A GENERAL-PURPOSE API FOR IWARP AND INFINIBAND

Robert D. Russell
University of New Hampshire InterOperability Laboratory
121 Technology Drive, Suite 2
Durham, NH 03824-4716
rdr@iol.unh.edu
(603) 862-3774

ABSTRACT

Remote Direct Memory Access (RDMA) allows data to be transferred over a network directly from the memory of one computer to the memory of another computer without CPU intervention. There are two major types of RDMA hardware on the market today: InfiniBand, and RDMA over IP, also known as iWARP. This hardware is supported by open software that was developed by the OpenFabrics Alliance (OFA) and that is known as the OpenFabrics Enterprise Distribution (OFED) stack. This stack provides a common interface to both types of RDMA hardware, but does not itself provide a general-purpose API that would be convenient to most network programmers. Rather, it supplies the tools by which such APIs can be constructed.

The Extended Sockets API (ES-API) is a specification published by the Open Group that defines extensions to the traditional socket API which include two major new features necessary to exploit the advantages of RDMA hardware and the OFED stack: asynchronous I/O and memory registration.

The UNH-EXS interface is a multi-threaded implementation of the ES-API plus additional extensions, which enables programmers to utilize RDMA hardware via the OFED stack in a convenient, relatively familiar manner. The UNH-EXS interface is implemented entirely in user space on the Linux operating system. This provides easy porting, modification and adoption of UNH-EXS, since it requires no changes to existing Linux kernels. We present results on the performance of some benchmark applications using the UNH-EXS interface on both iWARP and InfiniBand hardware.

KEY WORDS

iWARP, InfiniBand, Remote Direct Memory Access (RDMA), Extended Sockets (EXS).

1 Introduction

1.1 Remote Direct Memory Access

Remote Direct Memory Access (RDMA) is a technology that allows data to be transferred over a network directly from the memory of one computer to the memory of another without CPU intervention. There are two major types of RDMA hardware available today: InfiniBand [1], and RDMA over IP [2], also known as iWARP (*Internet Wide-Area RDMA Protocol*) [3, 4, 5]. In this paper, unless otherwise stated, the term RDMA applies to both.

The 10 Gigabit/second Ethernet (10GigEthernet) standard allows IP networks to achieve high transfer rates; however, without offloading onto specialized *Network Interface Cards* (NICs), TCP/IP is not sufficient to make use of the entire 10GigEthernet bandwidth. This is due to data copying, packet processing and interrupt handling on the CPUs at each end of a TCP/IP connection. In a traditional TCP/IP network stack, an interrupt occurs for every packet sent and received, data is copied at least once in each host computer's memory (between user space and the kernel's TCP/IP buffers), and the CPU is responsible for processing multiple nested packet headers for all protocol levels in all incoming and outgoing packets.

One type of RDMA hardware that can be used to eliminate these inefficiencies is iWARP. The entire iWARP, TCP and IP protocol suites are offloaded from the CPU onto the *RDMA Network Interface Card* (RNIC). This enables applications utilizing RNICs to have lower CPU usage than those utilizing standard NICs, and to achieve throughput close to the full capacity of 10GigEthernet.

1.2 OFED Stack

The *OpenFabrics Alliance* (OFA) [6] provides open-source software to interface with RDMA hardware of

both types. This software has an active development community and is now included in Linux kernel distributions. It provides an abstraction layer called the *Communication Manager Abstraction* (CMA), along with a verbs layer that is used to perform data transfers over RDMA hardware. This collection of software is referred to as the *OpenFabrics Enterprise Distribution* (OFED) stack [7]. This software runs in both user and kernel space, and on both the Linux and Windows operating systems.

This software stack is intentionally not designed as a conventional API. Rather, it is a set of functions, called "verbs", and data structures that can be utilized to build various APIs. Several versions already exist for the MPI API, for example [8, 9]. These verbs and data structures are not oriented toward user applications, because they require knowledge of the internal workings of RDMA hardware and its interactions with the OFED stack. This form of programming is very unfamiliar to most programmers and is not at all similar to the conventional sockets programming used by many network programs. It therefore seems desirable to provide a more familiar, general-purpose API on top of the OFED verbs layer that would make it easier to write general network programs that could nevertheless take advantage of the features provided by the OFED stack and RDMA hardware. This general-purpose API would coexist with more specialized APIs, such as MPI.

