detail our mechanism to take advantage of asynchronous communication for synchronous zero-copy sockets.

4.1 Application Transparent Asynchronism

The main idea of asynchronism is to avoid blocking the application while waiting for the communication to be completed, i.e, as soon as the data transmission is initiated, the control is returned to the application. With the asynchronous sockets interface, the application is provided with additional socket calls through which it can initiate data transfer in one call and wait for its completion in another. In the synchronous sockets interface, however, there are no such separate calls; there is just one call which initiates the data transfer *and* waits for its completion. Thus, the application cannot initiate multiple communications requests at the same time. Further, the semantics of synchronous sockets assumes that when the control is returned from the communication call, the buffer is free to be used (e.g., read from or write to). Thus, returning from a synchronous call asynchronously means that the application can assume that the data has been sent or received and try to write or read from the buffer irrespective of the completion of the operation. Accordingly, a scheme which asynchronously returns control from the communication call after initiating the communication, might result in data corruption for synchronous sockets.

To transparently provide asynchronous capabilities for synchronous sockets, two goals need to be met: (i) the interface

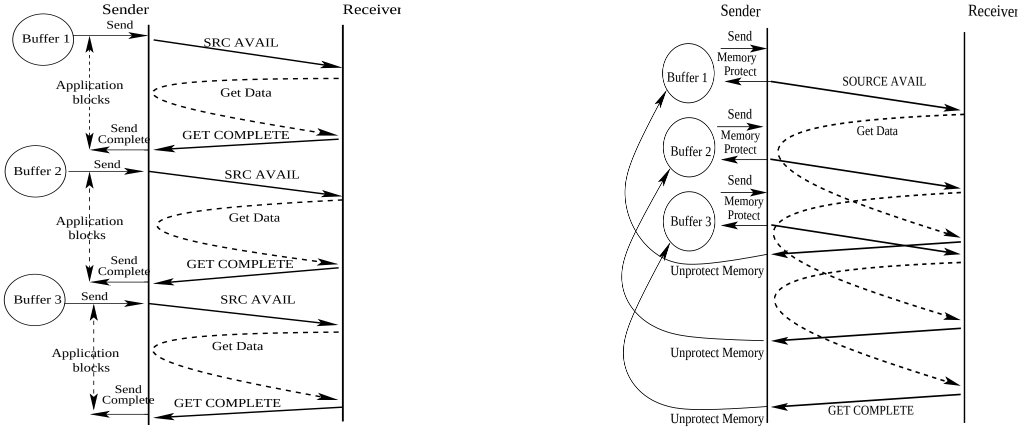

Figure 2: (a) Synchronous Zero-copy SDP (ZSDP) and (b) Asynchronous Zero-copy SDP (AZ-SDP)

should not change; the application can still use the same interface as earlier, i.e., the synchronous sockets interface and (ii) the application can assume the synchronous sockets semantics, i.e., once the control returns from the communication call, it can read from or write to the communication buffer. In our approach, the key idea in meeting these design goals is to memory-protect the user buffer (thus disallow the application from accessing it) and to carry out communication asynchronously from this buffer, while *tricking* the application into believing that we are carrying out data communication in a synchronous manner.

Figure 2 illustrates the designs of the synchronous zero-copy SDP (ZSDP) scheme and our asynchronous zero-copy SDP (AZ-SDP) scheme. As shown in Figure 2(a), in the ZSDP scheme, on a data transmission event, a *source-avail* message containing information about the source buffer is sent to the receiver side. The receiver, on seeing this request, initiates a *GET* on the source data to be fetched into the final destination buffer using an IBA RDMA read request. Once the *GET* has completed, the receiver sends a *GET_COMPLETE* message to the sender indicating that the communication has completed. The sender on receiving this *GET_COMPLETE* message, returns control to the application.

Figure 2(b) shows the design of the AZ-SDP scheme. This scheme is similar to the ZSDP scheme, except that it memory-protects the transmission application buffers and sends out several outstanding *source-avail* messages to the receiver. The receiver, on receiving these *source-avail* messages, memory-protects the receive application buffers and initiates several *GET* requests using multiple IBA RDMA read requests. On the completion of each of these *GET* requests, the receiver sends back *GET_COMPLETE* messages to the sender. Finally, on receiving the *GET_COMPLETE* requests, the sender unprotects the corresponding memory buffers. Thus, this approach allows for a better pipelining in the data communication providing a potential for much higher performance as compared to ZSDP.

