修士論文
Master's Thesis

Finding and Analyzing Performance Pitfalls of On-Demand Paging of InfiniBand

(InfiniBandのオンデマンドページングの
性能問題の発見と分析)

Takuya Fukuoka
福岡 拓也

学籍番号: 48-196438

January 28, 2021

Department of Information and Communication Engineering
Graduate School of Information Science and Technology
The University of Tokyo
東京大学大学院 情報理工学系研究科 電子情報学専攻

Advisor Prof. Kenjiro Taura
指導教員 田浦 健次朗 教授

Abstract

As the importance of distributed computing is increasing, the performance of interconnects, which connect each machine, is critical as well as the performance of individual machines. Remote direct memory access (RDMA) is a technology employed in interconnects such as InfiniBand and high-speed Ethernet, and is famous for its ultra-low latency and high bandwidth. RDMA achieves the benefits by kernel-bypassing and zero-copy transfer, that is, enabling communication without involving remote CPUs and directly transferring the memory between user-level buffers and NICs. Because of the kernel-bypassing nature, however, conventional RDMA technology requires registering the memory before issuing communication. This memory registration process includes pinning down communication buffers to make them accessible from remote CPUs, and is known to be a quite expensive operation with non-negligible latency and CPU load. There has been much work on alleviating the cost of memory registration from the software side, but there still remain problems both in the efficiency and productivity of memory management.

Recently, Mellanox has addressed these issues by introducing On-Demand Paging (ODP) into InfiniBand. ODP is a hardware functionality of InfiniBand to register/de-register communication buffers automatically depending on the situation. When the required memory is not registered under ODP, the RDMA NIC (RNIC) asks the OS to cause a *network page fault* and to provide physical memory and its mapping. Previous research showed the cost of network page fault was acceptable with latency of several hundreds of microseconds. However, it only measured the network page faults themselves and little investigated that of retransmission derived from the network page faults.

In this thesis, 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 while packet flood is massive repetitive retransmission incurred by a long delay of updating page statuses. We show their latencies are longer by 3–4 orders of magnitude than the overhead of a network page fault itself. We also demonstrate the revealed pitfalls are actually harmful to existing software systems through experiments with SparkUCX and ArgoDSM.

Contents

1 Introduction **1**

1.1 Overview 1

1.2 Contributions 3

1.3 Outline 3

2 Background **4**

2.1 Virtual Memory 4

2.2 InfiniBand 5

2.2.1 Remote Direct Memory Access (RDMA) 5

2.2.2 Memory Registration 6

2.2.3 InfiniBand Verbs 7

2.2.4 Transport Types 8

2.2.5 Retransmission Handling in RC 9

2.3 On-Demand Paging (ODP) 11

2.3.1 Basic Mechanism 12

2.3.2 Packet Handling 12

3 Related Work **13**

3.1 Memory Registration Methods 13

3.2 Performance Analysis of On-Demand Paging 14

3.3 Reliability of InfiniBand 15

3.4 Performance Difference between Mellanox RNICs Generations 15

3.5 Communication Libraries over Various Interconnects 16

3.5.1 Message Passing Interface (MPI) 16

3.5.2 Unified Communication X (UCX) 16

3.5.3 Libfabric 17

3.6 RDMA-based Systems 17

3.6.1 Key-value Store (KVS) Systems 17

3.6.2 Global Address Space (GAS) Systems . . . 18

4 On-Demand Paging in Reality **20**

4.1 Ibdump . . . 20

4.2 Actual Behaviors of ODP . . . 20

4.3 Timeout for the Worst Case . . . 22

5 Performance Pitfalls of On-Demand Paging **25**

5.1 Packet Damming . . . 25

5.1.1 Pitfalls with Two READ Operations . . . 25

5.1.2 Performance with More Than Two READ Operations . . . 27

5.1.3 Experiments with Other Conditions . . . 28

5.2 Packet Flood . . . 30

5.2.1 Impact of Packet Flood . . . 30

5.2.2 Further Analysis . . . 30

5.2.3 Discussion . . . 32

5.2.4 Analysis of WRITE Operations . . . 33

6 Impacts of Pitfalls on Applications **36**

6.1 ArgoDSM . . . 36

6.1.1 Evaluation of the Initialization and Finalization . . . 36

6.1.2 Evaluation of Simple Example in Tutorial . . . 37

6.2 SparkUCX . . . 38

7 Concluding Remarks **41**

7.1 Lessons Learned . . . 41

7.2 Future Work . . . 42

Acknowledgement **43**

Publications **44**

Bibliography **44**

A Performance Analysis with the Latest InfiniBand Adapter **52**

B Code snippets of the micro-benchmark **55**

List of Figures

2.1 InfiniBand network architecture 6
2.2 Illustration on how InfiniBand bypasses buffer copies 6
2.3 QP model introduced in InfiniBand verbs 8
2.4 The first situation in which retransmission occurs. 9
2.5 The second situation in which retransmission occurs. 10
2.6 The third situation in which retransmission occurs. 11
2.7 Page fault and invalidation flows of ODP 12

3.1 UCX architecture 17

4.1 Screenshot of Wireshark with rich GUIs 22
4.2 Workflow of ODP with a single READ operation observed with idbump. 23
4.3 Workflow of ODP with a single SEND operation observed with idbump. 23
4.4 Workflow of ODP with a single WRITE operation observed with idbump. 24
4.5 T_o measured by varying C_{ACK} on different InfiniBand systems 24

5.1 Micro-benchmark in simplified C code 26
5.2 Average execution time of the micro-benchmark out of 10 trials with varying intervals between two communications 26
5.3 Workflow of ODP with two READ operations based on packets obtained via ibdump. . 27
5.4 Probability of occurrence of timeout out of 10 trials with varying intervals of two READ operations in the server-side ODP and client-side ODP 28
5.5 Probability of timeout out of 10 trials with varying intervals of each READ operations in the both-side ODP 28
5.6 Workflow of ODP with three READ operations based on the packet captured with ibdump. 29
5.7 Effect of varying the number of QPs with the micro-benchmark in Figure 5.1 with READ operations. 31
5.8 Memory layout for communication buffer and assignment of QPs 31
5.9 Number of completed operations per page 32

5.10 Effect of varying the number of QPs with the micro-benchmark in Figure 5.1 and WRITE operations. 33
5.11 A difference between WRITE and READ operations when N operations are posted into one QP at the same time. 34
5.12 Execution time varying the number of WRITE operations with the micro-benchmark in Figure 5.1. 35

6.1 Execution time distribution when running an ArgoDSM benchmark which contains only argo::init() and argo::finalize(). 37
6.2 Fragment of packets while the ArgoDSM benchnark containing only argo::init() and argo::finalize() is running on KNL-4 37
6.3 Pseudocode of calculating the maximum value from an integer array using multiple threads. 38
6.4 Execution time distribution when running the Simple Example of ArgoDSM 39
6.5 Comparison of execution time of SparkUCX with ODP enabled and disabled. 40

A.1 Effect of varying the number of QPs with ConnectX-6 adapters. 52
A.2 Number of completed operations per page with ConnectX-6 adapters. 53
A.3 Effect of varying the number of QPs with the micro-benchmark in Figure 5.1 with WRITE operations and ConnectX-6 adapters. 54

B.1 List of configurations which we changed depending on the experiments in this thesis. . . 55

List of Tables

2.1 Transport Types of InfiniBand 9
2.2 Encoding for minimal RNR NAK delay 10

4.1 InfiniBand systems and details on their RNICs 21
4.2 Experimental environments 21

Chapter 1

Introduction

1.1 Overview

Distributed computing, which utilizes multiple computing machines, is essential in our daily life. Many companies possess datacenters to meet the demand of Artificial Intelligence (AI) and Big Data, and scientific computations are conducted on supercomputers. In these computing platforms, the performance of interconnects, which connect each machine, is critical as well as the performance of individual machines. Among many kinds of interconnects, datacenters and supercomputers tend to choose InfiniBand for high-performance computing (HPC) and AI infrastructures, and high-speed Ethernet for cloud and hyper-scale platforms [36].

Remote direct memory access (RDMA) is a popular technology employed in these interconnects for its ultra-low latency and high bandwidth. RDMA achieves the benefits by enabling communication without involving remote CPUs and directly transferring the memory between user-level buffers and NICs. These characteristics are called kernel-bypassing and zero-copy transfer, which makes it attractive for many distributed systems [13, 14, 23, 25, 34, 39, 54, 55, 59]. RDMA was originally used by the HPC community with InfiniBand clusters, but has become popular in datacenters with commodity Ethernet via RoCE and iWARP [18, 40, 61].

While RDMA decreases the latency and network bandwidth, there is one big challenge that should be addressed. Because of the nature of kernel bypassing, conventional RDMA technology requires registering the memory before issuing communication. This memory registration process includes pinning down communication buffers to physical memory to avoid swapping it out and make it accessible from remote CPUs. It is reported that, however, memory registration is a quite expensive operation with non-negligible latency and CPU load, and registering memory every time communication is invoked degrades performance [16, 38]. Thus, effective management of communication buffers is needed to get the most of the performance of RDMA. A lot of research has been conducted about managing registration buffers so far [10, 16, 31, 44, 45, 53, 56, 57, 60], and one common approach, which is called pin-down cache, is

reusing pin-down areas to decrease the invocation of registration primitives.

However, the pin-down cache scheme is highly problematic in the efficiency and productivity of memory management. First of all, it requires to pin down some communication buffers and limits the memory available for computation because the pinned buffer is never swapped out. As the number of the cores in one chip increases, total memory required for communication increases, but the total size of memory is still constant, which makes overlapping communication and computation difficult [31]. Therefore, a complex mechanism for reusing pin-down caches is needed, but it leads to a complicated programming model with less productivity for RDMA-based systems [30].

Recently, Mellanox has addressed these issues by introducing On-Demand Paging (ODP) [30, 32, 33, 37] into InfiniBand. ODP is a functionality of InfiniBand hardware to page-in/-out communication buffers automatically depending on the situation. It frees developers from manual memory registrations and management of pin-down caches, and achieves automatic memory management. When the required memory is not mapped into the physical memory under ODP, the RDMA NIC (RNIC) asks the OS to cause a network page fault and to provide physical memory and its mapping. ODP also allows the communication buffers to be swapped out without deregistration, which can ensure enough computation memory and, therefore, achieves good communication-computation overlap. Furthermore, existing work [30] showed that it took an acceptable cost of only several hundred microseconds on network page faults. Therefore, the ODP feature seems to be a promising approach to taking both performance and productivity in IB networks.

However, the work focused only on the cost of network page faults themselves and little investigated that of retransmission derived from page faults. ODP is expected to work inside various RDMA-based software systems, and should not degrade the performance badly under any circumstances. To understand the actual cost of ODP, we need an in-depth investigation in various network situations, including the retransmission and timeout of packets.

In this thesis, 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 results in a latency of several hundreds of milliseconds. Packet flood is massive repetitive retransmission incurred by a long delay of updating page statuses, which results in a latency of a few seconds. These latencies are longer by 3–4 orders of magnitude than the overhead of a network page fault itself reported in the previous work [30]. We also demonstrate that the revealed pitfalls are actually harmful to existing software systems through experiments with SparkUCX [49] and ArgoDSM [26].

