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

P. BALAJI, S. BHAGVAT, H. -W. JIN AND D. K. PANDA

Technical Report
Ohio State University (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
The Ohio State University
{balaji, bhagvat, jinhy, panda}@cse.ohio-state.edu

Abstract

The Sockets Direct Protocol (SDP) is a recently proposed industry standard to allow existing sockets applications 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 and Windows) and Asynchronous sockets (e.g., used by Windows and 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 applicable 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 one used by most sockets applications today. Thus, a mechanism in which the approaches proposed for asynchronous sockets can be used for synchronous sockets would be very beneficial for such applications. 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 in an asynchronous manner while maintaining the synchronous sockets semantics. 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, up to a factor of 2 benefit for computation-communication overlap tests and multi-connection benchmarks and significant benefits in other benchmarks as well.

Keywords: SDP, InfiniBand, SDP, Asynchronous Communication

1 Introduction

With several high-performance networks being introduced into the High Performance Computing (HPC) market, each exposing its own communication interface, application developers demand a common interface that they can utilize in order to achieve portability across the various networks. The Message Passing Interface (MPI) [16, 12, 7] and the Sockets interface have been two of the most popular choices towards achieving such portability. MPI has been the *de facto* standard for scientific applications, while Sockets has been more prominent in legacy scientific applications, as well as grid-based or heterogeneous-computing applications, file and storage systems, and other commercial applications.

Because traditional sockets over host-based TCP/IP [19, 21] has not been able to cope with the exponentially increasing network speeds, InfiniBand (IBA) [13] and other network technologies recently proposed a new standard known as the Sockets Direct Protocol (SDP) [1]. SDP is a pseudo 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) allow such deployment while retaining 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 and Windows) and Asynchronous sockets (e.g., used by Windows and 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 point of time providing a better overlap of the communication with the other computation going on in the application. Further, due to the inherent benefits of asynchronous sockets, the SDP standard also allows several intelligent approaches such as *source-avail and sink-avail based zero-copy* for these sockets. Unfortunately, most of these approaches are not as 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 in which the approaches proposed for asynchronous sockets can be used for synchronous sockets would be very beneficial for such applications.

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 in an asynchronous man-

*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.

†We would like to thank S. Liang for helping us with an initial implementation of this project. We would also like to thank R. Noronha for his valuable comments during the course of this project.

ner. Once the data transfer is completed, the memory 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, up to a factor of 2 benefit for computation-communication overlap tests and multi-connection benchmarks and significant benefits in other benchmarks as well.

The remaining part of the paper is organized as follows. In Section 2, we provide a brief background about InfiniBand and SDP. In Section 3, we present some of the related work in this area and mention our research contributions in improving the existing literature. The detailed description about the design and the implementation details are presented in Section 4. We evaluate our scheme against the other approaches with a wide range of micro-benchmarks in Section 5 and conclude the paper in Section 6.

2 Background

In this section, we first provide a brief background on the InfiniBand (IBA) architecture and the subset of its features that are used in this paper. Next, in Section 2.2, we give an overview of Sockets Direct Protocol (SDP).

2.1 Overview of InfiniBand Architecture

The InfiniBand Architecture (IBA) is an industry standard that defines a System Area Network (SAN) to design clusters offering low latency and high bandwidth. The compute nodes are connected to the IBA fabric by means of Host Channel Adapters (HCAs). IBA defines a semantic interface called as Verbs for the consumer applications to communicate with the HCAs. VAPI is one such interface developed by Mellanox Technologies. Other such Verbs interfaces including Gen2-verbs, etc., also exist.

IBA supports two types of communication semantics: channel semantics (send-receive communication model) and memory semantics (RDMA communication model). In the channel semantics, every send request has a corresponding receive request at the remote end. Thus, there is a one-to-one correspondence between every send and receive operation. In the memory semantics, Remote Direct Memory Access (RDMA) operations are used. These operations are transparent at the remote end since they do not require the remote end to involve in the communication. Therefore, an RDMA operation has to specify both the memory address for the local buffer as well as that for the remote buffer. There are two kinds of RDMA operations: RDMA Write and RDMA Read. In an RDMA write operation, the initiator directly writes data into the remote node's user buffer. Similarly, in an RDMA Read operation, the initiator directly reads data from the remote node's user buffer.

