Pitfalls of InfiniBand with On-Demand Paging

1st Takuya Fukuoka
The University of Tokyo

2nd Shigeyuki Sato*
The University of Tokyo
sato.shigeyuki@mi.u-tokyo.ac.jp

3rd Kenjiro Taura
The University of Tokyo
tau@eidos.ic.i.u-tokyo.ac.jp

***Abstract*—InfiniBand is a popular high-performance interconnect and offers Remote Direct Memory Access (RDMA), which enables low-latency communication based on kernel bypassing. Although the conventional RDMA technology necessitates manual physical memory management, an emerging extension, On-Demand Paging (ODP), implements automatic memory management based on RDMA-triggered page faults, which benefits productivity. Although the existing studies said the overhead of a page fault of ODP to be small enough, an in-depth investigation in various network situations including retransmission and timeout is missing. In this work, we conduct a comprehensive analysis of the actual behaviors of ODP on different devices and reveal two awful performance pitfalls, which incur longer latencies 3–4 orders of magnitude than a common-case page fault does. We also experimentally demonstrate that the revealed pitfalls are harmful to existing software systems. This paper presents our experimental analysis and lessons learned therefrom.**

***Index Terms*—InfiniBand, On-Demand Paging, Reliable Connection, reverse-engineering, packet capturing**

I. Introduction

High-performance interconnects play a crucial role in real-world computing. According to Mellanox's report [1], data centers and supercomputers tend to choose InfiniBand for high-performance computing (HPC) and artificial intelligence (AI) infrastructures, and high-speed Ethernet for cloud and hyperscale platforms.

A key technology of these interconnects is remote direct memory access (RDMA), which enables kernel-bypassing and zero-copy transfer of high bandwidth and ultra-low latency. As RDMA over Ethernet, such as RoCE and iWARP, has become common [2]–[4], RDMA has become utilized for developing various distributed systems [5]–[10].

The kernel-bypassing nature of RDMA is crucial for low-latency communication, whereas it incurs a large burden on system development because the standard RDMA primitives cannot benefit from the memory paging by kernels. They necessitate manual registering of physical memory segments ready for RDMA into network devices. Moreover, this memory registration is known to involve significant runtime overhead. To balance the spatial cost of registered memory and the runtime cost of registration for the performance of RDMA-based systems, we have to elaborate physical memory management [11]–[16]. To take both performance and productivity, on-demand memory management based on page faults as in kernels is promising and desirable.

This work was partially supported by JST ACT-I Grant Number JPMJPR17UC and JSPS KAKENHI Grant Number 16H01715.

*Corresponding author

The On-Demand Paging (ODP) [17], [18] functionality of InfiniBand hardware developed by Mellanox serves the very thing. ODP implements automatic memory management with RDMA-triggered page faults on network devices. Developers thus become free from hand-crafted physical memory management. Although page fault handling involves some runtime overhead, the existing study [17] reported that it merely costed several hundreds of microseconds per page fault and concluded it to be "small enough".

Now, we are faced with a basic question: *Is it always small enough?* ODP is for helping to develop general RDMA-based software systems and should be amenable to different sorts of communication. Particularly, the existing work [17], [19], [20] focused on the cost of page faults themselves and little investigated that of retransmission derived from page faults. To understand the actual cost of ODP, we need an in-depth investigation in various network situations, including the retransmission and timeout of packets.

To answer this question, we conduct a comprehensive investigation into the performance of ODP-enabled RDMA of InfiniBand, which includes the reverse-engineering of the hardware implementation of ODP. Then, we reveal two awful performance pitfalls: *packet damming* and *packet flood*. Packet damming is a situation wherein a stuck packet incurs a long timeout of the Reliable Connection of InfiniBand, which incurred a latency of several hundreds of milliseconds. Packet flood is massive repetitive retransmission incurred by a long delay of updating page statuses, which incurred a latency of a few seconds. These resultant latencies are longer by 3–4 orders of magnitude than the overhead of a network page fault itself reported in [17]. We also demonstrate that the revealed pitfalls are actually harmful to existing software systems through experiments with SparkUCX [21] and ArgoDSM [22].

This paper presents our experimental analysis and lessons learned therefrom. We believe that these are beneficial from both the hardware and software standpoints. Our experimental analysis pinpoints hardware-level flaws, which would serve as good clues for hardware vendors. Our practice and lessons would be good hints to identify and/or avoid ODP-related performance bugs for the developers of communication software that can enable ODP, such as MVAPICH2-X[1], UCX [23], and libfabric [24].

Our main contributions are summarized as follows.

[1]https://mvapich.cse.ohio-state.edu/

- We present an in-depth experimental analysis of the actual behaviors of ODP (Section IV). To the best of our knowledge, our work is the first to analyze one-sided RDMA operations under ODP experimentally and actual timeouts on different InfiniBand devices quantitatively.
- We identify two performance pitfalls of ODP [17] of InfiniBand, which we call *packet damming* (Section V) and *packet flood* (Section VI). We experimentally demonstrate that they can easily arise even under simple conditions.
- We evaluate the impacts of packet damming and packet flood on SparkUCX and ArgoDSM (Section VII). Our experimental results show that they are actually harmful to existing software systems.

II. InfiniBand

In this section, we briefly introduce the basic concepts of InfiniBand, and then describe the retransmission mechanism, which is a key to the hardware implementation of ODP. See [25] for the details of the standard specification of InfiniBand.

InfiniBand has four transport modes: Unreliable Datagram (UD), Unreliable Connection (UC), Reliable Datagram (RD), and Reliable Connection (RC). In this paper, we focus on RC, which is the most commonly used, because the implementation of ODP for InfiniBand is based on RC [17].

A. Memory Registration

To handle RDMA of kernel-bypassing transfers, InfiniBand adapter cards, i.e., RDMA network interface cards (RNICs), manage translation tables from virtual addresses to physical addresses. Memory registration is to register user-space memory regions (e.g., communication buffers) into translation tables of RNICs. Kernels generally change virtual-to-physical address mapping through paging or swapping, while the translation tables of RNICs do not follow it because of the kernel-bypassing nature. Therefore, memory registration generally involves pinning down a given region to guarantee to keep the translation tables of RNICs valid.

B. InfiniBand Verbs

InfiniBand verbs are low-level communication interfaces of RNICs for RDMA. They provide READ, WRITE, and ATOMIC of one-sided communication and SEND and RECV of two-sided communication.

Queue Pairs (QPs) are the endpoints of communication channels of InfiniBand verbs. Each QP consists of a Send Queue (SQ) and a Receive Queue (RQ). When issuing communication, an application posts a Work Request (WR) to a QP. QPs manage WRs as Work Queue Elements (WQEs) until RNICs complete to process the WRs. When a WR completes, the corresponding Completion Queue Entry (CQE) gets inserted into a Completion Queue (CQ), which is associated with QPs, to notify applications of completion. Even if a WR fails, a CQE that contains the corresponding error code gets inserted into the CQ.

