# Transparently Achieving Superior Socket Performance Using Zero Copy Socket Direct Protocol over 20Gb/s InfiniBand Links

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 protocol that utilizes the capabilities of the InfiniBand fabric to transparently achieve performance gains for existing socket-based networked applications. In this paper we discuss an implementation of Zero Copy support for synchronous send()/recv() socket calls, that uses the remote DMA capability of InfiniBand for SDP data transfers. We added this support to the open-source implementation of SDP over InfiniBand. We evaluate this implementation over a 20 Gb/s InfiniBand link. We demonstrate scalability of Zero Copy and show its benefits for systems that utilize multiple socket connections in parallel. For example, enabling Zero Copy with 8 active connections yields a bandwidth growth from 630MB/s to 1360MB/s, at the same time reducing the CPU utilization by a factor of ten.*

## 1. Introduction

The InfiniBand architecture, defined by the InfiniBand Trade Association [1], introduces a high bandwidth, low latency interconnect with RDMA capabilities, supporting link speeds up to 120 gigabit per second. Upper Layer Protocols (ULPs) have been developed, providing standard interfaces to existing OS frameworks and applications on top of InfiniBand. These ULPs include user-mode middleware and kernel protocol drivers, such as Socket Direct Protocol (SDP).

The initial InfiniBand open-source SDP implementation ([3], [4]) included only support for Buffer Copy (BCopy) mode. We have extended the SDP implementation, and added zero copy (ZCopy) support on the synchronous I/O path [5]. The implementation enables us to benchmark ZCopy performance.

Mellanox has recently implemented support for InfiniBand Double Data Rate (DDR) link speed in host adapters and switches. InfiniBand DDR increases the bandwidth of a 4x link from 10Gb/s to 20Gb/s. We present benchmarks of SDP performance over an InfiniBand DDR link. We demonstrate that our ZCopy implementation scales well to these new link rates, making it possible for applications to transfer data at close to full wire speed.

### 1.1. Introduction to InfiniBand

InfiniBand is a standard switched interconnect fabric with high bandwidth and low latency. The architecture defines a System Area Network that connects processor nodes and I/O devices. Processor nodes and I/O devices connect to the fabric through Host Channel Adapters (HCA) and Target Channel Adapters (TCA) respectively.

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

- Zero Copy – enables direct data exchange between application buffers without an intermediate copy
- Transport offload - includes segmentation, reassembly, retransmission, transport checks, timers, etc.
- Kernel Bypass – enables direct access from user mode applications to the HCA hardware

Zero Copy is achieved through RDMA operations. To perform such an operation, the consumer must perform memory registration which involves pin-down and registration with the HCA (see section 2.3, Memory Registration Scheme).

The InfiniBand specification defines an HCA interface called Verbs [2]. The Verbs interface provides operations for resource management and data transfer operations. The communication is based on Queue Pairs (QPs) and Completion Queues (CQs). InfiniBand supports Send/Receive, RDMA Read, RDMA Write and Atomic 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. Further information about the

InfiniBand architecture can be found in [2].

### 1.2. InfiniBand Double Data Rate

The InfiniBand Architecture specification rev 1.2 [2] extends the physical lane speeds from 2.5Gb/s to 5Gb/s (DDR) and 10Gb/s (QDR). InfiniBand Double Data Rate (DDR) technology increases the bandwidth and reduces the latency for the upper layer protocols. InfiniBand DDR uses the same cabling as single data rate InfiniBand, and is transparent to the software.

Mellanox has recently introduced InfiniBand DDR support on both HCAs and switches. The supported link speeds are 20Gb/s and 60Gb/s for 4x HCAs and 12x switches, respectively. In this paper, we present benchmarks of our SDP implementation on InfiniBand DDR enabled HCAs.

### 1.3. Sockets Direct Protocol (SDP)

Traditional implementations of TCP sockets typically require data copy between application buffers and NIC kernel buffers, segmentation, reassembly and transport handling. As network speed increases, these operations consume substantial CPU resources and memory bandwidth, and become a performance bottleneck. Sockets Direct Protocol (SDP), which was added as an annex to the InfiniBand Architecture Specification [2] on April 2002, eliminates this bottleneck.

SDP is a byte stream protocol that mimics TCP SOCK_STREAM semantics. Existing socket-based applications can use the SDP protocol without any code modification or recompilation. 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. The major software stack components are illustrated in Figure 1.