4.2 Buffer Protection Mechanisms

As described in Section 4.1, our asynchronous communication mechanism uses memory-protect operations to disallow the application from accessing the communication buffer. If the application tries to access the buffer, a *page fault* is generated; our scheme needs to appropriately handle this, such that the semantics of synchronous sockets is maintained.

As we will see in Section 5.1, if the application touches the communication buffer very frequently (thus generating *page faults* very frequently), it might impact the performance of AZ-SDP. However, the actual number of *page faults* that the application would generate depends closely on the kind of application we are trying to support. For example, if a middleware that supports non-blocking semantics is built on top of the sockets interface, we expect the number of *page faults* to be quite low. Considering MPI [14] to be one example of such a middleware, whenever the end application calls a non-blocking communication call, MPI will have to implement this using the blocking semantics of sockets. However, till the application actually checks for completion, the data will remain untouched, thus reducing the number of *page faults* that might occur. Another example, is applications which perform data prefetching. As network throughput is increasing at a much faster rate as compared to the decrease in point-to-point latency, several applications today attempt to intelligently prefetch data that they might use in the future. This, essentially implies that though the prefetched data is transferred, it might be used at a much later time, if at all it is used. Again, in such scenarios, we expect the number of *page faults* occuring to be quite less. In this section, we describe generic approaches for handling the *page faults*. The performance, though, would depend on the actual number of *page faults* that the application would generate (which is further discussed in Section 5.1).

On the receiver side, we use a simple approach for ensuring the synchronous sockets semantics. Specifically, if the application calls a *recv()* call, the buffer to which the data is arriving is protected and the control is returned to the application. Now, if the receiver tries to read from this buffer before the data has actually arrived, our scheme blocks the application in the *page fault* until the data arrives. From the application's perspective, this operation is completely transparent except that the memory access would seem to take a longer time. On the sender side, however, we can consider two different approaches to handle this and guarantee the synchronous communication semantics: (i) *block-on-write* and (ii) *copy-on-write*. We discuss these alternatives in Sections 4.2.1 and Sections 4.2.2, respectively.

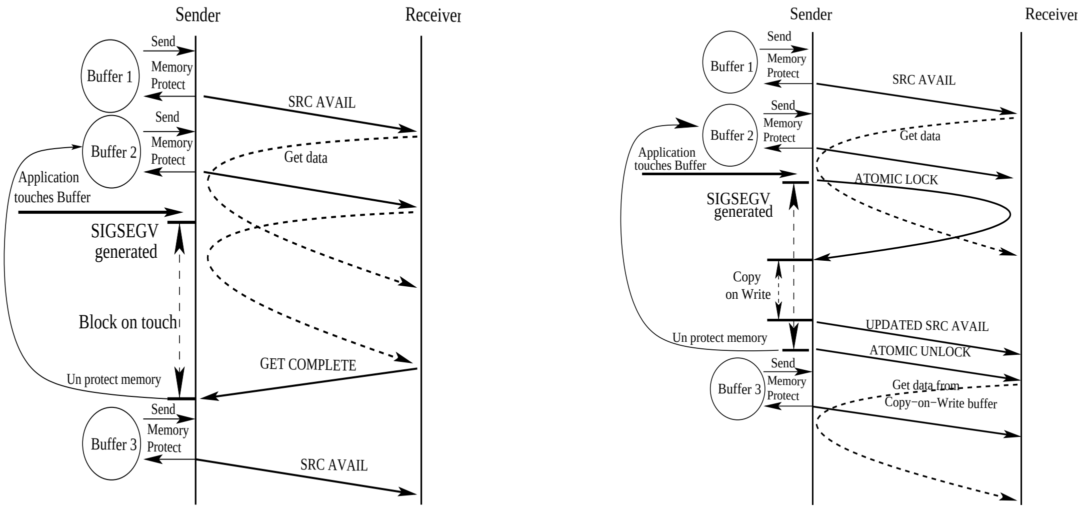

Figure 3: Buffer Protection Schemes for AZ-SDP: (a) Block-on-Write based buffer protection and (b) Copy-on-Write based buffer protection

4.2.1 Block on Write

This approach is similar to the approach used on the receiver side, i.e., if the application tries to access the communication buffer before the communication completes, we force the application to block (Figure 3(a)). The advantage of this approach is that we always achieve zero-copy communication (saving on CPU cycles by avoiding memory copies). The disadvantage of this approach is that it is not skew tolerant, i.e., if the receiver process is delayed because of some computation and cannot post a receive for the communication, the sender has to block waiting for the receiver to arrive.