C. Retransmission Handling

In RC, RNICs can detect packet loss through the timeout of requests. When a request goes to be a timeout, RNICs retransmit lost packets and guarantee the transmission of packets. Regarding timeout, QPs have two parameters: Local ACK Timeout C_{ACK} and Retry Count C_{Retry}. Local ACK Timeout C_{ACK} is a 5-bit counter to define the timeout interval $T_{tr} = 4.096 \cdot 2^{c_{\text{ACK}}}$ µs. Setting $C_{\text{ACK}} = 0$ means to disable the timeout. The timeout interval T_{tr} defines the amount of time T_o that RNICs take to detect the timeout condition such that $T_{tr} \leq T_o \leq 4T_{tr}$. Practically, we have to set C_{ACK}, taking To into consideration. Retry Count C_{Retry} is the maximum number of retransmissions allowed for a request. A process aborts with the `IBV_WC_RETRY_EXC_ERR` error when retransmission for a request fails C_{Retry} times.

Note that we generally cannot set T_{tr} to 8.192 µs with $C_{\text{ACK}} = 1$ because the specification [25] says "The minimum acceptable value of Local ACK Timeout, other than zero, shall be defined by the CA [RNIC] vendor." More specifically, letting c be set to C_{ACK} and c_0 be the minimum acceptable value for an RNIC, T_{tr} is calculated with $C_{\text{ACK}} = \max(c, c_0)$.

The specification [25] also says:

> Because of variabilities in the fabric, scheduling algorithms and architecture of the channel adapters and many other factors, it is not possible, nor desirable, to time outstanding requests with a high degree of precision.

This statement implies that the minimum acceptable value of C_{ACK} may not be small in practice.

InfiniBand verbs also provide a way of intentionally invoking retransmission via the Receiver-Not-Ready (RNR) Negative-Acknowledgment (NAK) packet, which is a special packet for the receiver of a concerned packet to ask the sender to retransmit it after a certain period. We can set the smallest period for which the sender has to wait via a verb parameter, which we call *minimal RNR NAK delay*.

III. On-Demand Paging

On-Demand Paging (ODP) [18] is an extension of InfiniBand verbs to allow us to be free from pinning down memory regions for memory registration. Under ODP, RNICs raise network page faults[2] [17] and device drivers handle them to manage translation tables in RNICs automatically. The usage of ODP has two types Explicit ODP, which is to enable ODP for registered memory regions, and Implicit ODP, which is to enable ODP for the entire address space and make users free from memory registration. The difference in usage between them, however, does not matter in this paper.

A. Basic Mechanism

We briefly review the basic mechanism of ODP according to the original paper [17].

First, an RNIC checks, on the access to memory regions, if a given virtual memory address is mapped into a physical

[2]In this paper, we call network page faults simply as page faults.

memory address. If not mapped, the RNIC asks a driver to raise an interrupt to query for the kernel. Then, the kernel returns the corresponding physical address by, e.g., allocating pages or retrieving them from secondary storage. Lastly, the driver passes the physical address to the RNIC, and the translation table gets updated.

Pages registered in RNICs have to get invalidated when kernels release them. The process of this page invalidation takes place in reverse order of that of page faults. When releasing a page, a kernel notifies a driver of the address mapping to be invalidated. Then, the driver conveys it to an RNIC to flush the entry in the translation table. Lastly, the driver notifies the kernel of the completion of this invalidation.

B. Packet Handling

The page fault handling and invalidation described above merely explain interactions between kernels and RNICs. They do not suffice for implementing ODP because RNICs communicate with remote RNICs. The implementation of ODP has to handle appropriately packets that result in page faults.

A primary technical issue is that RNICs have to manage a packet until its resultant page fault gets resolved, by using a limited size of cache memory. To cope with this issue, RNICs rely on the retransmission mechanism without storing pending packets locally. Specifically, RNICs leverage RNR NAK for suspending senders of a packet that causes a page fault. Receivers do not have to store any dropped packets until RNR NAK reaches either because the reliability of InfiniBand guarantees to leave them on the sender side.

IV. On-Demand Paging in Reality

In this section, we experimentally analyze an actual implementation of ODPs. Table I summarizes the detailed information of the InfiniBand RNICs used in our experiments. Table II summarizes experimental environments.

A. Actual Behaviors of ODP

The original paper [17] on ODP clearly described why RNR NAK and reliability enabled us to implement ODP with a limited memory. However, how to send RNR NAK and retransmit requests/response is not very clear, particularly for one-sided operations READ and WRITE. Then, to understand the actual behavior of ODP with RDMA operations, we observe the InfiniBand traffic of a single READ through the packet capturing tool ibdump[3]. For clarity, we here analyze the process of resolving a page fault in the client side and that in the server side, separately. We call these one-side page fault resolutions with ODP, *client-side ODP* and *server-side ODP*, respectively. We call a mixture of client-side ODP and server-side ODP, *both-side ODP*. We used KNL (private servers B) for this experiment, setting the minimal RNR NAK delay to be 1.28 ms.

Figure 1 illustrates the workflow of a single READ based on packets captured via ibdump and page fault counters.

[3]https://github.com/Mellanox/ibdump

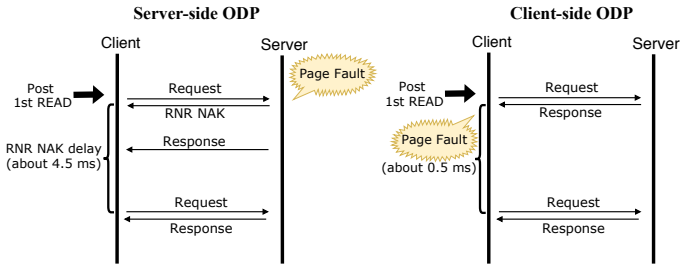


Fig. 1. Workflow of ODP with single READ observed with idbump.

In the server-side ODP, the server sent RNR NAK back to the client in the face of the page fault. Then, the client waited about 4.5 ms and retransmitted a request, while discarding responses sent back during the waiting time. We have found no specific reason for discarding responses and conjecture it to be related to the hardware-level matter of RNR NAK.

In the client-side ODP, the retransmission of a request also took place but was not based on RNR NAK. The client raised a page fault for a response and discarded the response because of limited memory. Then, the client retransmitted a request after about 0.5 ms regardless of the resolution of the page fault. That is, the client does not wait locally for the page fault to get resolved. Therefore, the retransmission of the same request takes place over and over if a page fault takes a long time to resolve. We conjecture that this implementation would be due to a large burden on hardware when many requests to be retransmitted remain.

