Designing MPI Library with On-Demand Paging (ODP) of InfiniBand: Challenges and Benefits

Mingzhe Li, Khaled Hamidouche, Xiaoyi Lu, Hari Subramoni, Jie Zhang, and Dhabaleswar K. Panda
Department of Computer Science and Engineering, The Ohio State University
{limin, hamidouc, luxi, subramon, zhanjie, panda}@cse.ohio-state.edu

Abstract—**Existing InfiniBand drivers require the communication buffers to be pinned in physical memory during communication. Most runtimes leave these buffers pinned until the end of the run. Such situation limits the swappable memory space for applications. To address these concerns, Mellanox has recently introduced the On-Demand Paging (ODP) feature for InfiniBand. With ODP, communication buffers are paged in when they are needed by the HCA and paged out when the OS needs to swap them. This paper presents a thorough analysis on ODP and studies its performance characteristics. With these studies, we propose novel designs of ODP-aware MPI communication protocols. To the best of our knowledge, this is the first work to study and analyze the ODP feature and design an ODP-aware MPI library. Performance evaluations with applications show that ODP-aware designs can reduce the size of pin-down buffers by 11X without performance degradation compared with the pin-down scheme.**

I. Introduction

High-performance interconnects have been the key drivers for high-performance computing systems. A key feature of today's high-performance networks is Remote Direct Memory Access (RDMA) which is able to maximize the computation and communication overlapping in applications to get optimal performance. RDMA is supported by networks such as InfiniBand [10], Aries [9], Gemini [3], Quadrics [16], IBM's BlueGene/Q [11], PERCS [4] and even RDMA over Converged Ethernet [5].

Most of these networks require all buffers which are to be made accessible for RDMA to be explicitly pinned in memory before the transfer can proceed. One big challenge with such a pin-down scheme is to achieve efficient communication buffer management. The cost of memory registration dramatically degrades the performance of RDMA and increases network latency in the critical paths of data transfers. The other issue is that this kind of network could require huge amount of memory for scalable communication. Parallel programming models such as MPI and PGAS running on such networks need to handle these challenges to deliver the best performance for applications.

Many studies [7, 12, 14, 20] have been made to reduce the overhead of memory registration and size of registered memory on high-performance interconnects. One common approach is incorporating a pin-down cache with the runtime systems. However, this design still needs to pin-down communication buffers, such as bounce buffers for two-sided data transfers and pinned user buffers for one-sided transfers. These pin-down buffers may consume a large amount of memory resources which leads to limited swappable memory space for computation in scientific applications. With modern multi/many-core platforms that are evolving very rapidly with 32/64 cores per node, the total size of pin-down buffers is expected to be higher compared to current generation systems, however per core memory resource is expected to decrease. This makes the memory contention between computation and communication even worse.

InfiniBand (IB), as one of the pinning-based network, also has the short-comings of the pin-down scheme. Recently, Mellanox has introduced a new feature named On-Demand Paging (ODP) that alleviates the shortcomings of pin-down memory on IB networks. The ODP feature brings the following benefits: (1) buffers for RDMA operations can be automatically paged in when the Host Channel Adapter (HCA) needs them and paged out when the OS needs to swap them, (2) overhead of pinning/unpinning buffers could be removed, and (3) size of registered buffers could be larger than physical memory size. Thus, the ODP feature could be a promising alternative solution for scalable communication on IB networks.

Inside an MPI stack which supports IB, there are many protocols and algorithms for point-to-point and one-sided primitives. All these designs are proposed under the assumption that communication buffers for RDMA operations have already been loaded and pinned in memory. With the ODP feature, a buffer for RDMA communication may not even be loaded into memory when the HCA tries to access it. What makes the situation worse, many high-performance scientific applications need a large amount of memory resources for computation. When these applications are running, even if some communication buffers have been loaded into physical memory, these buffers could potentially be swapped out when the computation needs more physical memory resources. When this happens, extra operations will be added to critical paths in data transfers by resolving page faults. All these features and implications of ODP may have a significant impact on MPI library designs for point-to-point and one-sided operations, but this has not yet been explored in the literature.

Furthermore, many algorithms and principles inside MPI library are guided by the LogGP [1] model. LogGP model

*This research is supported in part by National Science Foundation grants #OCI-0926691, #OCI-1148371, #ACI-1450440 and #CCF-1213084.

has been well studied for MPI communications with the pin-down scheme. With the emerging ODP feature, when a process issues an RDMA operation, the overhead of transferring a message will vary based on whether page fault is happening or not. In such case, should the LogGP model be adjusted to study RDMA communication performance with ODP? What parameters or patterns need to be modified in LogGP for modeling communications with ODP?

To fully understand all these unexplored questions, an ODP-aware MPI runtime must be designed by taking these aspects into consideration. For doing this, the following broad challenges need to be well investigated:

- What are the performance characteristics and behaviors of ODP at IB verbs-level? What kind of verbs-level benchmarks should be designed?
- What is the overhead of dynamically paging in/out communication buffers? How to alleviate or eliminate the overhead in critical paths for data transfer when the communication buffers for RDMA operations are paged out by the OS?
- Can we directly use or extend LogGP model for MPI communications with ODP?
- Can we come up with new protocols or algorithms for MPI point-to-point and one-sided primitives with ODP?

In this paper, we design a high-performance ODP-aware MPI runtime which tackles all the above challenges. To summarize, this paper makes the following contributions:

- Conduct a comprehensive performance evaluation of ODP performance. Analyze the performance results using performance statistics reported by the HCA. Design a set of verbs-level miro-benchmarks.
- Extend LogGP model for ODP to model expected communication time of all MPI point-to-point and one-sided primitives.
- Discuss all algorithms and protocols used in MPI point-to-point and one-sided primitives and analyze the performance and memory trade-offs using ODP. Propose new communication protocols for all MPI blocking/non-blocking point-to-point and one-sided primitives.
- Systematically evaluate all our proposed designs with MPI mirco-benchmarks and ten real-world applications. Provide insights of performance results.
- Evaluate our out-of-core verbs-level benchmark to show scenarios where better performance and reduced size of pin-down buffers could be achieved by ODP compared with existing pin-down scheme.

Experimental evaluations show that our proposed designs are able to reduce the size of pin-down buffers for LAMMPS by 11X while achieving comparable performance as current pin-down scheme using 128 processes. In addition, with our own verbs-level benchmark, the ODP scheme could outperform existing pin-down scheme by 4X. To the best of our knowledge, this is the first research work that studies and proposes high-performance ODP-aware MPI runtime on InfiniBand clusters.

II. Background

In this section, we describe the pin-down scheme and the new ODP feature.

A. Pin-down Scheme

Pin-down scheme requires the user to explicitly set up memory regions for RDMA operations. This setup procedure translates into marking the relevant memory pages as non-pageable in physical memory. Pinning user-level virtual memory notifies the OS that the corresponding underlying physical pages cannot be swapped out until the application terminates or explicitly unpins them. Due to this restriction, the upper bound on the amount of memory that can be pinned at one time is limited by the size of physical memory.