4.2.2 Copy on Write

The idea of this approach is to perform a copy-on-write operation from the communication buffer to a temporary internal buffer when a *page fault* is generated. However, before control is returned to the user, the AZ-SDP layer needs to ensure that the receiver has not already started the *GET* operation; otherwise, it could result in data corruption.

This scheme performs the following steps to maintain the synchronous sockets semantics (illustrated in Figure 3(b)):

1. The AZ-SDP layer maintains a lock at the receiver side for each *source-avail* message.
2. Once the receiver calls a *recv()* and sees this *source-avail* message, it sets the lock and initiates the *GET* operation for the data using the IBA RDMA read operation.
3. On the sender side, if a *page fault* occurs (due to the application trying to touch the buffer), the AZ-SDP layer attempts to obtain the lock on the receiver side using an IBA *compare-and-swap* atomic operation. Depending on whether the sender gets a *page fault* first or the receiver calls the *recv()* operation first, one of them will get the lock.
4. If the sender gets the lock, it means that the receiver has not called a *recv()* for the data yet. In this case, the sender copies the data into a *copy-on-write buffer*, sends an *updated-source-avail* message pointing to the *copy-on-write buffer* and returns the lock. During this time, if the receiver attempts to get the lock and fails, it just blocks waiting for the *updated-source-avail* message.
5. If the sender does not get the lock, it means that the receiver has already called the *recv()* call and is in the process of transferring data. In this case, the sender just blocks waiting for the receiver to complete the data transfer and send it a *GET_COMPLETE* message.

The advantage of this approach is that it is more skew tolerant as compared to the *block-on-write* approach, i.e., if the receiver is delayed because of some computation and does not call a *recv()* soon, the sender does not have to block. The disadvantages of this approach, on the other hand, are: (i) it requires an additional copy operation, so it consumes more CPU cycles as compared to the ZSDP scheme and (ii) it has an additional lock management phase which adds more overhead in the communication. Thus, this approach may result in higher overhead than even the copy-based scheme (BSDP) when there is no skew.

4.3 Handling Buffer Sharing

Several applications perform buffer sharing using approaches such as memory-mapping two different buffers (e.g., *mmap()* operation). Let us consider a scenario where buffer *B1* and buffer *B2* are memory-mapped to each other. In this case, it is possible that the application can perform a *send()* operation from *B1* and try to access *B2*. In our approach, we memory-protect *B1* and disallow all accesses to it. However, if the application writes to *B2*, this newly written data is reflected in *B1* as well (due to the memory-mapping); this can potentially take place before the data is actually transmitted from *B1* and can cause data corruption.

In order to handle this, we override the *mmap()* call from *libc* to call our own *mmap()* call. The new *mmap()* call, internally maintains a mapping of all memory-mapped buffers. Now, if any communication is initiated from one buffer, all buffers memory-mapped to this buffer are protected. Similarly, if a *page fault* occurs, memory access is blocked (or *copy-on-write* performed) till all communication for this and its associated memory-mapped buffers has completed.

4.4 Handling Unaligned Buffers

The *mprotect()* operation used to memory-protect buffers in Linux, performs memory-protects in a granularity of a physical page size, i.e., if a buffer is protected, all physical pages on which it resides are protected. However, when the application is performing communication from a buffer, it is not necessary that this buffer is aligned so that it starts on a physical page.

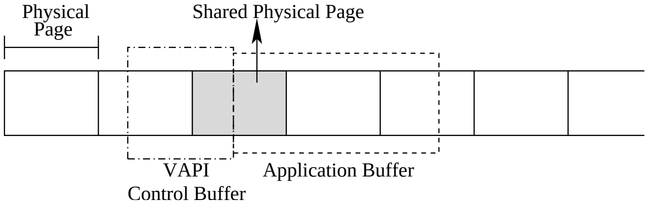

Figure 4: Physical Page Sharing Between Two Buffers

Let us consider the case depicted in Figure 4. In this case, the application buffer shares the same physical page with a control buffer used by the protocol layer, e.g, VAPI. Here, if we protect the application buffer disallowing any access to it, the protocol's internal control buffer is also protected. Now, suppose the protocol layer needs to access this control buffer to carry out the data transmission; this would result in a deadlock.