B. Timeout for the Worst Case

The observations in Figure 1 merely show the common-case latency. Considering that the implementation of ODP utilizes the retransmission mechanism, the worst-case scenario of resolving a page fault will incur the timeout of InfiniBand. However, a quantitative analysis of the actual timeouts T_o of different InfiniBand RNICs is missing. We have found merely a few articles [29], [30] on personal websites to complain about long delays in detecting anomalies.

To understand the extreme case of ODP quantitatively, we measured the actual T_o with various InfiniBand RNICs as described in Table I. In this experiment, we deliberately caused packet loss by specifying a wrong destination LID to a QP in the initialization phase. We set $C_{\text{Retry}} = 7$ and measured the time between the issue of the first request and the abortion of the process with `IBV_WC_RETRY_EXC_ERR`. Therefore, letting t be the measurement time, we obtain $T_o = t/(C_{\text{Retry}} + 1) = t/8$.

Figure 2 shows the results of T_o measured by varying C_{ACK}. We observed experimental lower limits of T_o to be around 30 ms for the ConnectX-5 and to be around 500 ms for the others. From these lower limits, we can estimate the minimum acceptable values of C_{ACK} to be 12 for the ConnectX-5 and to be 16 for the others.

We have two important observations from the experimental results. One is, considering that the usual round trip latency

TABLE I
INFINIBAND SYSTEMS AND DETAILS ON THEIR RNICS, WHERE REEDBUSH-H/L, ABCI, AND ITO ARE COMPUTING CLUSTERS.

System name	PSID	Model name	Driver version	Firmware version
Private servers A	MT_1100120019	ConnectX-3 56Gbps FDR	5.0-2.1.8.0	2.42.5000
Private servers B	MT_2170111021	ConnectX-4 56Gbps FDR	5.0-2.1.8.0	12.27.1016
Reedbush-H [26]	MT_2160110021	ConnectX-4 56Gbps FDR	4.5-0.1.0	12.24.1000
Reedbush-L [26]	MT_2180110032	ConnectX-4 100Gbps EDR	4.5-0.1.0	12.24.1000
ABCI [27]	MT_0000000095	ConnectX-4 100Gbps EDR	4.4-1.0.0	12.21.1000
ITO [28]	FJT2180110032	ConnectX-4 100Gbps EDR	4.4-1.0.0	12.23.1020
Azure VM HCr Series	MT_0000000010	ConnectX-5 100Gbps EDR	4.7-3.2.9	16.26.0206
Azure VM HBv2 Series	MT_0000000223	ConnectX-6 200Gbps HDR	5.0-2.1.8.0	20.26.6200

TABLE II
EXPERIMENTAL ENVIRONMENT

System name	CPU	# of logical cores	Memory
KNL (Private servers B)	Xeon Phi CPU 7250 @ 1.40GHz	272	196 GB + MCDRAM 16 GB
Reedbush-H	Xeon CPU E5-2695 v4 @ 2.10GHz	36 (= 18 × 2)	256 GB
ABCI	Xeon Gold 6148 CPU @ 2.40GHz	80 (= 20 × 4)	384 GB

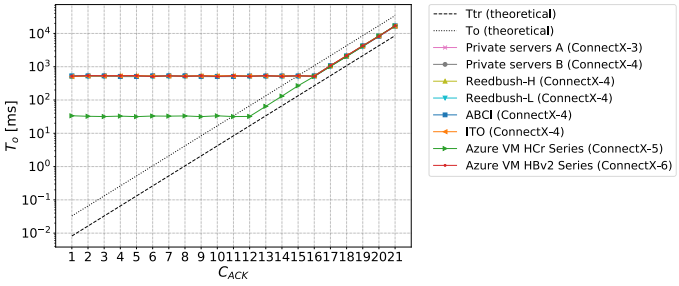


Fig. 2. T_o measured by varying C_{ACK} on different InfiniBand systems, where systems other than Azure VM HCr Series (green) lie on almost the same line.

```
1  init(local_buf, remote_buf, QP[num_QPs], ...);
2
3  for (i = 0; i < num_ops; i++) {
4    local  = &local_buf[size * i];
5    remote = &remote_buf[size * i];
6    QP     = QPs[i % num_QPs];
7
8    post_rdma_read(local, remote, QP, size);
9    usleep(interval);
10 }
11 wait();
```

Fig. 3. Our micro-benchmark in simplified C code, which allows us to specify `size` as the message size for each operation, `num_ops` as a number of READ operations, `num_qps` as a number of QPs, and `interval` as time intervals between communications. The `wait` function polls the CQ to check whether all communications have been completed.

of InfiniBand is about several µs, the timeout is tremendously and incomparably longer. The timeout is a disaster even if rare. The implementation of ODP should never incur it. The other is that a long timeout is ubiquitous. Even real-world computing clusters [26]–[28] and even the latest RNIC, the ConnectX-6 200Gbps HDR, can suffer from it.

V. PACKET DAMMING

In this section, we present the first performance pitfall of ODP: *packet damming*. Packet damming is a performance bug involving a delay of several hundreds of *milliseconds*, during which the packets are dammed. This is caused by READ operations and results in packet loss and a subsequent timeout.

We conducted experiments with a micro-benchmark to explore the conditions under which packet damming occurs and analyzed the root cause using ibdump. For these experiments, we used KNL (private servers B) described in Table I and Table II, where the minimal timeout which can be configured is approximately 500 ms. We used RC as the transport type with $C_{\text{ACK}} = 1$ (minimal) and $C_{\text{Retry}} = 7$.

We created a micro-benchmark with InfiniBand verbs, in which we allocated two processes to two different machines, and the client process issued multiple READ operations to the server process as shown in Figure 3. We introduced a sleep function to control the communication intervals. At the end of the benchmark, a blocking wait was inserted so that the completion of all the communications could be confirmed. The communication buffer was aligned with 4096-byte boundaries, considering the page size. In this section, we used a message size of 100 bytes, a single QP, and both-side ODP, if not specifically mentioned.

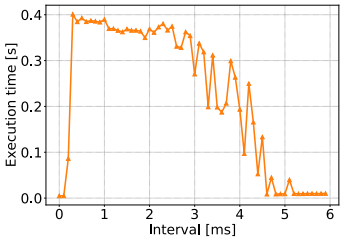


Fig. 4. Average execution time of our micro-benchmark out of 10 trials with varying intervals between two communications. Two READ operations were issued, and both-side ODP was adopted. Minimal RNR NAK delay was set to 1.28 ms.

