Protocols for Wide-Area Data-intensive Applications: Design and Performance Issues

Yufei Ren, Tan Li, Dantong Yu,
Shudong Jin, Thomas Robertazzi
Department of Electrical and Computer Engineering
Stony Brook University
Stony Brook, NY, 11794, USA
Email: yufren@ic.sunysb.edu, tanli@ic.sunysb.edu, dtyu@bnl.gov,
shujin@notes.cc.sunysb.edu, tom@ece.sunysb.edu

Brian L. Tierney, Eric Pouyoul
Lawrence Berkeley National Laboratory
One Cyclotron Road
Berkeley, CA, 94720, USA
Email: btierney@lbl.gov, epouyoul@lbl.gov

***Abstract*—Providing high-speed data transfer is vital to various data-intensive applications. While there have been remarkable technology advances to provide ultra-high-speed network bandwidth, existing protocols and applications may not be able to fully utilize the bare-metal bandwidth due to their inefficient design. We identify the same problem remains in the field of Remote Direct Memory Access (RDMA) networks. RDMA offloads TCP/IP protocols to hardware devices. However, its benefits have not been fully exploited due to the lack of efficient software and application protocols, in particular in wide-area networks. In this paper, we address the design choices to develop such protocols. We describe a protocol implemented as part of a communication middleware. The protocol has its flow control, connection management, and task synchronization. It maximizes the parallelism of RDMA operations. We demonstrate its performance benefit on various local and wide-area testbeds, including the DOE ANI testbed with RoCE links and InfiniBand links.**

I. Introduction

Data-intensive applications such as those in the grid and cloud computing environment are generating extremely high volumes of data. Data is often transferred, visualized, and analyzed by geographically distributed teams of users. High-performance network capabilities must be available to support these applications across both local and wide area networks. The efficient design of network protocols and software systems is a crucial aspect of research and development in data intensive computing.

Protocol offload and hardware acceleration are among the techniques used to achieve high data transfer rates with minimal consumption of host resources. The TCP/IP Offloading Engine (TOE) is one of the early examples of protocol offload to meet these requirements. Its underlying concept lies in the use of a dedicated hardware module on the network adapter card to process TCP/IP internal operations such as segmenting, framing, reassembling the payload, timing, and flow control. Research [1], [2] has demonstrated that TOE is a cost-effective technique to free host processors from excessive protocol processing, and therefore improves the concurrency between communication and computation. Thereafter, Remote Direct Memory Access (RDMA) was proposed as a hardware-based Protocol Offload Engine (POE) solution to move bulk data from the source host memory directly to the remote host's memory with kernel-bypass and zero-copy operations. Its use has recently became popular in environments implementing and utilizing converged Ethernet and data center bridging technologies.

Along with assuring near line-speed data transfer, another challenge is to manage a heterogeneity of underlying RDMA architectures for diverse applications. Various RDMA implementations offer opportunities to enhance the performance of a data transfer service, overcoming the limitations of a kernel-based TCP/IP approach [3], [4]. However, despite of the emergence of industry standards such as OpenFabrics Enterprise Distribution (OFED) [5], it is still difficult for applications to manage multiple devices.

In this paper, we study the issues related to designing a high-speed network protocol and improving application performance. In particular, we focus on Remote Direct Memory Access (RDMA), and investigate the interaction between application protocols/software and network hardware capabilities. We designed an application layer protocol for RDMA networks, as part of a middleware layer that integrates network access, memory management, and multitasking. We address various issues related to its efficient implementation such as buffer management and memory registration, and the parallelization of RDMA operations, all of which are vital to delivering the benefits of RDMA to applications. Using this protocol, we implemented an RDMA-based FTP software, RFTP. Our developmental work is part of a larger project to exploit the full capacity of a 100Gbps network in the U.S. Department of Energy's Energy Sciences Network (ESnet) [6].

The contributions of this paper encompass the following. First, we validate our design choice for RDMA-based data transfer protocols, and detail its performance benefits. Second, we propose and discuss the implementation of our data transfer protocol using OpenFabrics's asynchronous RDMA write, while maximizing the parallelism of data transmission. Third, we describe our extensive experiments to evaluate the performance of our protocol, particularly over wide-area networks. We show that our tool has higher performance compared with existing widely used data transfer tools such

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00

as GridFTP [7].

The remainder of this paper is organized as follows. In Section II we summarize previous work on RDMA technologies. In Section III we validate our design choices using some preliminary performance tests. Section IV describes our protocol design and implementation. Our experiments and results are reported and analyzed in Section V, followed by our conclusions.

II. Related Work

The outstanding performance benefits of RDMA technology for data center networks and high performance computing have attracted a great deal of interest from academia and industry. The original RDMA architecture, known as InfiniBand (IB) [8], supports a top-down RDMA message service with its own implementation of layer two to layer four protocol (sometimes including layer-1) of the OSI stack. It provides a message passing service to applications with all protocol processing operations offloaded to specialized hardware. Unlike the best-effort frame delivery service in Ethernet, the link layer of InfiniBand offers reliability and maintains packet order through its credit-based flow control and virtual lane mechanisms. However, extending IB on WAN requires proprietary hardware to encapsulate IB into the Ethernet frame. This limitation has prevented its wide adoption.

Two other implementations, Internet Wide Area RDMA Protocol (iWARP) and RDMA over Converged Ethernet (RoCE), were proposed to extend the advantages of RDMA to ubiquitous IP/Ethernet-based networks and integrate the traditional network structure with these advanced mechanisms. iWARP offloads the whole TCP/IP stack. The Direct Data Placement (DDP) layer of the iWARP stack implements and supports zero-copy and kernel-bypass; it transfers data in the user-space buffer directly to application memory on the remote server. iWARP enables RDMA to seamlessly run over best effort IP networks such as the Internet. RoCE techniques support the running of IB transport protocol over Ethernet and offer the advantages of IB in an Ethernet environment. Compared to iWARP, RoCE is a more natural extension of message-based data transfer, and therefore, of the two, offers better efficiency [9].

