# Zero Copy Sockets Direct Protocol over InfiniBand - Preliminary Implementation and Performance Analysis

Dror Goldenberg, Michael Kagan, Ran Ravid, Michael S. Tsirkin
*Mellanox Technologies Inc.*
{gdror, michael, ranr, mst} @ mellanox.co.il

## Abstract

*Sockets Direct Protocol (SDP) is a byte-stream transport protocol implementing the TCP SOCK_STREAM semantics utilizing transport offloading capabilities of the InfiniBand fabric. Under the hood, SDP supports Zero-Copy (ZCopy) operation mode, using the InfiniBand RDMA capability to transfer data directly between application buffers. Alternatively, in Buffer Copy (BCopy) mode, data is copied to and from transport buffers.*

*In the initial open-source SDP implementation, ZCopy mode was restricted to Asynchronous I/O operations. We added a prototype ZCopy support for send()/recv() synchronous socket calls.*

*This paper presents the major architectural aspects of the SDP protocol, the ZCopy implementation, and a preliminary performance evaluation. We show substantial benefits of ZCopy when multiple connections are running in parallel on the same host. For example, when 8 connections are simultaneously active, enabling ZCopy yields a bandwidth growth from 500MB/s to 700MB/s, while CPU utilization decreases 8 times.*

## 1. Introduction

The InfiniBand architecture [1] introduces a high bandwidth, low latency interconnect with RDMA capabilities, running at up to 120 gigabit per second link speeds. Current implementations support up to 20 gigabit per second wire speed on Host Channel Adapters (HCAs), and 60 gigabit per second on switches. Upper layer protocols have been developed providing standard interfaces to existing frameworks and applications on top of HCA devices. The most commonly used protocols are Message Passing Interface (MPI), IP over InfiniBand (IPoIB), Sockets Direct Protocol (SDP), SCSI RDMA Protocol (SRP), and Direct Access Provider Library (DAPL).

In this paper we discuss modifications to the InfiniBand open-source implementation of SDP ([4], [5]). We have extended the SDP implementation, and added zero copy (ZCopy) support on the synchronous I/O path. The implementation has been developed up to the prototype level, and enables us to benchmark the implementation performance.

### 1.1. Introduction to InfiniBand

The InfiniBand architecture defines a System Area Network that connects processor nodes and I/O devices. This network may comprise multiple subnets connected by routers. Each subnet may contain one or more InfiniBand Switches. Processor nodes and I/O devices connect to the InfiniBand fabric through Host Channel Adapters (HCA) and Target Channel Adapters (TCA) respectively, as illustrated in Figure 1.

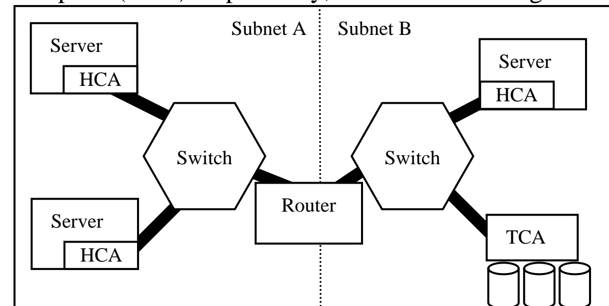


**Figure 1: InfiniBand fabric basic components**

The HCA device plugs into the host I/O subsystem. The HCA contains sophisticated DMA engine, transport engine, and management capabilities. The key contributions of the HCA to the system performance are:

- Transport offload – the HCA can perform all transport tasks, thus making it possible for the application to send and receive data reliably with minimal software overhead. This includes segmentation and reassembly, retransmission, transport checks, timers, etc.
- Zero Copy – the HCA is capable of exposing user or kernel memory buffers to the InfiniBand fabric. A remote HCA can then read or write data from and into these buffers by performing Remote DMA (RDMA) operations. RDMA operations make it possible to convey application buffers

Copyright 2005 IEEE

over the fabric, without involving the host CPU, while avoiding copying buffers into intermediate communication pre-allocated buffers.

- Kernel Bypass – through special protection enforcement mechanisms, the InfiniBand allows user applications to directly interact with the HCA hardware in order to send and receive messages. Kernel Bypass reduces the communication overhead for user-space applications.

In order to allow RDMA operations, the consumer must perform memory registration. Memory registration involves:

- Pin-Down – registered memory must be locked in physical memory and accessible by the HCA.
- Registration with the HCA – configuration of the HCA hardware to grant certain permissions to application buffers. This includes access right setting, association with a set of connections, and setup of HCA translation tables.

The InfiniBand specification defines an HCA interface called Verbs [1]. The Verbs provide operations for resource management and data transfer operations. The communication is based on Queue Pairs (QPs). InfiniBand supports Send/Receive, RDMA Read, and RDMA Write operations. These operations are initiated by posting Work Requests on the Send or Receive Queue (SQ/RQ). Data transfer operations are asynchronous. The HCA reports completion of Work Requests asynchronously by posting Completion Queue Elements (CQEs) to Completion Queues.