This thesis presents our experimental analysis and lessons learned therefrom. We believe that these are beneficial to both the hardware and software standpoints. Our experimental analysis pinpoints

hardware-level flaws, which would serve as useful clues for hardware vendors. Our practice and lessons would be useful hints to identify and/or avoid ODP-related performance bugs for the developers of communication software that can enable ODP, such as MVAPICH2-X [6], UCX [50], and libfabric [17].

1.2 Contributions

Our main contributions are summarized as follows.

- We present an in-depth experimental analysis of the actual behaviors of ODP (Section 4). 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 of InfiniBand, which we call *packet damming* (Section 5.1) and *packet flood* (Section 5.2) respectively. 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 6). Our experimental results show that they are actually harmful to existing software systems.

1.3 Outline

This thesis is organized as follows. Section 2 provides basic knowledge about virtual memory, InfiniBand, and RDMA. It also presents the details about the retransmission of InfiniBand and basic ODP mechanism based on the information from existing literature. Section 3 reviews previous work of memory registration methods, performance evaluation of ODP, reliability of RC, and performance difference between various RDMA NICs, which is important for understanding the positioning of our work. We also introduce existing work about RDMA in a broader scope, including communication libraries over various interconnects and RDMA-based systems. Section 4 provides comprehensive analysis about how ODP actually works based on packets gained via ibdump, a packet capturing tool for InfiniBand. Section 5 presents two performance pitfalls of ODP, *packet dammping* and *packet flood*. It clarifies how and when these pitfalls arise using various microbenchmarks. Section 6 shows that two performance pitfalls actually appear and be harmful in existing software systems. Finally, Section 7 presents lessons learned from our experiences and derives future work.

Chapter 2

Background

This chapter provides basic knowledge for understanding our work. First, we briefly review virtual memory, one of the important functionality the Operating System provides (Section 2.1). Then, we explain the fundamentals of InfiniBand and Remote Direct Memory Access (RDMA), including the details about the retransmission of InfiniBand (Section 2.2). Finally, we introduce On-Demand Paging (ODP) and provide its basic mechanism based on the information from existing literature (Section 2.3).

2.1 Virtual Memory

Virtual memory is a memory management technique used in the Operating System. Under virtual memory, application programmers handle virtual addresses instead of physical addresses, and the OS provides service of translating the virtual address to the physical address.

Virtual memory has three virtues as explained in the following [30].

1. Address space isolation

   A process should not be allowed to access the memory that other processes possess. Virtual memory assures that different physical memory regions are allocated to each process.

2. Simplified programming model

   Virtual memory provides a smooth, clean, and large enough memory space to the programmers. They do not have to manage non-contiguous physical memory regions resulted from repetitive memory allocation and de-allocation. The virtual address space can be also larger than the size of physical memory. This is achieved by a technology called memory swapping, which utilizes secondary storage. As the memory swapping is conducted by the OS automatically, the programmers do not have to be aware of it.

3. Canonical memory optimization

Virtual memory enables some optimization techniques such as demand paging and copy-on-write, which improves the utilization of memory and the performance of the system.

The translation between physical addresses and virtual addresses is managed in a unit of contiguous memory block called a page, and conducted by dedicated hardware on the CPU chip called Memory Management Unit (MMU). The address mappings are stored in a data structure called page table being placed on memory, and Translation Lookaside Buffer (TLB), which is a part of MMU, caches page table entries. If a necessary page table entry does not exist in TLB, a TLB miss occurs before the CPU or OS walks the page tables to fetch it.

Memory swapping is a technology to realize a larger size of virtual memory than that of physical memory. This is achieved by the OS automatically moving the content between memory and the secondary storage such as HDD and SSD. When the memory is full, the unused content is kicked out from the memory to the secondary storage. Conversely, when the content on the secondary storage is needed again, the hardware invokes an interruption called a page fault, and the content is returned back to the memory. The former operation is called swapping out, and the latter swapping in. In these operations, the update of page table entries is also needed. That is because when the content is on the secondary storage, the corresponding virtual address exists but the physical address does not. As memory swapping is an expensive operation, you may want to avoid swapping out the certain memory region with frequent usage. This can be done by invoking a system call *mlock()* and pinning the content to the memory.

2.2 InfiniBand

Here, we introduce basic concepts of InfiniBand, and then describe its retransmission mechanism, which is key to the implementation of ODP.

InfiniBand is a popular switched interconnection standard widely used in high-performance computing [52]. In InfiniBand clusters, each computing node being equipped with adapter cards[1] are connected via InfiniBand cables and switches as illustrated in Figure 2.1. The network topology in which computing machines are interconnected via several switches is called InfiniBand switched fabric.

InfiniBand is famous for its ultra-low latency of several microseconds and high bandwidth up to 200 Gbps, and is employed in many popular supercomputing systems such as Sierra, Selene, Reedbush, and ABCI [8].

2.2.1 Remote Direct Memory Access (RDMA)

The high performance of InfiniBand comes from remote direct memory access (RDMA) technology. Traditional socket interface which utilizes TCP/IP transport requires multiple buffer copies between

[1] sometimes called Host Channel Adapter (HCA) or RDMA NIC (RNIC)

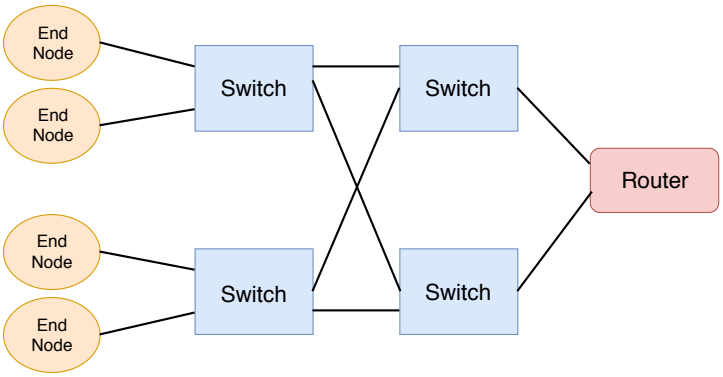


Figure 2.1: InfiniBand network architecture

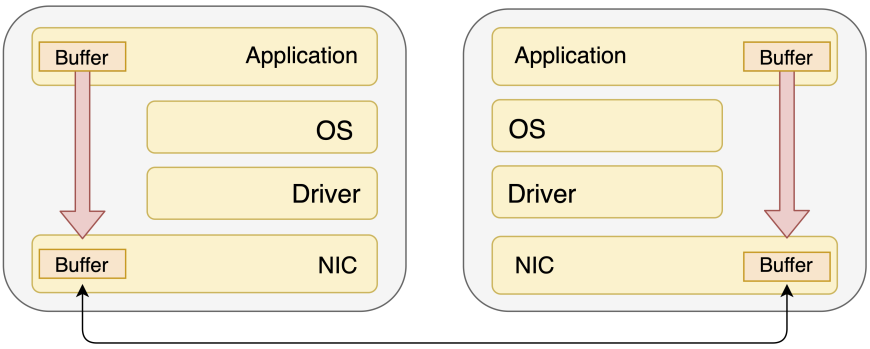


Figure 2.2: Illustration on how InfiniBand bypasses buffer copies

multiple layers such as user space, kernel space, device drivers, and NICs, which consumes CPUs. On the other hand, RDMA enables us to communicate without involving remote CPUs and directly transfer the data between user-level buffers and RNICs (Figure 2.2). These characteristics are called kernel-bypassing and zero-copy transfer, and they are key to the ultra-low latency and high bandwidth.

2.2.2 Memory Registration

Unlike the socket interface, RDMA requires users to register all the memory regions needed for communication beforehand. This process is called memory registration, a crucial process to make the memory region usable for RDMA communication. It is necessary for the following reasons.

1. Because of the kernel-bypassing nature of RDMA, communication buffers should be located in the memory and never swapped out. In other words, we should assure that their physical addresses always exist before issuing communication operations.

2. Since users specify a virtual address when posting communication, RNICs should be able to translate from a virtual address to a physical address. RNICs have virtual-physical address translation tables, and it should have the same mapping as the page table in the memory does.

3. The management of access rights for each memory region is needed.

Memory registration is supported by physical-virtual address translation tables in RNICs, and proceeds with the following three steps:

1. Checking whether the physical address of a target communication buffer exists and if not, a new page is allocated.
2. Pinning down the buffer to the physical memory to avoid swapping it out and make it always accessible from remote CPUs.
3. Updating the translation table in RNICs.

Memory deregistration is the opposite operation and it unpins the buffer and removes the translation entry in RNICs to free resources.

These operations are expensive operations with a latency of tens of microseconds and the bandwidth would increase by 3.3 times without them [16, 38].

2.2.3 InfiniBand Verbs

InfiniBand verbs is provided as an interface for the host operating systems to communicate with RNICs. InfiniBand verbs is categorized into kernel-level verbs and user-level verbs. In this thesis, we only focus on user-level verbs.

QP Model

Queue Pairs (QPs) are the endpoints of communication channels of InfiniBand verbs. A QP consists of two sets of queues: a Send Queue (SQ) and a Receive Queue (RQ). The communication progresses in a non-blocking way in InfninBand, and a QP is associated with another queue called Completion Queue (CQ) as Figure 2.3 illustrates.

When issuing communication, an application posts a Work Request (WR) to the corresponding queue through verbs, and it is managed as Work Queue Element (WQE). When the WR is processed by HCA and it completes, Completion Queue Entry (CQE) appears in CQ which is associated with the QP to notify the application. Even if the operation fails, CQE appears in the CQ, and in this case, CQE contains the corresponding error code.

Communication Operations

There are mainly two kinds of communication operations in InfiniBand verbs, two-sided operations, and one-sided operations.

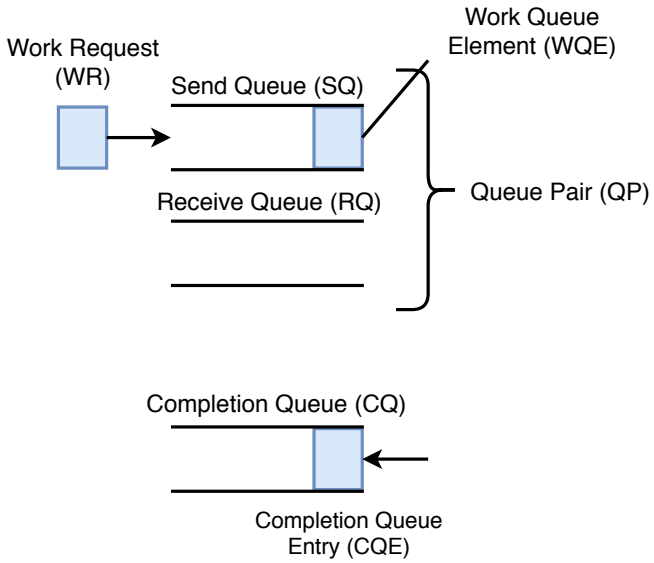


Figure 2.3: QP model introduced in InfiniBand verbs

Two-sided operations include SEND and RECEIVE operations where both sides of processes should be involved in the communication. To send data from a sender to a receiver, the sender posts a SEND WR to its SQ, and the receiver posts a RECEIVE WR to its RQ. Each WR designates the local (virtual) memory address to send/receive the data respectively. It is noted that the receiver should post it before data is received, or it would return RNR NAK to the sender (the details are described in Section 2.2.5).