In this section, we present two approaches for handling this: (i) Malloc Hook and (ii) Hybrid approach with BSDP.

4.4.1 Malloc Hook

In this approach, we provide a hook for the *malloc()* and *free()* calls, i.e., we override the *malloc()* and *free()* calls to be redirected to our own memory allocation and freeing functions. Now, in the new memory allocation function, if an allocation for *N* bytes is requested, we allocate *N* + *PAGE_SIZE* bytes and return a pointer to a portion of this large buffer such that the start address is aligned to a physical page boundary.

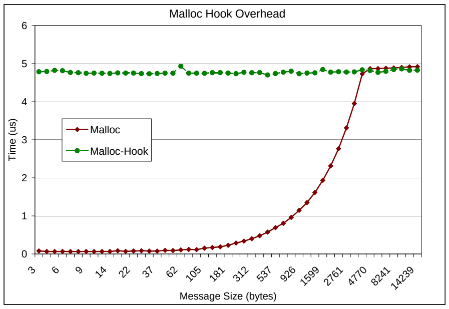

Figure 5: Overhead of the Malloc Hook

While this approach is simple, it has several disadvantages. First, if the application calls several small buffer allocations, for each call atleast a *PAGE_SIZE* amount of buffer is allocated. This might result in a lot of wastage. Second, as shown in Figure 5, the amount of time taken to perform a memory allocation operation increases significantly from a small buffer allocation to a PAGE_SIZE amount of buffer allocation. Thus, if we use a malloc hook, even a 40 byte memory allocation would take the amount of time equivalent to that of a complete physical page size, i.e., instead of 0.1μs, a 40 byte memory allocation would take about 4.8μs.

To understand the impact of the additional memory allocation time, we show the break up of the message transmission initiation phase in Table 1. As shown in the table, there are several steps involved in initiating a data transfer. Of these, the memory allocation overhead is of primary interest to us. For small message communication (e.g., source- and sink-avail messages), VAPI allocates a small buffer (40 bytes), copies the data into the buffer together with the descriptor describing the buffer itself and its protection attributes. This allows the network adapter to fetch both the descriptor as well as the actual buffer in a single DMA operation. Here, we calculate the memory allocation portion for the small buffer (40 bytes) as the fourth overhead. As we can see in the table, by adding our malloc hook, all the overheads remain the same, except for the memory allocation overhead which increases to 4.8μs, i.e., its portion in the entire transmission initiation overhead increases to about 45% from 1.5% making it the dominant overhead in the data transmission initiation part.

Table 1: Transmission Initiation Overhead

Operation	w/ Malloc (μs)	w/ Malloc Hook (μs)
Registration Check	1.4	1.4
Memory-Protect	1.4	1.4
Memory Copy	0.3	0.3
Malloc	**0.1**	**4.8**
Descriptor Post	1.6	1.6
Other	1.1	1.1

4.4.2 Hybrid Approach with Buffered SDP (BSDP)

In this approach, we use a hybrid mechanism between AZ-SDP and BSDP. Specifically, if the buffer is not page-aligned, we transmit the page-aligned portions of the buffer using AZ-SDP and the remaining portions of the buffer using BSDP. The beginning and end portions of the communication buffer are thus sent through BSDP while the intermediate portion over AZ-SDP.

This approach does not have any of the disadvantages mentioned for the previous malloc-hook based scheme. The only disadvantage is that a single message communication might need to be carried out in multiple communication operations (at most three). This might add some overhead when the communication buffers are not page-aligned. In our preliminary results, we noticed that this approach gives about 5% to 10% better throughput as compared to the malloc-hook based scheme. Hence, we went ahead with this approach in this paper.