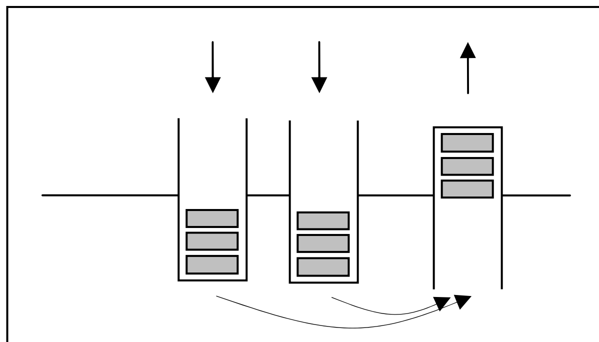

**Figure 2: Data operation interface**

The completion semantics depend on the transport service of the QP. For a reliable queue, a CQE is generated after data arrival to the remote destination is acknowledged. For an unreliable queue, a CQE is generated once data has been transmitted to the fabric. Completion of a receive request is generated when data arrives to the local receive queue.

Receive operations require that the consumer pre-posts receive buffers. Incoming RDMA operations do not require pre-posted buffers on the receive side and are handled at the transport level without involving the host CPU. The consumer can observe Work Request completions by polling the Completion Queue, or by requesting completion notifications that are delivered by means of hardware interrupts. Figure 2 illustrates the InfiniBand asynchronous interface for data operations exported through the Verbs.

### 1.2. Sockets Direct Protocol

Traditional implementations of TCP sockets typically require data copy between application buffers and NIC kernel buffers, segmentation, reassembly and other transport handling. These three operations consume CPU and memory resources, and become a performance bottleneck for high speed interconnects.

Data copying overhead has been identified in the 1990s as a significant CPU consumer in TCP stacks [2]. Protocol offload implementations addressed this overhead by implementing zero copy [3].

The Sockets Direct Protocol (SDP) (added as an annex to the InfiniBand architecture specification [1] on April 2002) eliminates the protocol stack bottlenecks for InfiniBand-based networks. SDP is a byte stream protocol that mimics TCP SOCK_STREAM semantics. Existing socket-based applications can use the SDP protocol for communication over the InfiniBand fabric, without any code modification or recompilation. Traditional TCP/IP is used outside the InfiniBand fabric.

As illustrated in Figure 5, an implementation of SDP resides between the software socket interface and the InfiniBand Verbs interface. Right under the socket interface, there is a Socket Switch that chooses between an SDP and a regular TCP socket for each particular connection, according to a configurable policy.

SDP is implemented on top of the InfiniBand Reliable Connection (RC) transport service, and leverages the reliability and transport checks implemented by the HCA hardware. SDP maps the socket send()/recv() calls to InfiniBand operations. Data messages are transferred by SEND and RDMA operations. Control messages are transferred by SEND messages. SDP supports two data transfer types: Buffer Copy (BCopy) and Zero Copy (ZCopy).

BCopy uses dedicated, pre-registered private SDP buffers. Data is copied from application buffers into SDP buffers; data transfer is then performed using the SEND operation. On the receive side, the data lands in pre-registered SDP buffers and is then copied into application buffers. Figure 3 illustrates the BCopy flow. In this example, the receiver (data sink) had been waiting for data before the data arrived. In case where the receiver is slow, the data will wait in the SDP

Copyright 2005 IEEE

buffers until the receiver calls recv(). A flow control mechanism makes it possible for the receiver to limit the number of outstanding SDP buffers. On the sender (data source) side, the send() call returns as soon as the data is copied into the local SDP buffer.

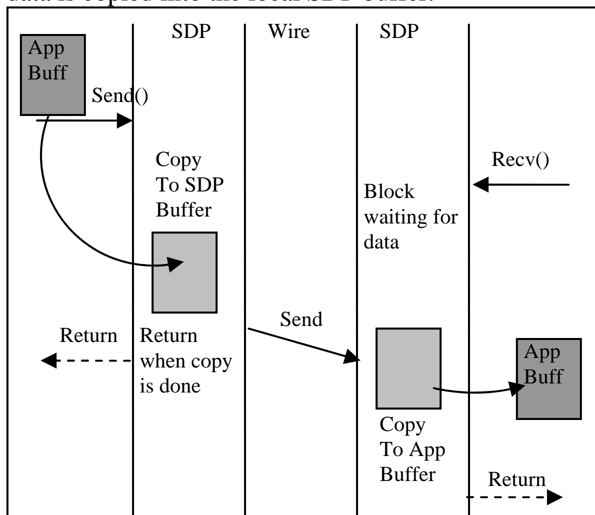


**Figure 3: BCopy flow**

ZCopy avoids the data copy overhead by direct data transfer between application buffers through RDMA operations.

Application buffers are typically not registered. Therefore, in order to enable RDMA operations, the buffers must be pinned and registered by the SDP stack implementation.

SDP supports two modes of ZCopy operation: Read ZCopy and Write ZCopy.

A Read ZCopy flow is illustrated in Figure 4. In this example, the data source gets an application buffer to send. If the buffer is large enough, the data source registers and advertises it by sending a SrcAvail message to the data sink. Later, a buffer is posted on the data sink. The buffer is registered and an RDMA Read is performed, copying the data source buffers into data sink application buffers. Once the RDMA Read is completed, the data sink indicates this by sending an RdmaRdCompl message to the data source. Note that it is also possible to perform an RDMA Read into local SDP buffers.

Write ZCopy uses an RDMA Write for data transfer. The data sink advertises a buffer by sending a SinkAvail message. The data source will then RDMA Write the buffer. Once the RDMA Write is completed, the data source indicates this by sending an RdmaWrCompl message. Read ZCopy is typically useful when the receiver is slower than the sender, while Write ZCopy is typically useful when the sender is slower than the receiver.