One-sided operations include RDMA READ and WRITE to the remote memory, where the remote side is not aware of the communication. To read data from the remote process, the requester posts a READ WR to SQ. This WR designates the remote memory address to get data and the local memory address to store received data. WRITE operations work similarly, but sometimes you want to notify the remote process of the completion of write. To meet the demand, RDMA WRITE with Immediate operation is provided as a completely different operation from RDMA WRITE. Other than READ, WRITE, and WRITE with Immediate operations, InfiniBand verbs provides atomic operations such as Fetch-and-Add and Compare-and-Swap operation as one-sided operations.

Regardless of the operations you use, all the memory buffers for communication should be registered beforehand.

2.2.4 Transport Types

InfiniBand fabric offers four types of transport modes: Reliable Connection (RC), Reliable Datagram (RD), Unreliable Connection (UC), and Unreliable Datagram (UD), among which RC and UD are commonly used. Table 2.1 represents the detailed comparison of four types of transport. Reliable transport assures the transmission of packets and conducts retransmission if an error occurs. While connection-

Table 2.1: Transport Types of InfiniBand

	RC	RD	UD	UC
Scalability (# of QPs required) [1]	N^2 QPs	N QPs	N QPs	N^2 QPs
Reliability	Yes	Yes	No	No
RDMA and Atomic Operations Support	Yes	Yes	No	WRITE only
Connection Oriented	Yes	No	No	Yes

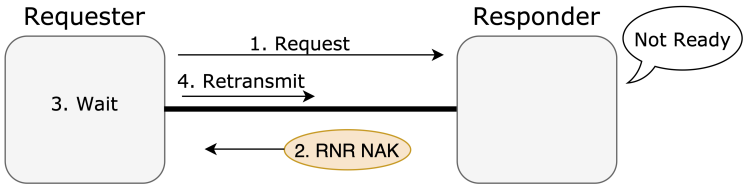


Figure 2.4: The first situation in which retransmission occurs.

oriented transport requires the establishment of the connection between two end-nodes, datagram transport enables one end-node to communicate with the other end-node without a one-to-one connection. Therefore, the number of QPs required in each end-nodes differs between them. While connection-oriented transport needs $O(N^2)$ QPs where N represents the number of end-nodes, datagram transport only requires $O(N)$ QPs because it does not create connections.

2.2.5 Retransmission Handling in RC

Reliable Service (RC) provides a guarantee that packets are correctly delivered from a one node to the other. One of the mechanisms that support reliability is retransmission. Here, we introduce three kinds of situations in which a request is retransmitted.

The first situation in which retransmission occurs is that the responder is not ready for sending the response right now as shown in Figure 2.4. The most typical example is when the corresponding WQE does not exist in the receive queue in two-sided communication. In this case, the receiver returns a special packet called Receiver-Not-Ready (RNR) Negative-Acknowledgment (NAK) to the sender in order to notify the sender about it. This is a packet that asks the sender to wait for a certain period and to retransmit the request. We can control the period via a 5-bit counter of InfiniBand verbs, *RNR Time*, which designate the *smallest* period for which the sender has to wait. We call the smallest period as *minimal RNR NAK delay*, and the relationship between RNR Time and minimal RNR NAK delay is shown in Table 2.2.

The second situation is that the responder detects sequence errors of packets as shown in Figure 2.5. All the packets of InfiniBand contain Packet Sequence Number (PSN), 24 bits field to identify the posi-

[1] N represents the number of end-nodes

Table 2.2: Encoding for minimal RNR NAK delay

RNR Time	minimal RNR NAK delay [ms]	RNR Time	minimal RNR NAK delay [ms]
0	655.36	16	2.56
1	0.01	17	3.84
2	0.02	18	5.12
3	0.03	19	7.68
4	0.04	20	10.24
5	0.06	21	15.36
6	0.08	22	20.48
7	0.12	23	30.72
8	0.16	24	40.96
9	0.24	25	61.44
10	0.32	26	81.92
11	0.48	27	122.88
12	0.64	28	163.84
13	0.96	29	245.76
14	1.28	30	327.68
15	1.96	31	491.52

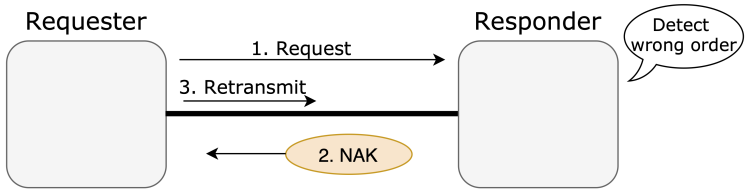


Figure 2.5: The second situation in which retransmission occurs.

tion within a sequence of packets. PSNs are used to detect missing or out-of-order packets. When the responder receives a packet that is out of PSN sequence, PSN Sequence Error occurs and a NAK packet is sent back to the requester. This NAK packet contains the responder's expected PSN, and once the requester received a NAK packet, it retransmits all the packets whose PSNs are no less than the responder's expected PSN.

The third situation is that the corresponding response does not arrived for a certain period as shown in Figure 2.6. This is called *timeout*, and in this case, 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`

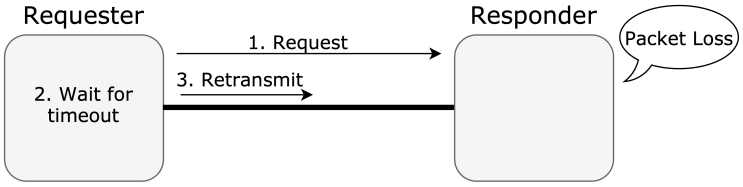


Figure 2.6: The third situation in which retransmission occurs.

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 [20] 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 [20] 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.

2.3 On-Demand Paging (ODP)

In this section, we introduce On-Demand Paging (ODP) [37], which is the main target functionality of our research. ODP is an extension of InfiniBand verbs to allow us to be free from memory registration. It enables us to page-in/-out memory regions automatically depending on the situation.

ODP is implemented with the help of device drivers and RNICs. When remote memory access is requested under ODP, RNICs raise network page faults[2] [30] and device drivers handle them. The previous research reported that the overhead of a page fault is about several hundred microseconds [30].

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.

ODP is currently supported by ConnectX-4 Mellanox InfiniBand adapter or higher generations, and is employed in production-grade communication software such as MVAPICH2-X [6], UCX [50], and libfabric [17].

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

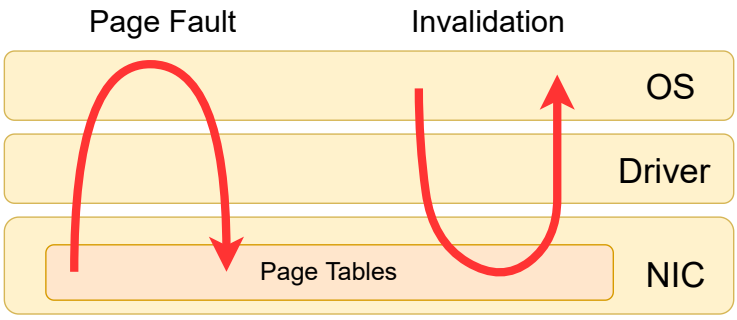


Figure 2.7: Page fault and invalidation flows of ODP

2.3.1 Basic Mechanism

We briefly review the basic mechanism of ODP according to the original paper [30]. We represent page fault and invalidation flows of ODP in Figure 2.7.

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

2.3.2 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.

Chapter 3

Related Work

This chapter reviews related work of our research. Section 3.1 introduces effective memory registration methods. They are software-based methods and had been popularly researched before the emergence of ODP. ODP is superior to them in that it provides a more simplified programming model and consumes less memory. Section 3.2 presents previous work about ODP itself including the performance analysis. While it focused the cost of page faults of ODP and concluded that ODP is performant, we revealed that there exist pitfalls of ODP in relation to retransmission mechanism. Section 3.3 introduces research on the reliability of InfiniBand. We present them here because the pitfalls we found are heavily related to the mechanism to assure reliability, especially the retransmission. Section 3.4 presents some work about the difference of performance property in various Mellanox RNICs. It is important to mention them because packet damming, one of the pitfalls we found, occurs only in the specific hardware. In Section 3.5 and Section 3.6, we also introduce existing work about RDMA in a broader scope, including communication libraries over various interconnects and RDMA-based systems

3.1 Memory Registration Methods

Many efforts to eliminate costly memory registration of RDMA by reusing pinned-down buffers were made. Tezuka et al. [53] proposed the basic idea of pin-down cache for the first time, which reused registered buffers by delaying deregistration. The registered buffers worked as a cache; when the total size of registered buffers exceeded the maximum size, the deregistration of a buffer occurred in the least recently used (LRU) order. Zhou et al. [60] also reused pinned buffers for dynamic registered regions and introduced batched deregistration to reduce the average cost of deregistering memory. Ou at al. [44, 45] proposed Memory Registration Region Cache and a replacement algorithm based on region size and recency. They also proposed a communication scheme between an RDMA client and server that overlapped memory registrations. Wu et al. [57] proposed a two-level architecture called Fast Memory Registration and Deregistration scheme, which adopted batched deregistration. It also made use of the

Mellanox fast memory region registration extension to the InfiniBand verbs API.

Even in the presence of pin-down cache, several performance tradeoffs are known to exist. The Unifier caching system [56] addressed a tradeoff between (de)registration cost and spatial cost. Larger pinned down buffers suppress on-demand pinning in dynamic registration, whereas they more occupy physical memory, which other computations should have used. Frey and Alonso [16] 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 order of magnitude.

Some studies dealt with memory registration methods aimed at specific situations. Firehose algorithm [10] was designed for global address space over distributed memory, which used reference counting of global address regions for (de)registration. Wu et al. [58] designed a new scheme, Optimistic Group Registration for RDMA Gather/Scatter for noncontiguous data transmission. Li et al. [31] implemented dynamic memory management efficiently on multicore platforms by employing a dedicated helper thread to which all the dynamic memory (de)registration requests are offloaded.

The breakdown of the memory registration process itself is known to be important for efficient memory management. Mietke et al. [38] analyzed the registration process inside the Mellanox InfiniBand driver and provided clues to improving it. For example, it drastically reduced the cost of `mlock()` by making a separate kernel thread zero-fill pages when they were not present.

3.2 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. [30] proposed the network page fault support for InfiniBand for the first time, and experimentally analyzed the overhead of page faults and invalidation. Their breakdowns showed that the overhead of page fault was dominated by the hardware including triggering the interrupt and resuming the transmission while the invalidation spent most of its time on the update of page tables. Li at al. [32] presented a thorough analysis of the performance 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 on Implicit ODP [33], they revealed that the overhead of page faults was able to mitigate through an elaborate tuning of 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.

3.3 Reliability of InfiniBand

The reliability of InfiniBand has been analyzed in some literature. Koop et al. [28] evaluated and compared the cost of hardware-based implementation and software-based implementation. They implemented MPI over the UC transport and experimentally compared it with MPI over the RC and UD transports. They showed a software-based approach was not only feasible but might provide higher performance because of its smaller memory consumption. They also developed MVAPICH-Aptus [27], which implemented MPI over multiple types of transport and was able to select transport protocols and message channels dynamically. Jose at al. [21] made a similar attempt to address the memory consumption and scalability issue of RC transport and implemented memcached with a hybrid use of RC and UD for improving performance.