5 Experimental Evaluation

In this section, we evaluate the AZ-SDP implementation and compare it with the other two implementations of SDP, i.e., BSDP and ZSDP. We perform two sets of evaluations. In the first set (section 5.1), we use single connection benchmarks such as ping-pong latency, uni-directional throughput, computation-communication overlap capabilities and effect of page faults. In the second set (section 5.2), we use multi-connection benchmarks such as hot-spot latency, multi-stream throughput and multi-client throughput tests. For AZ-SDP, our results are based on the *block-on-write* technique for page faults.

The experimental test-bed used in this paper consists of four nodes 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 (capable of a link-rate of

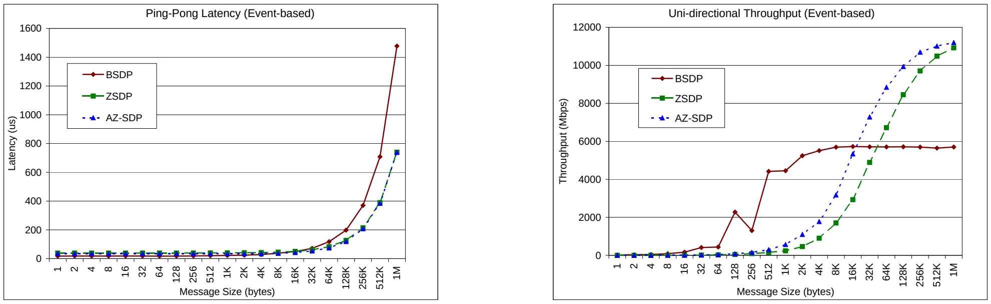

Figure 6: Micro-Benchmarks: (a) Ping-Pong Latency and (b) Unidirectional Throughput

20 Gbps) and are connected to a Mellanox MTS-2400, 24-port fully non-blocking DDR switch.

5.1 Single Connection Micro-Benchmarks

In this section, we evaluate the three stacks with a suite of single connection micro-benchmarks.

Ping-Pong Latency: Figure 6(a) shows the point-to-point latency achieved by the three stacks. In this test, the sender node first sends a message to the receiver; the receiver receives this message and sends back another message to the sender. Such exchange is carried out for several iterations, the total time calculated and averaged over the number of iterations. This gives the time for a complete message exchange. The ping-pong latency demonstrated in the figure is half of this amount (one-way communication).

As shown in the figure, both zero-copy schemes (ZSDP and AZ-SDP) achieve a superior ping-pong latency as compared to BSDP. However, there is no significant difference in the performance of ZSDP and AZ-SDP. This is due to the way the ping-pong latency test is designed. In this test, only one message is sent at a time and the node has to wait for a reply from its peer before it can send the next message, i.e., the test itself is completely synchronous and cannot utilize the capability of AZ-SDP with respect to allowing multiple outstanding requests on the network at any given time, resulting in no performance difference between the two schemes.

Uni-directional Throughput: Figure 6(b) shows the uni-directional throughput achieved by the three stacks. In this test, the sender node keeps streaming data and the receiver keeps receiving it. Once the data transfer completes, the time is measured and the data rate is calculated as the number of bytes sent out per unit time. We used the *ttcp* benchmark [17] version 1.4.7 for this experiment.

As shown in the figure, for small messages BSDP performs the best. The reason for this is two fold: (i) Both ZSDP and AZ-SDP rely on control message exchange for every message to be transferred. This causes an additional overhead for each data transfer which is significant for small messages and (ii) Our BSDP implementation uses an optimization technique known as reverse packetization to improve the throughput for small messages. More details about this can be found in [2].

For medium and large messages, on the other hand, AZ-SDP and ZSDP outperform BSDP because of the zero-copy communication. Also, for medium messages, AZ-SDP performs the best with about 35% improvement compared to ZSDP.

Computation-Communication Overlap: As mentioned earlier, IBA provides hardware offloaded network and transport layers to allow high performance communication. This also implies that the host CPU now has to do lesser amount of work for carrying out the communication, i.e., once the data transfer is initiated, the host is free to carry out its own computation while the actual communication is carried out by the network adapter. However, the amount of such overlap between the computation and communication that each of the schemes can exploit varies. In this experiment, we measure the capability of each scheme with respect to overlapping application computation with the network communication. Specifically, we modify the *ttcp* benchmark to add additional computation between every data transmission. We vary the amount of this computation and report the throughput delivered.