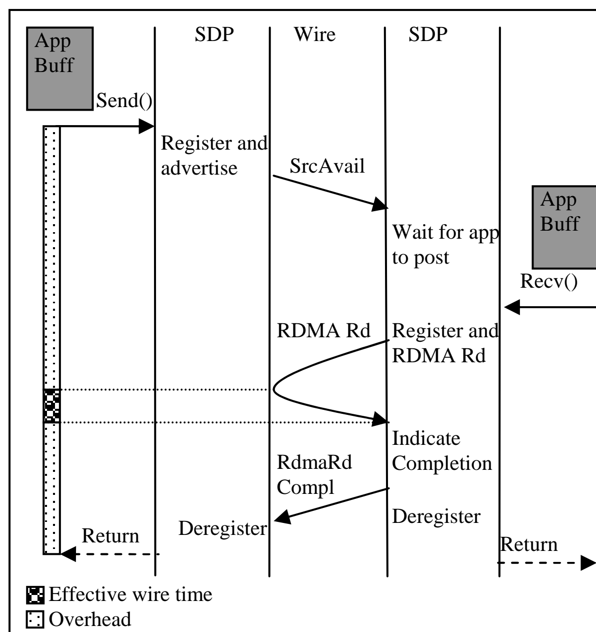


**Figure 4: Read ZCopy flow**

Both BCopy and ZCopy modes incur some communication overhead. While with BCopy we incur the overhead of a local data copy, in ZCopy mode we incur the overhead of locking the application buffers in physical memory, their registration with the HCA, and additional communication overhead. Thus, the SDP implementation uses a ZCopy threshold parameter to determine whether to take the ZCopy path or the BCopy path. We will discuss strategies for ZCopy threshold selection later on in this paper.

Each SDP half-connection has three possible data delivery modes:

- Buffered – only BCopy operations are allowed
- Combined – BCopy and Read ZCopy are allowed. It is not allowed to advertise more than one ZCopy buffer.
- Pipelined – Both BCopy and ZCopy are allowed. Pipelining of more than one advertisement is allowed.

The transition between these modes is up to the implementation. The protocol specification defines, for each mode, which side can initiate a mode transition.

## 1.3. InfiniBand Gold Collection

The InfiniBand Gold Collection [5] is an open-source Verbs and ULP stack implementation. It is based on the initial Linux open source code that was posted on the OpenIB Alliance web site [4], and has been further enhanced and stabilized by Mellanox.

The InfiniBand Gold Collection is implemented for the Linux operating system and supports numerous Linux kernels including several 2.4 and 2.6 kernels. As

Copyright 2005 IEEE

of version 1.7.0, it includes an SDP implementation with BCopy support. Originally, it did not utilize ZCopy for synchronous send()/recv() operations: ZCopy was utilized exclusively for asynchronous I/O (AIO), and supported only certain kernel versions. (See Section 2 below for a description of Zcopy design and implementation for synchronous send()/recv() operations.)

SDP is implemented as a kernel module with a socket switch in a user-mode library (libsdp), as illustrated in Figure 5. The socket switch makes it possible to listen simultaneously on both SDP and TCP sockets. The socket switch policy specifies whether an SDP or TCP socket shall be preferred based on the application name, IP address or port number.

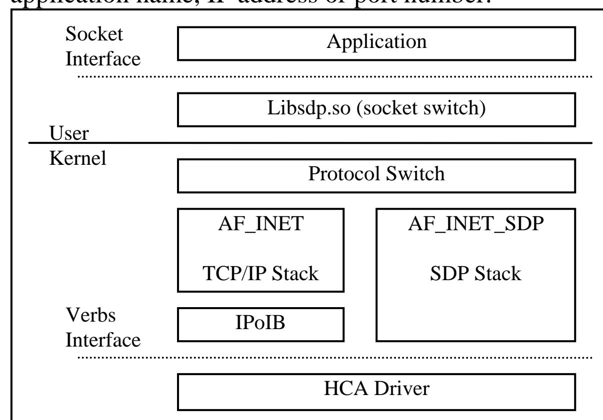


**Figure 5: Gold CD SDP stack components**

The SDP protocol is registered with the kernel as a new AF_INET_SDP protocol family. The socket switch is implemented as a user-mode shared library. When loaded, it overrides the application socket calls. When a socket is created, the socket switch determines whether a TCP or an SDP socket is required, and creates an AF_INET or an AF_INET_SDP socket object. An AF_INET_SDP object is implemented by the SDP stack kernel module. This module implements all socket entry points (net_proto_family and proto_ops) that are invoked by the kernel: create, release, bind, connect, accept, sendmsg, etc.

Since the entire SDP implementation is in the kernel, it does not leverage the kernel bypass capability of the InfiniBand architecture. Nevertheless, transport offload is fully utilized, which was shown to be beneficial to both bandwidth and CPU utilization [6]. In the following section, we discuss the design for making use of the RDMA hardware capability.

# 2. Zero Copy implementation

## 2.1. Design objectives

Our major design goal was to enable true zero copy support for synchronous socket operations in SDP.

The zero copy capability for synchronous operations was not included in the original open source stack [4]. Thus, the stack effectively behaved as if the ZCopy threshold was set to infinity. The ZCopy scheme was only supported for the asynchronous I/O (AIO) calls. This capability cannot be directly extended to support synchronous operations, for the following reasons:

- ZCopy was supported only for kernel version 2.4.21-15 Red Hat Advanced Server 3.0 update 2. On this kernel, the AIO subsystem takes care of pinning the buffers. To be more accurate, it maps the user-land buffers into the kernel address space. On the contrary, synchronous socket calls do not perform this mapping of buffers to kernel.
- The AIO subsystem is asynchronous in its nature, therefore it maps nicely to the InfiniBand model. Whereas synchronous calls require some blocking code to be implemented and some heuristics to be considered as to which operation path to take.
- The AIO code supports multiple simultaneous operations, while synchronous operations allow only one outstanding transaction at a certain time.

