Improving MPI One-sided Passive Communication Using I/OAT Offload Engines *

T. Gangadharappa, G. Santhanaraman, K. Gopalakrishnan, S. Potluri and D.K. Panda

{gangadha, santhana, gopalakk, potluri, panda}@cse.ohio-state.edu

Department of Computer Science and Engineering, The Ohio State University

Abstract. *MPI has become the de-facto standard for parallel programming model for distributed memory systems. The MPI-2 standard improved upon the MPI-1 by introducing the remote memory access (RMA) semantics. The remote memory access or one-sided communication in MPI parlance provides applications with more capability for communication-computation overlap. The passive synchronization semantics defined by MPI-2 allows for overlap as the target process is unaware of the communication. However, several interconnects (such as QLogic InfiniPath, naive 1/10 Gigabit Ethernet) still lack direct remote memory access interfaces and utilize two-sided primitives to emulate the one-sided interface. In such designs the overlap possible is limited by the inherent limitation of the two-sided designs requiring remote process involvement.*

In this paper we aim to improve the design of the MPI-2 one-sided communication to provide better communication overlap on interconnects that have limited or no RMA capabilities. In particular we focus on: (i) designing the one-sided passive synchronization using helper thread to provide faster communication progress and (ii) offloading CPU intensive memory copy operations to the DMA hardware to achieve memory copy-computation overlap. We demonstrate the benefits of our designs using microbenchmarks. Our results show an improvement in communication overlap of up to 25%, lower large message latency and and improved cache utilization.

Keywords: *MPI-2, Passive synchronization, One-sided operations, I/OAT*

1 Introduction

Scientific computing has seen unprecedented growth in recent years. The demand for computation power is ever on the increase as the scientists try to solve the grand challenge problems. At the same time, the supercomputing field has seen an explosive growth to meet the demands for more and more computational power. The emergence of multi-core processors and growth of high performance interconnects offering very low latency and high bandwidth being the primary contributors to this trend.

* This research is supported in part by U.S. Department of Energy grants #DE-FC02-06ER25749 and #DE-FC02-06ER25755; National Science Foundation grants #CNS-0403342, #CCF-0702675 and #CCF-0833169; Wright Center for Innovation grant #WCI04-010-OSU-0; grants from Intel, Mellanox, QLogic, Cisco, and Sun Microsystems; Equipment donations from Intel, Mellanox, AMD, Advanced Clustering, IBM, Appro, QLogic, and Sun Microsystems.

MPI has become the de-facto programming model for these distributed memory machines. MPI-1 defined the message passing standard with support for point to point and collective communications. The send/receive messaging model is also called the two-sided model. The MPI-2 standard provided further extensions and introduced the remote memory access (RMA) programming models. The RMA model is also known as the one-sided model in MPI terms. In the one-sided model, as the name suggests, ideally only one side is involved in data communication. MPI-2 defined two mechanisms of using the one-sided interface. An (i) active mode: in which the target and source processes synchronize explicitly and (ii) passive mode: in which the target process is unaware of the communication. On modern interconnects such as InfiniBand the one-sided models can be implemented efficiently using Remote Memory Direct Access (RDMA) mechanisms. The passive mode can provide a truly one-sided interface to the application writers. Designing the passive mechanism requires hardware support from the network to perform the operations in a truly one-sided manner inside the middleware library.

However, the RDMA feature is not available in all networking hardware. For instance, the QLogic Infinipath[1] adapters which is used in this work does not support RDMA semantics. In these cases the one-sided interface is designed over a two-sided communication interface internally. In this approach the remote/receiver process has to be involved in moving the data on the destination process. There are two main issues involved in this approach: (i) Since there are no communication calls made on the target process in the passive model, the data movement and synchronizations operations are delayed till a later MPI call invokes the progress engine and (ii) Since the receiver CPU is involved in the data communication, the capability to overlap computation and communication is drastically reduced.