A. Pitfalls with Two READ Operations

First, we observed the execution time of the micro-benchmark with only two READ operations in Figure 4. The execution took several hundred *milliseconds* with an interval of 100–4500 µs. This is surprising, given that the overhead of

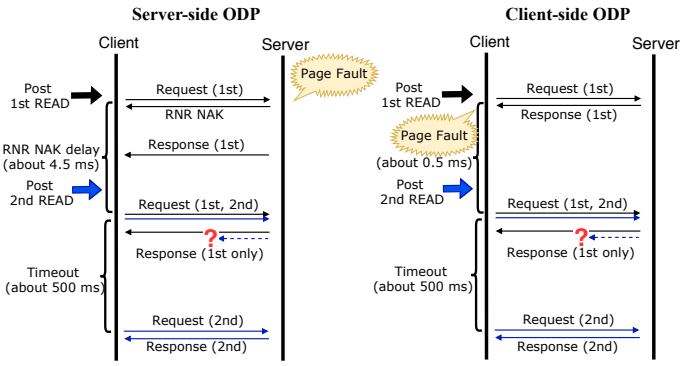
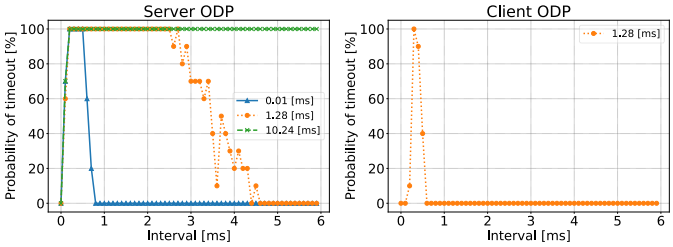


Fig. 5. Workflow of ODP with two READ operations based on the packets obtained via ibdump.



(a) Server-side ODP (b) Client-side ODP

Fig. 6. Probability of occurrence of timeout out of 10 trials with varying intervals of two READ in the server-side ODP and client-side ODP. The value of the legend represents minimal RNR NAK delay.

ODP is mainly due to the page fault, which usually takes only several hundred microseconds. We also ran the same micro-benchmark with the client-side ODP and server-side ODP, and the long execution time was similarly observed.

To analyze this exceptional phenomenon, we captured InfiniBand packets using ibdump. Figure 5 illustrates the occurrence based on the information from the packets and page fault counters. The figure shows that the long latency resulted from the timeout of the second READ operation. The response of the second READ seemed to disappear for some reason, and it forced the client to wait for the timeout. We term this phenomenon as packet damming because the transmission of packets is dammed for a long time.

To investigate exactly when packet damming occurred, we measured the probability of timeout in the server-side ODP and client-side ODP respectively. For the server-side ODP, we changed the pending period specified by minimal RNR NAK delay. Figure 6a shows that in the case of minimal RNR NAK delay of 1.28 ms, the timeout occurred up to around the interval of 4500 µs, which corresponded exactly to the actual RNR NAK delay represented in Figure 1. In addition, if the pending period was changed, the range of intervals followed it. Figure 6b shows the result of the client-side ODP and indicates that the timeout occurred up to around 500 µs of the interval, which corresponded to the retransmission interval of client-side ODP represented in Figure 1. In summary, packet damming occurred when the second request was posted

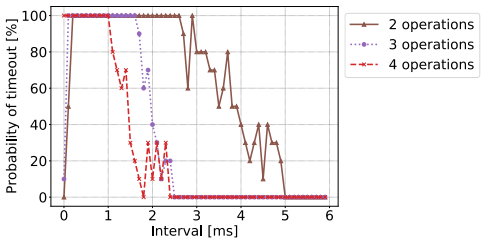


Fig. 7. Probability of timeout out of 10 trials with varying intervals of each READ operations in the both-side ODP. The number of READ operations is changed between 2 and 4. Minimal RNR NAK is set to 1.28 ms.

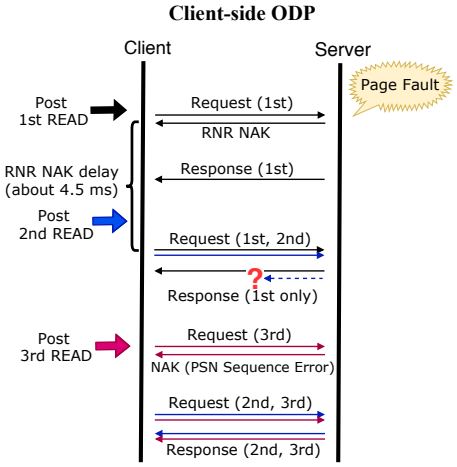


Fig. 8. Workflow of ODP with three READ operations based on the packet captured with ibdump.

during the first request's pending period when preparing for retransmission.

B. Performance with More Than Two READs

Figure 7 shows the probability of the timeout, with varying numbers of READ operations in the micro-benchmark. Increase in the number of READ operations surprisingly narrowed down the range of intervals in which the timeout occurred. This phenomenon can be explained by the Packet Sequence Number (PSN) management in RC. In InfiniBand, any packet contains a PSN to detect missing or out-of-order packets. When the responder detects a packet with an unexpected PSN, it returns the NAK to the requester with a PSN sequence error.

Figure 8 depicts the process with three READ operations. Even after the packet loss of the second READ operation, the server expected the second READ request to arrive. Therefore, if the third READ request was issued from the client after the packet loss, the server acknowledged the third request as an unexpected request, and returned the NAK with the PSN sequence error. After the client received the NAK, it retransmitted all the requests that were not completed. The noteworthy aspect is that the retransmission was conducted for the second and third READ operations immediately, and *the*

timeout never happened. However, the timeout still occurred even with more than two operations when the interval was small enough for all the READ operations to fit into the pending period of the first READ.

C. Experiments with Other Conditions

To explore and analyze the packet damming, we conducted experiments with other conditions and discovered the following.

- It occurred independent of other QPs. The QP still continued waiting for the timeout and caused a long latency even if new operations were posted in other QPs.
- It occurred regardless of whether the communication buffer in each communication operation was on the same page or not.
- It was not related to the page fault on the second (or later) communication. It occurred even when all the communication buffers were used and touched in advance, except for the one for the first communication. Even it occurred when the second operation was WRITE or SEND.
- It was irrelevant to the size of the communication buffer.
- It occurred in various systems with ConnectX-4 servers, including Reedbush-H/L, ABCI, and ITO in Table I. Nevertheless, we have not observed it with ConnectX-6 servers with the micro-benchmark so far.

VI. Packet Flood

In this section, we describe the second performance pitfall of ODP: *packet flood*. Packet flood is a performance bug involving a delay in the order of seconds, which is accompanied by a massive number of packets. It can occur when READ operations are issued from multiple QPs and cause simultaneous page faults.

As in the previous section, we reproduced the packet flood with the same micro-benchmark represented in Figure 3 and analyzed it using ibdump. While the experiment in the previous section involved only one QP, this one involved multiple QPs. The experimental environment was also the same as that introduced in the previous section. In this section, we set the minimal RNR NAK delay to be $1.28\ \mathrm{ms}$ and $C_{\mathrm{ACK}} = 18$.