Figure 7 shows the overlap capability for the different schemes with the amount of computation added represented on the x-axis and the throughput measured, on the y-axis. Figure 7(a) shows the overlap capability for a message size of 64Kbytes and Figure 7(b) shows that for a message size of 1Mbyte. As shown in the figures, AZ-SDP achieves much higher computation-communication overlap as compared to the other schemes, as illustrated by its capability to retain high performance even for a large amount of intermediate computation. For example, for a message size of 64Kbytes, AZ-SDP achieves an improvement of up to a factor of 2. Also, for a message size of 1Mbyte, we see absolutely no drop in the performance of AZ-SDP even with a computation amount of 200μs while the other schemes see a huge degradation in the performance.

The reason for this better performance of AZ-SDP is its capability to utilize the hardware offloaded protocol stack more efficiently, i.e., once the data buffer is protected and the transmission initiated, AZ-SDP returns control to the application allowing it to perform its computation while the network hardware takes care of the data transmission. ZSDP on the other hand waits for the actual data to be transmitted before returning control to the application, i.e., it takes absolutely no advantage of the network adapter's capability to carry out data transmission independently.

Impact of Page Faults: As described earlier, the AZ-SDP scheme protects memory locations and carries out communication from or to these memory locations asynchronously. If the application tries to touch the data buffer before the communication completes, a *page fault* is generated. The AZ-SDP implementation traps this event, blocks to make sure that the data communication completes and returns the control to the appli-

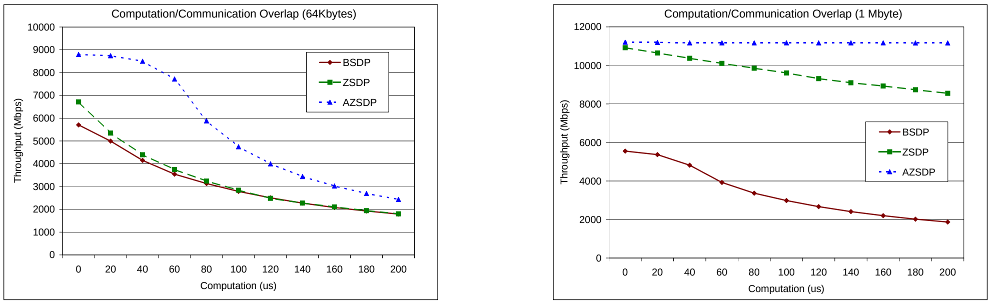

Figure 7: Computation and Communication Overlap Micro-Benchmark: (a) 64Kbyte message and (b) 1Mbyte message

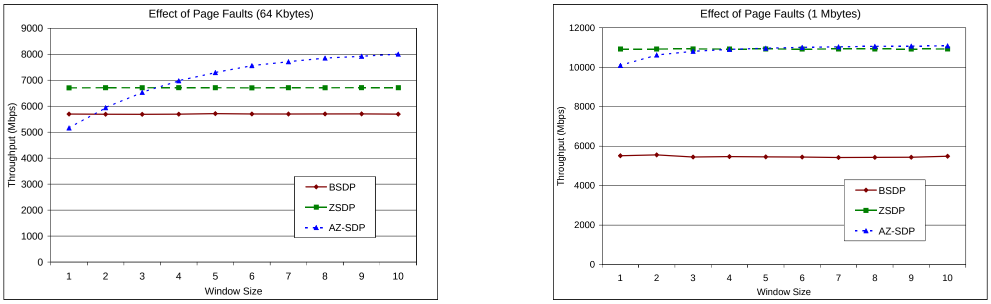

Figure 8: Impact of Page Faults: (a) 64Kbyte message and (b) 1Mbyte message

cation (allowing it to touch the buffer). Thus, in the case where the application very frequently touches the data buffer immediately after communication event, the AZ-SDP scheme has to handle several page faults adding some amount of overhead to this scheme. To characterize this overhead, we have modified the *ttcp* benchmark to touch data occasionally. We define a variable known as the *Window Size (W)* for this. The sender side first calls *W* data transmission calls and then writes a pattern into the transmission buffer. Similarly, the receiver calls *W* data reception calls and then reads the pattern from the reception buffer. This obviously does not impact the BSDP and ZSDP schemes since they do not perform any kind of protection of the application buffers. However, for the AZ-SDP scheme, each time the sender tries to write to the buffer or the receiver tries to read from the buffer, a *page fault* is generated, adding additional overhead.