1.3 The ES-API

The *Extended Sockets API* (ES-API) [10] is a specification published by the Open Group [11] that defines extensions to the traditional socket API in order to provide improved efficiency in network programming. It contains two major new features: asynchronous I/O and memory registration. These extensions can be a useful method for efficient, high-level access to the OFED stack and RDMA.

The UNH-EXS interface is a multi-threaded implementation of the ES-API that also provides some additional API facilities.

2 Design of UNH-EXS

Part of the OFED stack is a user-space library that allows user-space programs to use its verbs and data structures to interface directly with RDMA hardware. We decided to implement UNH-EXS entirely in user-space as a thin layer between application programs and the OFED verbs, as shown in Figure 1. Working entirely in user space rather than kernel space makes it much easier to implement and debug software, and it increases the portability of the resulting code. However, this introduces a significant difference between the UNH-EXS interface and the ES-API standard [10], because the ES-API was intended to be integrated into the operating system kernel. Thus the design of ES-API anticipates modifications to the existing kernel I/O interface in order to reuse a large number of existing OS functions for RDMA access. Some examples are – socket, bind, listen, close, getsockname, and getpeername.

<table>
<tr><td>user space</td><td colspan="2">user application program
UNH EXS library
OFED user library</td></tr>
<tr><td rowspan="2">kernel space</td><td colspan="2">OFED kernel modules</td></tr>
<tr><td>iWARP driver</td><td>IB driver</td></tr>
<tr><td>hardware</td><td>iWARP RNIC
10Gig Ethernet</td><td>InfiniBand HCA
InfiniBand Fabric</td></tr>
</table>

Figure 1. Layering of EXS, OFED, iWARP and InfiniBand

The ES-API also supplies many new functions to be used in place of or in addition to existing socket functions. This is necessary when the number or types of parameters of existing functions have to be changed to accommodate the expanded requirements of asynchronous operation and registered memory. Some examples are exs_connect, exs_accept, exs_send, and exs_recv.

standard sockets	UNH-EXS	origin
accept	exs_accept	ES-API standard
bind	exs_bind	UNH-IOL addition
close	exs_close	UNH-IOL addition
connect	exs_connect	ES-API standard
fcntl	exs_fcntl	UNH-IOL addition
getsockname	exs_getsockname	UNH-IOL addition
getpeername	exs_getpeername	UNH-IOL addition
	exs_init	ES-API standard
listen	exs_listen	UNH-IOL addition
poll	exs_poll	ES-API standard
recv	exs_recv	ES-API standard
	exs_recv_offset	UNH-IOL addition
send	exs_send	ES-API standard
sendfile	exs_sendfile	ES-API standard
	exs_send_offset	UNH-IOL addition
socket	exs_socket	UNH-IOL addition

Figure 2. Functions in UNH-EXS

Most ES-API functions, new and old, are based on the

use of a *file descriptor* (fd) to identify a network connection. These fds are used in UNIX as an index into a process-specific table in the kernel that points to all the control information about the file. User-level code has only limited access to this information, and cannot add the types of new information necessary to implement the EXS functionality. The ES-API expects this to be dealt with by modifications to the operating system kernel. But because UNH-EXS is implemented entirely in user space, with no changes to the kernel, it must provide, in addition to all new EXS-API functions, its own equivalent type of a file descriptor and its own versions of all standard functions that can be applied to RDMA sockets, as listed in Figure 2. In addition, UNH-EXS added some new functions to provide additional capabilities, as discussed in section 4 and in an earlier paper[12].

3 Implementing EXS functions on top of the OFED stack

3.1 Advertisements and Acknowledgments

All socket communication requires a sender and a receiver, and EXS provides the two corresponding functions: exs_send and exs_recv. The underlying RDMA transfer operations are not as straightforward, due to the requirements of memory registration on both ends of the connection and other information needed by OFED verbs. Therefore, one side of a connection has to "advertise" to the other side the memory it wishes to use in a transfer before the actual transfer can occur, and the EXS interface must match up advertisements from the remote side with requests on the local side before it can initiate an actual RDMA transfer.

In the UNH-EXS implementation, the recipient of a data transfer always controls the RDMA transfer, a decision that was based on our prior experience with iSCSI [13, 14, 15] and with the use of iSER [16] in conjunction with iSCSI over RDMA [17]. The EXS implementation translates an exs_send into an RDMA "send" operation that sends a short advertisement containing the size and the starting address of the data block to be sent, and a memory registration "handle" that gives the receiving side permission to transfer the block of data directly from the sender's registered memory.