As mentioned above, in such scenarios, the remote CPU is involved in copying or moving the data leading to an inefficient use of the compute cycles. In addition, such operations also affect the caching hierarchy since the CPU fetches the data into cache, thereby evicting other valuable cache entries. The problem gets even worse with the introduction of multi-core systems since several cores can concurrently access the memory leading to memory contention issues, CPU stalling issues, etc. Recently, Intel's I/O Acceleration Technology (I/OAT) [6, 9, 11] introduced an Asynchronous DMA Copy Engine (ADCE) in kernel space that has direct access to main memory.

This leads to the following challenges:

1. What kind of designs can improve asynchronous progress and overlap capabilties?
2. Can the I/OAT mechanisms be used to supplement the lack of network hardware capabilities to reduce the remote processor involvement for one-sided communication operations?

In this work we aim to address all the above issues. To address the first issue we provide a helper-thread based design to ensure quick communication progress. We also provide a copy-offload based design to overlap message copies and computation. Our copy-offload design utilizes the I/OAT technology to provide asychronous data movement on the target process. This reduces the remote processor involvement leading to improved overlap. Finally, we perform an in-depth analysis of our designs with latency, bandwidth, computation-copy overlap benchmarks as well as L2 cache misses and demonstrate the effectiveness of our I/OAT offloaded designs.

The rest of the paper is organized as follows. Section 2 provides a background for this work. Section 3 presents our designs. In Section 4 we present our evaluations.

Finally we look at some related work in Section 5 and provide a conclusion in Section 6.

2 Background

In this section we provide a background of the MPI one-sided communication model and the I/OAT DMA engine.

2.1 MPI one-sided communication

In MPI one-sided communication (also referred to as remote memory access or RMA), the origin process (the process that issues the RMA operation) can access a target process' memory address space directly. The origin process provides all the parameters such as target rank, target memory address, target datatype, etc. The memory area operated on is known as the *window* in MPI parlance. MPI defines three one-sided operations: MPI_Put, MPI_Get and MPI_Accumulate. The completion of the one-sided operations is ensured by using synchronization primitives. MPI semantics allows one-sided operations only within an *epoch*, which is the period between two synchronization events. MPI provides three ways of synchronization of which two are active synchronizations i.e. they require both origin and target to synchronize via a collective operation. It also provides a passive synchronization by which the origin process can lock a target *window*, perform one-sided operations and unlock the *window*. In the passive synchronization mode, the origin process alone makes the synchronization and issues data transfer operations. The remote process is not involved at all. This type of synchronization is very effective on large scale systems as it can minimize the coordination between the origin and target process. This kind of synchronization gives greater potential for computation/communication overlap. In this paper, we primarily concentrate on the passive mode of synchronization.

In this work, we mainly focus on designing MPI one-sided communication for networks that do not support one-sided RDMA semantics. As a case study for this work, we use the Infinipath [1] network that doesn't expose an RDMA (Remote Direct Memory Access) interface, unlike traditional verbs that support both send/receive as well as RDMA semantics.

2.2 Interconnect

InfiniBand overview The InfiniBand architecture (IBA) defines a switched network fabric for interconecting processor and I/O nodes. An InfiniBand network is made up of switches, InfiniBand adapters (called Host Channel Adapters) and media for communication. InfiniBand defines multiple transport classes: Reliable Connection (RC), Unreliable Connection (UC), Reliable Datagram (RD), Unreliable Datagram (UD).

Infinipath HCA overview Infinipath HCA provides a light-weight communication layer (called the PSM API layer) for designing the MPI interface. Infinipath uses a connectionless model for communication. Infinipath HCA's do not use DMA engines to send/receive data and thus do not require memory pinning.

Infinipath HCA's do not contain an embedded processor and all protocol operations are performed by the host processor. This additional burden means the host processor is not available for application processing. Our design explores ways by which we can alleviate the load on the host processor by offloading the copy operations specific to one-sided communications.

2.3 I/OAT

Intel I/O Acceleration Technology (I/OAT) is an I/O acceleration technology developed by Intel and available in most modern Intel Chipset. The technology provides a DMA engine to offload data-movement and reduce CPU overhead. On chipsets with this feature, the I/OAT device is a PCI resource with a respective I/OAT DMA driver. The I/OAT DMA engine can be used as copy-offload engine, allowing the processors to perform useful tasks.