B. On-Demand Paging

On-Demand Paging (ODP) is a new feature introduced by Mellanox to register communication buffers for RDMA operations. This feature is dependent on IB hardware, driver, and OS kernel version. ODP allows the communication buffers automatically be paged in when the HCA needs them and paged out when the OS needs to swap them. Compared with the pin-down scheme that requires all pinned buffers fitting into physical memory, ODP allows the size of registered memory regions to be larger than the size of physical memory.

ODP can be further divided into 2 sub-classes: Explicit and Implicit ODP. Explicit ODP means that applications explicitly specify the memory regions used with ODP. Implicit ODP means that applications specify its complete address space used with ODP. Since existing implicit ODP only allows local access to prevent security issues, we only focus on explicit ODP in this paper. In the following, when we use ODP, we refer to explicit ODP.

1) Page Fault and Invalidation: On IB networks, users post I/O virtual address to the HCA, which handles the translation from virtual address to physical address, for RDMA operations. This implies that when the HCA performs RDMA operations, it needs mapping information from virtual addresses to physical addresses. However, when a new buffer is allocated and registered, it is possible that this buffer has not been loaded into memory nor the mapping from virtual to physical address of this buffer exists in the HCA. In this case, when the HCA attempts to access this memory region, a page fault event will be generated by the HCA, which will be handled by the IB driver and OS. Figure 1 shows the flow to handle a page fault event. In the first step, it searches for the registered memory regions associated with the I/O virtual address which the HCA attempted to access. In the second step, it finds the pages for this registered memory region. In the third step, it brings these pages into memory and maps these pages to the HCA. After that, the driver notifies the HCA with the new mapping information. When the mapping is updated in the HCA, suspended communications start to proceed.

Page Invalidation happens when the OS needs to release some pages that have been registered by the HCA. When this

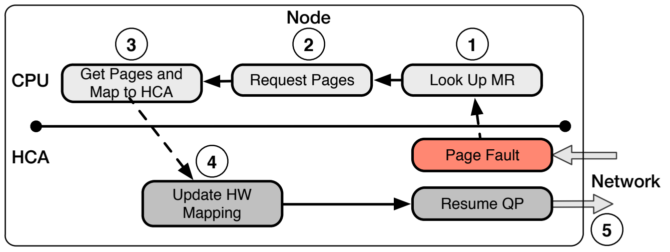


Fig. 1: Steps to Handle Page Fault Event

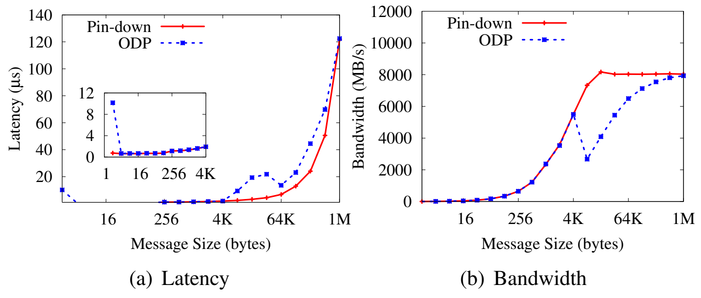


Fig. 2: IB Verbs-level Latency and Bandwidth with ODP

happens, an invalidation notification event is generated by the OS and sent to the IB driver. After the IB driver receives this notification, it will look for intersecting pages and notify the HCA about the invalidation request. The HCA flushes the hardware caches to invalidate these pages. After flushing the HCA's page table caches, the OS could swap these pages.

2) Page Pre-fetching: Page pre-fetching is designed to warm up new mappings by bringing buffers into memory and updating mapping from virtual address to physical address in the HCA. Multiple pages could be pre-fetched at once. The pre-fetch operation is an asynchronous verbs-level call and it is a best-effort hint. Pages pre-fetched by this verbs-level call are not guaranteed to remain resident in memory. They could be swapped out when the OS wants to reclaim these pages.

III. Evaluating ODP Performance and Insights

In this section, we use IB verbs-level point-to-point latency and bandwidth benchmarks to show the performance one can get with the ODP feature. We also evaluate the overhead of pre-fetch operation and the overhead of resolving page fault events, respectively. These experiments are conducted on the platform described in Section VII-A.

A. Latency and Bandwidth Performance with ODP

In this subsection, we aim to understand the basic performance of ODP for communication and compare its performance with existing pin-down scheme. We come up with two verbs-level benchmarks for latency and bandwidth experiments. For both benchmarks, we only allocate one send/receive buffer the size of which is equal to the max message size. We report numbers that are an average of 1,000 iterations. Figure 2(a) shows the verbs-level ping-pong latency results. From the results, we could see that the ODP scheme has a big degradation at 1 byte message size as well as message sizes larger than 8 KB. The same trend can be seen in Figure 2(b). On our systems, the page size is 4 KB that means for message sizes from 1 byte to 4 KB, all the communications only use one page on each side. Mellanox provides an extensive set of statistics reported as sysfs entries to help analyze the performance of applications running with ODP. We gather some statistics to analyze the performance degradation here. With the latency benchmark, we see that there is a page fault event for the first iteration of 1 byte message size, this is because the HCA does not have the mapping information for this communication buffer in the first iteration. Starting from 8 KB message size, page fault starts to happen for all following messages and these page faults lead to the performance degradation. At the bandwidth benchmark with 1MB message size, we see that the ODP scheme and the pin-down scheme perform the same; the reason is that the network is saturated by sending large messages with both schemes.

B. Page Fault Overhead

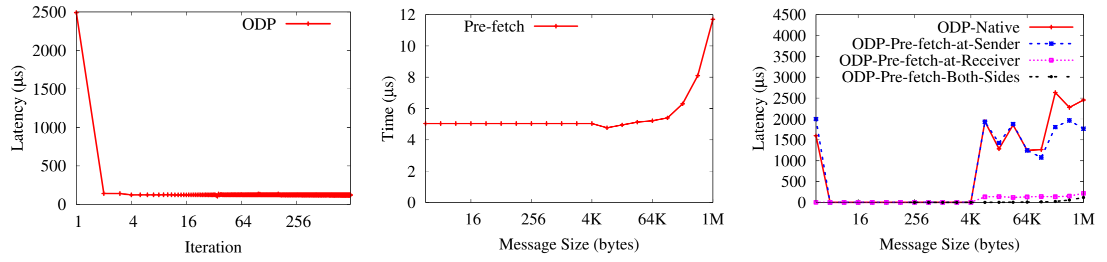
In this subsection, we aim to measure the overhead of a page fault event. We run our latency benchmark with 1MB message sizes and report the latency in each iteration. These numbers are shown in Figure 3(a). We observe two things from the results. The first thing is that the first iteration latency is significantly worse than the following iterations. The second thing is that after the HCA and CPU resolve the page fault event, following iterations achieve comparable performance in terms of latency as the default pin-down scheme.

C. Pre-fetch Overhead

In this subsection, we aim to measure the overhead of a pre-fetch operation. In our verbs-level benchmark, we issue a pre-fetch operation with the same memory region in 5 iterations. We run this benchmark from 2 bytes to 1 MB message sizes and report the average latency of each message size. These numbers are shown in Figure 3(b). Compared with the overhead of resolving a page fault event, issuing a pre-fetch operation is much lighter.