The EXS implementation of exs_recv attempts to match it with a previously received advertisement from the sending side, and waits (asynchronously) if necessary until such an advertisement is received. When a match is found, the receiving side's EXS implementation issues (asynchronously) an RDMA "read_request" operation that includes all the information from the advertisement plus the corresponding information from the exs_recv. This allows the RDMA hardware on both sides to cooperate in "pulling" the data from the memory of the sending machine directly into the memory of the receiving machine without any copying or CPU intervention.

Alternative designs, such as having the sender always "push" the data to the receiver by using the RDMA "write" operation, were rejected because the current design maps better onto the asymmetric RDMA instructions provided by iWARP RDMA hardware, and because the current design means the memory is being changed only on the machine controlling the transfer.

3.2 Credit-based Flow Control

Because an exs_send transfers user data from the sending user's memory directly into receiving user's memory, no additional buffer memory needs to be provided by the EXS implementation, the OFED stack or the operating system on either end.

However, an advertisement must be assembled in small buffers within the sending side EXS implementation, and received into small "untagged" buffers within the receiving side EXS. Advertisements are "unsolicited" – they can be sent at any time without prior warning to the receiver – and the implementation must have previously provided buffers to receive these unsolicited messages or the RDMA hardware will cause a fatal error. To deal with this, we have implemented a simple form of credit-based flow control to ensure that a sender will not send an advertisement unless it knows in advance that a receiver has a buffer ready to accept it.

Each side of a connection maintains two internal credit values: "send_credits" is the number of advertisements it is allowed to send to the other side, and "recv_credits" is the number of advertisements it is prepared to receive from the other side. When an EXS connection is first established, the EXS interfaces on both sides negotiate the initial values of these numbers. Prior to connection establishment, the user application can set the values to use in a negotiation with the exs_fcntl function.

Sending an advertisement requires that the sender have an unused send_credit; if not, it must wait (asynchronously) until one becomes available before sending the advertisement. Receiving an advertisement reduces the receiver's unused recv_credits. Both numbers are increased once the actual data transfer finishes, and this is indicated to the receiver's EXS interface by the OFED stack when the RDMA read_request operation completes. However, there is no corresponding completion notification given by the RDMA hardware on the sender side when the actual RDMA data transfer com-

pletes. Therefore, the receiver's EXS interface must also send a short unsolicited "acknowledgment" message back to the sender at the completion of a data transfer. This acknowledgment conveys to the sending EXS interface the completion status of the transfer, allowing it to increment its send_credits for this connection and to post the completion event to the sending application.

4 Additional Features of UNH-EXS

4.1 Immediate Data

As described in section 3, data transfers in EXS actually require several RDMA operations: one to send an unsolicited advertisement, one to start an RDMA read_request, and one to send an unsolicited acknowledgment. (A fourth, hidden RDMA "read_response" operation is performed entirely by the RNIC on the sending side in response to the RDMA read_request operation performed by the receiver's RNIC.)

When the amount of data in a transfer is small, the total transfer time will be dominated by the overhead needed to execute all these operations and to transfer the extra unsolicited messages. Therefore, the UNH-EXS interface enables users to utilize the exs_fcntl function to set a new "small_packet_max_size" parameter on a socket prior to establishing a connection on that socket. Whenever the user sends an amount of data not exceeding that limit, the UNH-EXS interface will actually send the user's data as an "immediate" part of the unsolicited advertisement itself. Since this data is registered on the user side, the RDMA hardware will still transfer it directly from the sending user's memory without any additional copying or buffering on the sending side.

When the receiving side EXS matches this advertisement with a local exs_recv, it just copies this immediate data into the memory area provided by the user in the exs_recv, avoiding the RDMA read_request and the hidden RDMA read_response operations, but introducing a previously unneeded data copy on the receiving side (only). The effect of this on performance is demonstrated in section 5.

4.2 Persistent Sends

Normally every byte sent in an exs_send is "consumed" when it is received by an exs_recv. However, the EXS socket interface to RDMA also makes possible new ways to share remote memory that are similar to the "one-sided" communications in MPI [18].

One side of the EXS socket can send a block of registered memory over that socket by calling the normal exs_send and supplying an "EXS_PERSISTENT" flag. What this means is that the other side of the EXS socket can repeatedly receive this same memory block – the sent data is not "consumed" as it normally would be for a socket send, but remains available for subsequent exs_recvs by the remote side of the EXS socket connection without subsequent corresponding exs_sends. Effectively this opens up the memory on the sending side of an EXS socket to repeated inspection by a remote receiver using the normal exs_recv operation. In such a situation, the sender is completely passive, since the remote receiver simply "pulls" the data (or parts of it) across the connection directly from the memory of the sender to the memory of the receiver without any CPU intervention on the sender's side. There is, in fact, no indication whatsoever to the sender that this has even happened, and the sender incurs no CPU overhead whatsoever when this happens.