One objective of our design is to support applications across all these RDMA architectures. We built our system with the common Verb Application Programming Interface (API) from the OpenFabrics Enterprise Distribution (OFED) [5], a unified, cross-platform, transport-independent software stack for RDMA. OFED offers a uniform application programming interface, known as native IB verbs, to access various RDMA architectures. Applications mainly use the *libibverbs* and *librdmacm* libraries. Figure 1 shows the layered structure, with applications at the top layer. OFED software also offers several middleware packages, such as IP over IB [10] and Sockets Direct Protocol (SDP) [11], to allow socket-based applications to run over RDMA devices without needed to rewrite the program. The User Direct Access Programming Library (uDAPL) [12] also has RDMA capabilities for applications,

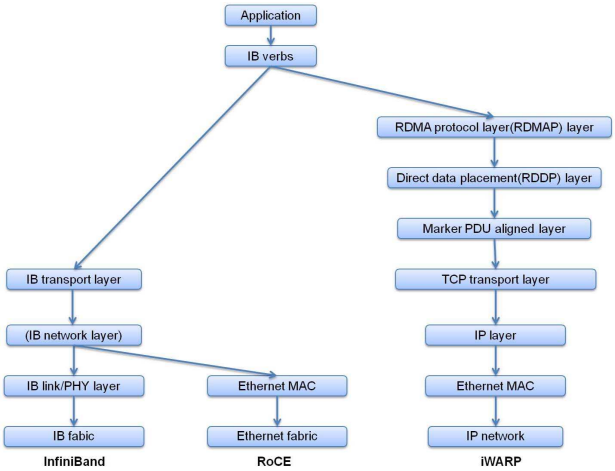


Fig. 1. Applications over different RDMA protocols

and was used in other studies [13], [14]. Nevertheless, these extensions introduce additional overhead and performance penalties compared to the native RDMA IB verbs [15].

RDMA offers two message transfer semantics: Channel and Memory. The former, SEND/RECEIVE, also referred to as a two-sided operation in RDMA, where both the source and sink kernels are involved in the data transfer once the connection is established [16]. The communication channel between the source and sink is modeled as queue pair (QP). Each QP consists of one sender and one receiver queues, whilst each queue represents one end of the channel. Before an application uses RDMA to transfer data, the receiver posts a work request to the receiver queue, after which the sender can post a work request to the send queue. Both the sender and receiver will get a completion event after the data transfer is finished. On the other hand, the semantics of the RDMA READ/WRITE mode is regarded as one-sided operation. The receiver advertises its available registered memory to the sender, including the information of memory region and the address, so that the sender can directly use RDMA WRITE to write data into the specified memory location within the receiver host.

Lai [15] implemented a RDMA FTP application based on the two-sided zero-copy operation of IB networks. However, SEND/RECEIVE operations originally were proposed for delivering control messages. The one-sided semantics of RDMA READ/WRITE is a better choice for high-speed large-scale data transfer because it can decouple the data transfer entirely from the kernel software of the host operation system. Other researchers [17], [18] demonstrated that even though there are some benefits of using RDMA over LAN and WAN with short latency, there are challenges in achieving good performance in WAN with a long latency due to the problem of its low performance with RDMA READ operation. Based on these related works and their prior studies [16], [17], [18], our middleware is designed to exploit the full benefit of the RDMA by using RDMA WRITE operation yielding better performance and lower communication cost for synchronizing

senders and receivers.

Tian et al. [19] has implemented a RDMA extension driver for GridFTP to utilize the high network bandwidth provided by InfiniBand. Similar to our approach, they employed RDMA WRITE to transfer large blocks of data. However, their design is not fully optimized. For example, the data source needs one RTT to get credits (tokens for data transfer) from the remote side's buffer, a drawback that will slow down data transfer in the wide area networks (WANs) with a large RTT. Moreover, it is not clear how their protocol enforces flow control between the two communicating parties, and whether it maximizes the parallelism of RDMA operations. Subramoni [20] also presented another driver to the GridFTP framework to merge the capability of InfiniBand and GridFTP. RDMA technology was extended and integrated into Message Passing Interface (MPI) [21], [14], [22] to allow parallel applications to take advantage of RDMA's low latency and high performance communication capability. However, its scalability and performance are not yet tested and validated in the newly available 40Gbps InfiniBand and Ethernet network environments.

III. Background and Design Choices

Our objective is to design data transfer protocols that transparently utilize the underlying RDMA network architecture, and offer superior bandwidth performance. To that end, we need to make intelligent decisions on the architecture of software and the way to use RDMA semantics. In this section, we describe our middleware architecture and validate our choice on RDMA semantics.

A. Middleware overview and our contributions

In [23], we detailed our preliminary middleware implementation. In this paper, we elaborate upon the updated design of the protocol and the implementation of the middleware layer. The middleware layer lies between the applications and RDMA network transport layer to meet our goal of creating a generalized and common interface and architecture that simplifies the process of developing various RDMA-based applications. This middleware can take advantage of RDMA techniques to attain high network throughput. It provides the necessary data communication and access functions, while maximizing the parallelism of data processing with advanced features such as zero-copy, reuse of memory regions, multi-stream parallel transfer, and multi-threading.

The middleware layer, as shown in Figure 2, implements a set of function modules, and provides an abstraction of computational resources including main memory and network cards. The middleware layer contains two primary components: the data structure component keeps track of the data structures necessary for data communication and memory access; the other component offers a pool of threads with all the functional modules related to data communication, synchronization, and task scheduling. The threads handle data transfer and the completion event (CE) asynchronously.

The middleware interacts with host computer adapters (HCA) via an array of queue pairs, supported by the OFED

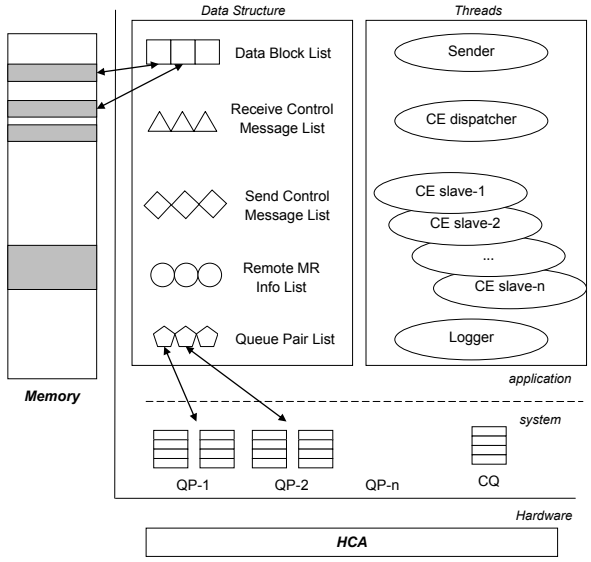


Fig. 2. Multi-threaded architecture and data structure of RDMA-based middleware

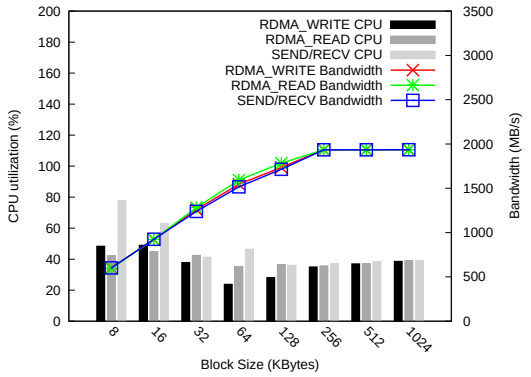
standard. The same standard also maintains a separate queue and the completion queue (CQ). The threads in the middleware layer gain access to the queue pairs and completion queue via a standard programming interface.