Together with these communication semantics, IBA also supports network-based atomic operations directly against the memory at the end node. Atomic operations are posted as normal data transmission or reception requests at the sender side as in any other type of communication. However, the operation is completely handled by the NIC and involves very little host intervention and resource consumption. The atomic operations supported are *fetch-and-add* and *compare-and-swap*, both on 64-bit data. The *fetch-and-add* operation performs an atomic addition at the remote end. The *compare-and-swap* operation is used to compare two 64-bit values and swap the remote value with the data provided if the comparison succeeds.

2.2 Sockets Direct Protocol

Sockets Direct Protocol (SDP) is an IBA specific protocol defined by the Software Working Group (SWG) of the IBA Trade Association (IBTA) [2]. 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 IBA and (ii) allow such deployment while retaining most of the raw performance provided by the network using features such as zero-copy data transfers, RDMA operations, etc. Figure 1 illustrates the SDP architecture.

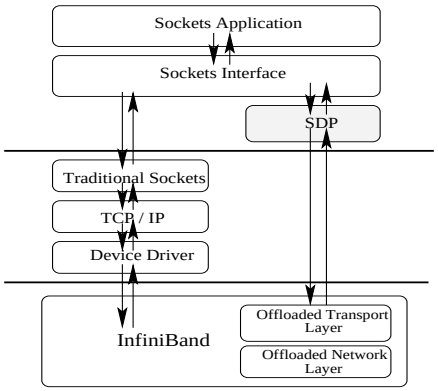


Figure 1: Sockets Direct Protocol

SDP's Upper Layer Protocol (ULP) interface is a byte-stream protocol that is layered on top of IBA's message-oriented transfer model. The mapping of the byte stream protocol to IBA's message-oriented semantics was designed to enable ULP 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 does the following steps: (i) registers the receive user-buffer (for large message reads) 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 directly RDMA write the data into the receive buffer.

Source-avail Message: If the data source has already posted a send buffer and the available SDP window is not large enough to contain the buffer, it does the following two steps: (i) registers the transmit user-buffer (for large message sends) 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 directly RDMA read the data into the receive buffer.

However, these control messages are most relevant only for the asynchronous sockets interface due to their 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 [11, 10]. In this scheme, 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

Though the Sockets Direct Protocol (SDP) standard has been recently proposed for InfiniBand (IBA), the concept of such high performance sockets has existed for quite some time over other networks. Several researchers, including ourselves, have performed a significant amount of research on such sockets implementations over various networks. Shah, et. al., from Intel, were the first to come up with such an implementation for VIA over the GigaNet cLAN network [18]. This was followed by other implementations over VIA by Kim et. al. [15] and Balaji et. al. [6]. Other networks soon adapted this solution to have their own implementations on top of Myrinet [17], Gigabit Ethernet [5], etc.

For the high performance sockets implementations over IBA, i.e., SDP, there has been some amount of previous research as well. 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 [11, 10]. 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, presents an approach by which we can carry out zero-copy communication while not being restricted to just one communication request allowing for a significant improvement in the performance.

To optimize the TCP/IP and UDP/IP protocol stacks itself, many researchers have been suggested several zero-copy schemes [14, 22, 8, 9]. However, most of these approaches are for asynchronous sockets and all of them require modifications of 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 the kernel or firmware modifications at all and can achieve very low packet processing overhead while preserving the sockets interface.

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 2.2, 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 using IBA's RDMA read operations. 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 written 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 simple 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 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 or write from/to the communication buffer. In our approach, the key idea in meeting these design goals is to memory protect the user buffer without allowing the application to access 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.

In this section, we present the basic design of the asynchronous zero-copy SDP scheme and describe its potential benefits with respect to the synchronous zero-copy SDP scheme. In Section 4.2 we describe our approach to handle scenarios where the application attempts to access the buffer involved in the communication operation and guarantee the synchronous communication semantics.