There are many interesting uses for such a feature. For example, suppose the sender is an ongoing simulation of a physical phenomenon, and the receiver is a remote display station that wishes to dynamically display the current state of the simulation in real-time. This can provide invaluable real-time feedback to the experimenter. Effectively the receiver can "sample" the results at its leisure, without in any way interfering with the ongoing simulation. In particular, there is no need to synchronize the display with the simulation. The simulation updates its memory whenever it is ready; the remote display receives that memory directly into its own memory whenever it is ready. For many display applications it does not matter if the transferred data is not completely consistent – on a display screen the inconsistency may be noticed as a few "bad pixels" or as a slightly skewed boundary pattern between two regions in the simulation grid. If the receiver were concerned about internal consistency of the data, a simple checksumming technique could be used with minimum disruption to the simulation.

This technique can also be of great benefit for remote debugging. The area of memory sent by the sender in a persistent fashion should contain the variables pertinent to the ongoing simulation. The receiver will have to know the layout of these variables in that memory, but that can be arranged at compile-time or sent dynamically over the same EXS socket prior to the actual start of the simulation run. The receiver can then display the current value of the variables in real-time, under the control of the experimenter at the display console, while the simulation is running and without interfering with the simulation. The receiver could also keep a trace of these variables during the simulation, which can be useful if the simulation eventually crashes or otherwise becomes unstable.

This technique can also be of value when an application wishes to explicitly control the synchronization of a

simulation and a remote display station (for example) by exchanging short control messages over a separate communications channel (i.e., another socket) that may or may not need to use RDMA.

We can also consider a completely different type of application that wishes to have a transaction exchange (i.e., request – response) between the communication partners. One way to implement this would be to have two EXS sockets between the partners, with each partner sending persistently on one of the sockets and receiving on the other. The sender would actually send only once, and would thereafter modify the data in its memory only – the receiver would pick up the latest values whenever it did a receive.

There is a need to synchronize changes to the data on each side, and there are several possible ways to do this. The most obvious would be for each sending side to send a short message to the receiving side whenever it has finished changing its data, and to not change the data again until it receives a short message back from the receiver indicating both that the receiver's reply data is ready to read and that the sender could once again change its data. This adds the overhead of the extra short message exchange.

EXS offers another way to do this synchronization, using a variant of the persistent send described above. In this variant the user gives a flag to the send that requests persistent data as well as an asynchronous event whenever the receiver actually reads this persistent data. When it gets this event, the sender can infer that its data in memory can safely be changed. However, to give the receiver time to utilize the data it just read in order to generate a response, the sender can not infer from just that one event that it is also time to read the response from the other side. A simple solution to this is to use a double-buffering setup, with one persistent send connection per buffer. The event generated when the receiver reads buffer A now indicates that the sender can change the data in buffer A (as before), and can also safely read the data in buffer B. An example of the performance of this technique is given in Figure 9 in section 5.

4.3 Persistent Receives

It is also possible for a receiver to issue a "persistent" receive on an EXS socket. In this case, the same exs_recv will be used to satisfy all subsequent exs_sends from the remote side of the EXS socket. In symmetry with the persistent send, the side issuing the persistent receive is not synchronized with or made aware of the matching remote sends from the other side of the EXS socket. Rather the application on the receiving side just uses the current values in its local memory whenever it needs them. The remote sender will update them asynchronously via RDMA using a normal exs_send.

A persistent receive is inherently more "dangerous" than a persistent send, because it allows the remote side to change local memory without notification to or synchronization with the receiver. This implies that the receiver must be totally impervious to inconsistencies in the memory block, must not itself be changing these values, or must be explicitly synchronizing the remote sends via a second communications channel.

Note that both sides of an EXS socket connection cannot be persistent, since one of the sides must be an active (non-persistent) driver of the passive (persistent) side.

4.4 Offsets

The EXS socket interface has been extended by the introduction of additional exs_send_offset() and exs_recv_offset() calls that are defined to take an extra parameter, which is the offset from the start of the remote "persistent memory" at which the transfer should start. The effect is to enable the active side to selectively transfer appropriate subparts of the persistent memory for purposes of inspection or modification.