Koop et al. 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 [29]. That was because InfiniBand employs lossless link-level flow control and therefore packets are never lost due to buffer overflows in switches, a principle cause of transport-layer retransmissions [35]. Some systems even chose the design sacrificing the reliability for fast common case performance at the cost of rare retries. HERD [23] was a key-value store designed to sacrifice transport-level retransmission for the performance with the general usage. FaSST [25] was a distributed in-memory transaction system built on remote procedure calls over the UD transport of InfiniBand. It was designed to detect packet loss with coarse-grained timeouts of RPCs.

3.4 Performance Difference between Mellanox RNICs Generations

The performance characteristics of each communication operation might differ depending on what generation of RNICs you use. As one of the benefits of RDMA is kernel bypassing, it is natural to consider that one-sided operations perform better than two-sided primitives. However, there is some work to show that one-sided primitives perform poorly in old generation RNICs such as Mellanox ConnectX-3 [23,25].

Novakovic et al. analyzed the cause of low performance of one-sided operations and divided them into three categories [43]. First, one-sided operations require reliable connections, which consumes the memory in NICs. Since NICs have only a limited size of memory, one-sided operations usually do not scale well. Second, they require virtual-physical address translation and memory protection metadata such as rkey and lkey, which is also stored in the RNICs. Finally, dynamic data structures designed for RDMA sometimes needs more number of invocations of communication in chasing remote pointers. They revisited the above problems of one-sided operations, and pointed out that the problems can be solved with newer hardware. Specifically, the first problem can be solved by increasing the size of memory as Mellanox ConnectX-4 and ConnectX-5 NICs does. Further, to tackle the second issue, they

also focus on the physical segment, a new feature available in newer RNICs to reduce the memory translation table metadata.

Similarly, Wei et al. [54] measured the performance of one-sided and two-sided operations with multiple generations of RNICs. They concluded similarly that using one-sided operations is a better design choice for ConnectX-4 and ConnectX-5 while two-sided operations are appropriate for the system based on ConnectX-3.

3.5 Communication Libraries over Various Interconnects

Usually, InfiniBand verbs is not invoked directly by application programmers. Rather, they prefer to issue communication operations through software communication libraries as there exist many types of customized interconnects with different low-level APIs other than InfiniBand. These communication libraries should provide high-level communication abstraction to treat various hardware with several generations in a unified way. Here, we present three major portable communication libraries over various interconnects: MPI, UCX, and libfabric.

3.5.1 Message Passing Interface (MPI)

Message Passing Interface (MPI) is a de-facto standard communication interface used for long years in the HPC community. Exactly speaking, MPI is not a library, but a standard that defines the syntax and semantics of library routines. There are many kinds of implementations such as Intel MPI [4], Open MPI [7], NVAPICH [6], and MPICH [5]. MPI is characterized by a rich set of over 430 routines which include point-to-point, collective, and synchronization operations. Although they are elaborately tuned for avoiding the overhead, however, their poor performance has been debated even recently [48].

3.5.2 Unified Communication X (UCX)

Unified Communication X (UCX) [50] is a promising open-source production-grade communication framework that aims to both portability and performance. It has been actively developed with the collaboration of national laboratories, industry, and academia for high-performance and highly-scalable network stack for next-generation applications and systems. UCX supports many kinds of hardware such as InfiniBand, RoCE, TCP sockets, shared memory, and Cray Gemini. Figure 3.1 shows UCX software stack, which is placed on top of many kinds of hardware. What is notable about the design of UCX is that it is composed of two kinds of APIs, and users can freely choose them depending on the situation. While UCT, the lower-level APIs of UCX, exposes basic network operations, UCP, the high-level APIs of UCX, construct protocols commonly found in applications. UCX is widely integrated into

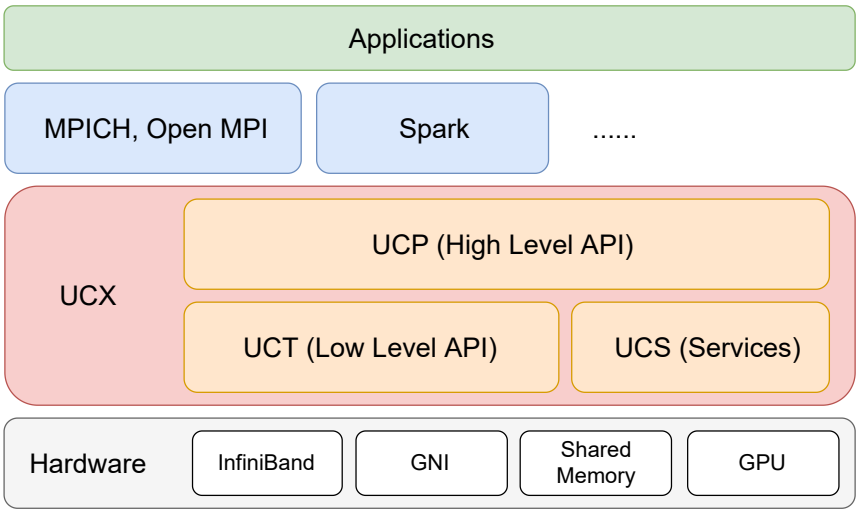


Figure 3.1: UCX architecture

many programming models such as MPI (MPICH, Open MPI), OpenSHMEM PGAS language [9, 12], and Charm++ [22].

3.5.3 Libfabric

As a general high-performance communication library, libfabric [17] is also famous and widely used nowadays. Libfabric is a framework focused on exporting fabric communication services to applications. Currently, it is utilized by a variety of open-source HPC middleware applications such as MPICH, OpenMPI, Open SHMEM, Charm++, GASNet. It also supports many kinds of network hardware including TCP/UDP networks, Omni-Path Architecture, InfiniBand, Cray GNI, Blue Gene Architecture, iWarp, RDMA Ethernet, and RoCE. The main difference between libfabric and UCX mainly comes from their APIs [46, 47]. Libfabric supports a richer set of endpoint options and applicable in the non-HPC, data-center application space whereas UCX's API is much simpler than libfabric. There is also a difference in completion models: CQs and/or poll sets for librabric and callback/progressing working model for UCX.

3.6 RDMA-based Systems

There are many types of distributed systems which utilize RDMA such as transaction system, key-value store, deep learning framework, GAS system, file system, and so forth [13, 14, 23, 25, 34, 39, 54, 55, 59]. In this section, we selectively introduce RDMA-based Key-value Store systems (Section 3.6.1) and GAS systems (Section 3.6.2).

3.6.1 Key-value Store (KVS) Systems

Several key-value store (KVS) systems that utilize RDMA have been developed. In KVS systems, the client issues two basic operations, get(key) and put(key, value), to the server. Get takes a key as an argument and returns the corresponding value which is associated with the key stored in the server. Put takes a key and a value as arguments and updates the value associated with the key. In incorporating RDMA into KVS systems, the main topic which invokes discussion is what kinds of RDMA operations should be used in each KVS operation.

Pilaf [39] is a classic RDMA-based key-value store system. They pinpointed that introducing RDMA to put operations leads to complex and fragile designs for several reasons including requiring expensive RDMA ATOMIC operations and decided to replace only get operations with RDMA. As Pilaf supports varying length of values, the server should have two data structures, fixed-size hash tables and extent areas. The fixed-size hash table is used to obtain a pointer to the value from the key, and the extent areas contain the actual value. In get operations of Pilaf, the client first issues no less than one RDMA READ to the fixed-size hash tables to get the pointer to the value, and then issues one more RDMA READ to the extent areas to get the actual value. They also introduced a self-verifying data structure in order to solve read-write races which arise in the face of concurrent RDMA READs and local writes.

In HERD [23], get operations are conducted by a mix of RDMA WRITE and SEND instead of RDMA READ, and put operations are done only by RDMA WRITE. They gave a detailed analysis of RDMA operations with micro-benchmarks and showed that RDMA READ was inefficient as the round trip was inevitable compared with WRITE. They also showed that the outbound performance of WRITE on RC did not scale well due to the memory limit on RNICs, and insisted that SEND on UD was appropriate for the response. In their later work [24], they further improved the performance of HERD by introducing doorbell batching and multi-queue.

3.6.2 Global Address Space (GAS) Systems

Global address space (GAS) programming model provides an abstraction of shared memory over processes that do not share physical memory. As programming with shared memory does not require the programmer to describe the communication explicitly, GAS models make the programming easy and productive. The typical systems which adopt GAS models are Distributed Shared Memory (DSM) and Partitioned Global Address Space (PGAS) systems. While DSM provides only global address space to the programmers, PGAS requires them to handle both global address space and local address space. Therefore, PGAS systems put more burden on them but is likely to perform better than DSM systems. GAS model hides communication between processes from the programmers, and instead, the systems take the responsibility for implicit communication. The challenge of GAS systems is known to be their performance and scalability, and recently, several GAS systems utilize RDMA to tackle this challenge.

ArgoDSM [26] is a software DSM system that provides a novel directory-based cache coherence protocol over RDMA. Typically, DSM systems employ caches to buffer remote memory accesses to minimize the latency. In the design of shared memory systems, consistency models define correct shared memory behavior in terms of loads and stores [51], which is ensured by cache coherence protocols. ArgoDSM adopts SC for DRF as a consistency model and a novel cache coherence protocol called Carina as a cache coherent protocol. Carina is based on self-invalidation, self-downgrade, and passive data classification directories that avoids the use of message handlers, and all operations are performed by RDMA READ and WRITE.

Grappa [42] is a GAS system designed for data-intensive applications represented in such frameworks as MapReduce, Vertex-centric, and Relational query execution. In designing Grappa, they focused on the fact that the performance of these applications depends not on the latency but on the bandwidth and that they have high parallelism. Grappa chooses to trade off the latency over throughput, and its underlying communication layer aggregates small messages into large ones. The interface of the communication layer was based on active messages, and the implementation utilizes RDMA over MPI. Grappa provides a PGAS-like programming model and does not have a coherence cache. Specifically, the access to data on remote nodes is conducted by delegate operation that runs on the owing node.

GAM [11] is a distributed in-memory platform that provides a cache coherence protocol over RDMA. GAM adopts PGAS addressing model and is equipped with a caching system on top of the global memory space. In GAM, Partial Store Order (PSO) is adopted as a consistency model, which is enforced by a coherence protocol based on RDMA. GAM has two communication channels for separate needs: for control messages and for data transmission. Since the control messages are relatively small and require instantaneous notification, two-sided RDMA SEND/RECEIVE operations are adopted. On the other hand, for the data transmission with a larger transmission volume, RDMA WRITE operations are utilized.

Menps [15] is a software DSM system that adopts a novel cache coherency protocol called floating home-based protocol and new invalidation methods called write notice invalidation and lease-based invalidation. Menps is implemented on top of UCX, and all the communication operations are conducted with RDMA. The first author's dissertation about Menps [15] reported that NAS BT benchmark running on top of Menps incur a long latency of a few seconds, which was caused by the retransmission of InfiniBand. This phenomenon was a start point of our research. After we dug deeper into it, we finally identified that the root cause comes from ODP. In Section 5.1, we describe how this phenomenon, packet damming, occurs with in-depth analysis using micro-benchmarks and packet capturing.