SDP is implemented on top of InfiniBand Reliable Connection (RC) transport service, and leverages the transport offload of the HCA. It maps the socket send()/recv() calls to InfiniBand operations. Control messages are transferred by SEND messages. Data messages are transferred through one of the two SDP 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. The BCopy flow is illustrated in Figure 2. Note that in this particular example the receiver had been waiting for data before data arrived. In case where the receiver is slow, the data will wait in the SDP buffers until the receiver calls recv(). A flow control mechanism makes it possible for the receiver to limit the number of outstanding SDP buffers.

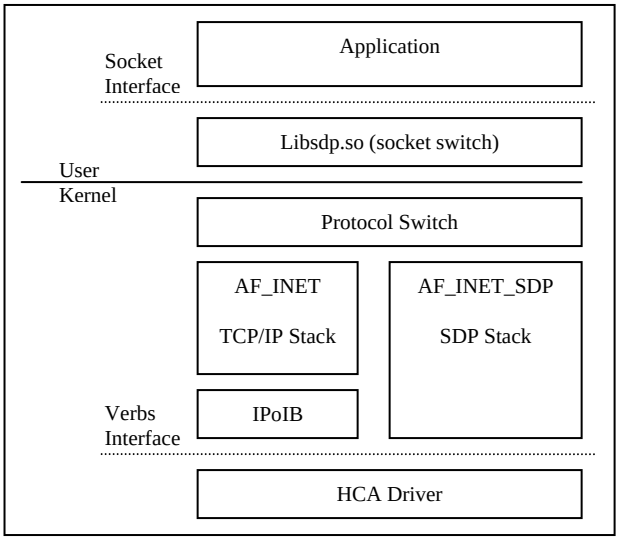


**Figure 1: IBGold SDP Stack Components**

When using ZCopy, the data is transferred directly 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. There are two modes of ZCopy operation: Read ZCopy and Write ZCopy.

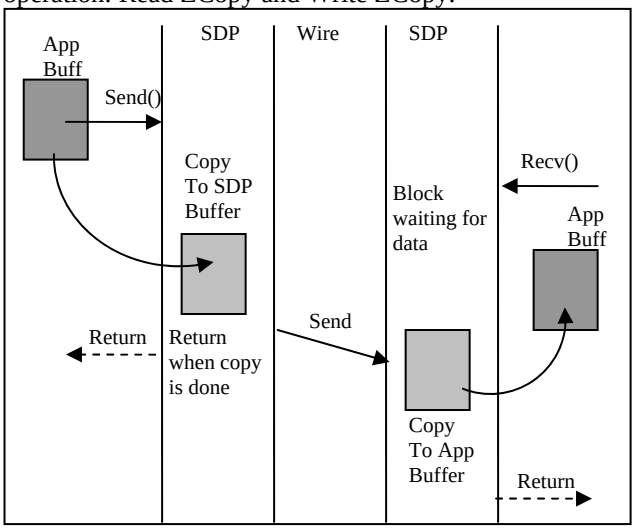


**Figure 2: BCopy Flow**

A Read ZCopy flow is illustrated in Figure 3. 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.

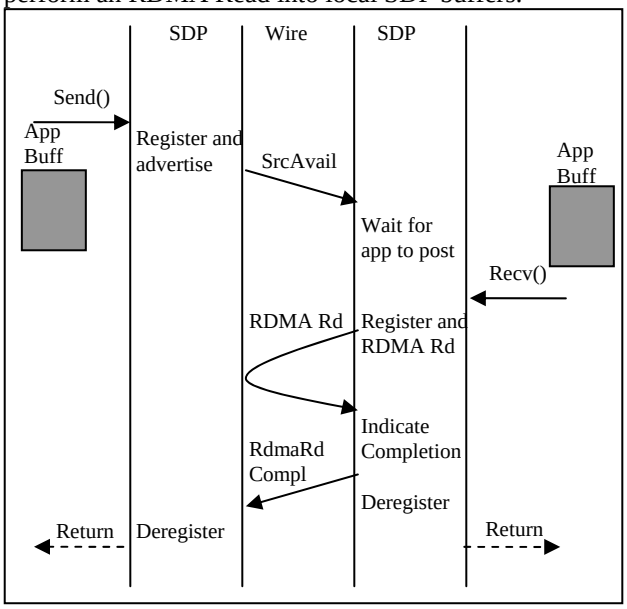


**Figure 3: Read ZCopy Flow**

Similarly, 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. The major differences between Read and Write ZCopy are the following:

- Write ZCopy is only allowed in Pipelined data delivery mode, while Read ZCopy is allowed in both Pipelined and Combined modes
- Read ZCopy advertisements must be completely consumed by the data sink, while Write ZCopy advertisements may be partially fulfilled by the data source. Therefore, depending on the recv() flags and the API semantics, Write ZCopy may not be suitable for pipelining in certain scenarios
- Certain HW may have different performance characteristics when performing RDMA write compared with RDMA read

Both BCopy and ZCopy modes incur some communication overhead. BCopy mode incurs the overhead of a local data copy. On the other hand, ZCopy mode incurs the overhead of locking the application buffers in physical memory, registering them with the HCA, and additional communication overhead associated with buffer advertisement. Thus, the SDP implementation includes a ZCopy threshold 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 (sink or source) can initiate a mode transition.

### 1.4. InfiniBand Gold (IBGold) Collection

The IBGold Collection [4] is an open-source Verbs and ULP stack implementation for the Linux operating system. It is based on the initial Linux open source code that was posted on the OpenIB Alliance web site [3], and has been further enhanced and stabilized by Mellanox. Originally, the IBGold Collection SDP did not utilize ZCopy for synchronous send()/recv() operations: ZCopy was exclusively used for asynchronous I/O (AIO), and only supported certain kernel versions.

SDP is implemented as a kernel module, registered with the kernel as a new protocol family (AF_INET_SDP), as illustrated in Figure 1. The socket switch is implemented as a user-mode shared library (libsdp). When preloaded, this library intercepts the application socket calls. For each socket, 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. AF_INET_SDP objects are created by the SDP kernel module. The module implements all socket entry points that are invoked by the kernel: create, release, bind, connect, listen, 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].

### 1.5. Socket APIs

Certain socket APIs have evolved over the years to facilitate better I/O performance. The original BSD/POSIX send()/recv() API [7], which to our

knowledge has the largest application base in Linux, only enables synchronous operation. Further extensions to the API include the Asynchronous I/O (AIO) [7] and Linux specific extensions such as io_submit(2), which makes it possible to submit I/O operations that complete asynchronously. Recently, the Open Group has released the Extended Socket API [8], which makes memory registration explicit to the application and enables asynchronous mode of operation.

The asynchronous APIs map to RDMA operations better than the synchronous calls. In order to benefit from the new API features, applications have to be rewritten, with all the implications of the new semantics. For some systems this can significantly improve I/O performance, for other systems such modifications are unacceptable or not applicable.

The original open source implementation supports AIO for certain Linux kernels. In those kernels, memory pinning is taken care of by the AIO subsystem. The asynchronous semantics of the API simplifies the underlying implementation, and also enabled pipelining of RDMA operations on the same socket. Registration is performed by the SDP module.

The ES-API is the most RDMA friendly API, as it requires the application to explicitly register memory with the socket provider. The API also allows asynchronous transfers. It can significantly simplify the SDP implementation and improve the performance. However, the ability to fully utilize the new functionality depends on the application.

This paper focuses on an implementation of ZCopy for the traditional synchronous socket calls. The main value of this implementation is its transparency to the application: existing applications can benefit from the underlying hardware capabilities, without code modification or recompilation. The actual implementation of the SDP provider is more complicated compared to asynchronous APIs, as the API is not RDMA friendly.

# 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 [3]. 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, which is a totally separate path.

ZCopy is only used for blocking synchronous socket calls: the non-blocking synchronous calls (MSG_DONTWAIT flag or O_NONBLOCK option) use BCopy mode in order to preserve their semantics.

## 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, the send() call can return as soon as the data is copied into the kernel SDP buffer. Therefore, the user may submit multiple send() operations to the SDP layer while some of the previous send() operations are in flight (see Figure 2: BCopy Flow). This pipelining capability makes it possible to sustain high bandwidth over a single connection. Nevertheless, the CPU utilization will be high due to data copying.

On the other hand, when ZCopy is used on synchronous socket calls, pipelining is not possible. The user buffer on the send side has to be kept pinned and registered until the RDMA operation completes. Since data transfer is somewhat deferred, the send() call 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 will be able to send the next message. Thus, regardless of the ZCopy flavor chosen, pipelining is disallowed in 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 small to medium message sizes.