5 Performance Results

We first present results obtained by "blasting" data from a user on one workstation to a user on another workstation using RDMA interfaces. Each workstation contains four 64-bit Intel 2.66 GHz x86_64 processors with 4 Gbytes of memory, a NetEffect 10 Gbps RNIC, and a Mellanox 8 Gbps Lion Mini SDR HCA, both mounted in PCI-E 8X slots. The matching interfaces are connected back-to-back with CX4 (copper) cables. The operating system was Red Hat Enterprise Linux 5.1, kernel 2.6.18-53 SMP, and the OFED stack was version 1.4.

In the throughput graphs, user-level throughput in Megabits per second (Mbps) is plotted on the y-axis. The x-axis is the user payload size in bytes sent by the application on one machine using one exs_send and received by the application on the other machine using one exs_recv. The scale on both axes is logarithmic because of the large ranges spanned.

The graph in Figure 3 shows the user throughput when using the iWARP interface. The solid line in the graph shows the throughput with normal sends, the long-dash line with immediate sends, and the short-dash line with persistent sends. Transfers containing up to around 100,000 bytes produce slightly better throughput when sent using immediate or persistent data. Note that the maximum user-level throughput actually achieved (with all three types of send) is 9.325 Gbps for very large packets. This was achieved using 1500 byte Ethernet frames

that allow for a maximum theoretical user payload of 9.363 Gbps which takes into account all required headers and CRCs. The data for this graph was taken on the receiving side. The results for the sending side are essentially identical.

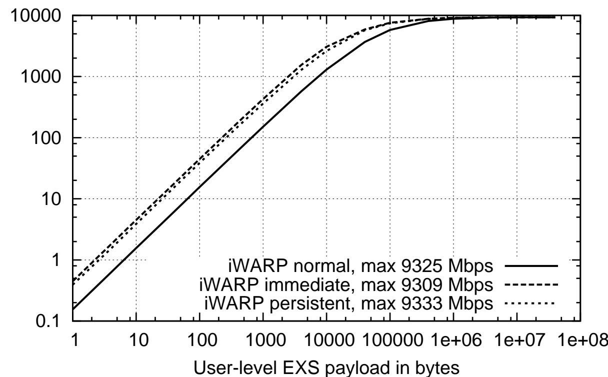

Figure 3. iWARP user payload throughput in Mbps

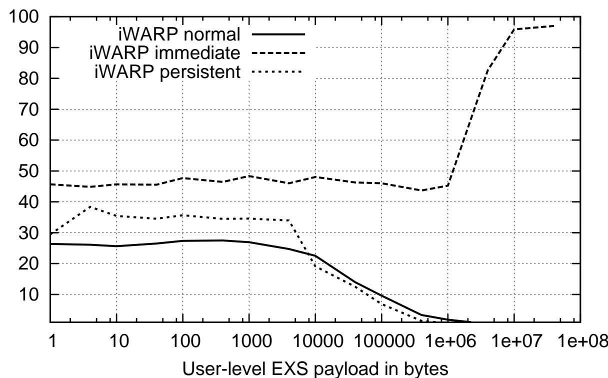

Figure 4. iWARP Percentage Utilization of one CPU during throughput tests

The graph in Figure 4 shows the percentage of CPU time used by the receiving side during the runs that produced Figure 3. The differences in CPU utilization are considerable. When using normal sends, the percent CPU utilization never exceeds 30%. When the size of a transfer reaches 100,000 bytes, the CPU utilization drops to 10%, and after 1,000,000 bytes it is essentially 0%. The CPU utilization curve for persistent sends is similar. But when the user-level data is transferred entirely as immediate data attached to advertisements, the percent CPU utilization never drops below 40%, and as the transfer size increases beyond 1,000,000 bytes, it rapidly increases to

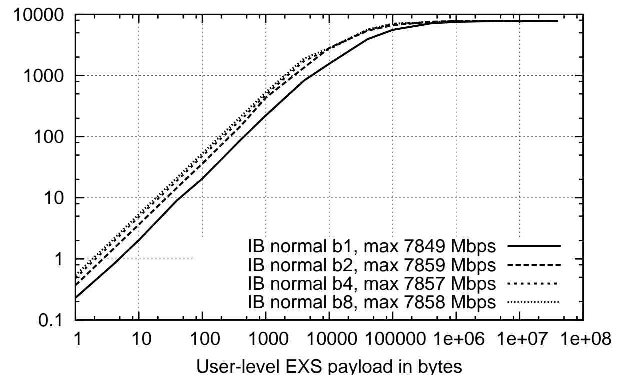

Figure 5. InfiniBand throughput in Mbps using simultaneous transactions

more than 90%. This is entirely attributable to the data copy from the advertisement's buffer to the user's memory area on the receiving side.