D. Pre-fetch at Sender/Receiver

In this subsection, we conduct experiments to evaluate the verbs-level pre-fetch call. We modify our latency benchmark for this experiment and report numbers of our ODP-based schemes. The first scheme uses the ODP feature without pre-fetch. The second scheme uses the pre-fetch call at the sender side. The third scheme uses the pre-fetch call at the receiver side. The fourth scheme uses the pre-fetch call both at the sender side and the receiver side. The experimental results are shown in Figure 3(c). There are several interesting observations that can be found from these results. First, we could see that the ODP-Pre-fetch-Both-Sides could significantly reduce the latency compared with the ODP-Native scheme. It means that the verbs-level pre-fetching call could work very effectively to load data into memory and update virtual address to physical address mapping in the HCA. Second, there is a big performance difference between the ODP-Pre-fetch-at-Sender



(a) Per Iteration Latency with 1 MB Message Size

(b) Page Pre-fetching Overhead

(c) Page Pre-fetching at Sender/Receiver

Fig. 3: Evaluation of Page Fault Overhead and Pre-fetch Overhead

and ODP-Pre-fetch-at-Receiver schemes. Based on [8, 18], we think the reason is that page fault events are handled differently at the sender side and the receiver side. When page fault happens at the sender, communication has not started yet. The sender only needs to handle the page fault without interacting with the receiver. This is shown as ODP-Pre-fetch-at-Receiver in the figure. However, when the page fault happens at the receiver side, the receiver first needs to notify the sender that a page fault has occurred. After that, the receiver starts to handle the page fault. After the page fault is resolved, the receiver can resume the suspending QP and restart communication. This is ODP-Pre-fetch-at-Sender shown in the figure.

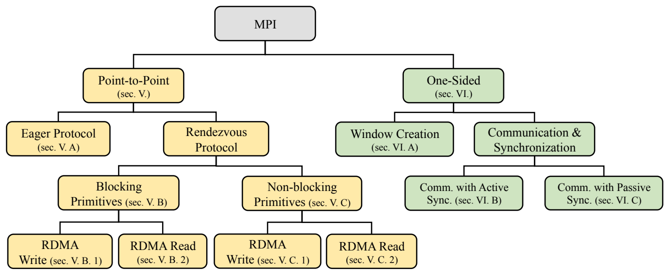
In summary, there are several things we learned from these IB verbs-level experiments:

- Page faults add significant overhead to data transfers.
- The verbs-level pre-fetching call could work very effectively to load pages and update mapping from virtual address to physical address.
- The overhead of page fault event at the sender side is different from that happening at the receiver side.
- Data transfer in several iterations only has overhead when page fault happens.

IV. Challenges in Using ODP at the MPI Level

To design an MPI stack taking advantage of ODP, there are two straightforward designs one could think of. The first design is blindly replacing existing pin-down design in an MPI stack with ODP. With the verbs-level performance numbers, we see that there are some overheads of using ODP. This naive design with ODP should expect huge performance degradation compared with the pin-down scheme. The second design is taking advantage of the pre-fetch operation to alleviate some of the page fault overheads. The pre-fetch operation could be used immediately after a user buffer is registered. This design could add the overhead of pre-fetch operation to critical paths of data transfers.

A high-performance ODP-aware MPI runtime should deliver equal or better performance than the pin-down scheme while reducing the size of pin-down buffers. In order to achieve that, one has to carefully analyze all the algorithms and protocols used in MPI primitives. We use a combination of in-depth analytical model of communication primitives and empirically observed communication behaviors to apply ODP in



MPI
Point-to-Point (sec. V.)
One-Sided (sec. VI.)
Eager Protocol (sec. V. A)
Rendezvous Protocol
Window Creation (sec. VI. A)
Communication & Synchronization
Blocking Primitives (sec. V. B)
Non-blocking Primitives (sec. V. C)
Comm. with Active Sync. (sec. VI. B)
Comm. with Passive Sync. (sec. VI. C)
RDMA Write (sec. V. B. 1)
RDMA Read (sec. V. B. 2)
RDMA Write (sec. V. C. 1)
RDMA Read (sec. V. C. 2)

Fig. 4: An Overview of the Design Scope for ODP-aware MPI

MPI primitives. Our analysis is based on the LogGP [1] model with extensions to the overhead parameter. With the ODP feature, the overhead of data transfers will be different if page fault or invalidation events are involved. We define $o_{pre-fetch}$ as the overhead of a pre-fetch operation, $o_{page\_fault\_s}$ and $o_{page\_fault\_r}$ as the overhead of resolving a page fault event at the sender and receiver, respectively, o_{post} as the overhead of posting a message to the HCA. In the following two sections, we present our analysis of applying ODP for both MPI point-to-point and one-sided programming models. Figure 4 gives an overview of the following two sections.

V. Designing ODP-aware MPI for Point-to-Point Primitives

In this section, we discuss the trade-offs of using ODP at sender and receiver with both eager and rendezvous protocols.

A. Eager Protocol: Sender & Receiver

Several MPI runtimes use a bounce buffer design by simply copying the user buffer to a pre-registered bounce buffer which could be directly read/write by the HCA. The expected time in the sender side is the copy overhead plus the time to initiate a data transfer. We define c to be the copy overhead. With the pin-down scheme, the expected communication time is: $c + o_{post}$, we do not include the G parameter from LogGP here, since the message size is small. With the ODP scheme, the bounce buffer is registered on demand, so the virtual to physical address mapping of the bounce buffer may not be present in the HCA when a transfer is initiated. To make the bounce buffer ready for transfer, there will be extra overhead of resolving a page fault event or issuing a pre-fetch operation. Based on the previous section, a pre-fetch operation should

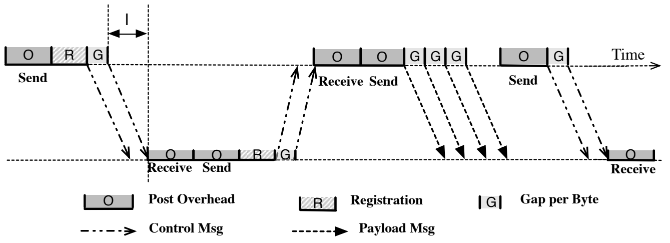


Fig. 5: RDMA Write Protocol with Pin-down Scheme under modified LogGP model

be used to update the mapping and load a bounce buffer. The total expected communication time is: $c+o_{post}+o_{pre-fetch}$. In most MPI stacks, the eager protocol is used to transfer message sizes smaller than 256 KB and is latency sensitive. The one-way latency of 64 bytes message size is 2 us. The overhead of issuing a pre-fetch operation with 64 bytes message size is 5 us. The extra overhead of using ODP could outweigh the gain of reduced pin-down buffers. So ODP should not be applied on the sender side.