Figures 2(a) and 2(b) illustrate the designs of the synchronous zero-copy SDP (ZSDP) scheme and our asyn-

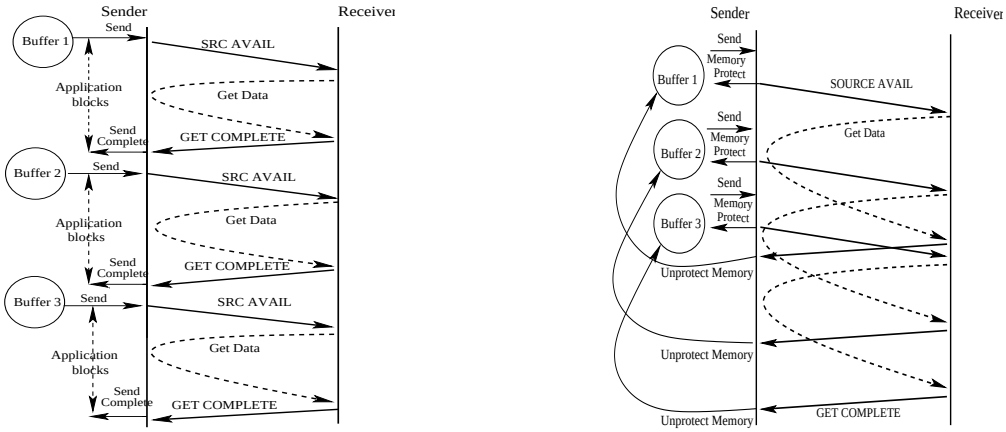


Figure 2: (a) Synchronous Zero-copy SDP (ZSDP) and (b) Asynchronous Zero-copy SDP (AZ-SDP)

chronous 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 the control to the application.

Figure 2(b) shows the design of the asynchronous zero-copy SDP (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 a much higher performance as compared to synchronous zero-copy SDP (ZSDP).

4.2 Buffer Protection Mechanisms

As described in Section 4.1, our asynchronous communication mechanism uses memory protection operations to disallow the application from accessing the buffer involved in communication. If the application tries to access the buffer, a *page fault* signal is generated; our scheme needs to appropriately handle this signal such that the semantics of synchronous sockets is maintained. In this section, we describe the different approaches possible for handling the *page fault* signal and maintaining the synchronous sockets semantics.

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* signal until the data arrives. From the application's perspective, however, 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 signal 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.

4.2.1 Block on Write

This approach is similar to the approach used on the receiver side. In this approach, 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 not only always achieve zero-copy communication (saving on precious CPU cycles by avoiding memory copy operations) but also overlap computation and communication till the application touches the buffer. 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 for the receiver to arrive and perform the data communication.

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* signal is generated and return the control to the user as soon as the data is copied. However, before this is done, the AZ-SDP layer needs to ensure that the receiver has not already started the *GET* operation; otherwise, it could result in corruption of the data received.

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 for each *SOURCE AVAIL* message it sends to the receiver. This lock is maintained at the receiver side and is initiated as soon as the *SOURCE AVAIL* message is sent to the receiver.
2. Once the receiver calls a *recv()* operation and sees this *SOURCE AVAIL* message, it sets the lock and initiates the

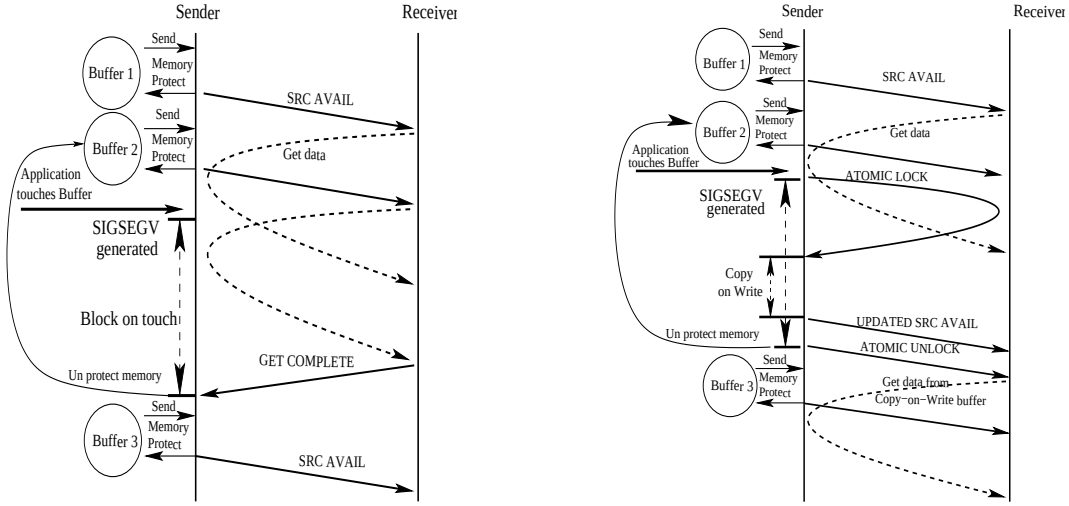


Figure 3: Buffer Protection Schemes for AZ-SDP: (a) Block-on-Write based buffer protection and (b) Copy-on-Write based buffer protection