B. RDMA semantics performance

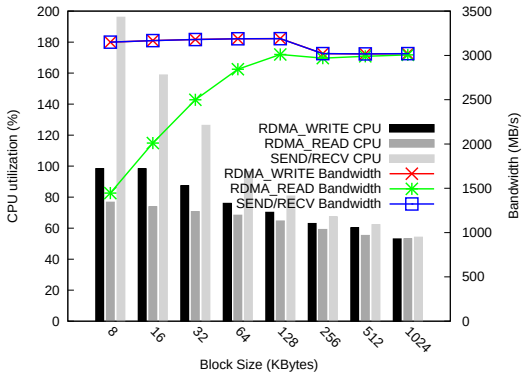
The second design choice is that of RDMA semantics. In making this decision, we considered various performance factors. To compare the performance of various RDMA channel and memory semantics, we designed a RDMA I/O engine and tested it with "fio" [24], an I/O benchmark and stress test tool that offers flexible parameter settings and excellent performance statistics reporting capabilities for both synchronous and asynchronous I/O. Compared with the standard OFED benchmark tools, our approach is easier to collect comprehensive performance statistics from the I/O module, including CPU usage, I/O latency, bandwidth, I/O performance distribution, and so on. The RDMA engine uses asynchronous I/O and allows our test program to simultaneously post multiple I/O requests. For both RDMA semantics (one-sided and two-sided), we conducted a comprehensive set of test cases with varying block sizes and maximum number of concurrent blocks in flight (also called I/O depths).

With low I/O depth, as shown Figure 3(a) and Figure 4(a), RDMA WRITE, RDMA READ and SEND/RECEIVE exhibit similar performance, while the CPU consumption of SEND/RECEIVE is much higher than that of the others. Its high CPU consumption reflects the fact that SEND/RECEIVE involves both the data source and sink during transfer, and the sink must process the same number of RDMA events as the source. However, RDMA READ/WRITE only handles RDMA events at one end.

A high I/O depth improves bandwidth performance as

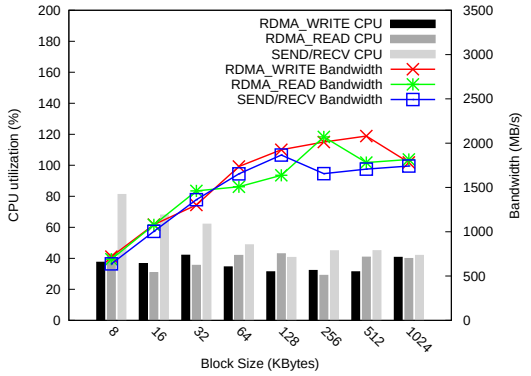


(a) I/O depth is 1

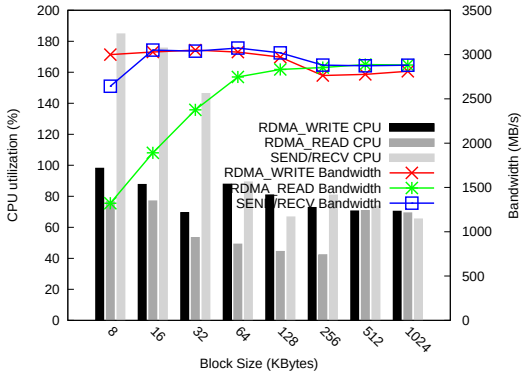


(b) I/O depth is 32

Fig. 3. RDMA semantics performance evaluation in RoCE Environment



(a) I/O depth is 1



(b) I/O depth is 32

Fig. 4. RDMA semantics performance evaluation in InfiniBand Environment

depicted in Figure 3(b) and Figure 4(b). To improve RDMA performance, an application must post multiple I/O tasks in flight to fully take advantage of OFED's asynchronous programming interface. Several observations resulted from these experiments: 1) RDMA WRITE and SEND/RECEIVE perform better than RDMA READ; 2) all test cases set block size in the range from 16KB to 128KB to achieve the best bandwidth; 3) performance saturates when the block size is bigger than 128KB; 4) CPU usage decreases when the block size increases because of fewer interrupts; and 5) during their peak performance, the CPU usage of SEND/RECEIVE is higher than that of RDMA WRITE.

Since the arrival rate of incoming data is unpredictable, the data sink must pre-post sufficient registered buffers in the receive queue before the data source transfers data. Otherwise, the data source may encounter the Receiver Not Ready (RNR) error indicating that the data sink's buffer is not available for receiving data. Then, the source must pause, causing low performance and under utilized network bandwidth. A control message mechanism was introduced to avoid the RNR error in Tian et. al.'s protocol implementation [19], where the source must wait for credits piggybacked with the message from data sink before it further posts more tasks into the send queue.

In summary, RDMA WRITE performs the best with the least CPU consumption in all test cases, and I/O depth should be set to a relatively large number, as identified in the previous testing results. Therefore, we designed a hybrid data transfer protocol that exchanges control messages via SEND/RECEIVE, and transfers bulk of data via RDMA WRITE.

IV. Protocol Design and Implementation

A. Protocol Overview

The OFED standard supports two types of queue pairs for host-to-host communication: Reliable Connected (RC) and Unreliable Datagram (UD). Considering the requirements of performance *and* reliability, we selected RC queue pairs for our protocol. The application can divide the entire dataset to be transferred into large blocks, a feature that usually leads to low processing overhead. On the other hand, the UD type is supported only in channel semantics, and the block size is limited by the size of the MTU [25]. A small block size may incur high CPU consumption, since many small blocks trigger a large number of queue pair events and interrupts that must be handled at both the data source and sink.

In our protocol, we use one dedicated queue pair for exchanging control messages between two communicating parties, and one or more for actual data transfer. Figure 5

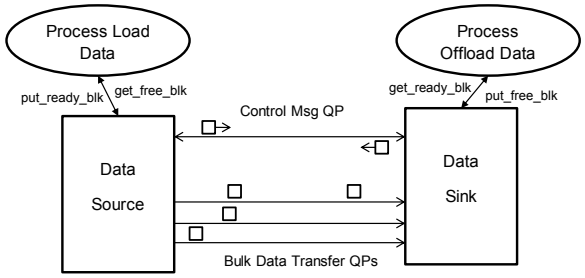


Fig. 5. Protocol Overview

illustrates how this protocol works. We use an event-driven design where different types of control message or regular data blocks trigger different events to be handled by pre-defined event routines.