With the eager protocol, a message from a sender may come before or after a receive is called at the receiver. When such a message arrives before a matching receive is posted at the receiver side, this message is queued in intermediate buffers at the receiver side which will be searched when a matching receive is called. We define s as the search time to find the data in the intermediate buffers. For this scenario, the expected time for both pin-down and ODP schemes is: $c + s$. When a matching send is absent in the intermediate buffers, a blocking receive waits for a matching send to arrive. We define $skew$ as the time a receiver needs to wait before a message from the sender arrives. With the pin-down scheme, the expected time is: $c + s + skew$. With the ODP scheme, the bounce buffer is registered on demand, there will also be extra overhead if the mapping of the buffer is not present in the HCA, the expected time is: $c + s + skew + o_{pre-fetch}$. Since the size of a bounce buffer used in the receiver side is also smaller than 256K, it is not worth adding extra overhead while only slightly reduces the total pin-down buffer size. So ODP should not be applied on the receiver side either.

B. Rendezvous Protocol in Blocking Primitives: Sender & Receiver

With the RDMA feature, most MPI runtimes use zero-copy designs to transfer medium and large messages with rendezvous protocols. There are two rendezvous protocols based on the RDMA operations used.

1) Rendezvous Protocol with RDMA Write: The typical protocol used with RDMA write has three steps: (1) The sender sends a control message to the receiver. (2) The receiver replies with a control message to notify the sender that the receive buffer is ready for RDMA operation. (3) The sender starts to directly write data to the receive buffer, and a finish message is sent to the receiver after the payload transfer is completed. We define reg as the overhead of registering send/receive buffers. Figure 5 shows the details of the RDMA write protocol under our modified LogGP model.

On the sender side, the expected communication time with the pin-down scheme is:

$$T_{spd} = 6 * o_{post} + 2 * reg + (3 * m_{control} + m_{data}) * G + 2l + g \tag{1}$$

When ODP is used with the rendezvous protocol, based on our discussion in the previous subsection, all the control messages will still be transferred with the eager protocol and the pin-down scheme. Only the data payload will be registered on demand. The expected time spent in the first and second step of the rendezvous protocol will be the same as that with the pin-down scheme. In the third step, there is extra overhead when the mapping in the HCA is not updated before the real data transfer is initiated. This is shown as "Scenario 1" in Figure 6. The expected time at the sender side is:

$$\begin{aligned} T_{sodp\_s1} = & 6 * o_{post} + 2 * reg + (3 * m_{control} + \\ & m_{data}) * G + o_{page\_fault\_s} + o_{page\_fault\_r} + 2l + g \end{aligned} \tag{2}$$

On the sender side, the real data transfer starts after two control messages have been exchanged with the receiver. So a pre-fetch operation could be issued after the first control message is sent out from the sender. This design to overlap control message exchange and pre-fetch operation could potentially remove all the extra overhead of ODP. Since the protocol is used with rendezvous protocol for large messages, an ODP-based scheme implies a large reduction in the registered buffer size. So ODP should be applied at the sender side.

On the receiver side, a control message needs to be replied to the sender. So, we could also apply the overlap design which issues pre-fetch operation after the control message is sent to the sender. So ODP should be applied at the receiver side.

2) Rendezvous Protocol with RDMA Read: The rendezvous protocol with RDMA read has three steps: (1) The sender sends a control message to notify the receiver that the sender buffer is ready for RDMA operation. (2) The receiver directly reads from the sender buffer after the control message is received. (3) The receiver sends a control message to the sender after the data transfer is complete. Compared with the RDMA write protocol, only two control messages are needed in the RDMA read protocol. But, the overlap design for the RDMA Write protocol could be applied here as well. So ODP should be applied at both sender and receiver side.

C. Rendezvous Protocol with Non-blocking Primitives: Sender & Receiver

In applications, non-blocking primitives are used to overlap computation and communication by inserting computation between non-blocking send/recv and completion primitives. When the pin-down scheme is used, the expected communication time is the same as that with blocking primitives.

With the ODP scheme, the overlapping design to overlap pre-fetch operation with control message exchanges should be used as well. The expected time for the control messages will be the same as that with blocking primitive. However, the

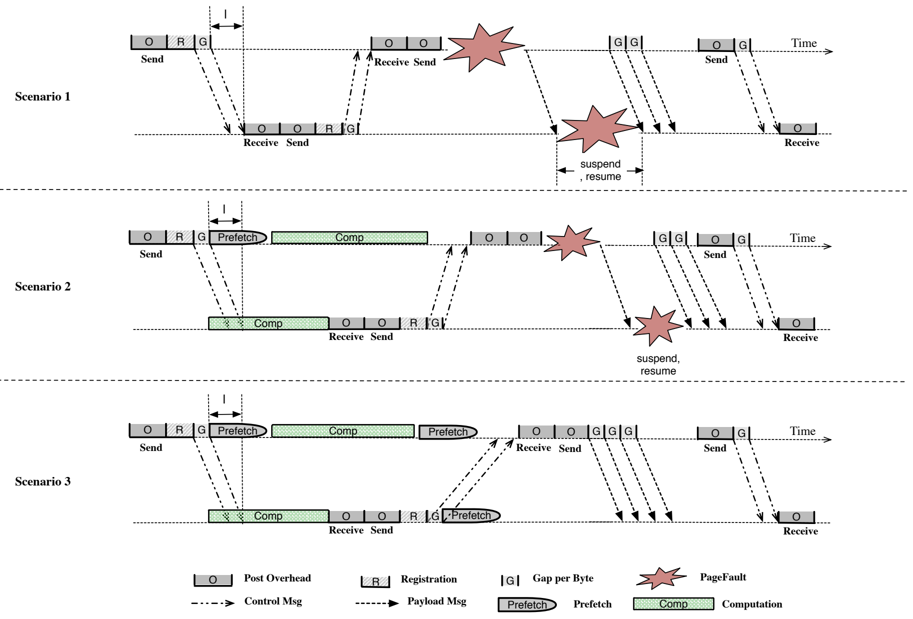


Fig. 6: RDMA Write Protocol with ODP under the extended LogGP model

expected time for the data payload will be different based on the communication pattern, as the OS could swap out these communication buffers. With computation between MPI send and completion primitive, the pre-fetched communication buffers could be invalidated by the OS before the data transfer starts. This scenario is shown as "Scenario 2" in Figure 6. The expected communication time at the sender will be:

$$\begin{aligned} T_{sodp\_s2} =& 6 * o_{post} + 2 * reg + (3 * m_{control} + m_{data}) * G \\ & + o_{pre-fetch} + o_{page\_fault\_s} + o_{page\_fault\_r} + 2l + g \end{aligned} \quad (3)$$

When computation is inserted between MPI send and completion primitive, there are two possible scenarios at the sender. The first scenario is when the sender enters the completion primitive, the reply control message has not arrived yet. In this case, the sender is sitting idle anyway, so it could issue another pre-fetch operation which could be hidden in the idle time. This scenario is shown as "Scenario 3" in Figure 6. The expected communication time at the sender will be:

$$\begin{aligned} T_{sodp\_s3} =& 6 * o_{post} + 2 * reg + (3 * m_{control} + m_{data}) * G \\ & + 2 * o_{pre-fetch} + 2l + g \end{aligned} \quad (4)$$

The other case is when the sender enters the completion primitive, the reply control message from the receiver has arrived. In this case, there will be overhead of $o_{pre-fetch}$ which can not be avoided. Considering large message transfer which is bounded by bandwidth, the extra overhead added to the latency is acceptable for rendezvous transfers. So ODP could be applied at the sender side.