Note that if the message is being sent or received with a MSG_DONTWAIT flag, or if the socket has been configured with the O_NONBLOCK option, then the socket implementation cannot block in order to exchange key information and RDMA the data. Thus, ZCopy is used only for blocking synchronous socket calls. The non-blocking synchronous calls default to BCopy mode in order to preserve their semantics.

Design and implementation are discussed in further detail in the following subsections.

## 2.2. Pipelining and data transfer modes

The fundamental difference between ZCopy and BCopy modes (when using the synchronous socket calls) is the ability to pipeline transactions on the wire.

In BCopy mode, user data is copied into kernel SDP buffers and is then sent over the wire. On the data source side, when processing the send() call, SDP will copy the user data into an SDP buffer, will post a work request to the HCA and return from the send() call. This way, the user may submit multiple send() operations to the SDP layer, while some of the previous send() operations are in flight (see Figure 3:

Copyright 2005 IEEE

BCopy ). This pipelining capability makes it possible to sustain high bandwidth over a single connection. Nevertheless, the CPU will be busy copying data.

In our implementation pipelining is not possible when using ZCopy on synchronous socket calls. Read ZCopy flow is illustrated in Figure 4. The user buffer on the send side must be kept unmodified, pinned and registered until the RDMA completes. To prevent the user from modifying (or even de-allocating) the data buffer, the send() operation must block until the transaction completes. Only after data has been fully transferred through an RDMA operation, including the handshake (RdmaRdCompl or RdmaWrCompl message), the send() call may return and the application may send the next message. Thus, regardless of the ZCopy flavor chosen, pipelining is not possible in our implementation for blocking synchronous socket calls. Therefore, the bandwidth over a single connection using ZCopy may not be as high as it is when using BCopy. Additionally, we expect the overhead imposed by RdmaRdCompl or RdmaWrCompl control messages to manifest itself at relatively small message sizes. Figure 4 shows for a specific transaction the effective time (where data flows through the wire) along with the non-effective time (protocol handshake). The larger the message size, the smaller the overhead part of the transaction, and therefore the better the bandwidth that can be obtained.

For the blocking socket operations we chose to implement only the Read ZCopy flow, which is used in Buffered and Combined data delivery modes. The implementation of Write ZCopy flow, used only in Pipelined mode, is suggested for future work.

While extending the scope to multiple socket connections running concurrently, the overhead of one connection overlaps with those of other connections as well as with data transfer on the wire. Thus, when running multiple connections concurrently, SDP delivers high performance using ZCopy for relatively low message sizes. We believe that the model of multiple sockets being active at the same time is applicable to many practical system configurations where multiple applications are run in parallel on the same server, or for multi-threaded applications that perform communication over multiple socket connections in parallel.

## 2.3. Memory registration scheme

As mentioned above (section 1.1), RDMA operations can address only registered memory for both sides of the data transfer. Memory registration involves pin-down and registration with the HCA.

The simplest approach to memory registration involves registering the memory before starting each send/receive operation, and deregistering it once the operation is completed. However, memory registration overhead was identified [7] [8] as a major bottleneck for zero-copy performance in existing processor and I/O device architectures. Memory registration latency measurements are presented in [7]. Caching or batching memory registrations are common techniques to reduce or amortize the registration overhead. Different cache management approaches have been proposed [9]. When such a pin-down/registration cache is used, good bandwidth and CPU utilization is demonstrated. When memory registration is not done explicitly by the application (as is the case for sockets-based applications), such cache design may be complicated by the need to track memory allocations and/or de-allocations, to avoid cache hits on de-allocated memory.

We took a different approach to memory registration, with different strategies for memory pinning and registration with the HCA. Our implementation performs page pinning at least once on each send() and recv() operation. The memory is unpinned once the operation is completed. On the other hand, we avoid the overhead of setting up and tearing down HCA translation tables for each RDMA operation by caching these operations. We further reduce the overhead of the registration with the HCA by using the Fast Memory Region (FMR), similarly to [10].

FMR infrastructure is the fastest interface for memory registration with the HCA. The FMR is a Mellanox feature extending the 1.1 InfiniBand specification. A similar feature was added later on to the 1.2 InfiniBand specification.

The FMR API defines a resource pool of blank memory regions. When a mapping is required, a blank memory region is taken out of the pool and a mapping is applied to it. In the mapping process we take a list of physical pages, combine them together into a virtual space, and create a memory region. This memory region has a virtual address and a key, and is accessible for DMA access (local or remote). Accesses to this memory region are mapped to the physical memory pages that were registered.

When a mapping is no longer required, the memory region is returned back to the pool and can be reused. We also maintain a cache of mapped regions, and perform a cache lookup by physical address each time a mapping is requested. If such a mapping already exists in the pool, the matching memory region is used.

FMR well suites kernel applications. Unlike regular memory registration techniques, FMR does not require

Copyright 2005 IEEE

any handshake (such as interrupts) with the hardware. Memory is being registered directly through direct access to the HCA translation tables. Because of this, FMR has significantly lower overhead of memory registration with the HCA compared with the regular registration.

In kernel, we use the get_user_pages() primitive to lock user pages in physical memory (pinning). Once the memory is locked, it can be made accessible for DMA by the device.