A. Impact of Packet Flood

Figure 9a shows how large impact the number of QPs had on the performance when using the same number of READ operations. When a small number of QPs were used, the ODP performance was generally normal and acceptable, with the execution time almost falling in the range of the common overhead of page faults. When the number of QPs exceeded 10, however, the ODP performance degraded drastically and became as much as 3,000 times worse than that for the case without ODP.

To analyze this inexplicable performance degradation, we also plotted the number of packets collected by ibdump as shown in Figure 9b. Surprisingly, the number of packets with the client-side ODP was hundreds of times greater than that without ODP, which indicated that a large number of retransmissions occurred with the client-side ODP. This observation implies that the page fault on the client side was not resolved for a long time, and retransmission from the client was repeated hundreds of times. We term this performance issue with multiple QPs as packet flood because of the tremendous number of packets involved. We conducted the same experiment on different environments and confirmed that it also occurred when using a ConnectX-6 adapter [31].

B. Further Analysis

To further investigate the root cause of packet flood, we conducted another experiment with the same micro-benchmark to observe how each communication progresses. At this time, we used a message size of 32 bytes and 128 QPs, and client-side ODP was adopted. Therefore, the memory layout would be as indicated in Figure 10.

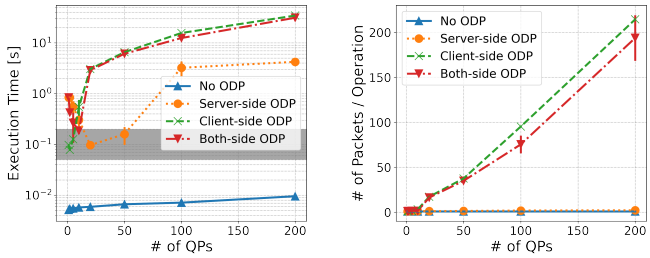
First, we focus on the result with 128 operations in Figure 11a. Because only one page was involved in this situation, we could expect that all the communications were completed immediately as soon as the single page fault was resolved. However, in reality, this was not the case. The result shows that the communication began to complete around 1 ms, and this can be interpreted as the resolution of the page fault in the RNIC. Nevertheless, the first 30 operations remained unfinished until 6 ms, which implied that they were unaware of the resolution of the page fault for approximately 5 ms. In other words, ODP with multiple QPs can cause *update failure of page statuses for a long time*.

During this period, the client kept repeating the retransmission of the request, receiving the response, and discarding it, assuming that the NIC did not have the corresponding entry. The period became longer and unacceptable when more operations and pages became involved, as shown in Figure 11b. In summary, packet flood occurs when multiple READ operations from multiple QPs cause simultaneous page faults followed by update failure of page statuses.

C. Discussion

The simultaneous page faults themselves could occur regardless of whether they adopted server-side ODP or client-side ODP, but packet flood occurred only with the client-side ODP as shown in Figure 9b. This is because update failure of page statuses occur only with the client-side ODP, which is explained by the fact that clients are always responsible for retransmission.

When a page fault occurs on the server side, all the servers should do is send RNR NAK back to the client, and the requests that cannot be processed can be completely ignored. On the other hand, client-side ODP works differently. When a page fault occurs on the client side, the client itself should retain the information about the communication for later retransmission. In other words, the client is stateful while the server is stateless. This explains why the information about the page statuses was kept and not updated for a long time only with client-side ODP.



(a) Execution Time (s) (b) Number of Packets

Fig. 9. Effect of varying the number of QPs with the micro-benchmark in Figure 3. We fixed the number of READ operations at 8192 and size of communication at 100 bytes with 200 pages involved. The gray bar in the left figure shows the range of unavoidable overheads of ODP, assuming that each page fault in RNICs takes 250–1000 µs to be resolved in common cases.

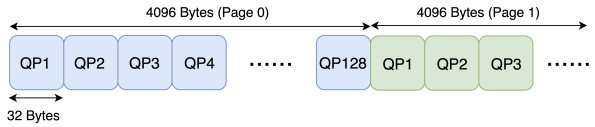
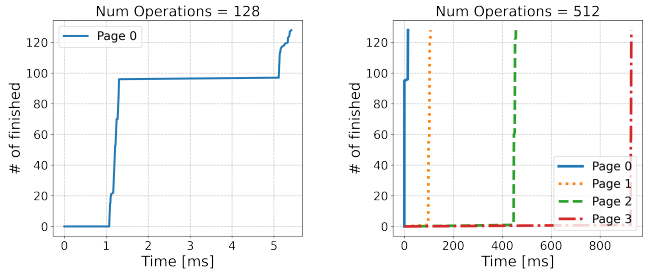
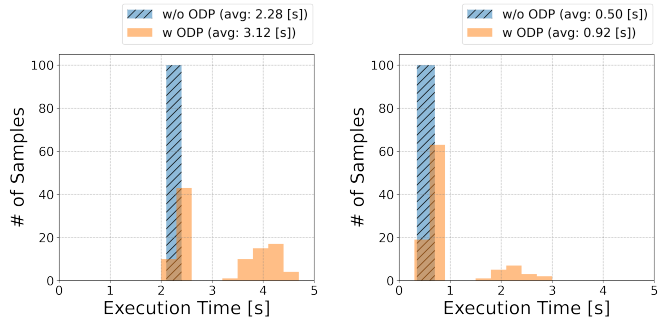


Fig. 10. Memory layout for communication buffer and assignment of QPs



(a) 128 Operations (b) 512 Operations

Fig. 11. Number of completed operation per page. 128 QPs and a message size of 32 bytes is utilized and client-side ODP is adopted.



(a) KNL (2 nodes) (b) Reedbush-H (2 nodes)

Fig. 12. Execution time distribution when running simple ArgoDSM benchmark which contains only argo::init() and argo::finalize() with ODP disabled/enabled in each system. 10 MB was passed to argo::init() as the memory size for the initialization.

Given this, a few aspects remain to be mentioned regarding Figure 9a. On the one hand, the performance of the both-side ODP and client-side ODP was able to be explained by packet flood, while on the other hand, the performance degradation of the server-side ODP was too large to ignore. We observed that with a more detailed investigation, it resulted from the packet loss and consequent timeout explained in Section V. More interestingly, we found that the timeout interval lengthened with multiple QPs compared with the case when only a single QP was used. This can be explained by the fact that the client should handle the number of communications proportional to the number of QPs concurrently, and a high load is imposed on the client by managing the RNR timer and retransmission.