To fully utilize the RDMA technology, our protocol design incorporates several optimizations. Firstly, the protocol keeps multiple data blocks in flight during the entire data transfer period. As we mentioned in the previous section, a high queue depth with several data blocks in flight is the key to achieving good performance. Secondly, the protocol is capable of using parallel queue pairs to transfer multiple data blocks simultaneously, eliminating the performance limitation of a single queue pair. With multiple queue pairs, there is the possibility of out-of-order arrivals of data blocks at the data sink. The protocol implementation must therefore be able to reassemble such out-of-order blocks. Thirdly, since the protocol uses RDMA WRITE to deliver bulk user payload, credits (tokens with destination address) are required before transmitting the data. It takes one additional round trip time (RTT) if the source explicitly requests credit information from the data sink. To save this RTT, our protocol adopts an active feedback mechanism. The data sink will proactively send the available data block information (credits) to the data source, and the data source keep track of all available ones.

B. Finite State Machines Modeling

To better illustrate our protocol, we used a finite state machine to model buffer blocks and their status at both the data source and sink. In our data transfer protocol, unlike TCP sockets, the sender does not explicitly copy data from user space to kernel space. Instead, the sender only posts tasks via the OFED interface, and afterwards the network card directly retrieves data from user space. With this model, the finite state machine of buffer blocks explains our protocol's behavior.

In the data source, a block (a chunk of memory resource for storing data) is initialized into a "free" state. A data transfer application can reserve a free block by `get_free_blk` which changes the state of the reserved block from "free" to "loading". The application then loads data from disk directly to the memory block, and the state then changes from "loading" to "loaded". Before the actual data transfer, the data source needs to know the remote memory's information, such as the unique identifier (rkey) and memory address of the data sink. Afterwards, the source encapsulates the block information into

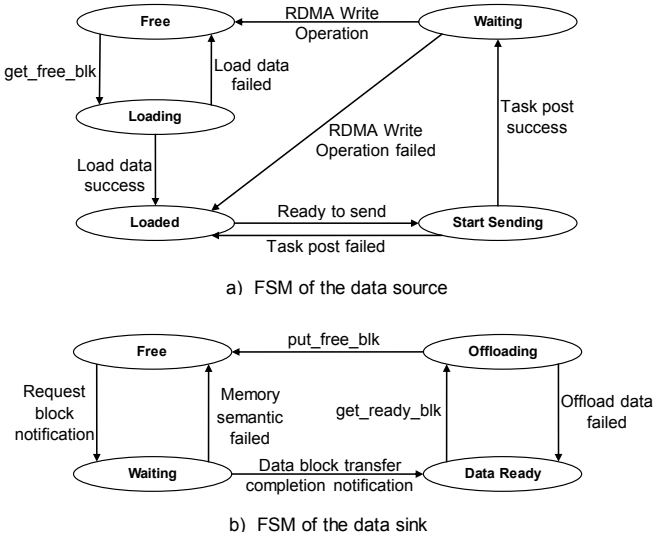


Fig. 6. User Payload Block's Finite State Machine

a memory semantic task, and posts it into the send queue. The state then changes from "Start sending" to "Waiting" if the task is posted successfully. "Waiting" means the content of the memory block is in flight. After the application polls the status of the memory semantic operation, the state is changed to "free" again if successful or "loaded" for re-sending if polling fails.

A block's state in the data sink finite state machine changes from free into waiting once either of the following two kinds of event is retrieved. One is a block request notification, which means the data source runs out of credits and is eager to get more credits as soon as possible. The other possible event is a completion notification of another memory block, which implies that the data source consumes one credit for that block. For efficient data transfer, a proactive feedback mechanism sends back one or two credits immediately to avoid the source running out of credit. A finish notification related to this block changes its state into "data ready". The application retrieves a block's payload from the protocol layer by `get_ready_blk`. After the application consumes the block's payload, i.e. offloading data into file system, the block's state is changed into "free" again by `put_free_blk`.

C. Connection Management and Message Format

As described in the previous subsection, the data source and sink manage buffer blocks and transfer data using asynchronous RDMA operations. Next, we detail connection management and message types during the process of moving data.

Each instance of data movement consists of three phases: (1) Initialization and parameter negotiation; (2) data transfer and reordering, and (3) connection teardown. Figure 7(a) shows the format of the control message exchanged through the dedicated control message queue pair, and Figure 7(b) shows the format of the user payload data delivered through multiple data channel queue pairs.

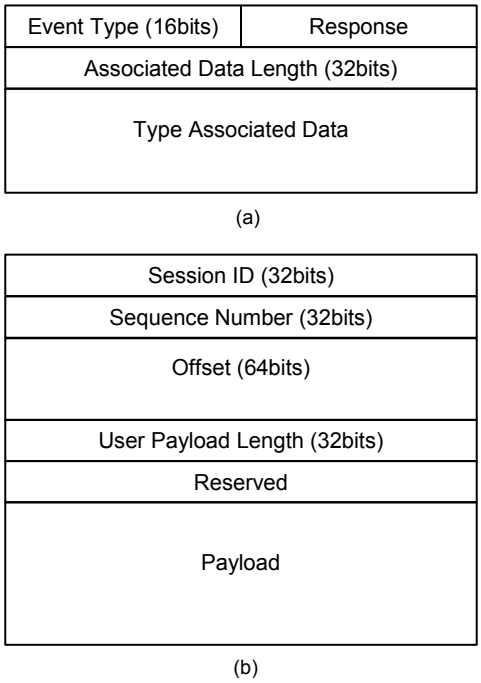


Fig. 7. Message Format of (a) Control message, and (b) User Payload Bulk Data Block.

In the first phase before data transfer, the data source sends requests to the sink to negotiate the block size, number of data channel queue pairs, and session identifier for each data transfer job.

- **Block size negotiation:** The data source selects a block size based on the user's input parameters, and copies the size information to the field of "Type Associated Data" of the control message to be sent to the sink. The sink sends back a reply on whether or not it accepts the block size for data transfer.
- **Number of data channels negotiation:** The protocol is designed to support multiple data channels, even when only transferring a single file. The source and the sink will exchange messages to agree on and establish a user-defined number of parallel queue pairs to deliver payload data.
- **Session identifier negotiation:** Each data transfer job, such as one file, is assigned a unique session identifier before the data is transferred. This identifier is placed into the header of every user payload block during the transfer of data. The application probably issues multiple data transfer tasks simultaneously. Each task is associated with a global session identifier which is available in both the source and sink. The sink is able to reassemble out-of-order blocks and deliver an in-order sequence of blocks to upper applications according to the session identifier and sequence number.