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 a IBA *compare-and-swap* network-based atomic operation. Depending on whether the sender gets a *page fault* signal first or whether 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 made a *recv()* call 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* instead of the application buffer and returns the lock.
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 scheme would just copy the data into a *copy-on-write buffer* and allow the sender to proceed with accessing the buffer. 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 a higher overhead than even the copy-based scheme (BSDP) because of these additional overheads 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* signal arrives, the 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 protections 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.

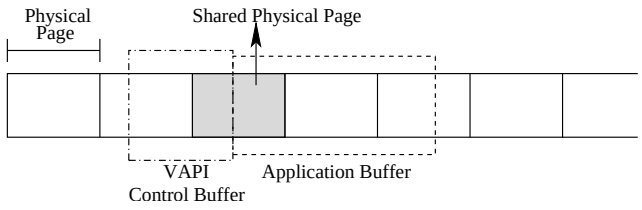


Figure 4: Physical Page Sharing Between Two Buffers

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 kind of scenarios: (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.

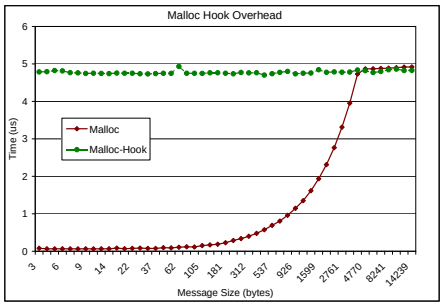


Figure 5: Overhead of the Malloc Hook

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.

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

On first sight, the second issue (with respect to the memory allocation time) does not seem to be a major issue since we typically do zero-copy communication (and hence asynchronous zero-copy communication) for large messages only. And for such communication, an additional 4.8μs might not add too much overhead. However, 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, initiating a data transfer request has several steps. First, we need to verify if the buffer used in the communication is registered with the network adapter (once a buffer is registered, we typically maintain a cache so that we do not need to register it again; here we need to check if the entry is present in our cache table). Assuming that the buffer is in our cache list, this operation takes about 1.4μs. Second, as we had mentioned earlier, in our AZ-SDP scheme we protect communication buffers from memory accesses in order to carry out the communication asynchronously. This protection operation takes about 1.4μs. Third, the *SOURCE AVAIL* control message needs to be created and copied into the transmission buffer; this takes about 0.3μs. The fourth and fifth overheads are related to the Verbs API (VAPI) by Mellanox Technologies, which we used in this paper. For small message sends, 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 (which takes about 0.1μs) and the actual posting of the descriptor to the network adapter as the fifth overhead (which takes about 1.6μs). Finally, the remaining operations such as checking the completion queue for any outstanding communication completion events, etc., take about 1.1μs. In summary, the memory allocation portion (done by VAPI) forms about 1.5% of the overhead.

Now, if we add our malloc hook, all the overheads remain the same, except for the fourth overhead, i.e., the memory allocation for VAPI to copy the buffer increases to 4.8μs; 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.

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, sent over AZ-SDP.

This approach does not have any of the disadvantages mentioned for the previous malloc-hook based scheme. The only disadvantage with this scheme is that a single message communication might need to be carried out in multiple communication operations (atmost 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., Buffered SDP (BSDP) and Zero-copy SDP (ZSDP). We perform two sets of evaluations. In the first set (section 5.1), we

use single connection micro-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 micro-benchmarks such as hot-spot latency, fan-in and fan-out streaming throughput tests.

For AZ-SDP, we currently only present results with the *block-on-write-based-fallback* technique on page faults due to performance issues with the *copy-on-write-based-fallback* technique.

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 adapter (capable of link-rate of 20 Gbps) and are connected to a Mellanox MTS-2400, 24-port completely 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, two nodes communicate with each other. 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 of messages 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, i.e., one-way communication latency.

As shown in the figure, both zero-copy based 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. This causes AZ-SDP to behave similar to ZSDP 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 is completed, 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 [20] version 1.4.7 for this experiment.