Chapter 4

On-Demand Paging in Reality

In this chapter, we experimentally analyze an actual implementation of ODPs. Table 4.1 summarizes the detailed information of the InfiniBand RNICs used in our experiments. Table 4.2 summarizes experimental environments.

4.1 Ibdump

Ibdump[1] is a packet capturing tool for InfiniBand traffic. It has the similar functionality of tcpdump on an Ethernet network, and it dumps collected packets into .pcap format. The format can be loaded and analyzed using Wireshark with rich GUIs as shown in Figure 4.1. Note that ibdump can be used only with sudo authority, which prevent us from using on major cluster computing systems. We also confirmed that it can not be used on Azure servers because the access right to the hardware is not granted to users.

4.2 Actual Behaviors of ODP

The original paper [30] on ODP clearly described why RNR NAK and reliability enable 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 operation through ibdump. 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-4 (private servers B) for this experiment, setting the minimal RNR NAK delay to be 1.28 ms.

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

Table 4.1: 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 [2]	MT_2160110021	ConnectX-4 56Gbps FDR	4.5-0.1.0	12.24.1000
Reedbush-L [2]	MT_2180110032	ConnectX-4 100Gbps EDR	4.5-0.1.0	12.24.1000
ABCI [1]	MT_0000000095	ConnectX-4 100Gbps EDR	4.4-1.0.0	12.21.1000
ITO [3]	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
Private servers C	MT_0000000224	ConnectX-6 100Gbps EDR	5.0-2.1.8.0	20.27.2008
Azure VM HBv2 Series	MT_0000000223	ConnectX-6 200Gbps HDR	5.0-2.1.8.0	20.26.6200

Table 4.2: Experimental environments

System name	CPU	# of logical cores	Memory
KNL-4 (Private servers B)	Xeon Phi CPU 7250 @ 1.40GHz	272	196 GB + MCDRAM 16 GB
KNL-6 (Private servers C)	Xeon Phi CPU 7250 @ 1.40GHz	272	196 GB + MCDRAM 16 GB
Reedbush-H	Xeon CPU E5-2695 v4 @ 2.10GHz	36 ($= 18 \times 2$)	256 GB
ABCI	Xeon Gold 6148 CPU @ 2.40GHz	80 ($= 20 \times 4$)	384 GB

Figure 4.2 illustrates the workflow of a single READ operation based on packets captured via ibdump and page fault counters.

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 4.0 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.

For reference, behaviors of SEND and WRITE operations with enabling ODP are presented in Figure 4.3 and Figure 4.4.

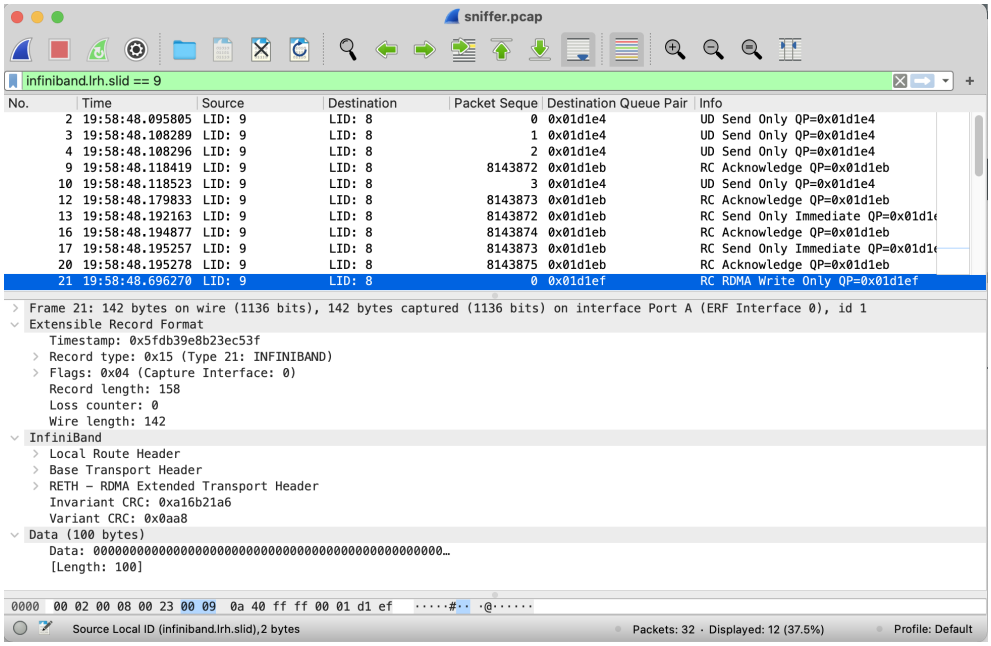


Figure 4.1: Screenshot of Wireshark with rich GUIs

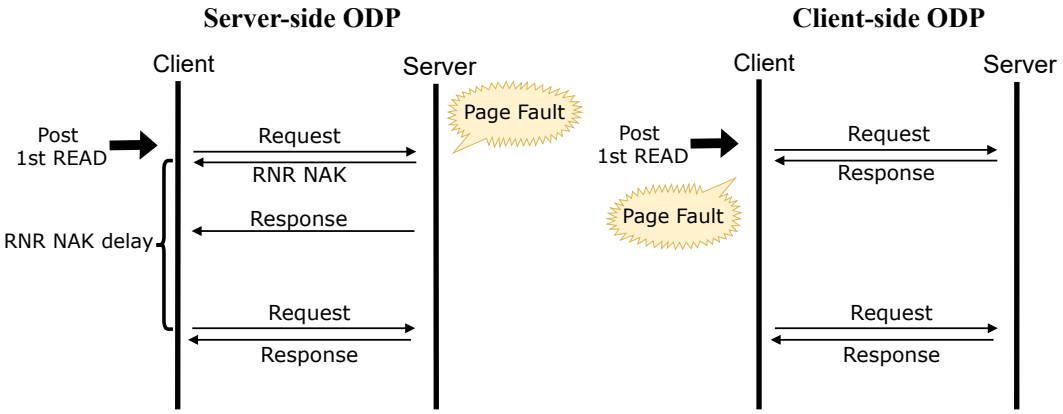
4.3 Timeout for the Worst Case

The observations above 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 [19, 41] 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 4.1. 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 4.5 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.

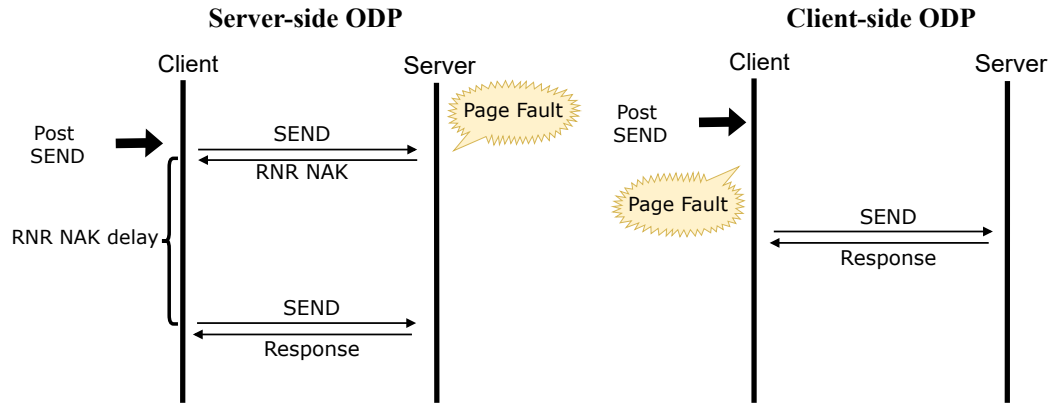
We have two important observations from the experimental results. One is, considering that the usual round trip latency of InfiniBand is about several μs, the timeout is tremendously and incomparably longer. The timeout is a disaster even if rare. The implementation of ODP should never incur it. The other is that a long timeout is ubiquitous. Even real-world computing clusters [1–3] and even the latest RNIC, the ConnectX-6 200Gbps HDR, can suffer from it.




```
Time [s]
0.000000     RDMA Read Request
0.000141     Acknowledge (RNR NAK)
0.000538     RDMA Read Response
0.004565     RDMA Read Request
0.004639     RDMA Read Response
```

```
Time [s]
0.000000     RDMA Read Request
0.000005     RDMA Read Response
0.000527     RDMA Read Request
0.000620     RDMA Read Response
```

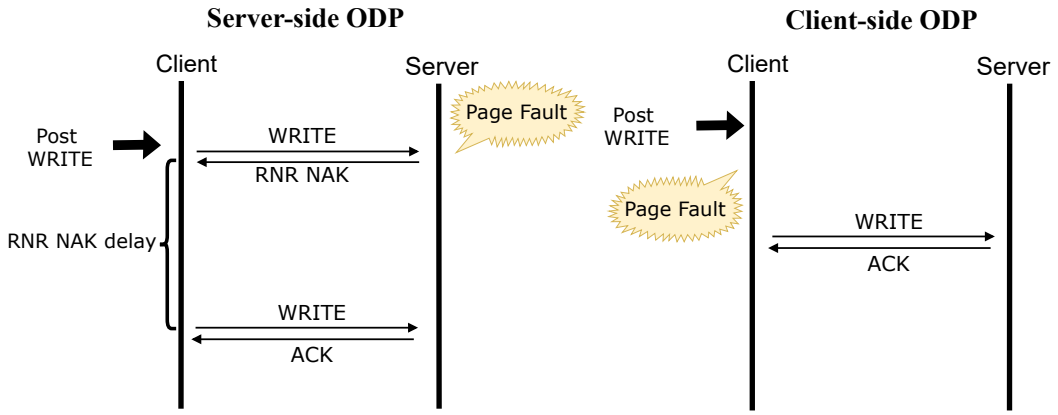
Figure 4.2: Workflow of ODP with a single READ operation observed with idbump.




```
Time [s]
0.000000     Send
0.000513     Acknowledge (RNR NAK)
0.003905     Send
0.003906     Acknowledge
```

```
Time [s]
0.000000     Send
0.000003     Acknowledge
```

Figure 4.3: Workflow of ODP with a single SEND operation observed with idbump.




```
Time [s]
0.000000    Write
0.000526    Acknowledge (RNR NAK)
0.006035    Write
0.006041    Acknowledge
```

```
Time [s]
0.000000    Write
0.000003    Acknowledge
```

Figure 4.4: Workflow of ODP with a single WRITE operation observed with idbump.