On the receiver side, the pre-fetched receive buffer could also be invalidated before real data transfer arrives. There are also two possible scenarios: One scenario is that the real data transfer is overlapped with the computation at the receiver side, there is no extra overhead even page fault events are generated. The other scenario is when the receiver enters the completion primitive, the data payload has not arrived yet. In this case, since the receiver needs to wait for the control message, it could issue another pre-fetch operation to re-update the mapping of the receive buffer in the HCA. This overhead of pre-fetch could be overlapped with the idle time. Based on this discussion, we should apply ODP at the receiver side.

VI. Designing ODP-aware MPI for RMA Primitives

We divide the RMA functionality into three separate concepts: (1) window creation, (2) communication functions, and (3) synchronization functions.

A. Window Creation

A window is a region of process memory that is made accessible to remote processes. This memory region needs to be registered with the communication subsystem and remote processes require a remote descriptor that is returned from the registration to access the memory. In the window creation phase, all the communications are with small messages which go through the point-to-point eager protocol. So the time of creating a window is the registration time plus the small messages transfer time. The expected time with the pin-down scheme will be the same as the ODP scheme. RMA programming model is prone to use large data working sets, which exists in most data-intensive applications. So ODP should be applied to RMA windows.

B. Communication with Active Synchronization

Active synchronization (global/group) involves both origin and target processes. The Get primitive is directly mapped to the RDMA read operation. The Put primitive is directly mapped to the RDMA write operation. Both primitives transfer data between a remote window and a local buffer. This local buffer could either be in a process's window or arbitrary address outside the window.

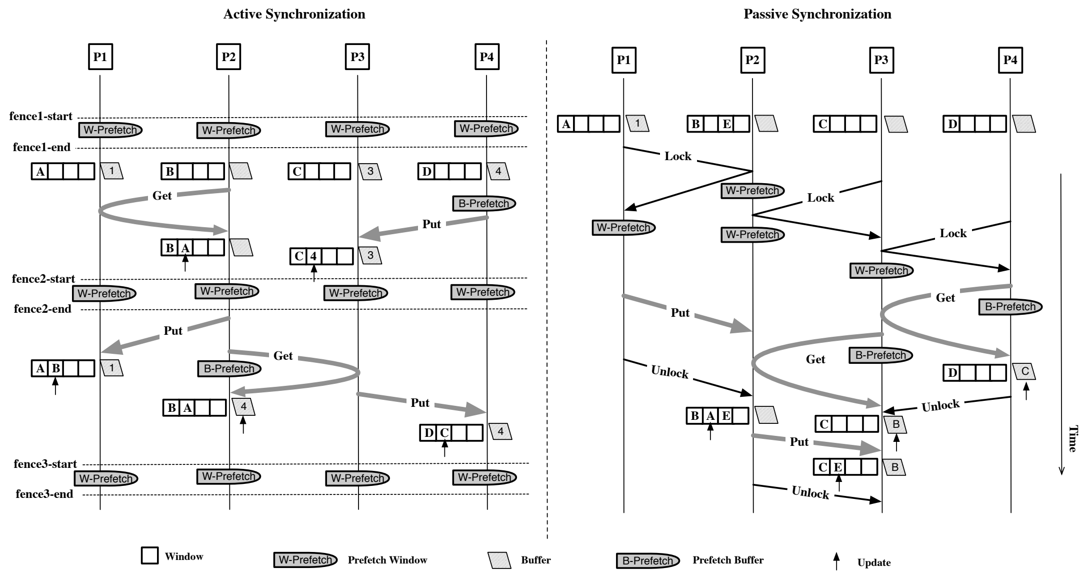


Fig. 7: RMA Communication with Active/Passive Synchronization

For the Put/Get primitive that transfers data between a remote window and a local window. With the pin-down scheme, the expected communication time is: $o_{post} + m_{data} * G + l$. With the ODP scheme, a Get/Put primitive will have extra overhead both at the origin and target processes by resolving page fault events. At the origin process, this page fault could be removed by issuing a pre-fetch operation, however this still adds the pre-fetch overhead to critical paths of data transfers. At the target process, this page fault could not be removed at this point, as it is not actively involved in the communication. The expected communication time with Put is:

$$T_{sodp} = o_{post} + m_{data} * G + o_{pre-fetch} + o_{page\_fault\_s} \qquad (5)$$

An alternative design is to issue pre-fetch operation in the synchronization primitive. With active synchronization, all processes are involved, so each process could issue pre-fetch operation with its window. With this design, the potential page fault events both at the origin and target processes could be avoided. So ODP should be applied in this case.

The Put/Get primitive transfers data between a remote window and local non-window buffer. Even after each process has issued a pre-fetch operation with its window, there is extra overhead related to the local buffer. One design is that local buffer is pre-fetched before RDMA operations start. However, for Put with small messages, this added overhead could outweigh the gain from reduced total pin-down buffer size. For Put with large messages, this added overhead is acceptable. For Get primitive, there is a gap between the point an RDMA operation is issued and the point real data arrives. This provides an opportunity to overlap the pre-fetch operation with the round-trip latency. In sum, ODP should be applied to Get primitives with all message sizes and Put primitives with large message sizes. All these scenarios are shown as Active Synchronization in Figure 7.

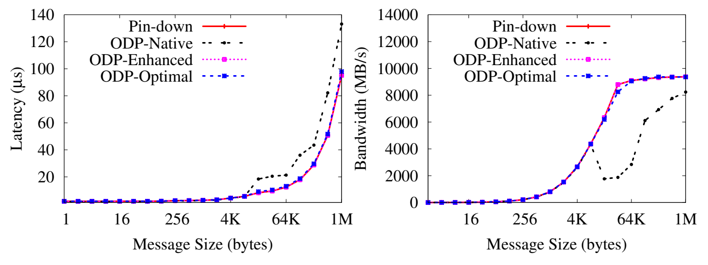
C. *Communication with Passive Synchronization*

Passive synchronization only involves the origin process. It opens an epoch by requesting a lock from the target process. Communication starts after the lock is granted. The epoch is closed by sending an unlock packet to the target process.

Most MPI runtimes implement passive synchronization operation over a two-sided based approach which involves both the origin and target processes. After the origin process gets a lock from a target process, communication operations mapped over RDMA operations could start. Similar to the active synchronization, target process can issue a pre-fetch operation with its window after a lock packet arrives. This could ensure that no page fault will be happening with its window when other processes access it. On the origin process, there will be an extra overhead of pre-fetching its local buffer for Put/Get primitives. In most MPI runtimes, all the RMA communication operations are queued in a list which will be issued back-to-back in the synchronization or completion operation. This means that at the time a lock request packet is sent out, all the communication operations have already been queued in a list. While the origin process is waiting for the lock to be granted. It could issue a pre-fetch operation for local communication buffers. For the Get primitive, the pre-fetch operation could also be issued after an RDMA operation is posted. With these designs, ODP should be applied here. All these scenarios are presented as Passive Synchronization in Figure 7.