Get_user_pages() locks pages in physical memory, and prevents this physical memory from being reassigned to another process or another virtual address. However it does not guarantee that the virtual address mapping to these pages will not be remapped to another physical page. Thus, for the receive side, extra care must be taken to ensure that once the RDMA operation is complete, the application can access the transferred data through the virtual address.

Memory unpinning is performed by the put_page() primitive. When buffers have been used to receive data, we also need to mark the pages as dirty to ensure that the virtual memory subsystem performs a write-back when a page is swapped out.

## 2.4. RDMA implementation at the data source

For the data source, the decision whether to go through the ZCopy or BCopy path is taken according to the message size and the ZCopy threshold values. There are two other parameters that are taken into consideration: current SDP data transfer mode (ZCopy not allowed if in Buffered mode) and the socket/message flag (ZCopy is not allowed if the operation is non-blocking). When the ZCopy path is selected, the user buffer is locked through the get_user_pages() primitive. The buffer is then registered through an FMR, and a SrcAvail message is sent to the remote peer. SDP then blocks until an RdmaRdCompl message is received. This message indicates that the data sink has completed reading the buffer.

The RdmaRdCompl message is received by a kernel thread, which awakens the blocking process. The FMR is then returned to the FMR resource pool, and the buffer is unpinned by a call to put_page(). The send() call then returns.

Each FMR allocated in the FMR pool contains 32 pages. In other words, a single FMR can map up to 128KB of user buffer memory. Longer messages are broken into smaller chunks and each one of them is pinned and registered separately. Then, depending on the current SDP data transfer mode, an advertisement is generated. If the current data transfer mode is Combined, only a single advertisement is generated. If the current mode is Pipelined, multiple advertisements are generated; the number of advertisements is negotiable at connection establishment time. The calling process then blocks until all SrcAvail advertisements have been consumed and all corresponding RdmaRdCompl messages have been received. If the data source has multiple SrcAvail to advertise, it will typically request transition to the Pipelined mode.

Signal handling is not fully implemented in our prototype. When a process is blocked waiting for an RdmaRdCompl message to arrive, a signal may arrive that requires returning from the send() call. Since an advertisement is outstanding at this time, it must be revoked by sending a SrcAvailCancel message. When this message is acknowledged (at the SDP protocol level), we can safely return from the send() call - having unpinned and deregistered the buffer. If the acknowledgement does not arrive within a reasonable time, the implementation may assume a protocol error and abort the connection.

## 2.5. RDMA implementation at the data sink

The receive path is triggered through two entry points: recv() calls and arrival of SrcAvail messages.

In case of the recv() call, the ZCopy or BCopy path is selected, according to the message size and the ZCopy threshold values. Note that there are other parameters that may bias our decision to perform ZCopy, e.g., if there is already buffered data in SDP buffers. If the ZCopy path is chosen, the buffers are pinned through the get_user_pages() call and an FMR mapping is obtained for these pages. We then check if there is an outstanding SrcAvail. If there is none, the calling process is blocked waiting for SrcAvail. If a SrcAvail is pending, an RDMA Read is issued to get the buffers. Once RDMA Read is completed, an RdmaRdCompl message is sent, the FMR is returned to the FMR pool, the buffer is unpinned, and the recv() call returns.

If recv() is called with a buffer size below ZCopy threshold while there is an outstanding SrcAvail, RDMA Read is performed into kernel buffers and data is copied into user buffers through copy_to_user() call.

Upon arrival of a SrcAvail message while some user buffers are outstanding, RDMA Read operations are performed. Once all RDMA Read operations complete, an RdmaRdCompl message is sent and the process unblocks. The buffers are then deregistered, unpinned, and the recv() call returns. If buffers are not available, the SrcAvail is left to be picked up later on by further recv() calls.

Copyright 2005 IEEE

Since each FMR is restricted to a 128KB block, multiple blocks may be required for a single recv() call. If SrcAvail messages arrive for only a few of these blocks and no RDMA Read operations are outstanding, the untouched blocks are deregistered and the recv() call returns to the user. This is done to satisfy the recv() low watermark condition. We note that this scenario can cause extra pinning, unpinning and registration, which can result in high CPU utilization on large messages when the data source and sink are not synchronized with their send/receive sizes.

A similar case occurs when the receiver is blocked waiting for SrcAvail, but instead BCopy data arrives. The data is then copied from the SDP buffers into the application buffer, the buffers are unpinned, and the FMR is returned to the FMRs pool. Essentially, these buffers were pinned unnecessarily.

The unpinning process on the data sink side is more complicated. Although get_user_pages() locks the buffer in physical memory preventing the operating system from granting this memory to another process, it is not guaranteed that the virtual address space of the process will be permanently mapped to those same physical pages. For example, when a process calls fork(), its virtual space is marked for copy-on-write. A memory write access will then force the accessed page to be copied to another physical address, and the virtual address to be remapped to the new location – even if the physical page is locked.

Once the RDMA Read operation is completed, the data lands in physical buffers that were locked through a call to get_user_pages(). To make sure that the user virtual address has not been remapped, we perform another call to get_user_pages() and compare the two page lists. If the lists are identical, the virtual to physical mapping of the process has been preserved, so that the data is already located in the right pages. If get_user_pages() returns a different page list, copy_to_user() is called to place the data into user buffers. Finally, put_page() is called twice for each page to unlock it. Before unlocking the memory, we call set_page_dirty() for each physical page to indicate to the memory manager that the HCA has performed DMA write operations into these buffers.