Results on the sending side are similar, except that the CPU utilization for persistent sends is always 0%! This is because the sender issues only a single exs_send with the EXS_PERSISTENT flag set, and this single exs_send matches all the exs_recvs issued by the receiver.

Figure 5 demonstrates throughput when using multiple asynchronous transmissions in EXS, this time over InfiniBand hardware. In this graph there are lines illustrating 4 situations, all using normal sends. These situations are: only 1 transfer "in-progress" at any time; 2 transfers simultaneously "in-progress" at the same time; 4 simultaneous "in-progress" transfers, and 8 simultaneous "in-progress" transfers. As can be seen, throughput increases as more simultaneous transfers are performed, although the size of the increase diminishes after introducing the second simultaneous transfer (i.e., using classic "double buffering").

Figure 6 compares the throughput performance of normal iWARP sends and normal InfiniBand sends. The InfiniBand throughput is slightly higher for payload sizes less than about 40,000 bytes; above that, iWARP performance is slightly better. Note that for large transfers, the maximum observed throughput for iWARP is 9325 Mbps, whereas for InfiniBand it is only 7849 Mbps. This is because the Ethernet link supports a maximum data transfer rate of 10 Gbps, whereas the maximum data transfer rate of an InfiniBand channel is 8 Gbps.

Figure 7 shows a similar throughput comparison when all the data is attached as immediate data in an advertisement. This graph shows essentially no difference in performance between the two technologies. Figure 8 shows the comparison when persistent sends are used. Here