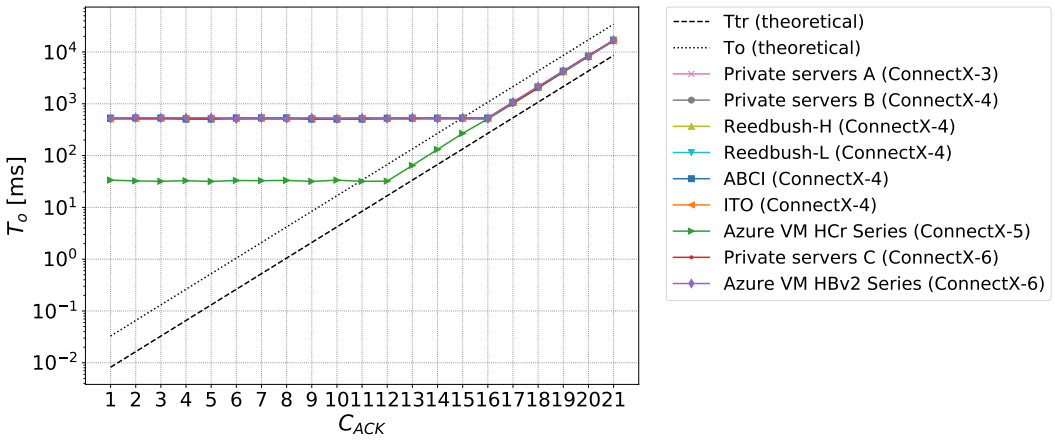


Figure 4.5: T_o measured by varying C_{ACK} on different InfiniBand systems, where systems other than Azure VM HCr Series (green) lie on almost the same line.

Chapter 5

Performance Pitfalls of On-Demand Paging

5.1 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, which 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-4 (private servers B) described in Table 4.1 and Table 4.2, where the minimal timeout which can be configured is approximately 500 ms. We used RC as the transport type with $C_{\mathrm{ACK}} = 1$ (minimal) and $C_{\mathrm{Retry}} = 7$.

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 5.1. We introduced a sleep function to control the communication intervals. At the end of the benchmark, a wait block was inserted so that the completion of all the communications could be confirmed. The communication buffer was aligned with 4096-bytes 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.

5.1.1 Pitfalls with Two READ Operations

First, we observed the execution time of the micro-benchmark with only two READ operations in Figure 5.2. The execution took several hundred *milliseconds* with an interval of 100–4500 μs. This is surprising, given that the overhead of ODP is mainly due to the page fault, which usually takes only

```
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();
```

Figure 5.1: 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.

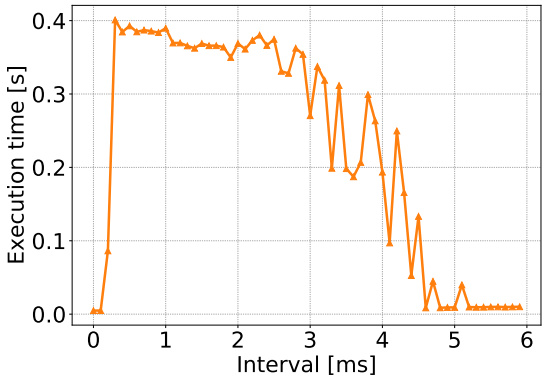
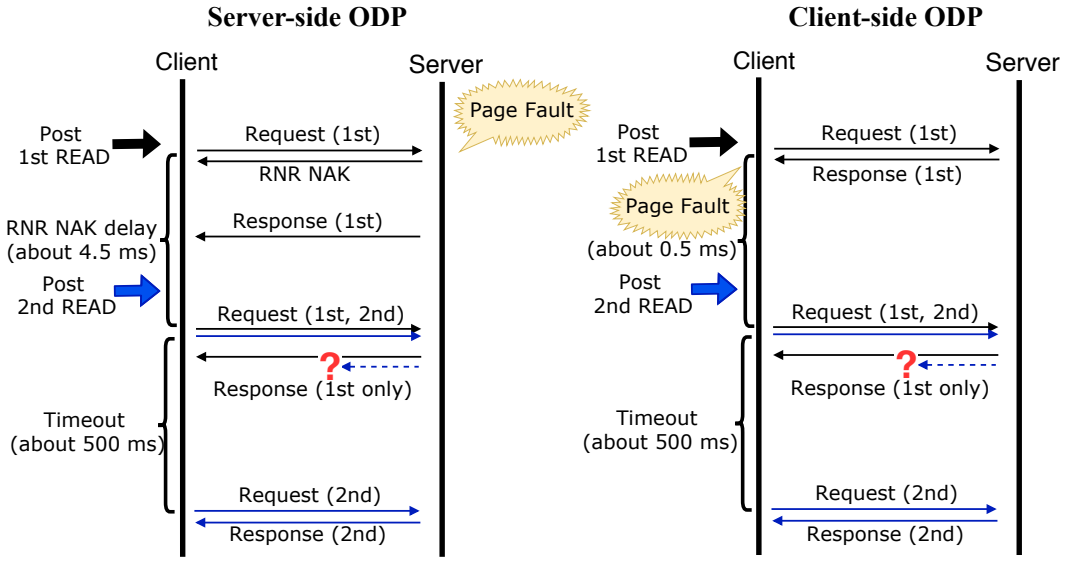


Figure 5.2: Average execution time of the micro-benchmark out of 10 trials with varying intervals between two communications. Two READ operations are issued, and both-side ODP is adopted. Minimal RNR NAK delay is set to 1.28 ms.

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 5.3 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 5.4a shows that in the case of minimal RNR NAK




```
Time [s]
0.000000     RDMA Read Request (1st)
0.000140     Acknowledge (RNR NACK)
0.000451     RDMA Read Response (1st)
0.003968     RDMA Read Request (1st)
0.003970     RDMA Read Request (2nd)
0.004051     RDMA Read Response (1st)
0.541748     RDMA Read Request (2nd)
0.541755     RDMA Read Response (2nd)
```

```
Time [s]
0.000000     RDMA Read Request (1st)
0.000005     RDMA Read Response (1st)
0.000778     RDMA Read Request (1st)
0.000782     RDMA Read Request (2nd)
0.000861     RDMA Read Response (1st)
0.435909     RDMA Read Request (2nd)
0.435912     RDMA Read Response (2nd)
```

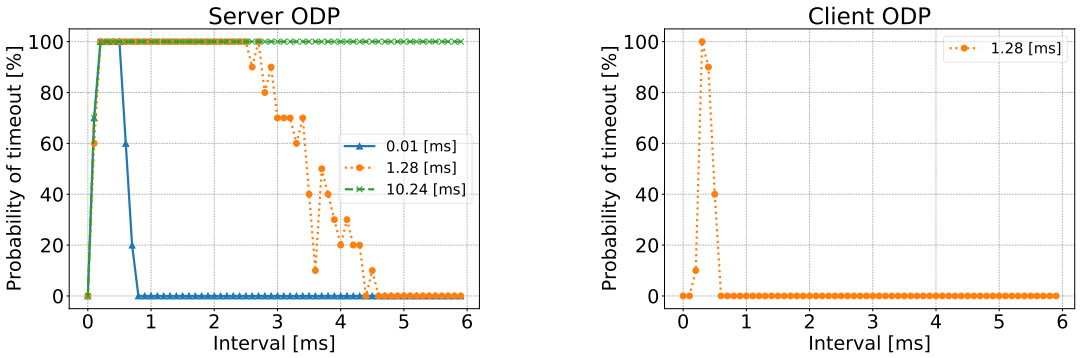
Figure 5.3: Workflow of ODP with two READ operations based on packets obtained via ibdump.

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 4.2. In addition, if the pending period was changed, the range of intervals followed it. Figure 5.4b 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 4.2. In summary, packet damming occurred when the second request was posted during the first request's pending period when preparing for retransmission.

5.1.2 Performance with More Than Two READ Operations

Figure 5.5 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 ocurred. 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 5.6 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



(a) Server-side ODP

(b) Client-side ODP

Figure 5.4: Probability of occurrence of timeout out of 10 trials with varying intervals of two READ operations in the server-side ODP and client-side ODP. The value of the legend represents minimal RNR NAK delay.

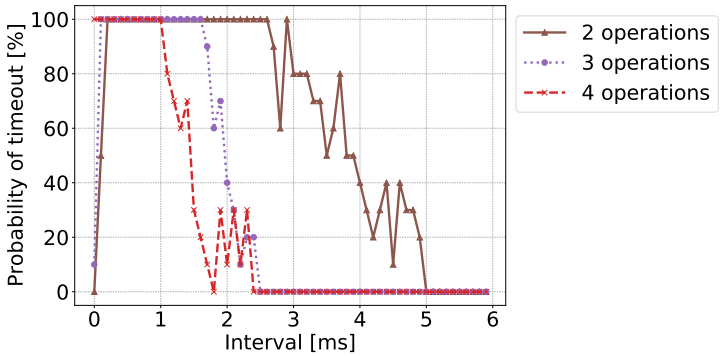
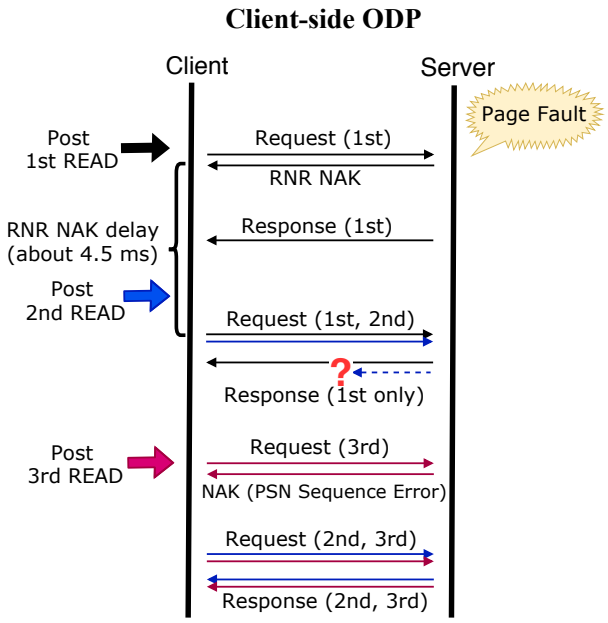


Figure 5.5: Probability of timeout out of 10 trials with varying intervals of each READ operations in the both-side ODP. The number of READ operations is changed between 2 and 4. Minimal RNR NAK is set to 1.28 ms.

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.

5.1.3 Experiments with Other Conditions

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




```
Time [s]
0.000000    RDMA Read Request (1st)
0.000143    Acknowledge (RNR NAK)
0.000464    RDMA Read Response (1st)
0.003913    RDMA Read Request (1st)
0.003915    RDMA Read Request (2nd)
0.004005    RDMA Read Response (1st)
0.006414    RDMA Read Request (3rd)
0.006415    Acknowledge (NAK)
0.006615    RDMA Read Request (2nd)
0.006616    RDMA Read Request (3rd)
0.006617    RDMA Read Response (2nd)
0.006618    RDMA Read Response (3rd)
```

Figure 5.6: Workflow of ODP with three READ operations based on the packet captured with ibdump.

- It occurred independent of other QPs. The QP still continued to wait 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 4.1. Nevertheless, we have not observed it with ConnectX-6 servers with the micro-benchmark so far.

5.2 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 5.1 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 $C_{\mathrm{ACK}} = 18$.