VII. Impacts on Applications

In this section, we show that the two performance pitfalls of ODP can affect applications in several environments and degrade the performance. Specifically, we show that packet damming occurs in ArgoDSM and packet flood occurs in SparkUCX. These systems do not use ODP in the default configuration, but we enable it using environmental variables of UCX [23] to evaluate the potential of ODP. The default configuration of UCX uses minimal RNR NAK delay of 0.96 ms and $C_{\mathrm{ACK}} = 18$. The MPI library used in the experiments of ArgoDSM was MPICH 3.3. The machines used in the experiments are shown in Table II, and only KNL system allow us to use sudo authority and ibdump.

A. ArgoDSM

ArgoDSM[4] [22] is a software distributed shared memory (DSM) system that utilizes RDMA to maintain cache coherency. It employs a home-node directory protocol without message handlers, and all the operations are performed by RDMA over MPI RMA, which invokes UCX internally. By running a tutorial and test codes of ArgoDSM with ODP enabled, we found that packet damming occurs in the initialization phase of ArgoDSM, where first touches and page faults are abundant.

To illustrate this more quantitatively, we created a simple benchmark that contained only the initialization and finalization of ArgoDSM and measured the execution time for 100 trials. We plotted the result with a histogram as in Figure 12, which shows that the samples with enabling ODP can be divided into two groups by the execution time in each system. We used ibdump to analyze the KNL and confirmed that the timeout actually appeared in the group with the longer execution time, while it did not appear in the group with shorter execution time. The timeout was caused by one READ operation followed by one SEND operation, whose packet dropped in the middle of communication when one node tried to take the global lock and accessed the remote address. Although we cannot analyze the Reedbush-H using ibdump, we can make a similar conjecture that the group with the longer execution time also causes packet damming.

[4]https://github.com/etascale/argodsm

SparkTC

	QPs	Disable [s]	Enable [s]	**Enable/Disable**
KNL (2)	411	303 ± 20	473 ± 65	1.56
Reedbush-H (2)	980	39.7 ± 0.38	256 ± 150	6.46
ABCI (2)	2191	83.9 ± 2.4	84.9 ± 1.5	1.01
ABCI (4)	2858	41.7 ± 1.4	59.3 ± 3	1.42

mllib.RecommendationExample

	QPs	Disable [s]	Enable [s]	**Enable/Disable**
KNL (2)	210	100 ± 3.2	151 ± 67	1.51
Reedbush-H (2)	980	21.9 ± 0.82	78.6 ± 46	3.59
ABCI (2)	2191	29 ± 0.55	31.2 ± 1.9	1.07
ABCI (4)	1953	24.3 ± 0.64	28.6 ± 1.1	1.18

mllib.RankingMetricsExample

	QPs	Disable [s]	Enable [s]	**Enable/Disable**
KNL (2)	389	517 ± 3.7	674 ± 140	1.30
Reedbush-H (2)	980	46.6 ± 1.4	111 ± 56	2.38
ABCI (2)	2191	107 ± 1.8	147 ± 1.6	1.37
ABCI (4)	2667	83.2 ± 3.4	197 ± 4.7	2.37

Fig. 13. Comparison of execution time of SparkUCX with ODP enabled and disabled. We conducted 10 trials and omitted the samples that failed to finish, especially with `IBV_WC_RETRY_EXC_ERR` when the retransmission count exceeded the limit.

B. SparkUCX

Apache Spark is a distributed in-memory data processing framework widely used in industry and academia. Spark shuffling is a process to redistribute or re-partition data and is known to be costly in terms of CPU, memory, and network. SparkUCX[5] [21] is an RDMA acceleration plugin of Spark, which aims to accelerate the shuffling by utilizing UCX. The SparkUCX-enabled Spark will be referred to as SparkUCX from herein. In this experiment, we ran several examples included in the Spark package and compared the performance between the case enabled and disabled ODP. We constructed our Spark cluster using standalone mode by starting a master process and worker processes by hand. For Reedbush-H and ABCI, we allocated one worker process to each worker machine and prepared two additional machines for the master process and the job-submission process. For KNL, which we only possess two machines, we allocated the master process and one worker process to one machine and one worker process and the job-submission process to the other machine. We separate the processes using numactl with half of the cores assigned to each process. The examples we chose are SparkTC, mllib.RecommendationExample and mllib.RankingMetricsExample, all of which include join operations of Spark, which issues READ operations internally. To configure SparkUCX, we adjusted the timeout of Spark, memory, and parallelism of executors for each environment.

Table 13 shows that the examples created several hundreds to several thousands QPs, and enabling ODP degrades the performance by up to 6.46 times. We observed that SparkUCX sometimes got stuck intermittently for a few seconds with ODP enabled. Packets in the middle of these period in the KNL showed that packet flood occurred and many READ packets were retransmitted every several tens of milliseconds. Although we are not permitted to use ibdump with sudo authority in Reedbush-H and ABCI systems, we can presume that the same phenomenon takes place in these systems. The degree of performance degradation with ODP differs from each system and each example because packet flood is intimately related to the timing issue.

[5]https://github.com/openucx/sparkucx

VIII. Related Work

A. Memory Registration Methods

Memory (de)registration of InfiniBand (or other RDMA-capable interconnects) is known to involve considerable runtime overhead mainly because of the (un)pinning operation of user-space pages, while leaving excessive memory registered wastes physical memory directly. The standard approach to balancing the runtime overhead and the spatial overhead is pin-down cache [16], which enables us to reuse pinned buffers internally by postponing actual deregistration (i.e., unpinning). With the pin-down cache by Tezuka et al. [16], the actual deregistration of a buffer occurred in the least recently used (LRU) order when the total size of registered buffers exceeded the maximum size. Zhou et al. [15] introduced batched deregistration to suppress the average cost of deregistring buffers. Ou et al. [12], [32] developed a more sophisticated LRU-based cache replacement scheme that divided the LRU stack into three sections.

Even in the presence of pin-down cache, several performance tradeoffs are known to exist. The Unifier [14] caching system addressed a tradeoff between (de)registration cost and spatial cost. Larger pinned down buffers suppress on-demand pinning in dynamic registration, whereas they occupy more physical memory, which other computations should have used. Frey and Alonso [11] argued a tradeoff between pinning (i.e., newly registering) and copying. They experimentally showed that for 256-KB or larger regions, it was more efficient to newly register by an order of magnitude.

The breakdown of the memory registration process is also important for efficient memory management. Mietke et al. [13] analyzed the registration process inside the Mellanox InfiniBand driver and provided clues to improve it. For example, it drastically reduced the cost of the `mlock` system call by making a separate kernel thread zero-fill pages when they were not present.

B. Performance Analysis of On-Demand Paging

Only a few studies have analyzed the performance characteristics of ODP because it is an emerging technology. Lesokhin et al. [17] presented the first implementation of network page faults for InfiniBand and experimentally analyzed the overhead of page faults and invalidation. Their breakdowns showed that the overhead of page faults was dominated by hardware-level interrupts and transmission resuming, while the invalidation spent most of its time on updating page tables. Li et al. [20] presented a performance analysis of Explicit ODP to design a memory-efficient MPI library. They compared Explicit ODP with pin-down cache in latency and bandwidth

and revealed that page faults incurred performance degradation dominantly. Their results also indicated that the characteristics of page faults on the sender-side were different from those on the receiver side and that prefetching on the receiver side effectively worked. In their later work [19] on Implicit ODP, they revealed that the overhead of page faults was able to mitigate through an elaborate tuning of the RNR NACK timer.