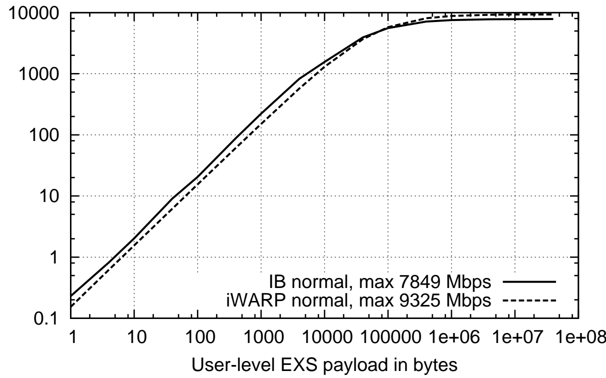

Figure 6. iWARP vs InfiniBand throughput in Mbps using normal EXS sends

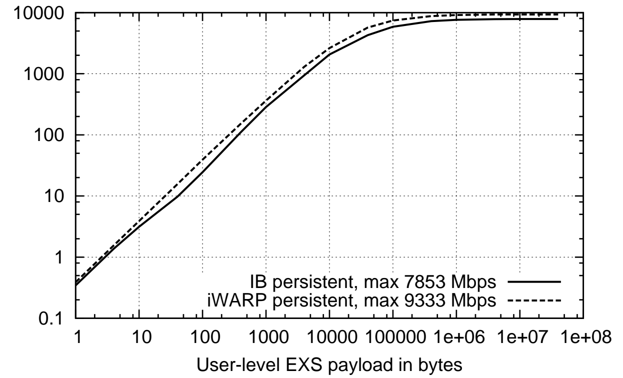

Figure 8. iWARP vs InfiniBand throughput in Mbps for persistent EXS sends

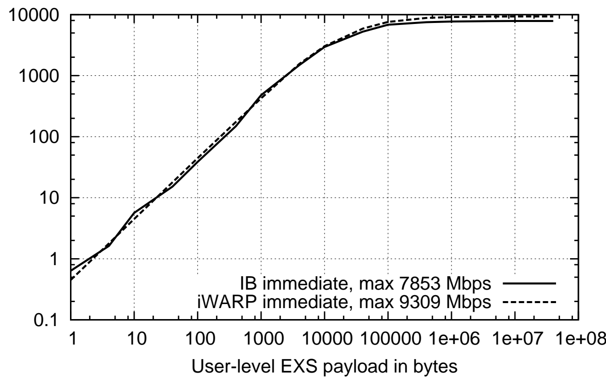

Figure 7. iWARP vs InfiniBand throughput in Mbps using EXS immediate data

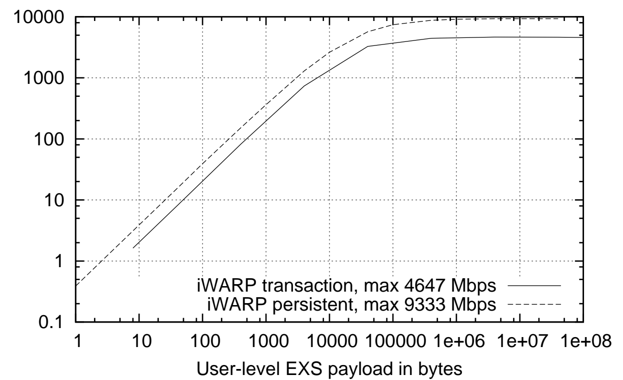

Figure 9. iWARP throughput in Mbps for transaction and persistent EXS sends

the iWARP NIC consistently outperforms the InfiniBand HCA by a small amount.

The table in Figure 9 shows the throughput achieved in the double-buffered transaction exchange discussed at the end of section 4.2 when compared with the simple "blast" using persistent sends just shown in Figure 8. The lower transaction performance is caused by the added synchronization. The payload is measured in only one direction, although in the transaction situation a similar payload is simultaneously traveling in the opposite direction.

The table in Figure 10 compares the one-way latency in microseconds using normal, immediate and persistent sends for both technologies. For this test, a classic "ping" user application was modified to use EXS functions. One-way latency is calculated by averaging the round-trip times for several million "ping" messages and then dividing by two.

For normal sends Figure 10 shows a one-way latency of 33 microseconds for both iWARP and InfiniBand. For sends that attach all their data as immediate data in the advertisement, iWARP one-way latency of 17 microseconds is somewhat better than the InfiniBand latency of about 19 microseconds. For persistent sends, iWARP one-way latency of 10 microseconds is much better than the InfiniBand latency of about 15 microseconds.

There are significant observable differences among the results of the various types of sends, regardless of the technology. Immediate sends have only about half the one-way latency of normal sends for both technologies. This is not surprising, since for such small amounts of data (1, 10 or 100 bytes), overhead accounts for the vast majority of the total time, and attaching data to the advertisement eliminates entirely the read_request, read_response actions of the iWARP RNIC. Persistent

Payload bytes	Normal		Immediate		Persistent	
	iW	IB	iW	IB	iW	IB
1	33	33	17	19	10	16
10	33	33	17	19	10	15
100	34	33	17	18	10	15

Figure 10. iWARP vs InfiniBand one-way latency in microseconds

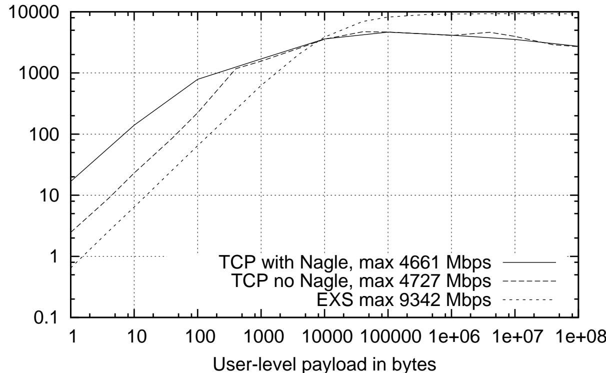

Figure 11. iWARP vs TCP throughput in Mbps

sends have only 30% of the one-way latency of normal sends on iWARP, 45% on InfiniBand. This reduction in latency is due to the omission of both the advertisement and the acknowledgment messages that normally surround the RDMA read_request, read_response actions. With further tuning and optimization we should be able to reduce these latency times.

Figure 11 compares the throughput of iWARP and TCP over the same 10GigEthernet NICs (TCP cannot use the RDMA feature of those NICs). Lines are plotted for TCP performance with and without the Nagle algorithm, as it makes an obvious difference when "blasting" small packets. Both TCP throughputs are better for payloads up to about 6,000 bytes, where they achieve their maximum of around 4700 Mbps, about the same as for EXS at that payload. For larger payloads, both TCP throughputs gradually decline, whereas EXS throughput continues to rise to almost full available bandwidth. The poor performance of EXS for small payload sizes is due to the fact that the initial EXS implementation reflects the message-oriented nature of the underlying RDMA protocols, which are more like TCP without the Nagle algorithm and are closer to SOCK_SEQPACKET semantics. TCP implements SOCK_STREAM semantics and