As mentioned in [10], using a copy-offload engine provides a reduction in CPU usage and yields better performance. Copy engines can move data in blocks larger than word-size and hence can provide higher performance on larger data sets. Also, since the memory copy progresses asynchronously with computation using the offload engine, it provides copy-communication and also reduces cache pollution.

3 Designs

In this section we discuss the challenges and design issues for implementing passive synchronization based MPI one-sided interface over an interconnect that provides two-sided send/recv primitives.

First we describe a basic design that is very commonly used to support MPI over Ethernet networks. For instance, the MPICH2 [3] implementation uses a similar two-sided approach as a generic solution for all interconnects.

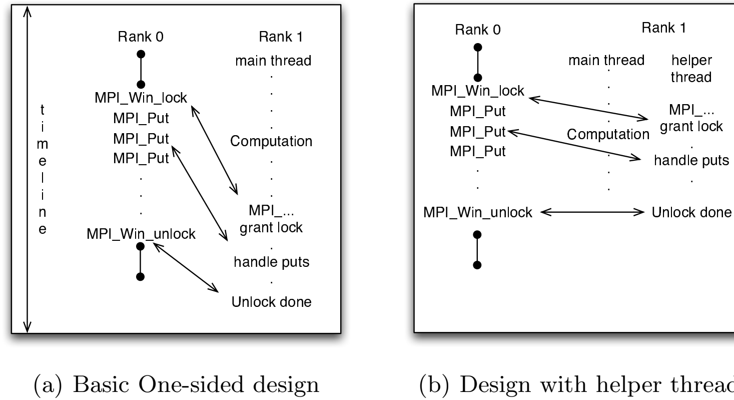
3.1 Basic Design

The basic design for the passive one-sided interface using a two-sided model requires active involvement of the target process. Since some interconnects lack RDMA and remote atomic locking capabilities, designs on such interconnects use a a two-sided protocol. Figure 1(a) shows a one-sided operation being performed from rank 0 to rank 1. Rank 0 initiates the passive one-sided operations by issuing the window lock operation. A lock message is sent to rank 1 and rank 0 waits for the lock to be granted. Once the lock is granted, rank 0 performs the *MPI_Put's* followed by the unlocking of the window. MPI semantics ensures that only after all the Put's are complete, the lock is released. All the one-sided messages are received into pre-posted buffers and then copied to the actual location in the target window.

Figure 1(a) clearly shows the problem with this design. The target rank is performing computation and only when it enters the MPI library does it respond to the lock protocol. This causes a large sender latency and also provides zero overlap of computation-communication on the receiver side.

3.2 Design with Helper Thread Mechanism

The basic design presented in Section 3.1 does not provide any communication-computation overlap and has a high sender overhead. The target rank does not enter the MPI library and hence the origin rank does not progress with the one-sided communication. One solution to alleviate this issue is shown in 1(b). In this design we have an additional helper thread running in the MPI library. With this design, if the main thread is not in the MPI library, the helper thread can ensure progress. This enables the one-sided communication to progress immediately. Like the basic design, the lock message is first received into pre-posted buffers. But due to helper thread, the lock is granted immediatly. Subsequently the Put's and the unlock operations are also handled by the helper