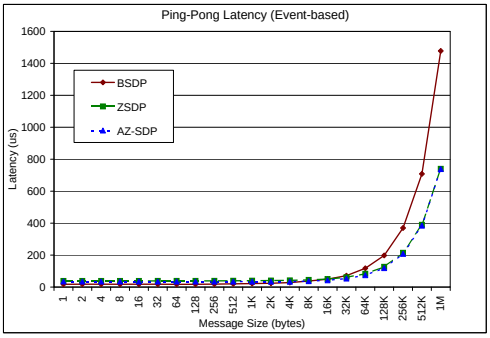
As shown in the figure, for small messages BSDP performs the best. The reason for this is two fold:

1. 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.
2. Our BSDP implementation uses an optimization technique known as reverse packetization for small messages. In this technique when the application tries to transmit data, if it has enough flow-control credits to send the data across, it copies the data and also goes ahead and sends it. On the other hand, if it does not have enough credits, it copies the data into the local socket buffer and returns the control to the application without actually transmitting the data. Now, if the application makes a second data transmission request, the new data is coalesced with the existing data in the local socket buffer and the coalesced data is scheduled for transmission. Thus, the actual data being sent out on the wire is made up of larger messages causing the throughput to increase more sharply. This technique is, however, not valid for the ZSDP and AZ-SDP implementations.

Computation-Communication Overlap: As mentioned earlier, IBA provides hardware offloaded network and transport layers to allow a high performance communication. This also implies that the host CPU now has to do a lesser amount of work for carrying out the communication, i.e., once the data transfer is initiated, the host is completely 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 (BSDP, ZSDP, AZ-SDP) 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 by the benchmark.

Figure 7 shows the overlap capability for the different schemes. 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 a significantly higher computation-communication overlap as compared to the other schemes, as illustrated by its capability to retain a high performance even for a significant amount of intermediate computation. For example, for a message size of 64Kbytes, AZ-SDP achieves an improvement of up to a factor of 2 times. 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 significant 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 the control to the application allowing it to perform its computation while the data transmission is going on, thus causing no degradation in performance. 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, thus causing a significant degradation in performance. The overlap capability of



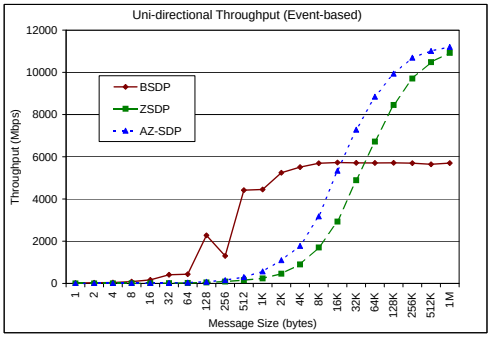
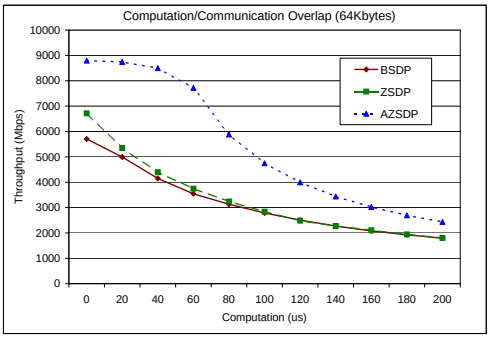


Figure 6: Micro-Benchmarks: (a) Ping-Pong Latency and (b) Unidirectional Throughput



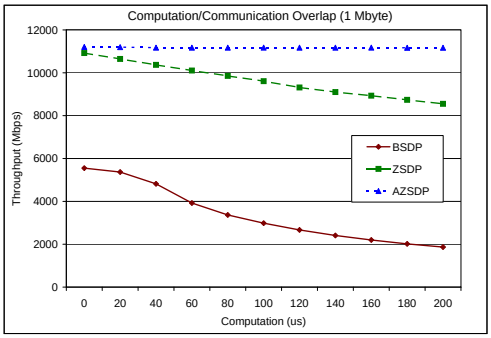


Figure 7: Computation and Communication Overlap Micro-Benchmark: (a) 64Kbyte message and (b) 1Mbyte message

BSDP depends on the amount of socket buffer allocated to it. As long as the message size is smaller than its socket buffer, BSDP can copy the data, initiate the data transfer with the network adapter and return the control back to the application to carry out its computation. If the data to be transmitted is larger than the socket buffer, BSDP blocks waiting for the data to be either received by the receiver or copied into its local socket buffer. In our experiments, we used a 64Kbyte socket buffer – the first transmission just copies the data into the socket buffer and initiates the data transfer; but the successive transmissions need to block waiting for the socket buffer to be cleared before they can copy the data.