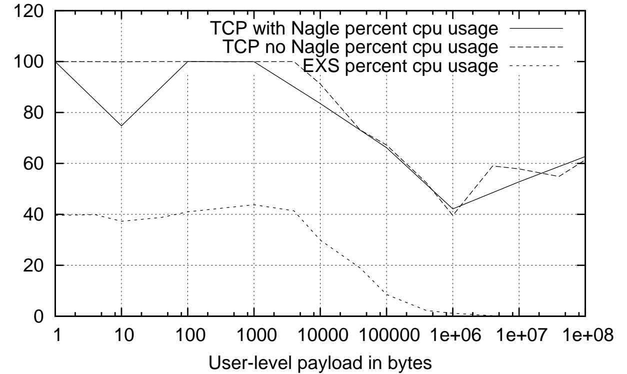

Figure 12. iWARP vs TCP cpu utilization

thus performs better when "blasting" small packets, with or without the Nagle algorithm. We are working to improve this performance of our EXS implementation.

Figure 12 compares the cpu utilization for these same runs, and shows that EXS uses significantly less of the cpu than TCP for all payload sizes. Furthermore, EXS cpu utilization, never more than 43%, decreases to essentially zero for the largest payloads, whereas for both types of TCP, several payloads sizes use 100% of the CPU.

6 Conclusion

The EXS interface provides convenient, high-level access to RDMA network hardware. It is much easier to use than the OFED verbs, and is much more familiar to programmers who have used conventional sockets. It also enables new methods of using RDMA communications by providing persistent receives and sends. Results shown in this paper demonstrate that application programs using the UNH-EXS implementation can attain reasonable performance with both types of RDMA network hardware, iWARP and InfiniBand.

References

[1] InfiniBand Trade Association. InfiniBand Architecture Specification version 1.2.1, November 2007.

[2] A. Romanow, J. Mogul, T. Talpey, and S. Bailey. Remote Direct Memory Access (RDMA) over IP Problem Statement. RFC 4297 (Informational), December 2005.

[3] P. Culley, U. Elzur, R. Recio, S. Bailey, and J. Carrier. Marker PDU Aligned Framing for TCP Specification. RFC 5044 (Standards Track), October 2007.

[4] H. Shah, J. Pinkerton, R. Recio, and P. Culley. Direct Data Placement over Reliable Transports. RFC 5041 (Standards Track), October 2007.

[5] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia. A Remote Direct Memory Access Protocol Specification. RFC 5040 (Standards Track), October 2007.

[6] OpenFabrics Alliance. http://www.openfabrics.org, 2009.

[7] OpenFabrics Enterprise Distribution. http://www.openfabrics.org/, 2009.

[8] Open MPI: Open Source High Performance Computing. http://www.open-mpi.org, 2009.

[9] MVAPICH: MPI over InfiniBand and iWARP. http://mvapich.cse.ohio-state.edu, 2009.

[10] Interconnect Software Consortium in association with the Open Group. Extended Sockets API (ES-API) Issue 1.0, January 2005.

[11] Open Group. http://www.opengroup.org, 2009.

[12] R.D. Russell. The Extended Sockets Interface for Accessing RDMA Hardware. In T.F. Gonzalez, editor, *Proceedings of the 20th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2008)*, pages 279–284, November 2008.

[13] R.D. Russell. iSCSI: Past, Present, Future. In *Proceedings of the 2nd JST CREST Workshop on Advanced Storage Systems*, pages 121–148, December 2005.

[14] Y. Shastry, S. Klotz, and R.D. Russell. Evaluating the Effect of iSCSI Protocol Parameters on Performance. In T. Fahringer and M.H. Hamza, editors, *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN 2005)*, pages 159–166, February 2005.

[15] A. Chadda, A.A. Palekar, R.D. Russell, and N. Ganapathy. Design, Implementation, and Performance Analysis of the iSCSI Protocol for SCSI over TCP/IP. In *Internetworking 2003 International Conference*, June 2003.

[16] M. Ko, M. Chadalapaka, J. Hufferd, U. Elzur, H. Shah, and P. Thaler. Internet Small Computer System Interface (iSCSI) Extensions for Remote Direct Memory Access (RDMA). RFC 5046 (Standards Track), October 2007.

[17] E. Burns and R.D. Russell. Implementation and Evaluation of iSCSI over RDMA. In *Proceedings of the 5th International Workshop on Storage Network Architecture and Parallel I/O (SNAPI'08)*, September 2008.

[18] MPI Forum. MPI: A Message-Passing Interface Standard, June 2008. http://www.mpi-forum.org/docs/mpi21-report.pdf.