(a) Basic One-sided design (b) Design with helper thread

Fig. 1. Basic One-sided Passive Design & Optimization

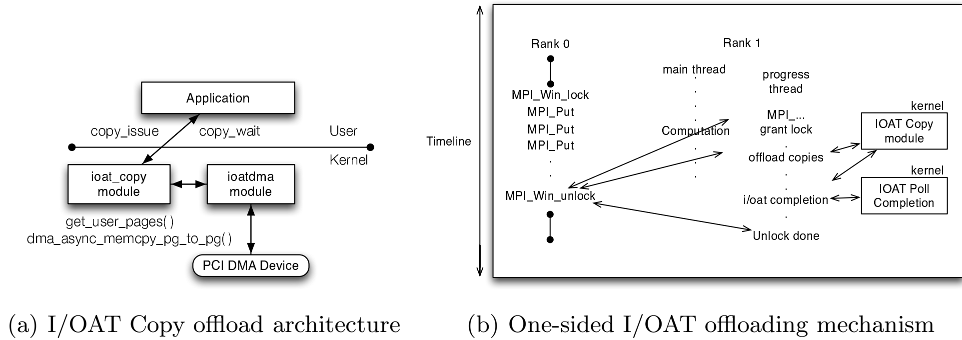
thread. The helper thread copies out the received Put's from pre-posted buffers into target memory windows. In our threaded design we also address the following design issues:

- One major demerit of this approach is the CPU contention between the main thread and the helper thread. We resolve this by cancelling the helper thread (*pthread_cancel*) if the main thread enters the MPI library. When the main thread exits the MPI library the helper thread is re-created. This ensures no lock or CPU contention, the relatively cheap cost of creating a new thread makes this approach feasible.
- The helper thread actively polls for communication progress. This would cause CPU contention with the main thread. To resolve this we keep the completion polling less aggresive in the helper thread. The helper thread sleeps intermittently if no completion event is detected. If a completion event is detected, we aggressively complete the processing of the polled event.
- One issue with this design is that if the amount of data transferred is very large, the helper thread will spend a lot of time performing memory copies. The memory copy is unavoidable without RDMA support. We try to solve this by using the I/OAT copy offloading. Using the I/OAT engine for copying large data also prevents the cache from being polluted thus enabling the foreground computation to proceed without cache effects.

3.3 Design with I/OAT

We improve upon our design in Section 3.2 using the I/OAT offload engine.

I/OAT Copy offload: The architecture of the copy offload is shown in Figure 2(a). The copy offload engine is implemented as a Linux kernel module. User space applications can issue copy requests to the module via an ioctl call. The kernel queues the requests unless an explicit issue is performed, allowing multiple requests to be batch issued. On issueing a copy, the dma-engine provides a cookie that can be polled



(a) I/OAT Copy offload architecture (b) One-sided I/OAT offloading mechanism

Fig. 2. I/OAT Based Design

for completion. The *ioat_copy* module locks the user memory space using the kernel API *get_user_pages* and issues one DMA operation per page. The I/OAT DMA kernel interface exposes the *dma_async_memcpy_pg_to_pg* interface which issues one copy request per page. The cookies returned by the DMA engine are stored for future polling. Completion of all the issued copies are ensured by a single system call similar to MPI_Waitall, which returns only when all outstanding copies are complete.

One-sided with I/OAT offload: Figure 2(b) shows the design of the one-sided interface with the I/OAT copy offload. In this design, the one-sided operations are not issued by the origin at the time of the lock operation. Instead the origin rank queues all the issued one-sided requests and delays issue until the unlock. At the unlock stage, the origin process knows the exact size and number of operations in the queue. The lock request now performs two tasks: (i) it requests for a lock from the target rank and (ii) it informs the target rank the exact number of operations it will issue. Prescence of the helper thread enables us to process the lock request immediately. Additionally, the helper thread maintains the count of the incoming one-sided requests. Incoming Put's and Accumulate's are issued for memory copy to the I/OAT module. Knowing the count of the one-sided operations allows us to batch multiple DMA issues in a single system call. Once all the one-sided operations from our origin rank is processed, the I/OAT engine is polled for completion and the lock is released.