5.2.1 Impact of Packet Flood

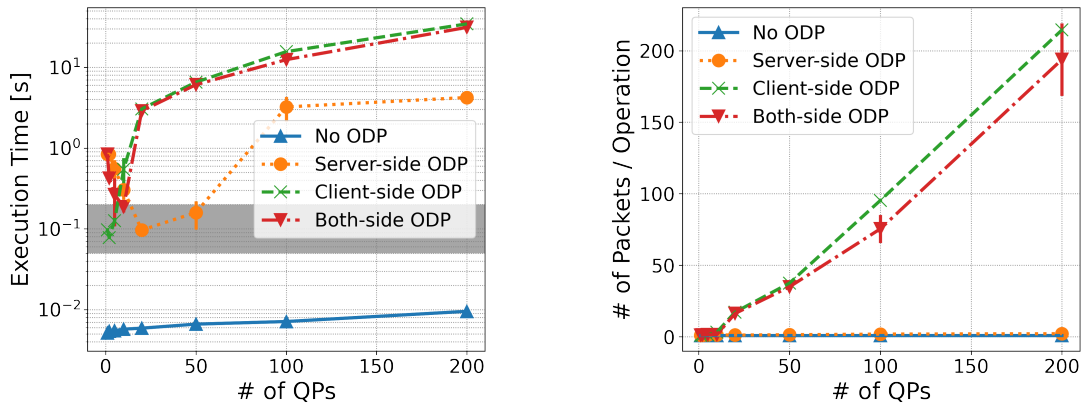
Figure 5.7a shows how the number of QPs had an impact on the performance when using the same number of READ operations. When a small number of QPs were used, the ODP performance was essentially normal and acceptable, with the execution time almost fitting into the range of common overhead of page faults. When the number of QPs exceeded 10, however, the ODP performance degraded drastically and became as much as 3000 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 5.7b. 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 (see Section A).

5.2.2 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 5.8.

First, we focus on the result with 128 operations. 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 communica-



(a) Execution Time (s)

(b) Number of Packets

Figure 5.7: Effect of varying the number of QPs with the micro-benchmark in Figure 5.1 with READ operations. 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.

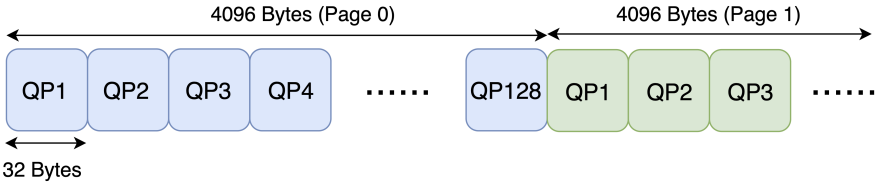
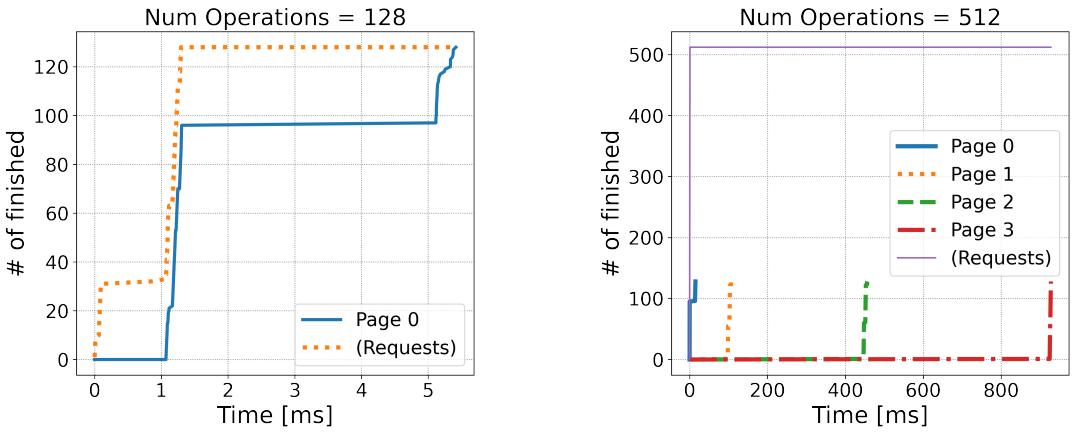


Figure 5.8: Memory layout for communication buffer and assignment of QPs

tion 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 the result of 512 operations.

In summary, packet flood occurs when muitlple READ operations from multiple QPs cause simultaneous page faults followed by update failure of page statuses.



(a) 128 Operations

(b) 512 Operations

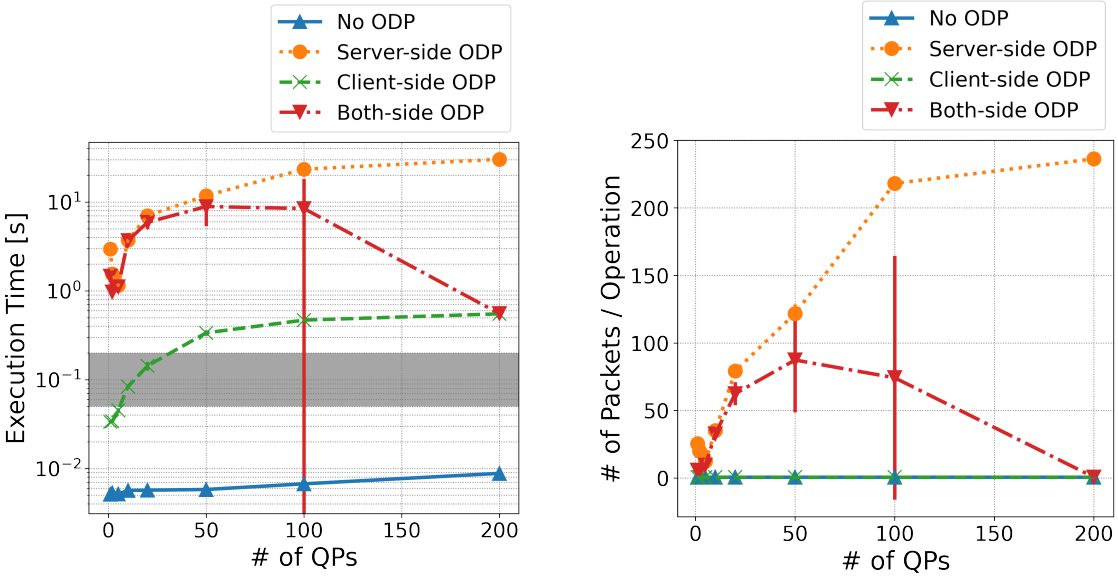
Figure 5.9: Number of completed operations per page. 128 QPs and a message size of 32 bytes is utilized and client-side ODP is adopted.

5.2.3 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 5.7b. 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 occurrs 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.

Given this, a few aspects remain to be mentioned regarding Figure 5.7a. 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 5.1. 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.



(a) 128 Operations

(b) 512 Operations

Figure 5.10: Effect of varying the number of QPs with the micro-benchmark in Figure 5.1 and WRITE operations. The parameters are the same as Figure 5.7 shows.

5.2.4 Analysis of WRITE Operations

We have discussed only READ operations so far, and from here, we investigate whether WRITE operations cause packet flood. We conducted the same experiments as shown in Section 5.2.1 using WRITE operations instead of READ operations. Figure 5.10 shows the performance results with WRITE operations. It indicates that the client-side ODP degraded performance with increasing the number of QPs while server-side ODP did not. As the performance degradation accompanies the increase of WRITE packets, the same phenomenon as in the case of READ operations is expected to be happening with WRITE operations.

However, it appears to be a completely different phenomenon for two reasons. First, the server-side ODP with WRITE operations performed poorly accompanied by larger packets by several tens times even when one QP was used. This was not observed in the packet flood of the client-side ODP with READ operations, where the increase of packets did not happen with only one QP. Second, this experimental result might be inconsistent with the analysis we presented in Section 5.2.3. According to our analysis, even when WRITE operations are issued, only the client-side ODP should degrade performance since the discussion about the statefulness could apply to both READ and WRITE. For the reasons, we distingish this phenomenon with WRITE operations from packet flood with READ operations.

To investigate what was happening with the client-side ODP with WRITE operations, we observed the packets with only one QPs. Then, we found that all the WRITE packets were issued immediately

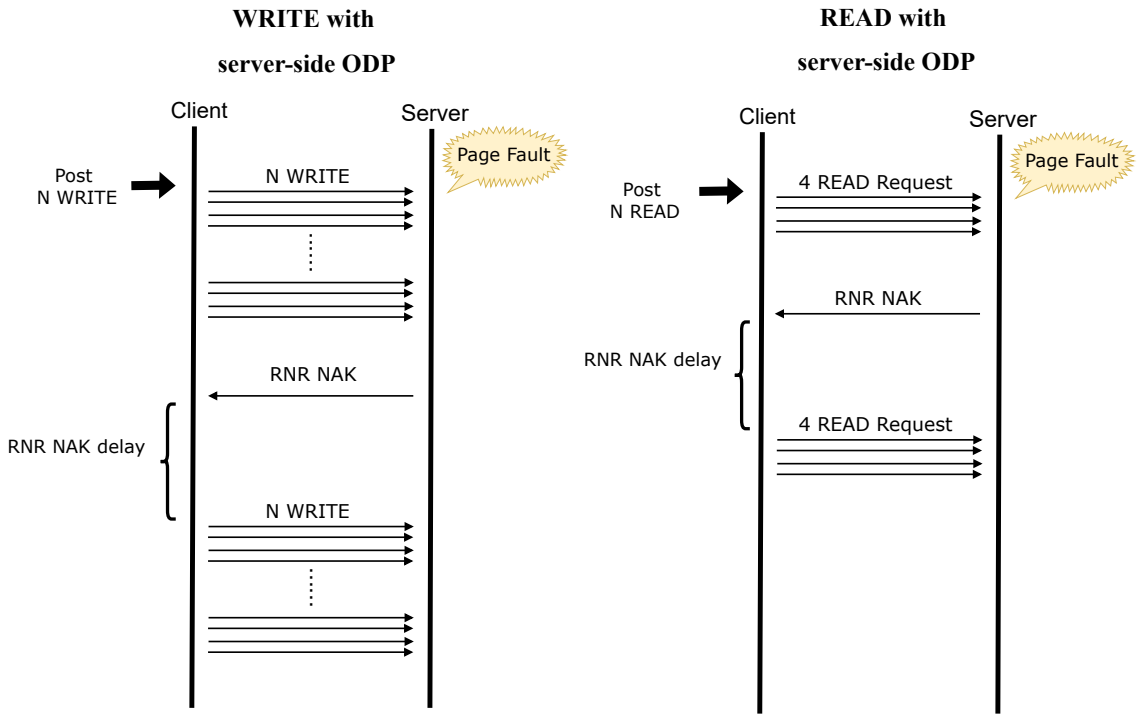
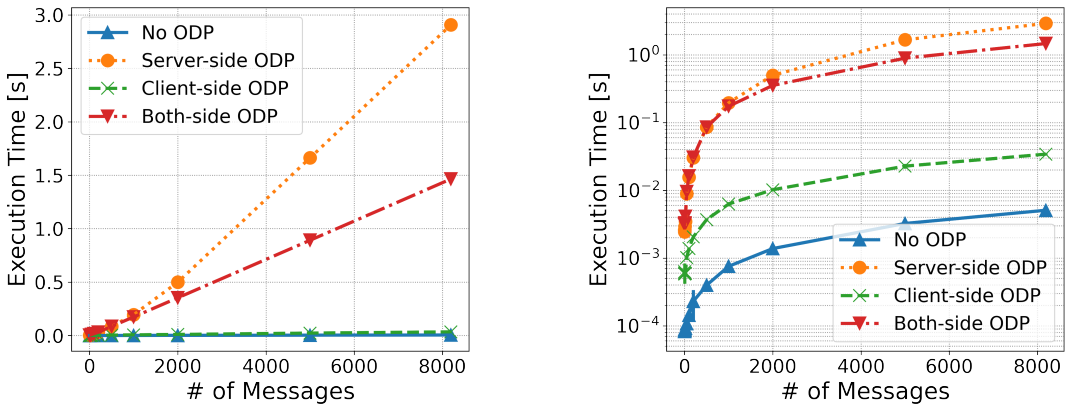


Figure 5.11: A difference between WRITE and READ when N operations are posted into one QP at the same time. In the case of WRITE operations, all the packets are retransmitted after the client received RNR NAK packets. Meanwhile, in the case of READ operations, there are not many wasted packets (READ requests) owing to the limitation of the maximum number of concurrent operations.

when they were posted through InfiniBand verbs. This does not happen with READ operations usually because InfiniBand verbs has a parameter called max_rd_atomic, which controls the maximum number of outstanding READ or ATOMIC operations per QP. In our experiments in this thesis, we chose to set it to be 4 as we should set the value under 16 for the limitation of the hardware. With this setting, even when you post more than four READ operations through InfiniBand verbs, it is assured that only four READ requests are issued immediately. Therefore, READ and WRITE operations behave differently as illustrated in Figure 5.11.