Our protocol supports asynchronous data transfer using OFED, viz., the key to enabling higher performance over the traditional TCP-based approaches. The source posts multiple payload data blocks in flight, and the sink actively acknowledges the successful receipt of data and returns the available memory region for the subsequent data transfer. There are three types of control message in this phase.

- **Memory Region (MR) block information request:** Once there is no available remote memory region for storing data before transferring, the data source sends this message to the data sink to request the next available memory region. The source is blocked until the sink sends back a response with MR information.
- **Block transfer completion notification:** The source sends a completion notification to notify a data sink that a data block is finished and available for the sink to read. This notification includes the block's ID and address, allowing the data sink to extract the payload from the memory block.
- **Memory region block information response:** The previous two types of control messages from the data source trigger the sink to send back any available memory region information. If the sink gets a **memory region block information request**, this indicates the source is idle and waiting for credits to proceed. The sink sends back one or multiple available addresses information according to the runtime status of the data transfer. If the sink gets a **block transfer completion notification**, the source must consume an available data address, and the sink grants back, at most, information on two available memory regions. This results in an exponential increase in the number of available remote MR in the data source at the beginning of a data transfer session. Such a design is similar to the slow start of TCP which allows the data transfer protocol to quickly fill up the available bandwidth. If at that time there is no available memory region in the data sink, the completion notification is simply ignored and the sink does not have to send a response. However, for the **memory region block information request**, the sink must send a response once there is at least one available memory regions. Otherwise, the responder will be delayed until one becomes available.

Finally, in the teardown phase, the source issues a **data set transfer completion** message indicating that the whole data set was transferred completely to the sink.

V. Experimental Results

To validate our middleware and protocol and its reference implementation, RFTP (RDMA-enabled FTP), we conducted comprehensive experimental studies on several LAN and WAN test environments. We begin this section describing the test configuration based on various RDMA architectures, including RoCE and InfiniBand, in LAN and WAN network environments. We then compare the performance of RFTP with GridFTP, a high performance data transfer tool widely used in the data-intensive science applications.

TABLE I
TESTBED DESCRIPTION

	InfniBand LAN	RoCE LAN	RoCE WAN
CPU * Cores	Intel Xeon X5550 2.67GHz 8 Cores	Intel Xeon X5650 2.67GHz 12 Cores	ANL: AMD Opteron Processor 6140 2.6GHz 16 Cores NERSC: Intel Xeon E5530 2.40GHz 8 Cores
Mem(GBytes)	48	24	ANL: 64 NERSC: 24
NICs(Gbps)	40	40	10
OS	RHEL 5.5	CentOS 6.2	ANL: CentOS 5.7 NERSC: CentOS 6.2
Kernel Version	2.6.18-238	2.6.32-220	ANL: 2.6.32-220 NERSC: 2.6.32.27
OFED Version	1.5.3.1	MLNX OFED 1.5.3	1.5.3
TCP Congestion Control Algorithm	cubic	bic	ANL: cubic NERSC:htcp
MTU Size	65520	9000	9000
RTT(ms)	0.013	0.025	49

A. Testbed Setup

We consider both memory-to-memory and memory-to-disk data transfer between local and remote hosts. For the former, memory data in the source is generated from /dev/zero, transferred via RDMA, and copied into /dev/null at the sink. In this configuration, our focus is to evaluate the performance in terms of network bandwidth and the efficacy of protocol offloading. We did not access the performance of a test scenario with a file system considered since it is much slowed than our 40 Gbps network testbed. For modern data center applications, as suggested in [15], it is a reasonable simplification to avoid the disk I/O bottleneck. We consider a variety of network environments include the LAN (which plays a key role in today's data center and cloud computing applications) and the WAN (which is essential to inter-data center transfers and to upload or download data to remote clients). The details of our three configurations are as follows.

1) High-bandwidth low-latency RoCE and InfiniBand LANs: To test application performance over different RDMA architectures, we set up two local-area test platforms. The first one is a back-to-back connection testbed in Stony Brook University. The propagation delay between hosts is less than 0.1ms. Each host is equipped with a 40Gbps RoCE HCA. The second test platform includes two nodes at the NERSC Computational Center. Each node has a Mellanox InfiniBand HCA interconnected by a 4X QDR InfiniBand switch, theoretically providing 32 Gb/s of point-to-point bandwidth. The vendor reported that the actual bandwidth is about 25 Gbps during their product validation.

2) High-bandwidth long-latency WAN RoCE Testbed: For the WAN test with long-latency links, we used the Advanced Networking Initiative (ANI) 100Gbps testbed[1] between Argonne National Laboratory near Chicago, IL, and the National Energy Research Scientific Computing Center (NERSC) in Oakland, CA, about 2000 miles away. The hosts on the ANI Testbed are equipped with a 10 Gbps RoCE NIC.

[1]ANI Testbed: http://ani-testbed.lbl.gov/

B. Parameter Configuration and Tuning

For a fair application comparison of these applications, we ran our test cases of RFTP and GridFTP on the same set of well-tuned hosts, and in a common network environment. Table I lists the detailed configurations for all the nodes, in all three testbeds described.

To improve the performance of TCP for transferring bulk data, we tuned the parameters of O.S. kernel, NIC and the host's power setup according to the vendor supplied manual [26]. For certain hosts in the testbeds, we employed some variants of TCP algorithms. But we always evaluate RFTP and GridFTP with the same TCP variants. The size of MTU was set to 9000 bytes on all hosts.

We also have optimized the configuration of GridFTP to ensure that it reached the best performance for network link with a large bandwidth-delay product (BDP). The GridFTP client, the globus-url-copy with extended block mode (MODE E) [27] was utilized for all data transfer; authentication was intentionally turned off to minimize the extra cost for data security. Both the GridFTP client and server here are threaded [28]. The size of TCP buffer is set to be the BDP of the link, a proven value for the optimal network performance.

An important characteristic for GridFTP and RFTP is that they can both transfer a large file via multiple streams. Since there is no disk bottleneck in the memory-to-memory test, we transferred one file in each test case to assess the impact of the number of parallel streams. For the memory-to-disk test, we created a group of 400GB files spread across multiple RAID disks to achieve the best performance of the disk system.