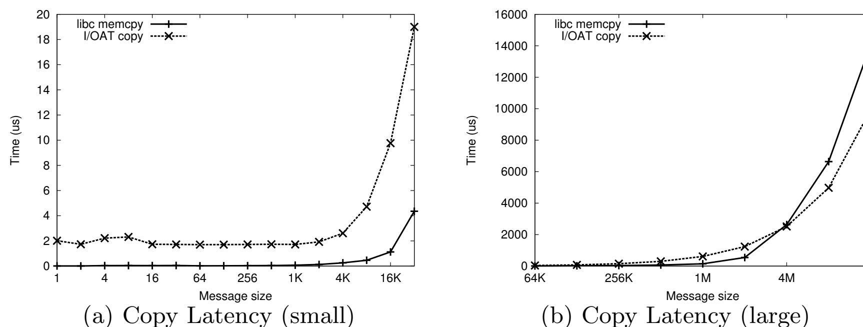
Currently, I/OAT does not expose an interrupt driven completion semantic. Thus we need to perform active polling to check for completion, however, polling is not required for progress. Since polling is done only to detect completion, the operation is kept lightweight. I/OAT completion semantics provides us the last completed cookie. Since we internally maintain the list of pendings requests we can use this informatio to calculate the number of outstanding DMA requests. If any pending requests exist, the polling is suspended and the thread is put to sleep for a small time interval. Upon waking the thread again checks for pending requests and iterates until it finds all requests to have completed. Once all the DMA requests are completed the lock is released and the origin rank is informed. Since, most of the data movement is handled by the I/OAT engine and completion polling requires only a single system call by the helper thread, the main thread can proceed with computation with minimal overhead.

4 Performance Evaluation

In this section we present the experimental evaluation of our designs. The testbed used is an Intel cluster. Each node is a dual processor (2.33 GHz quad-core) system running an Intel 5000X Chipset with I/OAT support. Each node has 4 GB main memory. The CPUs support the EM64T technology and run in 64 bit mode. The nodes support 8x PCI Express interfaces and are InfiniPath_QLE7140 HCAs with PCI Express interfaces. The operating system used is RedHat Linux 2.6.18-92. We implement our designs in the MVAPICH2 [5] library.

4.1 I/OAT Microbenchmarks

In this section we present the basic I/OAT performance figures. Figure 3(a) shows the I/OAT memory copy latencies for small sizes. The I/OAT copy offload has an initial overhead which leads to sub-optimal performance for small copies. However, as seen in 3(b) the performance of the copy offload is better for large data sizes. For data sizes of 4MB and above the I/OAT provides a better latency. We believe this is due to the fact that I/OAT is able to move data in blocks while *memcpy* can move data only word size at a time.



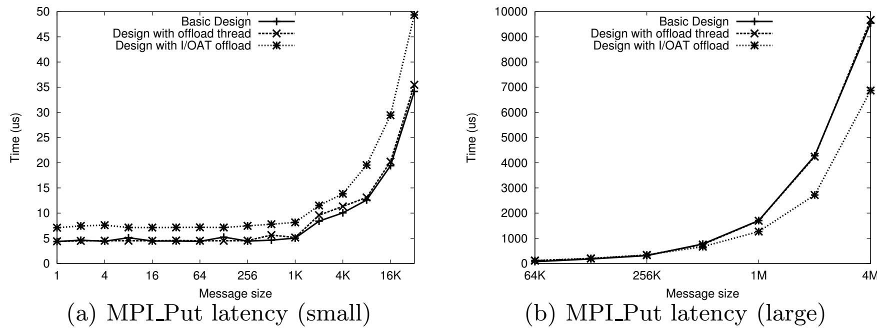
(a) Copy Latency (small) (b) Copy Latency (large)