These existing studies did not focus on retransmission and timeout, relying upon the hardware-level reliability of InfiniBand. To the best of our knowledge, our work is the first to experimentally analyze the retransmission and timeout of packets derived from page faults.

C. Reliability of InfiniBand

There is some literature on analyzing the reliability of InfiniBand. Koop et al. [33] evaluated the cost of reliability by comparing the hardware-based implementation and the software-based implementation. They implemented MPI over the UC transport and compared it experimentally with MPI over the RC and UD transports. They showed that a software-based approach was not only feasible but able to achieve higher performance because of its smaller memory consumption. Meanwhile, they also showed that their UD-based implementation incurred no packet loss for NAS Parallel Benchmarks with 256 processes and for three applications with 1024 processes [34]. This was due to the reliability provided by the link-layer protocol. Assuming this link-layer reliability, Kalia et al. [8] developed remote procedure calls over the UD transport of InfiniBand. They designed it to detect packet loss with coarse-grained timeouts of RPCs because they did not observe practically packet loss. For the same reason, the key-value store HERD [10] was also designed to sacrifice transport-level retransmission for common-case performance at the cost of rare application-level retries.

As seen from the literature above, even if we use unreliable transports, we are rarely faced with long timeouts of InfiniBand. Long timeouts were known to few practitioners [29], [30] but have never been a technical issue in research because of the rarity. To the best of our knowledge, through the case of packet damming under ODP, our work is the first to find that long timeouts can actually be harmful to InfiniBand systems.

IX. Concluding Remarks

In this study, we have reverse-engineered the behavior of ODP using ibdump and have found two severe retransmission-related performance pitfalls: packet damming and packet flood. Packet damming occurs when a READ operation involves other operations within a certain interval, and its root cause is packet loss and subsequent timeout. Packet flood occurs when READ operations invoke simultaneous page faults using multiple QPs, and its root cause is update failure of page statuses. We have shown that they can appear in the software systems running on major computing clusters and degrade their performance by up to 6.3 times.

A high-level insight of this work is the hardness of the hardware implementation of network page fault handling by fully utilizing the RC mechanism. In particular, handling concurrent page faults in the RNICs is technically difficult because of their limited memory and functionality.

A. Lessons Learned

The pitfalls are naturally critical in terms of the super-long latency, but they are problematic for the difficulty of the detection as well. First, they are related to retransmission and appear without spitting out the associated errors. Second, they are highly affected by the timing of communication operations, which prevents us from reproducing them invariably. Detecting the pitfalls becomes extremely hard without observing the raw packets, and in Section VII, we struggled to verify the occurrence of pitfalls in the non-sudo environments. In addition, it is difficult to detect them from the application side because there are usually many software layers between the application and InfiniBand verbs. In fact, it took several months for us to identify that the performance degradation resulted from ODP since we had encountered packet damming with running high-level applications on top of UCX for the first time. Why we took so long is partly because UCX prioritized ODP over direct memory registration by default, and we were even unaware of the use of ODP in the first place. This is the worst scenario possible, but we should manage to cope with them by some means.

However, changing and modifying the hardware currently deployed is not realistic. From here, we present workarounds for the pitfalls and guidelines for ODP from the software side. Regarding packet damming, we have two workarounds conceivable. First, we should set the smallest value to minimal RNR NAK delay to narrow down the range of the intervals in which the timeout occurs. As shown in the previous study [19], this setting can also reduce the resolution time of the client-side ODP and, therefore, we should actively adopt it. Second, we can avoid the long delay by posting an additional communication. Section V-B has shown that increasing the number of communication operations has provided more chances for the responder to detect the PSN Sequence Error and, subsequently, reduced the possibility of the timeout. The naive way to achieve this functionality is by implementing a software timer with appropriate granularity to issue a dummy communication periodically.

As for packet flood, issuing the same communication again might work because the page fault itself is actually solved during the packet flood, and the update failure of page statuses no longer applies to the new one. However, this is not a straightforward solution and requires careful design of an additional communication layer. As a guideline of ODP, occurrence of packet flood should be kept in mind with multiple QPs, and ODP should be carefully applied for regions that can be accessed from multiple QPs with a high probability.

B. Future Work

Even with these pitfalls, the concept of ODP is still worthy to pursue. As shown in the previous study [17], the common-case performance of ODP is surely acceptable as long as they

are not involved. To overcome these flaws in the use of READ essentially, not only software-level workarounds described in the previous subsection, hardware-level improvement (or bugfix) is needed.

Packet damming can be easily solved with a short timeout, and therefore, we should investigate whether the lower limit can be changed easily or it encounters other big problems. Regarding packet flood, revealing mechanisms of interaction between QPs and page statuses is urgent. To proceed with the investigation, cooperation with hardware vendors is necessary, and, in fact, we have already reported the two pitfalls to Mellanox for contributing to the improvement of ODP. They have reported that packet damming is a problem derived from a method specific to ConnectX-4 for processing page faults, and it vanishes in later models. Nevertheless, considering that the long timeout remains even in the latest InfiniBand cards, it is still valuable to investigate other performance pitfalls related to the long timeout. Packet flood is more serious in that it remains in the latest InfiniBand cards, and we are waiting for the investigation report from them.

Acknowledgments

We would like to thank NVIDIA Corporation (formerly Mellanox Technologies) for their beneficial feedback based on the in-depth investigation of the pitfalls. We thank Wataru Endo for his giving the starting point of this work through his system on top of UCX. We thank Shumpei Shiina for his comments on our manuscript useful for improving the presentation. We also acknowledge the feedback of anonymous reviewers.

References

[1] Mellanox, "Interconnect your future—enabling the best datacenter return on investment," https://www.mellanox.com/related-docs/solutions/hpc/TOP500-NOVEMBER-2019.pdf, 2019.

[2] R. Mittal, A. Shpiner, A. Panda, E. Zahavi, A. Krishnamurthy, S. Ratnasamy, and S. Shenker, "Revisiting network support for RDMA," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. ACM, 2018, pp. 313–326.

[3] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "RDMA over commodity ethernet at scale," in *Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '16. ACM, 2016, pp. 202–215.

[4] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. Haj Yahia, and M. Zhang, "Congestion control for large-scale RDMA deployments," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. ACM, 2015, pp. 323–536.