Figure 8 shows the impact of *page faults* on the three schemes for message sizes 64Kbytes and 1Mbyte respectively. As shown in the figure, for small window sizes, the performance of AZ-SDP is poor. Though this degradation is lesser for larger message sizes (Figure 8(b)), there is still some amount of drop. There are two reasons for this: (i) When a *page fault* is generated, no additional data transmission or reception requests are initiated; thus, during this time, the behavior of ZSDP and AZ-SDP would be similar and (ii) Each *page fault* adds about 6μs overhead. Thus, though AZ-SDP falls back to the ZSDP scheme, it still has to deal with the *page faults* for previous protected buffers causing worse performance than ZSDP[1].

5.2 Multi-Connection Micro-Benchmarks

In this section, we present the evaluation of the stacks with several benchmarks which carry out communication over multiple connections simultaneously.

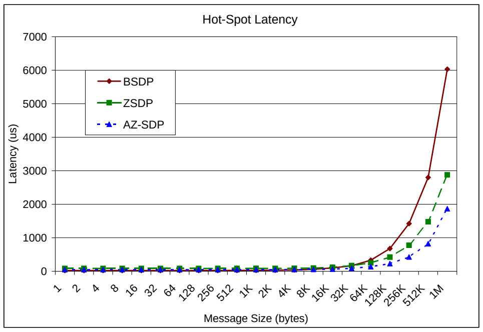

Figure 9: Hot-Spot Latency Test

Hot-Spot Latency Test: Figure 9 shows the impact of multiple connections in message transaction kind of environments. In this experiment, a number of client nodes perform point-to-point latency test with the same server, forming a hot-spot on the server. We perform this experiment with one node acting as a server node and three other dual-processor nodes hosting a total of 6 client processes and report the average of the latencies observed by each client process. As shown in the figure, AZ-SDP significantly outperforms the other two schemes especially for large messages. The key to the performance difference in this experiment lies in the usage of multiple connections for the test. Since the server has to deal with several connections at the same time, it can initiate a request to the first client and handle the other connections while the first data transfer is taking place. Thus, though each connection is synchronous, the overall experiment provides some asynchronism with respect to the number of clients the server has to deal with. Further, we expect this benefit to grow with the number of clients allowing a better scalability for the AZ-SDP scheme.

Multi-Stream Throughput Test: The multi-stream throughput test is similar to the uni-directional throughput test, except that

[1] We tackle this problem by allowing AZ-SDP to completely fall back to ZSDP if the application has generated more *page faults* than a certain threshold. However, to avoid diluting the results, we set this threshold to a very high number so that it is never triggered in the experiments.

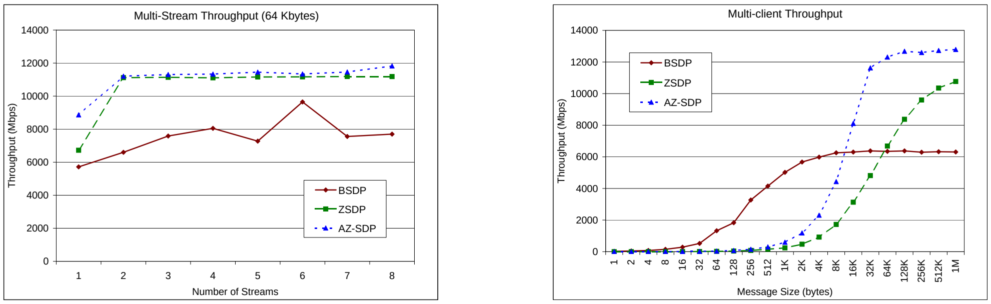

Figure 10: Multi-Connection Micro-Benchmarks: (a) Multi-Stream Throughput test and (b) Multi-Client Throughput test

multiple threads on the same pair of physical nodes carry out uni-directional communication separately. We measure the aggregate throughput of all the threads together and report it in Figure 10(a). The message size used for the test is 64Kbytes; the x-axis gives the number of threads used and the y-axis gives the throughput achieved. As shown in the figure, when the number of streams is *one*, the test behaves similar to the uni-directional throughput test with AZ-SDP outperforming the other schemes. However, when we have more streams performing communication as well, the performance of ZSDP is also similar to what AZ-SDP can achieve. To understand this behavior, we briefly reiterate on the way the ZSDP scheme works. In the ZSDP scheme, when a process tries to send the data out to a remote process, it sends the *buffer availability notification* message and *waits* till the remote process completes the data communication and informs it about the completion. Now, in a multi-threaded environment, while the first process is waiting, the remaining processes can go ahead and send out messages. Thus, though each thread is blocking for progress in ZSDP, the network is not left unutilized due to several threads accessing it simultaneously. This results in ZSDP achieving a similar performance as AZ-SDP in this environment.