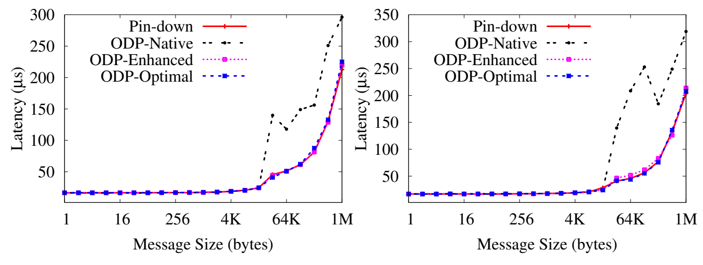
VII. Performance Evaluation

We have implemented all three ODP-based schemes in the popular MVAPICH2 [15] MPI library. The first scheme is ODP-Native which registers buffers with ODP but without pre-fetch. The second scheme is ODP-Enhanced which issues pre-fetch operation after a memory region is registered. The third scheme is ODP-Optimal which is implemented following the discussions in Sections V and VI. Since the ODP feature



(a) Latency (b) Bandwidth

Fig. 8: Point-to-Point Latency and Bandwidth Experiments



(a) Put (b) Get

Fig. 9: One-sided Latency Experiments

is a cutting-edge technology, no major supercomputing centers have this kind of configuration yet. So all our experiments are conducted on a local cluster described in the next subsection. We configure our local cluster with various OS kernel versions and we see the same performance results, so we only report numbers with one OS kernel version.

A. Experimental Setup

The setup consists of 4 Sandy Bridge Compute nodes, 2 Ivy Bridge Computer nodes and 2 Haswell Compute nodes interconnected by Mellanox FDR switch SX6036. The Intel Sandy Bridge processors consist of Xeon dual eight-core sockets operating at 2.60 GHz with 32 GB RAM. The Intel Ivy Bridge processors consist of Xeon dual ten-core sockets operating at 2.80 GHz with 64 GB RAM. The Haswell processors consist of dual ten-core sockets operating at 2.60 GHz with 64 GB RAM. Each node is equipped with MT4113 Connect-IB HCAs (56 Gbps data rate) with PCI-Ex Gen3 interfaces. The OS used is RHEL 7.1.1503 with kernel version 4.2.3, and Mellanox OpenFabrics version 3.1-1.0.3.

B. Performance Counters

In our ODP-aware MPI runtime, we have added designs to keep track of ODP statistics reported by the HCA. These statistics include the number of page fault events, pre-fetch operations, and page invalidation events. They could be used to analyze application performance running with ODP.

C. MPI Level Micro-benchmarks

In this subsection, we evaluate the influence of our proposed designs on communication performance with OSU micro-benchmarks [13].

Figure 8 shows the MPI level point-to-point latency and bandwidth with existing pin-down scheme and our proposed ODP-based schemes. From the figures, we see that the ODP-Native scheme performs worse than the Pin-down scheme for large messages. The degradation is caused by resolveing page fault events in the first iteration. Other ODP-based schemes that take advantage of pre-fetch operations are able to deliver similar performance as the Pin-down scheme. The reason is because these designs have already warmed up the buffers and updated the mapping before the data transfer starts.

Figure 9 shows the MPI one-sided Put and Get latency performance with different schemes. In this benchmark, the window object is created with MPI_Win_create and the synchronization operation is MPI_Win_lock. The results show that ODP-Optimal and ODP-Enhanced designs are able to achieve the same performance as the pin-down scheme.

D. Application Results

We evaluate our proposed designs with applications that use MPI point-to-point and one-sided primitives. Applications using MPI point-to-point primitives are AWP-ODC, NAS Parallel Benchmark and Lammps. Boltzmann and Lenard use Global Arrays going through MPI one-sided primitives. We evaluate these applications with varied input sizes and system sizes. In the results, all the reported page faults are minor page faults which could reclaim frames. We do not see any major page fault with any of these applications.

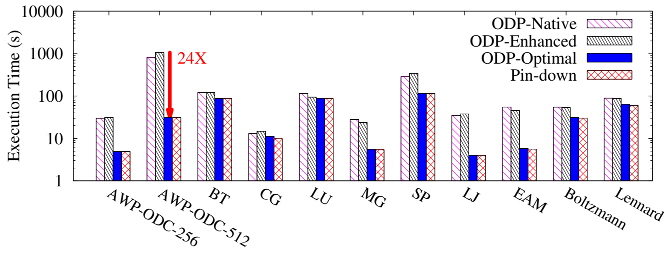
1) AWP-ODC: AWP-ODC is an elastic wave propagation code which simulates the dynamic rupture and wave propagation that occurs during an earthquake. It mainly uses non-blocking MPI primitives. We evaluate its performance with 64 processes (process grid: 4x4x4) and 128 processes (process grid: 8x4x4) running with different input sizes (256x256x256, 512x512x512). The performance results are shown in Figure 10(a) and Figure 11(a). For the problem size of 512x512x512 with 64 and 128 processes, compared with the ODP-Enhanced scheme, the ODP-Optimal scheme is able to reduce the execution time by 24X and 32X, respectively. In this application, computation is overlapped with communication, the pre-fetched communication buffers could be invalidated before payload data is transferred. Table I shows the statistics for each scheme in this run. We see that the page fault events of the ODP-Optimal scheme is less than 1% of that with other schemes. Although the ODP-Optimal scheme has a higher number of pre-fetch operations, it still achieves the same performance as the Pin-down scheme. The reason is that all our pre-fetch operations have been overlapped with data transfers or computation. With all ODP schemes, the pin-down buffer size could be reduced by 10X compared with the Pin-down sheme.

2) NAS Parallel Benchmarks: The NAS Parallel Benchmarks (NPB) are a set of programs that are designed to be typical of several MPI applications. We evaluate using the Class ‘C’. We run BT, CG, LU, MG, and SP in the experiments. Figure 10(a) and Figure 11(a) show the execution time of the NAS Benchmarks. The ODP-Optimal scheme maintains better performance than other ODP-based schemes for all benchmarks. Tables I and II show the statistics of each scheme. For the LU benchmark, we see that the ODP-Enhanced and ODP-Optimal schemes deliver equal performance. The reason is that LU mostly uses blocking MPI send primitives and non-blocking MPI recv primitives. Also, there is no computation happening between the MPI recv and completion primitives. So pre-fetched communication buffers are not invalidated before payload message is transferred. For the SP benchmark, there is computation happening between non-blocking MPI send and completion operation, the same pattern as AWP-ODC. That is why ODP-Optimal performs much better than other two ODP-based schemes for SP.