Each SDP transaction includes some protocol overhead associated with the buffer advertisement (see Figure 4). The larger the message size is, the smaller the overhead part of the transaction becomes, and therefore the better the bandwidth that can be obtained.

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

It is worth pointing out that using asynchronous socket API such as AIO or ES-API, the application can submit

multiple send operations to the same socket. The underlying SDP implementation can then pipeline the requests on the socket and advertise more than one buffer at each time. For some application, this can improve the performance of a single connection. We expect to obtain similar performance for the following two scenarios: 1) an application that has n sockets and uses synchronous calls and 2) an application that has one socket and uses n outstanding I/O submissions on this socket. However, some applications may be unable to submit more than one message at a time on a single socket.

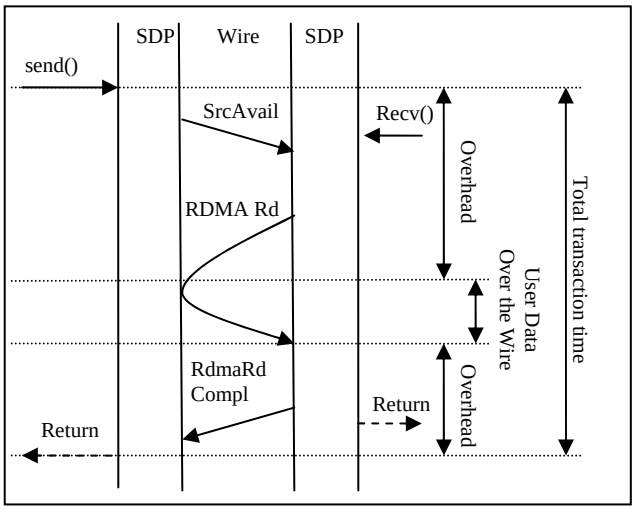


**Figure 4: Read ZCopy Effective Wire Time**

### 2.3. Memory Registration Scheme

Making applications virtual memory accessible for RDMA requires:

- Pin-Down – locking RDMA buffers in physical memory, accessible by the HCA
- Registration with the HCA – configuration of the HCA translation tables and granting access permissions to the RDMA buffers

The simplest approach involves registering the memory before starting each send/receive operation, and deregistering it once the operation is completed. However, memory registration overhead was identified as a major bottleneck for Zero Copy performance, e.g. [9], [10].

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 the registration. We further reduce this overhead by using the Fast Memory Region (FMR), similarly to the approach suggested by [11]. Using FMR, the registration with the HCA is divided into two steps: allocating the memory region in the HCA, and mapping it for RDMA. Allocated FMRs may be cached and in this way the allocation step is kept out of the data path.

We use the get_user_pages() primitive to lock user pages in physical memory. Once the memory is locked, it can be made accessible for DMA by the HCA. Memory unpinning is performed by the put_page() primitive. If buffers have been used to receive data, it is necessary to mark their pages as dirty to ensure that the virtual memory subsystem performs a write-back when a page is swapped out.

We use FMR to implement registration with the hardware. FMR infrastructure is the fastest interface for memory registration with the HCA. 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 FMR 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 addresses each time a mapping is requested. If such a mapping already exists in the pool, the matching memory region is used.

Unlike regular memory registration techniques, FMR mapping does not require any handshake with the hardware. Memory is registered directly through direct access to the HCA translation tables. FMR well-suits kernel applications and reduces the overhead of memory registration with the HCA.

### 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: the current SDP data transfer mode (ZCopy is not allowed in Buffered mode) and the socket options/message flags (ZCopy is not used if the operation is non-blocking). For simplicity, we chose to implement only the Read ZCopy flow for the synchronous socket operations.

When the ZCopy path is selected, the user buffer is pinned. It 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, the buffer is unpinned and the send() call returns.

Each FMR allocated in the FMR pool 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 SrcAvails to advertise, it will typically request transition to the Pipelined mode.

While a process is blocked waiting for an RdmaRdCompl message to arrive, a signal may arrive that requires returning from the send() call immediately. 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 safely return from the send() call - having unpinned and deregistered the buffer. If the acknowledgement does not arrive within a reasonable time, we treat that as a protocol error and abort the connection.