Fig. 3. I/OAT Copy Latency

4.2 MPI Benchmarks

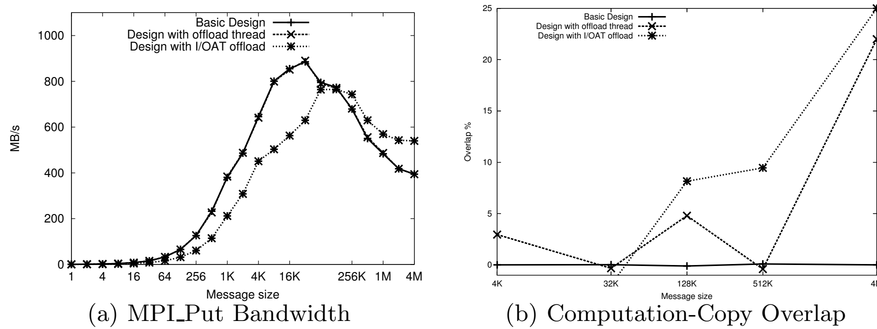
Figures 4(a) and 4(b) show the MPI_Put latency with the three designs discussed. The MPI_Put latency measurement was done without any computation on the receiver side. The basic design and the design with helper thread perform very similarly. We do not see any significant overhead incurred due to the thread cancellation in the helper thread based design. With the I/OAT design, small messages suffer from a higher latency due to the higher cost of issueing and completing the DMA request. However, since multiple DMA requests can be issued for a copy the overall latency per Put operation starts to improve with our I/OAT design. For large message sizes, Figure 4(b) clearly shows significantly lower latencies. At 4MB message sizes, the I/OAT provides upto 30% lower latency than the basic design.

We measure the MPI_Put messaging bandwidth using passive synchronization. Figure 5(a) shows the bandwidth achieved by the three designs. For small messages the basic design and the helper thread design perform well. The I/OAT design has a lower



(a) MPI_Put latency (small) (b) MPI_Put latency (large)

Fig. 4. MPI_Put Latency



(a) MPI_Put Bandwidth (b) Computation-Copy Overlap

Fig. 5. Bandwidth and Overlap Measurements

bandwidth due to higher overhead incurred due to the system call and the DMA polling. However, for larger messages (> 32k) we can see the benefits of the new design. The benefits start as low as 32K because multiple Put operations are concurrently issued for DMA. The I/OAT DMA engine can assure progress of multiple DMA requests and this allows concurrent copy operation corresponding to the Puts, as opposed to the serialized copy of all the Puts in the other two designs.

4.3 Computation-Communication Overlap

In this section we measure the communication-computation overlap obtained by our designs. In our design, the helper thread is always assigned to the same CPU as the main thread. However, using other CPU's in the multi-core systems to run the helper thread is not a realistic solution, hence this limits the overlap achievable. The overlap experiment, considered the time spent by the receiver without computation as the basic communication time. We introduce a comparable amount of computation in the receiver process in the overlap test. If the overall latency of the receiver does not change it signifies a 100% overlap of the computation and the communication. As seen in Figure 5(b), with the basic design, there is zero overlap and this is expected.

With the helper thread based design, due to low transmission time of small messages, the overlap achieved is minimal. But for higher message sizes (> 128K), because the helper thread ensures progress the lock is acquired quickly and the sender can

initiate all the data transmissions. This transmission time is now overlapped with computation time. With the I/OAT based design, the tranmission time is overlapped with computation time as in the previous case. Additionally, the copy operations progress concurrently with the computation, thus giving a significantly higher overlap value. We see upto 25% with 4MB message sizes using the I/OAT design.

4.4 Effect on Caches

To evaluate the effect of our designs on the caches we designed an experiment to measure the L2 caches misses incurred. The experiment performs passive synchronization based one-sided operations to a target rank, while the target rank performs a computation. We measure the L2 cache misses seen by the computation loop. At the end of each loop of the computation the two ranks synchronize and repeat the computation and one-sided communication. We use the *oprofile* Linux profiler to measure the L2 cache misses incurred.

As seen in Table 1 we see that the basic design has a high L2 cache miss rate. Due to the iterative nature of the test we see that the program suffers from the compulsary misses during the computation loop. However, completing the one-sided communication at the synchronization point pollutes the cache and some of the compulsary misses reoccur for next iteration. The helper thread based design, has a high cache miss rate due to the same L2 cache being shared by two threads. Each thread brings in data to the cache that is evicted by the other thread. The third column shows the lowest cache miss numbers, for the design with I/OAT offload. The primary reason being I/OAT moves data from memory to memory and hence the CPU caches remain unaffected. The computation loop alone without any communication suffers from a total of 358,000 L2 misses. This experiment clearly demonstrates the efficiency of the I/OAT offload design with respect to CPU caches.