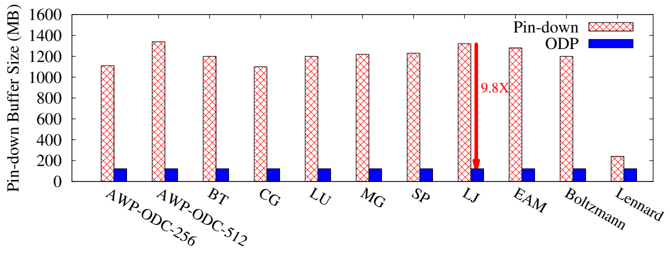
3) LAMMPS: LAMMPS [17] is a Large-scale Atomic/-Molecular Massively Parallel Simulator. We evaluate the LAMMPS LJ and EAM benchmarks. For both benchmarks, we use the out-of-box input files 'in.lj' for LJ and 'in.eam' for EAM. We run each benchmark with 64 processes (process grid: 4x4x4) and 128 processes (process grid: 8x4x4). With 128 processes, the execution time of LJ with the Pin-down, ODP-Enhanced, ODP-Optimal, and ODP-Native is 7s, 71.2s, 7.1s, and 70.9s, respectively. Compared with the ODP-Enhanced and ODP-Native schemes, the ODP-Optimal scheme could significantly reduce the execution time by avoiding page fault events. Table II includes the statistics for this run. For the EAM benchmark with 64 processes, the execution time with the Pin-down, ODP-Enhanced, ODP-Optimal, and ODP-Native schemes is 4s, 37.8s, 4s, and 35s, respectively. For both benchmarks with 128 processes, the ODP shemes could reduce the pin-down buffer size for LJ and EAM by 9X and 11X, respectively.

4) Boltzmann: The Boltzmann application benchmark runs a simple lattice Boltzmann simulation of flow in lid-driven square cavity. This benchmark is using ComEx-over-MPI backend. We use parameters to run at a Reynolds number of 128. Figures 10 and 11 present the execution time and pin-down buffer sizes. With 64 processes, the execution time for the Pin-down, ODP-Enhanced, ODP-Optimal, and ODP-Native schemes is 30.13s, 53.7s, 31.2s, and 54.8s, respectively. The pin-down buffer size is reduced by 9X with ODP schemes.

5) Lennard Jones: Molecular Dynamics of Lennard Jones System is using performed Global Arrays based on force decomposition (using ComEx-over-MPI backend). The input parameters for the experiment include 4,096 atoms, 128x128-sized blocks, density set to 1.05, temperature to 2.0 and use of 7,500 iteration steps. The application spends a majority of execution time in progressing Get operations using Unlock synchronization. Figures 10 and 11 depict the pin-down buffer size and execution time profiles. Compared with the ODP-Enhanced and ODP-Native design, the ODP-Optimal scheme is able to reduce the execution time by 30% and 31%, respectively. Also, the ODP-Optimal design is able to achieve the same performance as the Pin-down scheme.

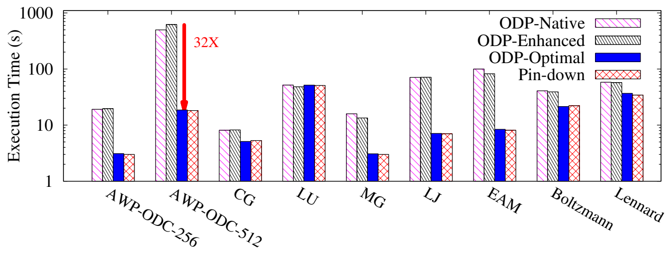


(a) Execution Time

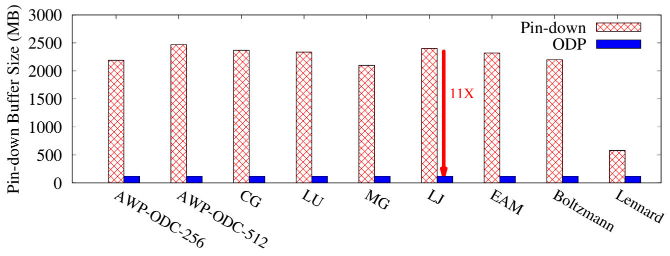


(b) Pin-down Buffer Sizes

Fig. 10: Application Performance (64 Processes)



(a) Execution Time



(b) Pin-down Buffer Sizes

Fig. 11: Application Performance (128 Processes)

E. Verbs-level Benchmark with Buffers Larger Than Physical Memory Size

In this subsection, we come up with a verbs-level benchmark to evaluate the performance when the total size of communication and computation buffers is larger than the physical memory size. We pin-down a large communication buffer in the beginning then exchange data between processes. The computation phases start after the communications are completed. In the end, the total execution time is reported. We run three inputs with this benchmark. The first input, second input, and third input sizes are 100%, 120%, and 140% of the physical memory size, respectively. With the ODP scheme, the execution time for these three inputs is 65.16s, 64.88s, and 65.02s, respectively. With the pin-down scheme,

TABLE I: ODP Statistics for Applications (64 processes)

Apps.	# Page-Fault Events			# Pre-Fetched Pages	
	ODP-Native	ODP-Enhanced	ODP-Optimal	ODP-Enhanced	ODP-Optimal
NAS-BT	2643	2212	0	10930	76510
NAS-CG	219	182	0	1792	7392
NAS-LU	273	230	0	2160	19980
NAS-MG	779	612	0	4288	8776
NAS-SP	4777	4798	24	10929	76235
Lammps-LJ	2903	2447	3	19635	47884
Lammps-EAM	5011	3945	5	16689	53597
AWP-ODC 256^3	3213	2368	26	5837	14123
AWP-ODC 512^3	28650	23037	107	20164	97156
Boltzmann	831	723	0	1287	2359
Lennard	1629	1447	0	2307	4816

TABLE II: ODP Statistics for Applications (128 processes)

Apps.	# Page-Fault Events			# Pre-Fetched Pages	
	ODP-Native	ODP-Enhanced	ODP-Optimal	ODP-Enhanced	ODP-Optimal
NAS-CG	245	143	0	1224	6177
NAS-LU	280	180	0	1912	18630
NAS-MG	986	730	0	5330	10944
Lammps-LJ	3023	2880	4	22378	52266
Lammps-EAM	5701	4430	4	20849	63790
AWP-ODC 256^3	2986	2014	0	4358	11229
AWP-ODC 512^3	20638	17252	53	28730	67332
Boltzmann	1052	704	0	3528	17139
Lennard	1745	1623	0	4032	25382

the execution time for these three inputs is 65.73s, 150.21s, and 257.67s, respectively. When the total memory requirement of this benchmark is larger than the physical memor size, with the ODP-based scheme, the OS kernel could swap all the communication buffers for the computation phase. However, with the pin-down scheme, all communication buffers are still pinned in memory, so memory thrashing is happening for the computation buffers.

VIII. Related Work

Woodall et al. [22] described a pipeline protocol by overlapping the dynamic registration and deregistration of memory buffers with data transfer. Li et al. [12] proposed a memory registration strategy that hides the cost of dynamic memory management by offloading all dynamic memory registration/deregistration requests to a dedicated memory management helper thread. Bell and Bonachea [7] proposed an RDMA algorithm for a shared memory system that determines the largest amount of memory that remote machines can share and then registers all shared memory regions and links them to a firehose interface, to which remote machines can write and read shared memory at any time. Tezuka et al. [20] proposed the pin-down cache technique which reuses the pinned-down area to decrease the number of calls to pin-down and release primitives. Shipman and Brightwell [19] investigated network buffer utilization and introduced a new protocol to increase the efficiency of receiver buffer utilization for IB. Ou et al. [14] proposed a memory registration region cache strategy that manages memory in terms of memory region containing one or more memory pages to minimize the cost of memory registration and deregistration in the critical data path.