An interesting aspect of ZCopy SDP implementation is the ability to emulate socket buffering. For example, certain applications send messages, smaller than the socket buffer size, simultaneously on both half connections of a socket. With BCopy and traditional TCP socket each half connection will buffer the request, and the send() will be unblocked. With ZCopy, implementations may deadlock: both half connections advertise data source buffers, block the send() call and wait for the remote side to consume the buffer. When an SDP implementation detects this condition, it can avoid the deadlock by: canceling its buffer advertisement, read the remote buffer into a local SDP buffer or force the remote side to send the data using BCopy. In this way SDP preserves the socket semantics even when both halves of the connection are sending data simultaneously.

## 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 in a similar fashion to the data source. 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 and an FMR mapping is obtained. 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 data. 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 the SDP pre registered local buffers and data is copied then into user buffers.

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.

Since each FMR is restricted to a 128Kbyte 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 possible optimization strategy may include deferring the memory pinning and registration until SrcAvail is observed at the data sink.

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 same solution, suggested in the previous paragraph, applies here too.

## 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 – controls the default value assigned for each socket that is created
- Per connection threshold – a socket option allows setting the ZCopy threshold per socket

A performance-tuned individual application would rather set the ZCopy threshold value at the crossover point of ZCopy and BCopy bandwidth equilibrium. We believe that ZCopy threshold should be set at the system

level. While this may result in lower bandwidth for a single socket connection, it is possible to obtain a higher aggregate bandwidth along with minimal CPU overhead for multiple connections. In the next section we focus on the results collected and further discuss how a lower ZCopy threshold can be beneficial at the system level.

# 3. Benchmarking Environment

## 3.1. Benchmarks

Iperf [12] benchmark version 1.7.0 was used for bandwidth measurements. This utility creates a number of TCP socket connections between two nodes: server and client. Messages of fixed size are then sent repeatedly through each socket, from client to server (unidirectional). 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. CPU utilization was measured by running the standard vmstat(1) utility 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.

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% of CPU utilization, it means that 4 logical CPUs are fully utilized. 100% accounts for a single CPU equivalent. The CPU utilization that we present is the average of the CPU utilization over time, and between the send and receive sides.

SDP latency for BCopy has been studied in [6] and [14]. We measured the impact of the ZCopy addition on the latency. For latency benchmarking we used Netperf 2.4.0 [13] transaction test (TCP_RR). This test performs request-response exchanges of fixed sizes over a single socket and measures the transaction rate. We calculated the one way latency as 0.5/transaction-rate. The CPU utilization numbers represent the average between the values reported by Netperf on local and remote sides.

## 3.2. Benchmarking Platform

Our benchmarking platform includes two Intel SE7520JR2 servers, each one with a Mellanox MemFree InfiniHost III Ex HCA installed in a PCI Express x8 slot. The servers are connected back-to-back through a 4X InfiniBand link running at double data rate (20Gb/s).

Each server has two EM64T Intel Xeon 3.2GHz CPUs in an SMP configuration with 1MB cache each, and 1.5Gbyte 266MHz DDR SDRAM. Hyper-Threading (HT) is enabled, such that 4 logical CPUs are presented to the operating system. The servers are based on the Intel E7520 chipset. Both HCAs are flashed with the 5.1.0 firmware version. The software stack is based on the Mellanox IBGold Collection version 1.7.0 (VAPI 4.0.3) with the additional implementation of ZCopy support for synchronous operations.

# 4. Performance Results and Analysis

## 4.1. Single Connection Bandwidth

The first experiment used a single active socket connection. We measured the bandwidth obtained and the CPU utilization.