[5] J. Xue, Y. Miao, C. Chen, M. Wu, L. Zhang, and L. Zhou, "Fast distributed deep learning over RDMA," in *Proceedings of the 14th EuroSys Conference 2019*, ser. EuroSys '19. ACM, 2019, pp. 1–14.

[6] X. Wei, Z. Dong, R. Chen, and H. Chen, "Deconstructing RDMA-enabled distributed transactions: Hybrid is better!" in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '18. USENIX, 2018, pp. 233–251.

[7] Y. Lu, J. Shu, Y. Chen, and T. Li, "Octopus: an RDMA-enabled distributed persistent memory file system," in *Proceedings of the 2017 USENIX Annual Technical Conference*, ser. USENIX ATC '17. USENIX, 2017, pp. 773–785.

[8] A. Kalia, M. Kaminsky, and D. G. Andersen, "FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '16. USENIX, 2016, pp. 185–201.

[9] ——, "Design guidelines for high performance RDMA systems," in *Proceedings of the 2016 USENIX Annual Technical Conference*, ser. USENIX ATC '16. USENIX, 2016, pp. 437–450.

[10] ——, "Using RDMA efficiently for key-value services," in *Proceedings of the 2014 ACM conference on SIGCOMM*, ser. SIGCOMM '14. ACM, 2014, pp. 295–306.

[11] P. W. Frey and G. Alonso, "Minimizing the hidden cost of RDMA," in *Proceedings of 2009 29th IEEE International Conference on Distributed Computing Systems*, ser. ICDCS '09. IEEE, 2009, pp. 553–560.

[12] L. Ou, X. He, and J. Han, "MRRC: An effective cache for fast memory registration in RDMA," in *Proceedings of 14th NASA Goddard, 23rd IEEE Conference on Mass Storage Systems and Technologies*, ser. MSST '06, 2006.

[13] F. Mietke, R. Rex, R. Baumgartl, T. Mehlan, T. Hoefler, and W. Rehm, "Analysis of the memory registration process in the Mellanox InfiniBand software stack," in *Euro-Par 2006 Parallel Processing*, ser. Lecture Notes in Computer Science, vol. 4128. Springer, 2006, pp. 124–133.

[14] J. Wu, P. Wyckoff, D. Panda, and R. Ross, "Unifier: Unifying cache management and communication buffer management for PVFS over InfiniBand," in *Proceedings of 2004 IEEE International Symposium on Cluster Computing and the Grid*, ser. CCGrid '04. IEEE, 2004, pp. 523–530.

[15] Y. Zhou, A. Bilas, S. Jagannathan, C. Dubnicki, J. F. Philbin, and K. Li, "Experiences with VI communication for database storage," in *Proceedings 29th Annual International Symposium on Computer Architecture*, ser. ISCA '02. IEEE, 2002, pp. 257–268.

[16] 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 the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, ser. IPPS/SPDP '98. IEEE, 1998, pp. 308–314.

[17] I. Lesokhin, H. Eran, S. Raindel, G. Shapiro, S. Grimberg, L. Liss, M. Ben-Yehuda, N. Amit, and D. Tsafrir, "Page fault support for network controllers," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. ACM, 2017, pp. 449–466.

[18] *Understanding On Demand Paging (ODP)*, Mellanox, 2019, https://community.mellanox.com/s/article/understanding-on-demand-paging--odp-x.

[19] M. Li, X. Lu, H. Subramoni, and D. K. Panda, "Designing registration caching free high-performance MPI library with implicit on-demand paging (ODP) of InfiniBand," in *Proceedings of 2017 IEEE 24th International Conference on High Performance Computing*, ser. HiPC '17. IEEE, 2017, pp. 62–71.

[20] M. Li, K. Hamidouche, X. Lu, H. Subramoni, J. Zhang, and D. K. Panda, "Designing MPI library with on-demand paging (ODP) of InfiniBand: Challenges and benefits," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. IEEE, 2016, pp. 433–443.

[21] P. Rudenko, "Sparkucx – rdma acceleration plugin for spark," 2020, 2020 Virtual OFA Workshop.

[22] S. Kaxiras, D. Klaftenegger, M. Norgren, A. Ros, and K. Sagonas, "Turning centralized coherence and distributed critical-section execution on their head," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '15. ACM, 2015, pp. 3–14.

[23] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, Y. Shahar, S. Potluri, D. Rossetti, D. Becker, D. Poole, C. Lamb, S. Kumar, C. Stunkel, G. Bosilca, and A. Bouteiller, "UCX: An open source framework for HPC network APIs and beyond," in *Proceedings of 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, ser. HOTI '15. IEEE, 2015, pp. 40–43.

[24] P. Grun, S. Hefty, S. Sur, D. Goodell, R. D. Russell, H. Pritchard, and J. M. Squyres, "A brief introduction to the OpenFabrics interfaces: A new network API for maximizing high performance application efficiency," in *Proceedings of 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, ser. HOTI '15. IEEE, 2015, pp. 34–39.

[25] *InfiniBand Architecture Specification Volume 1*, Release 1.4 ed., InfiniBand Trade Association, 2020, https://cw.infinibandta.org/document/dl/8567.
[26] "Introduction to the reedbush supercomputer system," https://www.cc.u-tokyo.ac.jp/en/supercomputer/reedbush/system.php.
[27] "About ABCI: Computing resources," https://abci.ai/en/about_abci/computing_resource.html.
[28] "Introduction of ITO," https://www.cc.kyushu-u.ac.jp/scp/eng/system/ITO/01_intro.html, 2018.
[29] M. Nakamura, "Understanding retransmission control in infiniband," http://www.nminoru.jp/~nminoru/network/infiniband/iba-retransmission.html, 2014, in Japanese.
[30] B. Hudzia, "On allowing shorter timeout on mellanox cards and other tips and tricks," https://www.reflectionsofthevoid.com/2014/02/on-allowing-shorter-timeout-on-mellanox.html, 2014.
[31] T. Fukuoka, "Finding and analyzing performance pitfalls of on-demand paging of infiniband," Master's thesis, The University of Tokyo, 2021.
[32] L. Ou, X. He, and J. Han, "An efficient design for fast memory registration in RDMA," *Journal of Network and Computer Applications*, vol. 32, no. 3, pp. 642–651, 2009.
[33] M. J. Koop, R. Kumar, and D. K. Panda, "Can software reliability outperform hardware reliability on high performance interconnects?: A case study with MPI over InfiniBand," in *Proceedings of the 22nd Annual International Conference on Supercomputing*, ser. ICS '08. ACM, 2008, pp. 145–154.
[34] M. J. Koop, S. Sur, Q. Gao, and D. K. Panda, "High performance MPI design using unreliable datagram for ultra-scale InfiniBand clusters," in *Proceedings of the 21st annual international conference on Supercomputing*, ser. ICS '07. ACM, 2007, pp. 180–189.