Impact of Page Faults: As described earlier, the AZ-SDP scheme protects memory locations and carries out communication from or to these memory locations in an asynchronous manner. If before the communication completes, the application tries to touch the data buffer, a *page fault* event is generated. The AZ-SDP implementation traps this event, blocks to make sure that the data communication completes and returns the control to the application (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 occasionally touch data[1]. We define a variable known as the *Window Size (W)* for this. The transmission 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 a 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.

Figures 8(a) and 8(b) show the impact of *page faults* on the different schemes for message sizes 64Kbytes and 1Mbyte respectively. As shown in the figure, when the window size is very small, 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. The reason for this is two fold:

1. When the application attempts to touch the communication buffer and the *page fault* is generated, no additional data transmission or reception requests are initiated. Thus, during this time, the behavior of AZ-SDP would be similar to that of ZSDP with respect to the number of outstanding communication requests it can sustain.
2. Handling the *page fault* adds close to 6μs overhead to the AZ-SDP scheme. Thus, though the network behavior falls back to the ZSDP scheme, AZ-SDP still has to deal with the *page faults* for the buffers it has already protected. This causes its performance to be worse than ZSDP. As an optimization, we went ahead and implemented a patch to this problem where AZ-SDP completely falls back to ZSDP if the application is generating too many *page faults*, i.e., if the number of *page faults* generated is above a certain threshold, the AZ-SDP scheme completely avoids protecting any more buffers and carries out communication in a synchronous manner (like ZSDP). However, to avoid diluting the results, we set this threshold to a very high number so that it is never triggered in the experiments.

[1]The micro-benchmark uses the same communication buffer for every iteration.

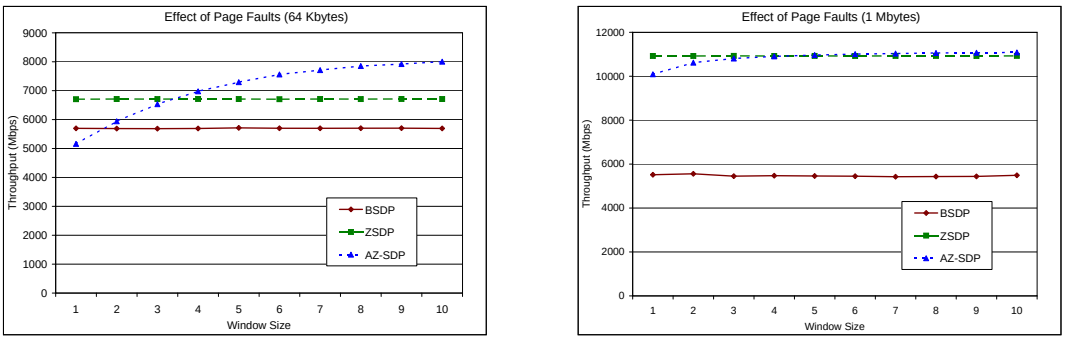


Figure 8: Impact of Page Faults: (a) 64Kbyte message and (b) 1Mbyte message

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.

Hot-Spot Latency Test: Figure 9(a) shows the impact of multiple connections in message transaction kind of environments. In this experiment, a number of client nodes perform a point-to-point latency test with the same server, forming a hot-spot on the server. We performed this experiment with one node acting as a server node and two other dual-processor nodes hosting a total of 4 client processes. As shown in the figure, AZ-SDP significantly outperforms the other two schemes especially for large messages. On first sight, this is a little surprising since this test is similar to the ping-pong latency test shown in Figure 6(a) where AZ-SDP and ZSDP show a similar performance. However, 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 being carried out. Thus, though each connection is synchronous, the overall experiment provides some amount of 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 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 9(b). The message size used for the test is 64Kbytes. 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.

Fan-in and Fan-out Throughput Tests: In the fan-in and fan-out throughput tests, similar to the hot-spot test, we use one server and 4 clients (spread over two dual-processor physical nodes). In this setup, we perform streaming throughput tests between each of the clients and the same server. The difference in the two tests is that, in the fan-in test all clients send data to the server while in the fan-out test all clients receive data from the server. We measure the aggregate throughput the server sees and report it in Figure 10.