Multi-client Throughput Test: In the multi-client throughput test, similar to the hot-spot test, we use one server and 6 clients (spread over three dual-processor physical nodes). In this setup, we perform the streaming throughput test between each of the clients and the same server. As shown in Figure 10(b), AZ-SDP performs significantly better than both ZSDP and BSDP in this test. Like the hot-spot test, the improvement in the performance of AZ-SDP is attributed to its ability to perform communication over the different connections simultaneously while ZSDP and BSDP perform communication one connection at a time.

6 Conclusions and Future Work

In this paper we proposed a mechanism, termed as *AZ-SDP (Asynchronous Zero-Copy SDP)*, which allows the approaches proposed for asynchronous sockets to be used for synchronous sockets, while maintaining the synchronous sockets semantics. We presented our detailed design in this paper and evaluated the stack with an extensive set of micro-benchmarks. The experimental results demonstrate that our approach can provide an improvement of close to 35% for medium-message uni-directional throughput and up to a factor of 2 benefit for computation-communication overlap tests and multi-connection benchmarks.

As future work, we plan to evaluate the AZ-SDP scheme with several applications from various domains. Also, we plan to extend our previous work on an extended sockets API [3] to AZ-SDP. This would not only provide a good performance for existing applications, but also allow for minor modifications in the applications to utilize the advanced features provided by modern networks such as one-sided communication.

References

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

[2] 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. Technical Report OSU-CISRC-10/05-TR68, Ohio State University, Columbus, Ohio, 2005.

[3] P. Balaji, H. W. Jin, K. Vaidyanathan, and D. K. Panda. Supporting iWARP Compatibility and Features for Regular Network Adapters. In *RAIT*, 2005.

[4] 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 *ISPASS '04*.

[5] P. Balaji, P. Shivam, P. Wyckoff, and D. K. Panda. High Performance User Level Sockets over Gigabit Ethernet. In *Cluster Computing '02*.

[6] P. Balaji, J. Wu, T. Kurc, U. Catalyurek, D. K. Panda, and J. Saltz. Impact of High Performance Sockets on Data Intensive Applications. In *HPDC '03*.

[7] J. Chase, A. Gallatin, and K. Yocum. End-System Optimizations for High-Speed TCP. *IEEE Communications Magazine*, 39(4):68–75, April 2001.

[8] H. J. Chu. Zero-Copy TCP in Solaris. In *Proceedings of 1996 Winter USENIX*, 1996.

[9] 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 *RAIT*, 2005.

[10] D. Goldenberg, M. Kagan, R. Ravid, and M. Tsirkin. Zero Copy Sockets Direct Protocol over InfiniBand - Preliminary Implementation and Performance Analysis. In *HotI*, 2005.

[11] Infiniband Trade Association. http://www.infinibandta.org.

[12] H. W. Jin, P. Balaji, C. Yoo, J . Y. Choi, and D. K. Panda. Exploiting NIC Architectural Support for Enhancing IP based Protocols on High Performance Networks. *JPDC '05*.

[13] J. S. Kim, K. Kim, and S. I. Jung. SOVIA: A User-level Sockets Layer Over Virtual Interface Architecture. In *Cluster Computing '01*.

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

[15] Myricom Inc. Sockets-GM Overview and Performance.

[16] H. V. Shah, C. Pu, and R. S. Madukkarumukumana. High Performance Sockets and RPC over Virtual Interface (VI) Architecture. In *CANPC Workshop '99*.

[17] USNA. TTCP: A test of TCP and UDP performance, December 1984.

[18] C. Yoo, H. W. Jin, and S. C. Kwon. Asynchronous UDP. *IEICE Transactions on Communications*, E84-B(12):3243–3251, December 2001.