### 2.6. Choosing the ZCopy threshold

The ZCopy Threshold is one of the most important parameters that should be tuned in an SDP implementation. Two mechanisms are used to select the threshold value:

- Default threshold – is settable through an SDP module parameter.
- Per connection threshold – a socket option allows setting/reading the ZCopy threshold per socket.

The following considerations have to be taken into account when tuning the ZCopy threshold at the system level: In cases where a single socket is used, setting the ZCopy threshold at the crossover point of the ZCopy/BCopy bandwidth equilibrium will yield the best bandwidth with the optimal CPU utilization; In the case where multiple sockets are used in parallel, ZCopy threshold can be set to a lower value and provide better results in terms of bandwidth and CPU utilization; ZCopy threshold may impact latency, which we haven't analyzed in this paper.

## 3. Benchmarking environment

In this section we focus on the results collected and discuss how a lower ZCopy threshold can perform well at the system level.

### 3.1. Benchmarks

Iperf [11] benchmark version 1.7.0 was used for bandwidth measurements. This utility creates a number of TCP socket connections between two computers: server and client. Messages of fixed size are sent repeatedly through each socket, from client to server. Each socket is operated from a dedicated thread. The TCP bandwidth is calculated as the aggregate number of bytes transferred per second over the sockets. We note that the bandwidth measured is uni-directional.

CPU utilization was measured by running vmstat(1) in parallel with the benchmark. Vmstat reports the CPU idle time in % in 1 second intervals. The CPU utilization is calculated as 100% minus CPU idle time. Since the CPU may be occupied by other system activities, this represents an upper bound on the aggregate CPU utilization by the SDP stack.

The machines we benchmarked are dual CPU with Hyper Threading (HT) enabled, resulting in 4 logical CPUs reported by the OS. The vmstat measurement reports 100% CPU utilization that accounts for 4 fully utilized CPUs. For simplicity, we normalized the results to the actual number of CPUs by multiplying the results by 4. In other words, when our graphs indicate 400%, it means that 4 CPUs are fully utilized, and 100% accounts for a single CPU equivalent. The CPU utilization that we present is averaged over time, and between the send and receive sides.

This study does not include latency measurements. Our goal is to compare SDP ZCopy and BCopy performance. We assume that in practice latency is typically relevant only for small message sizes, which do not cross the ZCopy threshold. In these cases, SDP uses the BCopy mechanism following the original SDP

Copyright 2005 IEEE

implementation BCopy path, without changes. For this reason, our implementation is expected to have the same latency for small messages as the original implementation. SDP latency for BCopy has already been studied in [6] and [12].

### 3.2. Benchmarking platform

Our benchmarking platform included two Dell 1750 servers with a Mellanox InfiniHost host channel adapter installed in a PCI-X 64/133MHz slot on each server, and connected back-to-back through a 4X InfiniBand cable. Each server had two Intel Xeon 3GHz CPUs in an SMP configuration with 0.5MB cache each, and 2GB 266MHz DDR SDRAM. Hyper-Threading (HT) was enabled such that 4 logical CPUs are presented to the operating system (preliminary experiment showed no HT impact on CPU utilization normalized into a 0% to 100% scale). Both HCAs were flashed with 3.3.2 firmware version.

The software stack is based on the Mellanox Gold Collection version 1.7.0 with the additional implementation of ZCopy support for synchronous operations.

## 4. Performance results and analysis

### 4.1. Single connection

The first experiment used a single active socket connection. We measured the bandwidth obtained and the CPU utilization. The CPU utilization numbers below are averaged over time as well as between server and client. We compared ZCopy and BCopy by running the benchmark twice, with ZCopy threshold set to 0 and to infinity respectively. As shown in Figure 6, for small messages up to 256KB, BCopy path provides better bandwidth. Starting at 256KB, the cross-over point, ZCopy achieves better bandwidth than BCopy. The CPU overhead is lower throughout the experiment for ZCopy path. For small messages BCopy CPU utilization is around 100% on average, where for ZCopy it is around 40%. As message sizes increase, ZCopy CPU utilization decreases to as low as 20%.

The drop in BCopy bandwidth for messages longer than 256KB can be explained by cache flushing: copying 256KB requires 512KB of memory, utilizing the entire 512KB CPU cache. We note that this data copy behavior has been observed in other papers, for SDP [6] as well as other protocols. E.g. [8], observes a similar drop for PM 1.0 protocol over Myrinet.

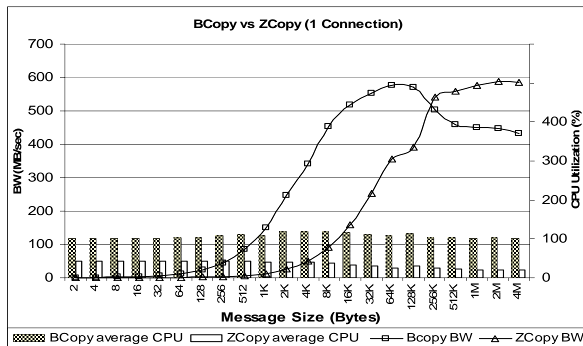


**Figure 6: ZCopy vs BCopy bandwidth and CPU utilization for a single connection**

### 4.2. Multiple simultaneous connections

In the second set of experiments multiple socket connections were run in parallel. We compared BCopy and ZCopy performance. A sample of our results is presented in Figure 7 through Figure 9. The BCopy/ZCopy cross-over point decreases as the number of connections grows: 256KB for 1 connection, 64KB for 2 connections, 32KB for 4 connections and 16KB for 8 connections. This makes ZCopy more compelling as the number of connections increases.