Petrini et al. [16] described (Quadrics interconnection network) QsNet that extends the native operating system in the nodes with a network operating system and specialized hardware support in the network interface. QsNet does not require memory to be registered before communication. MPICH-QsNet and Open MPI were used to provide support over Quadrics [23]. Since the Quadrics network is not available anymore, so we do not have access to such systems and evaluate its performance. UNet-MM [21], an extension U-Net, stores address translation in a translation cache on the network interface. Misses in the translation cache are handled by the host OS which pins virtual pages and installs their translations on the network interface. Meiko CS-2 [6] uses a protocol processor to deal with address translation and data transfer. Cray Aries [2] network uses I/O Memory Management Unit (IOMMU) to translate virtual-to-physical addresses for data transfers.

Our study goes beyond the existing studies presented in this section. First, we develop a performance model to evaluate the communication time with the pin-down scheme and the ODP scheme. Second, we use the model to formally analyze the performance and memory trade-offs using both schemes. We identify and propose solutions to several challenges with ODP that do not exist in the pin-down scheme. MPI runtimes supported Quadrics do not handle the internals of virtual to physical mapping and update them appropriately. Our studies handle them in order to achieve optimal performance at MPI level.

IX. Conclusion and Future Work

In this paper, we presented the new ODP feature and evaluated its performance characteristics with proposed IB verbs-level micro-benchmarks. Based on this knowledge, we designed an ODP-aware MPI runtime. We showed that simply integrating ODP with an MPI stack (ODP-Native and ODP-Enhanced schemes) could lead to significant performance degradation compared to existing pin-down scheme. We then proposed the ODP-Optimal scheme in MVAPICH2, which significantly reduces the size of pin-down buffers while sustaining performance. This scheme is designed based on our extended LogGP model. We systematically evaluated proposed schemes with MPI level micro-benchmarks and ten real-world applications. The experimental results at the micro-benchmark level show that the ODP-Optimal scheme is able to deliver comparable performance as existing pin-down scheme. For LAMMPS application, our proposed designs could reduce the pin-down buffer sizes by 11X without performance degradation. For AWP-ODC application, the ODP-Optimal scheme is able to reduce the execution time by 32x over the ODP-Enhanced scheme. In addition, with our proposed verbs-level benchmark, the ODP scheme could outperform existing pin-down scheme by 4X. All the proposed designs in this paper have been released in MVAPICH2-X. In future work, we plan to carry out the large-scale tests based on the availability of ODP on cutting-edge HPC clusters.

References

[1] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model – One Step Closer Towards a Realistic Model for Parallel Computation. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, pages 95–105, New York, NY, USA, 1995. ACM.

[2] B. Alverson, E. Froese, L. Kaplan, and D. Roweth. Cray XC Series Network. Technical report, 2012.

[3] R. Alverson, D. Roweth, and L. Kaplan. The Gemini System Interconnect. In *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*, pages 83–87, Aug 2010.

[4] B. Arimilli, R. Arimilli, V. Chung, S. Clark, W. Denzel, B. Drerup, T. Hoefler, J. Joyner, J. Lewis, J. Li, N. Ni, and R. Rajamony. The PERCS High-Performance Interconnect. In *Proceedings of the 2010 18th IEEE Symposium on High Performance Interconnects*, HOTI '10, pages 75–82, Washington, DC, USA, 2010.

[5] M. Beck and M. Kagan. Performance Evaluation of the RDMA over Ethernet (RoCE) Standard in Enterprise Data Centers Infrastructure. In *Proceedings of the 3rd Workshop on Data Center - Converged and Virtual Ethernet Switching*, DC-CaVES '11, pages 9–15. International Teletraffic Congress, 2011.

[6] J. Beecroft, M. Homewood, and M. McLaren. Meiko cs-2 interconnect elan-elite design. *Parallel Computing*, 20(10):1627 – 1638, 1994.

[7] C. Bell and D. Bonachea. A New DMA Registration Strategy for Pinning-based High Performance Networks. In *Proceedings of the 17th International Conference on Parallel and Distributed Processing*, IPDPS'03, 2003.

[8] N. Bloch, S. Raindel, H. Eran, and L. Liss. Use of Free Pages in Handling of Page Faults, June 3 2014. US Patent 8,745,276.

[9] G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard. Cray Cascade: A Scalable HPC System Based on A Dragonfly Network. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–9, Nov 2012.

[10] InfiniBand Trade Association. http://www.infinibandta.com.

[11] S. Kumar, A. R. Mamidala, D. A. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger, D. Chen, and B. Steinmacher-Burrow. PAMI: A Parallel Active Message Interface for the Blue Gene/Q Supercomputer. In *Proceedings of the 2012 IEEE 26th International Parallel Distributed Processing Symposium (IPDPS)*, pages 763–773, May 2012.

[12] D. Li, K. W. Cameron, D. S. Nikolopoulos, B. R. d. Supinski, and M. Schulz. Scalable Memory Registration for High Performance Networks Using Helper Threads. In *ACM International Conference on Computing Frontiers Supercomputing (CF11)*, Ischia, Italy, May 2011.

[13] Network Based Computing Laboratory. OSU Micro-benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/.

[14] L. Ou, X. He, and J. Han. MRRC: An Effective Cache For Fast Memory Registration in RDMA. In *23rd IEEE Conference on Mass Storage Systems and Technologies (MSST2006)*, College Park, Maryland, 2006.

[15] D. K. Panda, K. Tomko, K. Schulz, and A. Majumdar. The MVAPICH Project: Evolution and Sustainability of an Open Source Production Quality MPI library for HPC. In *Workshop on Sustainable Software for Science: Practice and Experiences, held in conjunction with Int'l Conference on Supercomputing (WSSPE)*, 2013.

[16] F. Petrini, W.-c. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network: High-Performance Clustering Technology. *IEEE Micro*, 22(1):46–57, Jan. 2002.

[17] S. Plimpton. Fast Parallel Algorithms for Short-range Molecular Dynamics. *J. Comput. Phys.*, 117(1):1–19, Mar. 1995.

[18] S. Raindel, H. Eran, L. Liss, and N. Bloch. Look-ahead Handling of Page Faults in I/O Operations, Dec. 16 2014. US Patent 8,914,458.

[19] G. M. Shipman, R. Brightwell, B. Barrett, J. M. Squyres, and G. Bloch. Investigations on InfiniBand: Efficient Network Buffer Utilization at Scale. In *Proceedings, Euro PVM/MPI*, Paris, France, October 2007.

[20] H. Tezuka, F. O. Carroll, A. Hori, and Y. Ishikawa. Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication. In *Proceedings of 12th Int. Parallel Processing Symposium*, March 1998.

[21] M. Welsh, A. Basu, and T. von Eicken. Incorporating memory management into user-level network interfaces. Technical report, Ithaca, NY, USA, 1997.

[22] T. S. Woodall, G. M. Shipman, G. Bosilca, R. L. Graham, and A. B. Maccabe. High Performance RDMA Protocols in HPC. In *Proceedings of the 13th European PVM/MPI User's Group Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, EuroPVM/MPI'06, pages 76–85, Berlin, Heidelberg, 2006. Springer-Verlag.

[23] W. Yu, T. S. Woodall, R. L. Graham, and D. K. Panda. Design and Implementation of Open MPI over Quadrics/Elan4. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 96a–96a, April 2005.