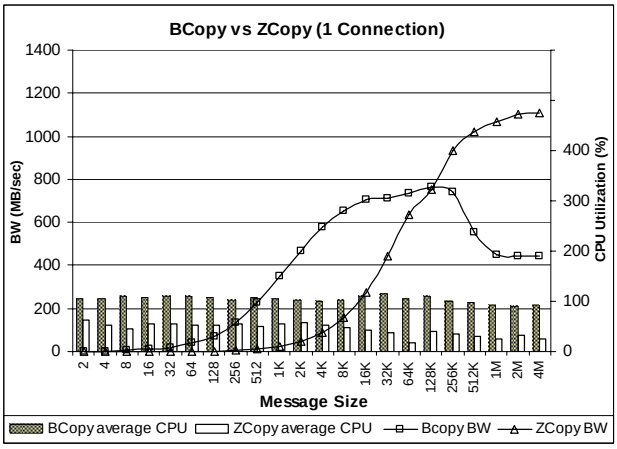


**Figure 5: BW and CPU Utilization for a Single Connection**

We compared ZCopy and BCopy by running the benchmark twice, with ZCopy threshold set to 0 and to infinity respectively. As shown in Figure 5, for small messages up to 128KB, the BCopy path achieves better bandwidth than ZCopy. Starting at 128KB, the crossover point, ZCopy achieves better bandwidth than BCopy. The CPU overhead is lower throughout the experiment for the ZCopy path. For small messages BCopy CPU utilization is around 100% on average, whereas for ZCopy it is around 50%. As message sizes increase, ZCopy CPU utilization decreases to values as low as 25%. Also note that the ZCopy bandwidth, even for the single socket scenario, scales beyond 4x single data rate bandwidth (presented in [5]) and reaches the 1100MB/s maximum bandwidth.

BCopy bandwidth drop at messages larger than 512KB

appears to be related to CPU data cache trashing. This hypothesis is supported by the following evidence: in this experiment the CPU cache size is 1MB; while copying 512KB, the cache is entirely occupied by the source and destination data. Going to 512KB and beyond flushes the cache and therefore degrades performance. On CPU architectures with smaller cache size, the bandwidth drop is observed at smaller message sizes (e.g. in [5] we observe bandwidth drop at message size 256KB in a system with 512KB CPU cache).

### 4.2. Multiple Simultaneous Connections BW

In the second set of experiments, we ran multiple socket connections in parallel. A sample of our results is presented in Figure 6 and Figure 7. The BCopy/ZCopy crossover point decreases as the number of connections grows: 128KB for 1 connection, 32KB for 2 connections, 16KB for 4 connections, etc. That makes the use of ZCopy more compelling as the number of connections increases.

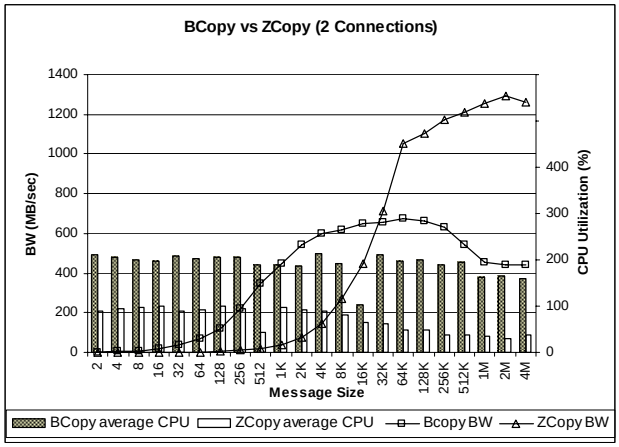


**Figure 6: BW and CPU Utilization for Two Connections**

The average CPU utilization is substantially lower with ZCopy than with BCopy. ZCopy CPU utilization is below 50% for a single CPU at 256KB messages and longer. In BCopy mode, all 4 CPUs get saturated to achieve maximum bandwidth.

The aggregate bandwidth that ZCopy is able to deliver is substantially higher than that of BCopy. When running 8 simultaneous connections, ZCopy sustains a bandwidth higher than 1360MB/s while BCopy delivers at most 630MB/s of sustained bandwidth. We note that ZCopy scales well as the number of connections and the network speed increase. ZCopy obtains close to native system bandwidth. BCopy, on the other hand, is limited in scalability and is CPU bound.

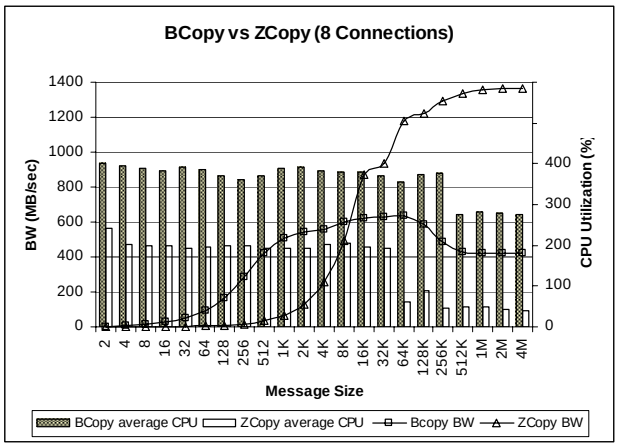


**Figure 7: BW and CPU Utilization for Eight Connections**

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.