Table 1. L2 Cache misses

Message size	Basic design	Design with helper thread	Design with I/OAT
32K	591,500	889,000	376,000
256K	467,500	849,500	396,500
1M	692,500	668,000	374,000
4M	799,500	555,500	366,000

5 Related Work

Several MPI-2 implementations support the one-sided communication model, MPICH2 [3], MVAPICH2 [5], OpenMPI [4], are some of the open-source implementations of MPI-2. Thread based design of passive synchronization have been proposed in [9], however their designs do not solve the CPU contention issue when the main progress thread executes the communication progress. Also, the naive usage of an helper thread, leads to the helper thread consuming CPU even when no one-sided communication exists. Our approach defines the helper thread as a generic progress thread that can progress any communication event seen on the network. Asychronous progress for rendezvous communication using a thread based approach has been studied in [2].

I/OAT feature has been used to achieve asynchronous memory copy in the context of data-centers [8, 6, 7]. The Linux TCP/IP receive stack introduced the Network DMA feature using I/OAT to reduce server overheads. Our work differs in the aspect

that we use I/OAT to offload data movement on the target side in MPI-2 one-sided communication.

6 Conclusions

The one-sided (RMA) communication model in MPI provides one-sided semantics to the application writers. The passive one-sided communication mode can minimize the coordination between the origin and target process and can ideally provide good computation communication overlap. However, this requires hardware support from the networks in the form of RDMA read/write and remote locking capabilities. When the network cannot provide such capabilities, the implementation is usually done on top of two-sided semantics leading to sub-optimal performance.

Thus efficient designs of the one-sided interface needs to be explored on RDMA-incapable networks. In this paper, we present a common basic design of implementing the one-sided interface over two-sided communications. We extended the basic design by proposing a new helper thread based design to ensure quick communication progress in passive one-sided models and an I/OAT copy offload based design to alleviate CPU consumption. The experimental evaluations showed significant performance benefits for large messages when I/OAT offloading is used. Using the I/OAT engine provided the added benefit of keeping the caches unpolluted, a major side-effect in *memcpy* based designs. In this work we use Infinipath network as a case-study, however the designs presented are generic and are applicable to other RDMA-incapable interconnects (such as naive 1/10 Gigabit Ethernet) and systems with DMA based copy support.

For future work we plan to extend our design to utilize an interrupt driven I/OAT completion semantics (which current I/OAT drivers do not support). Such an extension will further reduce CPU consumption. We also plan to evaluate the effectiveness of our designs on applications by developing one-sided versions of common scientific applications.

References

1. QLogic Corporation. http://www.qlogic.com/default.aspx/.
2. R. Kumar, A. Mamidala, M. Koop, G. Santhanaraman, and D. K. Panda. Lock-free Asynchronous Rendezvous Design for MPI Point-to-point communication. In *EuroPVM/MPI 2008*, September 2008.
3. MPICH2. http://www.mcs.anl.gov/research/projects/mpich2/.
4. OpenMPI. http://www.open-mpi.org/.
5. MVAPICH2: High Performance MPI over InfiniBand and iWARP. http://mvapich.cse.ohio-state.edu/.
6. K. Vaidyanathan, L. Chai, W. Huang, and D. K. Panda. Efficient Asynchronous Memory Copy Operations on Multi-Core Systems and I/OAT. In *International Conference on Cluster Computing*, 2007.
7. K. Vaidyanathan, W. Huang, L. Chai, and D. K. Panda. Designing Efficient Asynchronous Memory Operations Using Hardware Copy Engine: A Case Study with I/OAT. In *CAC*, 2007.
8. K. Vaidyanathan and D. K. Panda. Benefits of I/O Acceleration Technology (I/OAT) in Clusters. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2007)*, 2007.
9. H. W. Jin D. K. Panda D. Buntinas R. Thakur Weihang Jiang, J. Liu and W. Gropp. Efficient implementation of mpi-2 passive one-sided communication on infiniband clusters.
10. Li Zhao, R. Iyer, S. Makineni, L. Bhuyan, and D. Newell. Hardware support for bulk data movement in server platforms. pages 53–60, Oct. 2005.