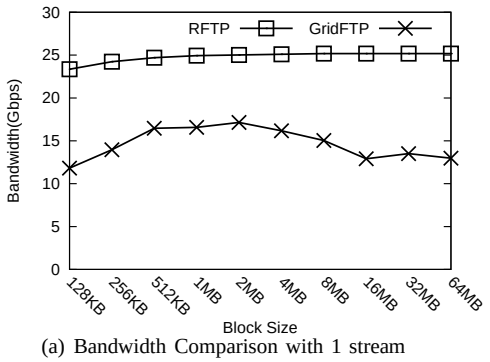
C. Experimental Results over LAN

In this set of experiments, we used memory-to-memory data transfer as the baseline results to compare the performances of RFTP and GridFTP.

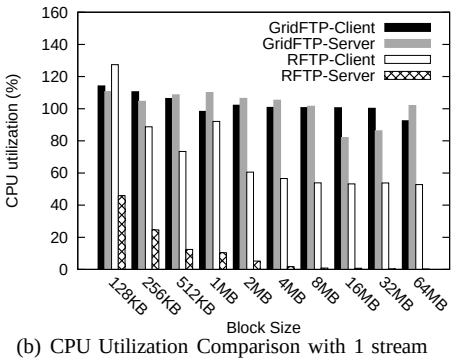
1) Bandwidth and CPU usage comparison over the RoCE link: We consider the aggregate application bandwidth and CPU utilization as the primary performance metrics. The performance numbers obtained are as follows. For each test, we transferred 900GB data with both GridFTP and RFTP. The aggregate bandwidth was obtained by collecting the average transfer performance of all streams. To calculate the CPU usage, we employed the "nmon" [29] tool to record the CPU utilization of the application during the entire transfer period, and then determined the average usage. We note that if the host has 12 cores, the total CPU utilization can be up to $12 \times 100\%$.

Figure 8 shows the bandwidth and CPU utilization performance of GridFTP and RFTP over RoCE in LAN, with different block sizes and numbers of streams. We made the following observations:

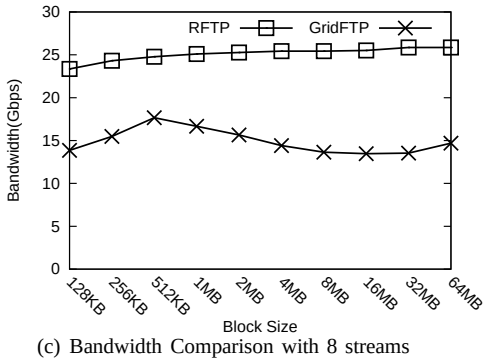
- RFTP saturates the bare-metal bandwidth with different block sizes while CPU utilization declines as the block size increases. Block sizes play an important role in



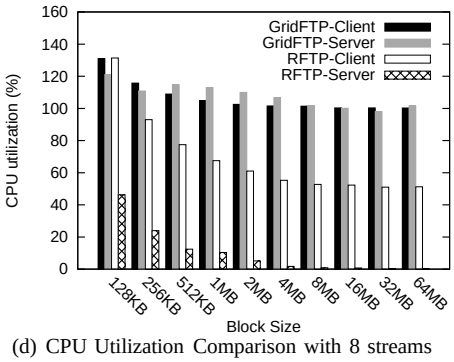
(a) Bandwidth Comparison with 1 stream



(b) CPU Utilization Comparison with 1 stream



(c) Bandwidth Comparison with 8 streams



(d) CPU Utilization Comparison with 8 streams

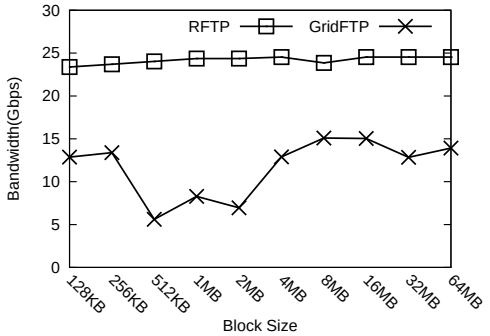
Fig. 8. Bandwidth and CPU Utilization comparison between GridFTP and RFTP over RoCE in LAN

reducing the CPU load, since the number of control messages and CPU interruptions are fewer with larger blocks.

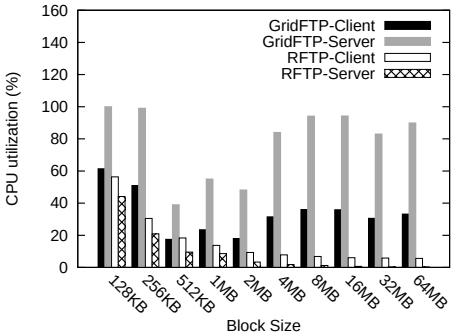
- Although the data transfer application can load data from /dev/zero with a high throughput it generates excessive CPU load to reset the memory content with `0x00`s. We monitored the CPU usage of the data loading thread using the "top" tool, finding that loading data from /dev/zero at 25Gbps leads to a 50% utilization of one core. According to Amdahl's law, the improvement to CPU utilization will be limited if loading data consumes a dominant share of the application's CPU usage. This is the case when the block size exceeds a certain threshold; for example, CPU utilization does not improve further when the block size is increased from 4MB to 64MB.
- A single GridFTP runtime process cannot archive bare-metal bandwidth, even with multiple streams or large block sizes. After we used the application debug tool "strace" to capture the underlying software behavior of the GridFTP application, we found that GridFTP only used a single thread to handle regular file operations, such as reading and writing data, and also network events, such as multiplexing, sending and receiving data. Consequently, good performance was not achieved once a single CPU became the bottleneck. As shown in Figure 8, both the GridFTP client and server always consume more than 100% of the CPU resource in a high bandwidth network environment. Furthermore, GridFTP's performance will be limited by a single core, while RFTP can take advantage of multi-core combined with multi-thread architecture simultaneously to handle more network events for a better transfer performance.

2) Comparison of Bandwidth and CPU usage with the InfiniBand link: Figure 9 compares the bandwidth and CPU utilization between GridFTP and RFTP in the LAN environment with a 40Gbps InfiniBand link. We ran RFTP with one stream and eight streams. We also tested GridFTP with a single TCP connection and eight parallel connections. RFTP consistently outperforms GridFTP and attains high bandwidth in this setting. We also note that with RFTP, the bare-metal bandwidth is almost fully utilized when block size is sufficiently large, for example, 512K bytes. The bare-metal bandwidth is limited by the eight-lane PCI 2.0 (Peripheral Component Interconnect) network adapter.