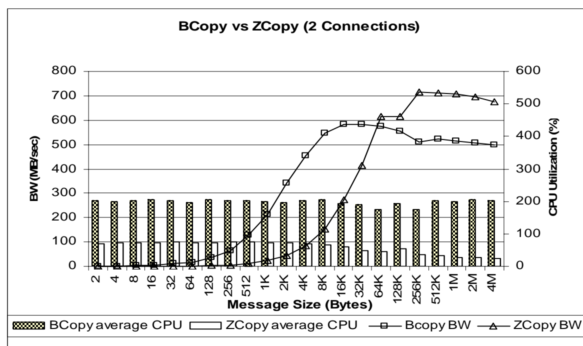


**Figure 7: BCopy vs ZCopy bandwidth and CPU utilization for two simultaneous connections**

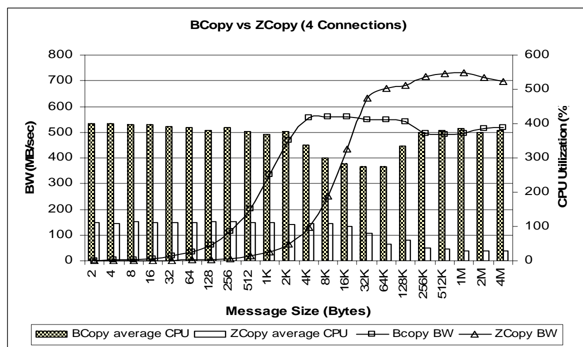


**Figure 8: BCopy vs ZCopy bandwidth and CPU utilization for four simultaneous connections**

Copyright 2005 IEEE

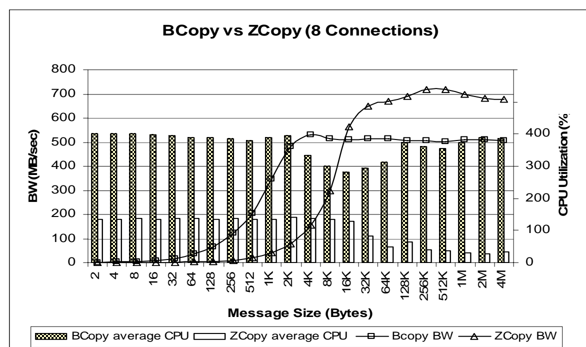


**Figure 9: BCopy vs ZCopy bandwidth and CPU utilization for eight simultaneous connections**

The average CPU utilization is substantially lower with ZCopy than with BCopy. ZCopy CPU utilization is below 50% for messages larger than 128KB, while for BCopy all the CPUs get saturated. The aggregate bandwidth that ZCopy is able to deliver is substantially higher than that of BCopy. When running 8 simultaneous connections, ZCopy sustains more than 700MB/s while BCopy only delivers 500MB/s.

Presenting the same results from a different angle, we compare the bandwidth and the CPU utilization for 64KB and 1MB messages running ZCopy and BCopy as a function of the number of concurrent connections.

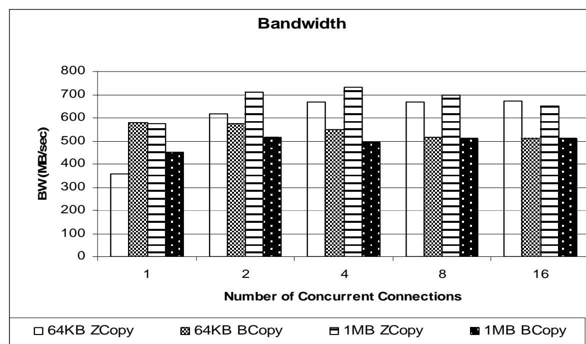


**Figure 10: Bandwidth, variable message size, concurrent connections**

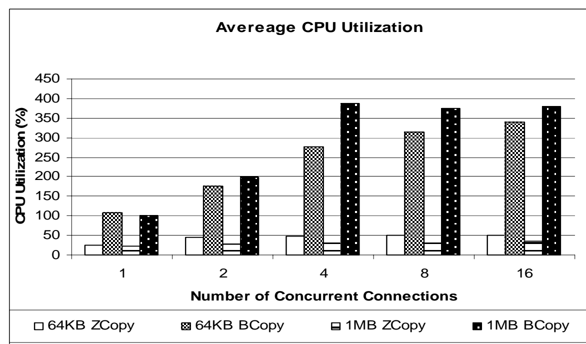


**Figure 11: Average CPU utilization, variable message size, concurrent connections**

Figure 10 and Figure 11 show that for 2 or more connections for 64KB messages, and for any number of connections for 1MB messages, ZCopy achieves superior bandwidth to BCopy while maintaining significantly lower CPU utilization. While BCopy substantially overloads the 4-CPU server (dual CPU with HT enabled), ZCopy achieves the same results utilizing less than 50% of a single CPU!

### 4.3. Per byte overhead

The per-byte overhead is measured in units of usec/KB and is calculated as follows: *Per-byte overhead = CPU utilization / Bandwidth.*
The lower the per-byte overhead, the less CPU is spent for sending a networking message with a certain size. Figure 12 shows the per-byte overhead for variable message size while 1, 2 and 8 connections are active simultaneously. Message sizes between 4 to 16 KB represent a cross-over point where ZCopy usage becomes more compelling than BCopy, as it requires less CPU to deliver the same message. As the number of concurrent connections increases, the per-byte overhead for the BCopy path increases. This turns ZCopy to beneficial at even smaller messages than 4KB.