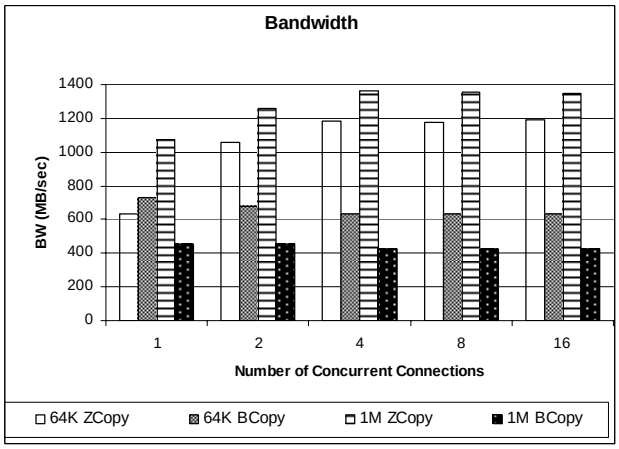


**Figure 8: Bandwidth for 64KB and 1MB Messages with Variable Concurrent Connections**

Figure 8 and Figure 9 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 CPUs, ZCopy triples the bandwidth utilizing 50% of a single CPU!

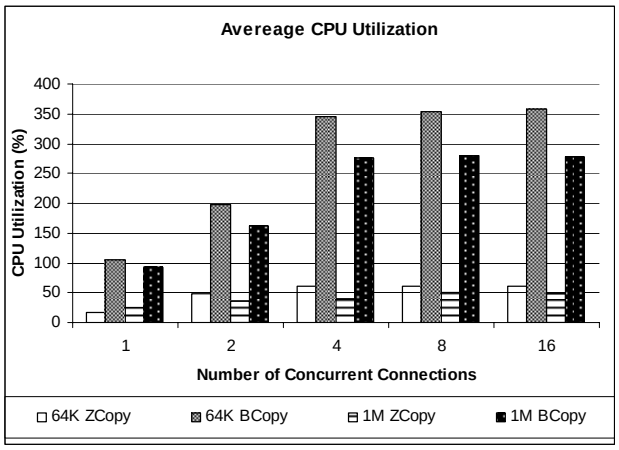


**Figure 9: Average CPU Utilization for 64KB and 1MB Messages with Variable Concurrent Connections**

### 4.3. Latency Impact

The addition of Zero copy to SDP raises the concern of whether an additional latency is incurred, especially for small messages where BCopy is typically being used. We benchmark the latency running SDP with and without the new ZCopy code (see Figure 10). We demonstrate that at small messages, there is only a negligible difference in latency (the lines mostly overlap in the graph). This is because the only change made to the BCopy path is the addition of an extra if statement to determine whether to use ZCopy path or not. ZCopy, for small messages, is wasteful as it delivers worse bandwidth and consumes more CPU resources. This is because of the protocol overhead involved (pinning, registration and handshake).

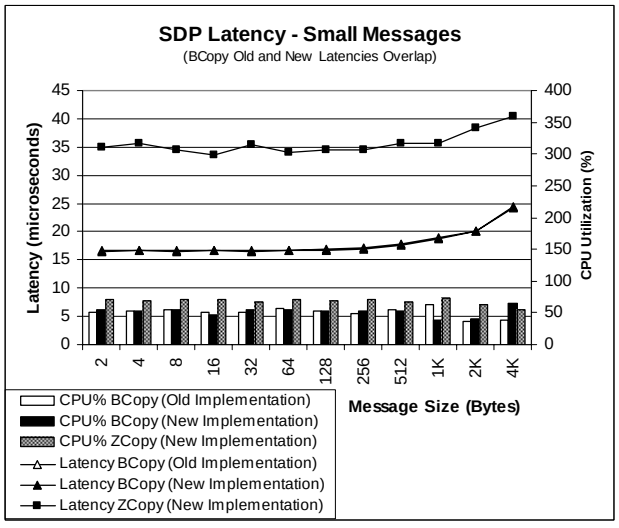


**Figure 10: SDP Latency (Small Messages)**

As we move to larger messages (see Figure 11), ZCopy becomes more compelling by being able to deliver better performance. ZCopy latency decreases, because of the increase in bandwidth and the CPU utilization decreases due to zero copy. The BCopy results for both implementations remain the same (the lines mostly overlap in the graph).