As shown in Figure 10(a), AZ-SDP performs significantly better than both ZSDP and BSDP in the fan-in throughput 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. In the fan-out throughput test (Figure 10(b)), surprisingly we do not see this kind of a difference between ZSDP and AZ-SDP. We are currently looking into this to understand this behavior.

6 Conclusions and Future Work

Because traditional sockets over host-based TCP/IP has not been able to cope with the exponentially increasing network speeds, InfiniBand (IBA) and other network technologies recently proposed a new standard known as the Sockets Direct Protocol (SDP). The idea of SDP is to allow existing sockets applications directly and transparently take advantage of the advanced features of current generation networks such as IBA. The SDP standard supports two kinds of sockets semantics: synchronous and asynchronous. 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 as 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 one used by most sockets applications today. 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

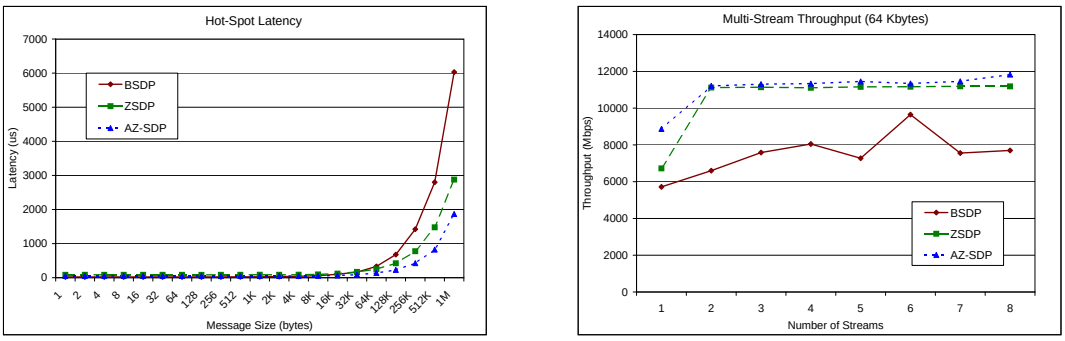


Figure 9: Multi-Connection Micro-Benchmarks: (a) Hot-Spot Latency test and (b) Multi-Stream Throughput test

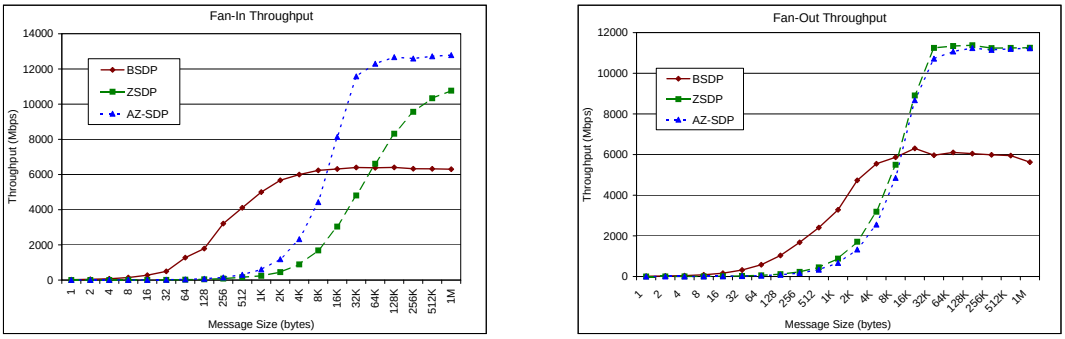


Figure 10: Multi-Connection Micro-Benchmarks: (a) Fan-in Throughput test and (b) Fan-out Throughput test

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, up to a factor of 2 benefit for computation-communication overlap tests and multi-connection benchmarks and significant benefits in other benchmarks as well.

As a part of the future work, we plan to evaluate the AZ-SDP scheme with several applications from various domains. Further, we also plan to utilize a similar idea in other networks as well to provide zero-copy and asynchronous sockets communication. Finally, we plan to extend our previous work on implementing 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, etc.

References

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

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

[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] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Supercomputing Symposium*.

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

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

[10] 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.

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

[12] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*.

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

[14] 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*.

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

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

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

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

[19] W. R. Stevens. *TCP/IP Illustrated, Volume I: The Protocols*. Addison Wesley, 2nd edition, 2000.

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

[21] G. R. Wright and W. R. Stevens. *TCP/IP Illustrated, Volume II: The Implementation*. Addison Wesley, 2nd edition, 2000.

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