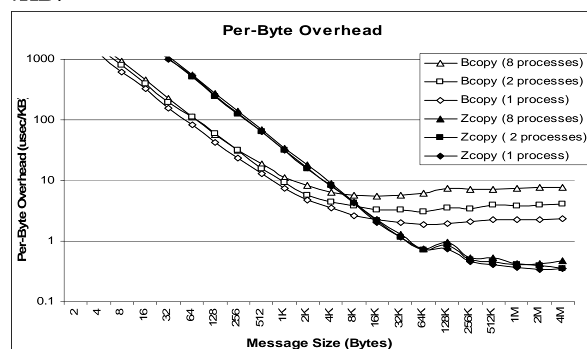


**Figure 12: Per-byte overhead**

## 5. Future work

Multiple enhancement options for improving SDP performance remain to be explored, including: user-land implementation of SDP and comparison with the kernel implementation; full implementation of pipelined mode and usage of Write ZCopy; delayed registration strategies may be applied for the receive flows.

More rigorous analysis is encouraged: detailed analysis of individual, bandwidth rather than aggregate bandwidth and per side CPU utilization, rather than average; more latency and bandwidth scenarios; more CPU and cache configurations including non x86 CPU analysis; ZCopy SDP benchmarking on data center applications.

Copyright 2005 IEEE

SDP ZCopy scalability in multiple node clusters has to be studied. Scalability enhancements need to be considered, such as: using Shared Receive Queues; reducing the number of Completion Queues; tuning the buffer allocation algorithm.

## 6. Conclusion

As network speeds increase, CPU copying becomes expensive unless zero copy techniques are being used. No matter how strong the CPU is, without zero copy it will end up choked by copying at soaring networking speeds.

SDP with the ZCopy path does a great job of increasing the CPU effectiveness for application processing.

SDP allows existing applications to transparently utilize InfiniBand high performance capabilities without any code changes. While existing SDP implementations only allowed ZCopy on AIO path, we demonstrate a portable ZCopy on the standard socket blocking synchronous calls.

We demonstrated an approach where there is no pin-down cache; all memory is pinned and unpinned on each access. Caching was only used for hardware translations.

We demonstrated benefits of utilizing the ZCopy path for socket synchronous calls. For a single connection, the bandwidth cross-over point between BCopy and ZCopy occurs at a relatively large message size. This is caused by the inability of the SDP implementation to pipeline messages because of the synchronous nature of blocking socket calls. Pipelining can be achieved amongst multiple concurrent connections. As the number of connections increases, the cross-over point decreases down to 8KB message size (running a handful of connections) and below. The CPU utilization is also reduced significantly. We believe this makes ZCopy beneficial to a wide range of applications and systems.

System-level ZCopy threshold tuning for general multi-tasking applications (rather than a single-connection benchmark) achieves the best overall system performance (aggregate bandwidth and CPU utilization) without need for application-specific tuning.

## 7. References


[1] The InfiniBand Architecture Specification, Volume 1, Release 1.2 - www.infinibandta.org/specs

[2] D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An Analysis of TCP Processing Overheads", *IEEE Communication Magazine*, Vol. 27, No. 2, June 1989, pp. 23 – 29.

[3] J.P.G. Sterbenz, and G. M. Parulkar, "Axon: A High-Speed Communication Architecture for Distributed Applications", *in proc. of the 9th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'90)*, Vol. II, June 1990, pp. 415–425.

[4] Open InfiniBand Alliance – www.openib.org

[5] Mellanox InfiniBand Gold Collection – www.mellanox.com

[6] P. Balaji, S. Narravula, K. Vaidyanathan, S. Krishnamoorthy, J. Wu, and D. K. Panda, "Sockets Direct Procotol over InfiniBand in Clusters: Is it Beneficial?", *in proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 04)*, Mar. 2004, pp. 28-35.

[7] V. Tipparaju, G. Santhanaraman, J. Nieplocha, and D. K. Panda, "Host-Assisted Zero-Copy Remote Memory Access Communication on InfiniBand", *in proc. of the 18th International Parallel and Distributed Processing Symposium (IPDPS 04)*, Apr. 2004.

[8] H. Tezuka, F. O'Carroll, A. Hori, and Yutaka Ishikawa, "Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication", *in proc. of the 12th International Parallel Processing Symposium*, Mar. 1998, pp. 308-314.

[9] C. Bell, and D. Bonachea, "A New DMA Registration Strategy for Pinning-Based High Performance Networks", *in proc. of the Workshop on Communication Architecture for Clusters (CAC'03) of IPDPS 03*, Apr. 2003, p. 198.1.

[10] J. Wu, P. Wyckoff, and D. K. Panda, "PVFS over InfiniBand: Design and Performance Evaluation", *in proc. of the 32nd International Conference on Parallel Processing (ICPP 03)*, Oct. 2003, pp. 125-132.

[11] Iperf – dast.nlanr.net/Projects/Iperf/

[12] S. Narravula, P. Balaji, K. Vaidyanathan, S. Krishnamoorthy, J. Wu, and D. K. Panda, "Supporting Strong Coherency for Active Caches in Multi-Tier Data-Centers over InfiniBand", *in proc. of the 3rd Annual Workshop on System Area Networks (SAN 03) in conjunction with HPCA 10*, Feb. 2004.


Copyright 2005 IEEE