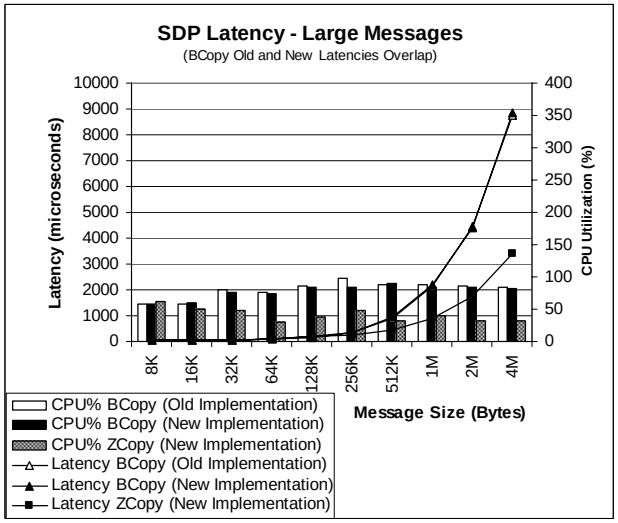


**Figure 11: SDP Latency (Large Messages)**

Latency analysis shows that the new ZCopy implementation only has a negligible impact on the BCopy path. It also shows that ZCopy can improve latency for large messages by providing both better latency and reduced CPU 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 for the receive flows; analysis of the effect of mixture or random message sizes on SDP performance.

It is desirable to benchmark ZCopy SDP on data center applications and characterize the performance benefits. The ZCopy threshold has to be tuned at the system level in these real-life workloads to obtain the best performance.

Scalability enhancements need to be considered, such as using Shared Receive Queues, reducing the number of Completion Queues, and tuning the buffer allocation algorithm.

As new socket APIs evolve, it is desirable to implement and benchmark the various APIs, better quantify their impact on performance, port certain applications to the new APIs and understand the applicability and complexity involved in such a rewrite.

## 6. Conclusion

Sockets Direct Protocol (SDP) implementation on Linux provides a way for applications to use InfiniBand capabilities without a need for source code modifications. While new APIs may serve to improve I/O performance, modifying applications may be complicated or expensive. This motivated us to support Zero Copy in commonly used synchronous socket operation, making it possible for such applications to transparently benefit from the RDMA capabilities of the InfiniBand fabric.

We implemented Zero Copy (ZCopy) mode for the synchronous socket calls in SDP and demonstrated the performance benefits. ZCopy eliminates the CPU buffer copy bottleneck and thus provides better bandwidth with very low CPU utilization. SDP ZCopy performance also scales as network speed increases, as opposed to BCopy performance, which is CPU bound. For InfiniBand 4x DDR link, ZCopy SDP reaches data transfer rates up to 1360MB/s, which is close to native InfiniBand system performance, while BCopy peaks at 630MB/s.

The addition of ZCopy path to SDP doesn't incur any noticeable latency overhead for small messages. For large messages, latency is actually improved due to the higher bandwidth and lower CPU utilization.

We demonstrate that, especially for applications that utilize multiple socket connections in parallel, enabling ZCopy improves performance even for fairly small message sizes. For a single connection, the bandwidth crossover point between BCopy and ZCopy occurs at a fairly 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 crossover point decreases below 16KB message size (running just a handful of connections). The CPU utilization is also reduced significantly. 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 the need for application-specific tuning.

## References


[1] The InfiniBand Trade Association – www.infinibandta.org

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

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

[4] Mellanox InfiniBand Gold Collection (IBGold) – www.mellanox.com

[5] D. Goldenberg, M. Kagan, R. Ravid, M. S. Tsirkin, "Zero Copy Sockets Direct Protocol over InfiniBand – Preliminary Implementation and Performance Analysis", *in proc. of the 13th Annual IEEE Symposium on High Performance Interconnects (Hot Interconnects 13)*, Aug. 2005.

[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] IEEE Std 1003.1, 2004 Edition

[8] Extended Socket API (ES-API), Issue 1.0 – www.opengroup.org/icsc

[9] 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.

[10] H. Tezuka, F. O'Carroll, A. Hori and Y. Ishikawa, "Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication", *in proc. of the 12th International Parallel Processing Symposium (IPPS/SPDP 98)*, Mar. 1998, pp. 308-314.

[11] 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.

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

[13] Netperf – www.netperf.org

[14] 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.