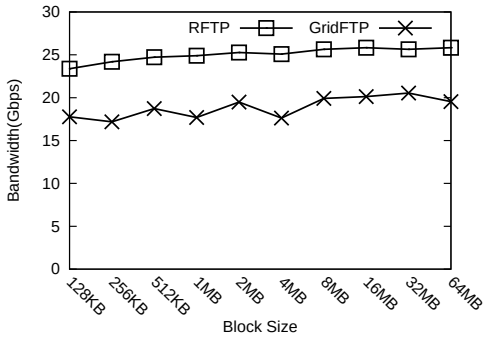
The observations in the previous section also are applicable in the InfiniBand environment. In addition, we made two more observations. First, compared with the results from the RoCE environment, the RFTP consumes less CPU in the InfiniBand environment. The reason is that *libibverbs* has lower overhead in the latter environment than that in the former one. Second, GridFTP's bandwidth performance fluctuates at different block sizes. This instability again might reflect GridFTP's single thread, and CPU power must be split between loading file data and network operations.



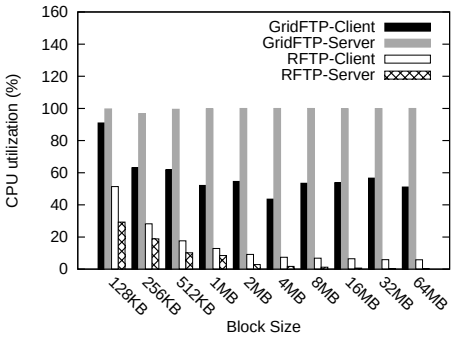
(a) Bandwidth Comparison with 1 stream



(b) CPU Utilization Comparison with 1 stream

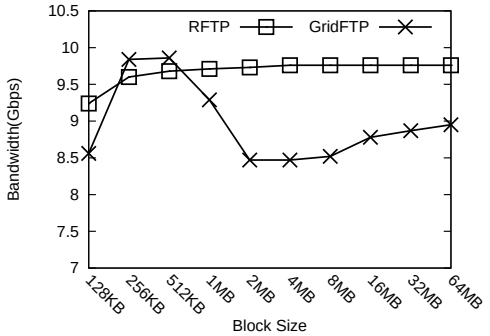


(c) Bandwidth Comparison with 8 streams

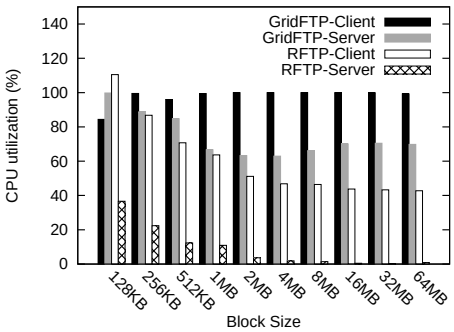


(d) CPU Utilization Comparison with 8 streams

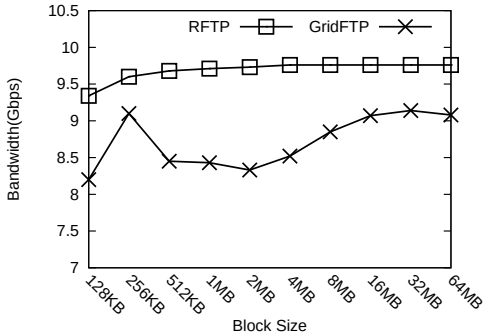
Fig. 9. Bandwidth and CPU Utilization comparison between GridFTP and RFTP over InfiniBand in LAN



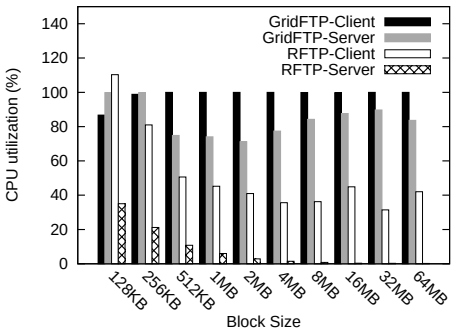
(a) Bandwidth Comparison with 1 stream



(b) CPU Utilization Comparison with 1 stream



(c) Bandwidth Comparison with 8 streams



(d) CPU Utilization Comparison with 8 streams

Fig. 10. Bandwidth and CPU comparison between GridFTP and RFTP over RoCE in WAN

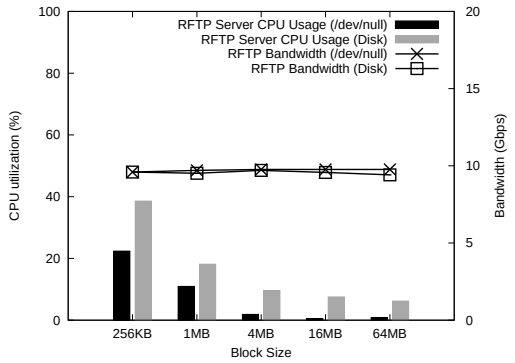


Fig. 11. RFTP Bandwidth and CPU utilization comparison between Memory-to-Memory and Memory-to-Disk

D. Experimental Results over the WAN link

We ran RFTP and GridFTP over the long-haul WAN RoCE link in the ANI testbed (the DOE's Advanced Network Initiative). In this set of experiments, we used both memory-to-memory and memory-to-disk data transfer to demonstrate the efficacy of our protocol design. Figure 10 compared the bandwidth and CPU utilization with one stream and eight streams. In most cases, RFTP again outperforms GridFTP in getting full bare-metal bandwidth with lower CPU utilization. The reason for bandwidth fluctuation of GridFTP is the same as we discussed in the previous subsection.

Figure 11 shows the bandwidth and CPU utilization of the RFTP server in the memory-to-memory and memory-to-disk test cases. We enabled the direct I/O feature of RFTP to save CPU usage and accelerate the RAID disk performance. To the best of our knowledge, GridFTP has not yet integrated direct I/O. Since writing data to disks with standard POSIX I/O consumes much more CPU time than direct I/O, GridFTP's performance is not comparable with RFTP using direct I/O. This figure shows that RFTP maintains the same bandwidth performance between memory and disk tests, with slightly higher CPU usage at the RFTP server since moving data into disk is more CPU intensive than simply writing into /dev/null. Hence, the design of our protocol and application are flexible in various testbed environments, including with disk operations.

VI. Conclusions

RDMA is known as a promising high-performance protocol offload technique that supports zero-copy and kernel bypass. Several factors limit the use of RDMA techniques, including the lack of middleware support to RDMA hardware and the lack of efficient protocols to fully utilize the available network bandwidth. In this paper, we described our study of the design and performance issues of data transfer tools for high-speed networks such as 40 Gbps Ethernet and InfiniBand. Our work provides an RDMA-based middleware layer that provides simple resource abstraction and management, task scheduling, and parallel data transfer. Based on this middleware, we designed a data transfer protocol that supports high performance flow control and parallel data transfer.

To demonstrate the efficiency of our protocol and software design, we developed a reference implementation for the proposed FTP protocol. We set up testbeds with various RDMA technologies in various network environments to cover many different real-life data transfer scenarios. In particular, we demonstrated the performance of our protocol over the Department of Energy's ANI Testbed that includes multiple 10Gbps RoCE links over a 2000 mile path. The experiments show that our protocol and its intelligent design achieved remarkable bandwidth performance and fully maximized the RDMA hardware capacities.

Acknowledgments

The authors are grateful to the facility and hardware donation of Mellanox Technologies, Inc. and Fusion-io, Inc. The authors have benefited from numerous technical discussions with Gilad Shainer, Roi Dayan, Erin Filliater, Yaron Haviv, Bill Lee, Dudu Slama, and Todd Wilde, from Mellanox, Paul Grun and David McMillen from System Fabric Works, Inc., Ezra Kissel and Martin Swany from Indiana University, and David Strohmeyer from Intel. This research is supported by United States Department of Energy, Grant No. DE-SC0003361. The ESnet Advanced Network Initiative (ANI) Testbed, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-05CH11231. Both contracts are funded through the The American Recovery and Reinvestment Act of 2009.

References

[1] W. Feng, P. Balaji, C. Baron, L. N. Bhuyan, and D. K. Panda, "Performance characterization of a 10-Gigabit ethernet TOE," in *Proceedings of 13th Symposium on High Performance Interconnects (HOTI)*, August 2005.

[2] H. Jang, S.-H. Chung, and D.-H. Yoo, "Implementation of an efficient RDMA mechanism tightly coupled with a TCP/IP offload engine," in *Proceedings of International Symposium on Industrial Embedded Systems (SIES)*, June 2008.

[3] N. Bierbaum, "MPI and embedded TCP/IP Gigabit Ethernet cluster computing," in *Proceedings of 27th Annual IEEE Conference on Local Computer Networks*, Tampa, Florida, USA, November 2002, pp. 733–734.

[4] E. Yeh, H. Chao, V. Mannem, J. Gervais, and B. Booth, "Introduction to TCP/IP Offload Engine (TOE)," *10 Gigabit Ethernet Alliance (10GEA)*, October 2002.

[5] OpenFabrics, "OpenFabrics Alliance: http://www.openfabrics.org/," 2012.

[6] ESnet, "Energy Sciences Network: http://www.es.net/," 2012.

[7] Globus Group, "GridFTP online page: http://www.globus.org/toolkit/docs/latest-stable/gridftp/," 2012.

[8] InfiniBand Trade Association, "InfiniBand Architecture Specification," *Release 1.2.1*, 2006.

[9] D. Cohen, T. Talpey, A. Kanevsky, U. Cummings, M. Krause, R. Recio, D. Crupnicoff, L. Dickman, and P. Grun, "Remote Direct Memory Access over the Converged Enhanced Ethernet fabric: Evaluating the options," in *2009 17th IEEE Symposium on High Performance Interconnects (HOTI)*, 2009, pp. 123–130.

[10] The Internet Engineering Task Force (IETF), "RFC 4392 - IP over InfiniBand (IPoIB) Architecture," April 2006.

[11] IBTA, "Infiniband Trade Association. http://www.infinibandta.org/," 2010.

[12] D. Collaborative, "uDAPL: User Direct Access Programming Library. http://www.datcollaborative.org/udapl_doc_062102.pdf," June 2002.

[13] A. Danalis, A. Brown, L. Pollock, and M. Swany, "Introducing gravel: An MPI companion library," in *Proceedings of IEEE International Symposium of Parallel and Distributed Processing (IPDPS)*, Miami, Florida USA, April 2008.
[14] A. Danalis, A. Brown, L. Pollock, M. Swany, and J. Cavazos, "Gravel: A communication library to fast path MPI," in *Euro PVM/MPI 2008*, October 2008.
[15] P. Lai, H. Subramoni, S. Narravula, A. Mamidala, and D. K. Panda, "Designing efficient FTP mechanisms for high performance data-transfer over InfiniBand," in *Proceedings of International Conference on Parallel Processing (ICPP)*, September 2009.
[16] P. W. Frey and G. Alonso, "Minimizing the hidden cost of RDMA," in *Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS)*, June 2009.
[17] N. S. V. Rao, W. Yu, W. R. Wing, S. W. Poole, and J. S. Vetter, "Wide-area performance profiling of 10GigE and InfiniBand technologies," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, November 2008.
[18] W. Yu, N. S. Rao, P. Wyckoff, and J. S. Vette, "Performance of RDMA-capable storage protocols on wide-area network," in *Proceedings of Petascale Data Storage Workshop*, November 2008.
[19] Y. Tian, W. Yu, and J. Vetter, "Rxio: Design and implementation of high performance rdma-capable gridftp," 2011.
[20] H. Subramoni, P. Lai, R. Kettimuthu, and D. K. Panda, "High performance data transfer in grid environment using gridftp over infiniband," in *Int'l Symposium on Cluster Computing and the Grid (CCGrid)*, May 2010.
[21] M. Luo, S. Potluri, P. Lai, Emilio, P. Mancini, H. Subramoni, K. C. Kandalla, S. Sur, and D. K. Panda, "High performance design and implementation of nemesis communication layer for two-sided and one-sided mpi semantics in mvapich," in *ICPPW '10 Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, 2010.
[22] H. Subramoni, P. Lai, M. Luo, and D. K. Panda, "RDMA over Ethernet: A preliminary study," in *Proceedings of Cluster Computing Workshops, CLUSTER'09*, August 2009.
[23] Y. Ren, T. Li, D. Yu, S. Jin, and T. Robertazzi, "Middleware support for rdma-based data transfer in cloud computing," in *Proceedings of High-Performance Grid and Cloud Computing Workshop*, May 2012.
[24] J. Axboe, "Flexible I/O Tester: http://freecode.com/projects/fio."
[25] Mellanox, "Rdma aware networks programming user manual," Jan 2010.
[26] "Performance tuning guidelines for mellanox network adapters," March 2012.
[27] Globus Group, "GT 4.0 GridFTP Glossary: http://www.globus.org/toolkit/docs/4.0/data/gridftp/gridftp_glossary.html," 2012.
[28] Globus Developer Group, "GridFTP Threaded Flavors: http://www.globus.org/toolkit/docs/5.0/5.0.0/data/gridftp/admin/," 2012.
[29] N. Griffiths, "nmon performance: A free tool to analyze AIX and Linux performance: http://www.ibm.com/developerworks/aix/library/au-analyze_aix/."