From here, we describe why the limitation of the maximum number of outstanding communication is critical for the performance. In the case of WRITE operations in Figure 5.11, after the client receives all the WRITE packets, an RNR NAK packet for the first WRITE is sent back from the server to the client. Then, the client retransmits all the WRITE packets after some period designated by RNR NAK Time. The important point is that $O(NM)$ WRITE packets are issued eventually, where N represents the number of WRITE operations posted through InfiniBand verbs and M represents the number of pages, that corresponds to the number of RNR NAK packets. Meanwhile, READ operations only need $O(M)$ packets to be issued, owing to the limitation of concurrent operations. Figure 5.12 is measurement results that verify this tendency. The execution time of server-side ODP increases *quadratically* as the number



(a) Linear scale

(b) Log scale

Figure 5.12: Execution time varying the number of WRITE operations with the micro-benchmark in Figure 5.1. The size of communication used is 32 bytes.

of operations increases. Since the increase in the number of operations accompanies the increase in the number of pages, this is the behavior we expect. In conclusion, this tremendous number of packets of WRITE operations themselves become the performance bottleneck.

Finally, we give a little comment about the behavior of both-side ODP with WRITE operations. The execution time of both-side ODP represented in Figure 5.10a is strange at first sight because the performance got back to normal when the number of QPs is 200. The key to this phenomenon is the fact that a WRITE packet is issued only after the local page fault is solved. Since it is reasonable for the page fault to be processed serially, WRITE packets whose memory regions belong to different pages should be issued at different times when client-side ODP is applied. Therefore, operations that handle memory regions with different pages tend not to affect each other because their packets are issued with sufficiently large intervals. As one QP handles a smaller number of packets in the same pages as the number of QPs increases, it is reasonable that both-side ODP performs normally with a sufficiently large number of QPs

Chapter 6

Impacts of Pitfalls 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 did not use ODP in the default configuration, but we enabled it using environmental variables of UCX [50] 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 4.2, and only KNL-4 system allowed us to use sudo authority and ibdump.

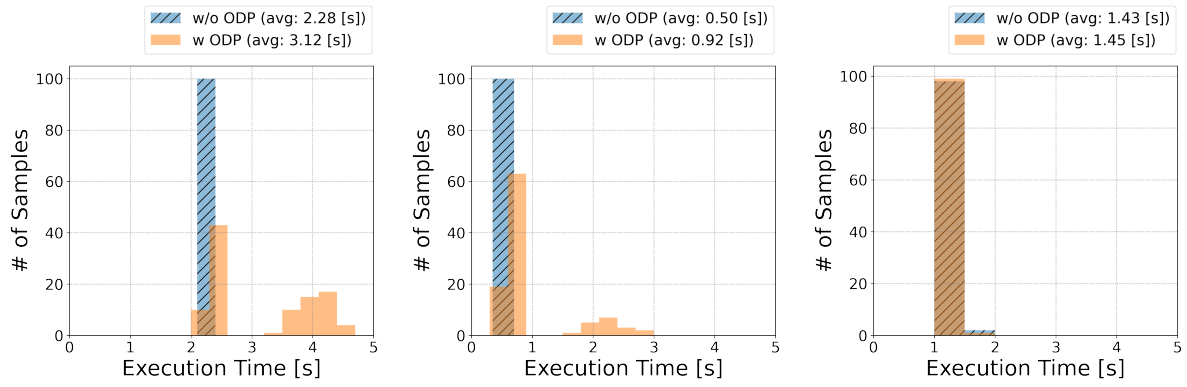
6.1 ArgoDSM

ArgoDSM[1] [26] 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 enebled, we found that packet damming occurs in the initialization phase of ArgoDSM, where first touches and page faults are abundant.

6.1.1 Evaluation of the Initialization and Finalization

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 6.1, which shows that the samples can be divided into two groups by the execution time in each system. We used ibdump to analyze the KNL-4 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. We attach packets which represent the occurrence of packet damming on KNL-4 in

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



(a) KNL-4 (2 nodes) (b) Reedbush-H (2 nodes) (c) ABCI (2 nodes)

Figure 6.1: Execution time distribution when running an ArgoDSM benchmark which contains only argo::init() and argo::finalize() with ODP enabled in each system. 10 MB was passed to argo::init() as the memory size for the initialization.

```
No. Time [s]     Source  Dest    PSN
117 0.070816     LID: 9  LID: 8  26  RC RDMA Read Request
118 0.072510     LID: 8  LID: 9  26  RC Acknowledge
119 0.077290     LID: 9  LID: 8  26  RC RDMA Read Request
120 0.077993     LID: 8  LID: 9  26  RC RDMA Read Response Only
121 0.078019     LID: 9  LID: 8  27  RC Send Only
122 0.078845     LID: 8  LID: 9  26  RC RDMA Read Response Only
123 1.739339     LID: 9  LID: 8  27  RC Send Only
124 1.739344     LID: 8  LID: 9  27  RC Acknowledge
```

Figure 6.2: Fragment of packets while the ArgoDSM benchnark containing only argo::init() and argo::finalize() is running on KNL-4. A long interval can be observed between packet 122 and packet 123.

Figure 6.2. The timeout was caused by one READ operation followed by one SEND operation[2], 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. As for ABCI, we tried the same experiment but no timeout happend because it is highly sensitive to the intervals between operations.

6.1.2 Evaluation of Simple Example in Tutorial

We also evaluated the performance of Simple Example [3] presented in the tutorial of ArgoDSM. We show the pseudocode of Simple Example in Figure 6.3. It calculates the biggest number in an integer array using multiple threads. Figure 6.4 shows the comparison between the case of enabling ODP and the

[2]MPI_Win_unlock invokes this SEND operation internally

[3]https://etascale.github.io/argodsm/tutorial.html

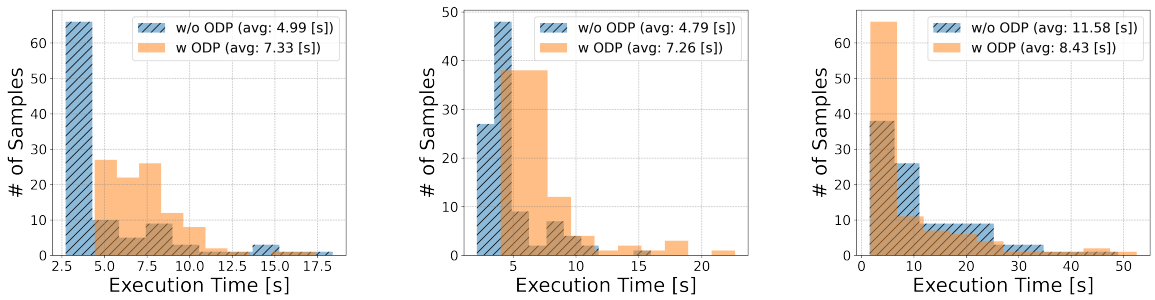
```
int global_max;

void calc_local_max(data) {
    int local_max = max(data);

    global_lock()
    if (local_max > global_max)
        global_max = local_max;
    global_unlock();
}

int main(int argc, char* argv[]) {
    int data_length = 1600000;
    int num_threads = 16;

    argo::init(total_memory=10MB);

    // Allocate array
    auto data = new int[data_length];
    init_array(data);

    // Start threads
    int chunk = data_length / num_threads;
    for (int i = 0; i < num_threads; i++) {
        thread_create(func=calc_local_max, data=data[chunk*i : chunk*(i + 1)]);
    }
    join_all_threads();

    argo::finalize();
}
```

Figure 6.3: Pseudocode of calculating the maximum value from an integer array using multiple threads. We changed the value of data_length from 160,000 to 1,600,000 and the value of total initialized memory from 10GB to 10MB from the original configuration.

case of disabling ODP. According to the result, the case of disabling ODP was more performant than the case of enabling ODP in the case of KNL-4 and Reedbush while the opposite result was got in the case of ABCI. Unfortunately, the difference mainly comes from the degree of contention of the global lock, not from packet damming. That is, the cost of the global lock in ArgoDSM is so heavy that the effect of packet damming did not show up in the overall execution time. Basically, the execution time of the global lock was influenced by the number of threads trying to take the global lock simultaneously. The important point here is that the delay caused by some page faults of ODP somehow affects the execution time of each thread, which makes the timing of updating the global max different. This is the reason why the overall execution time differs depending on the experimental conditions.

6.2 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,



(a) KNL-4 (2 nodes) (b) Reedbush-H (2 nodes) (c) ABCI (2 nodes)

Figure 6.4: Execution time distribution when running the Simple Example of ArgoDSM

memory, and network. SparkUCX[4] [49] 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. SparkUCX creates a large number of QPs because they are created per thread conducting shuffle operations. 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-4, 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 for each environment so as not to abort in the middle of execution. We set the parallelism of executors to be the same as the number of cores that each process occupies. In this case, the amount of calculation of a join operation is $O(N^2)$, where N represents the number of cores.

Table 6.5 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-4 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.

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

SparkTC

	QPs	Disable [s]	Enable [s]	**Enable/Disable**
KNL-4 (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-4 (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-4 (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

Figure 6.5: 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.

Chapter 7

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.

7.1 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 6, 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 [33], 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 5.1.2 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.

7.2 Future Work

Even with these pitfalls, the concept of ODP is still worthy to pursue. As shown in the previous study [30], 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 packet damming to Mellanox. 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 still remains 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 have also reported it to contribute to the improvement of ODP.

Acknowledgments

I would like to thank following people for helping my research during my master's course.

First and foremost, I would first express my deep and sincere gratitude to my supervisor, Professor Kenjiro Taura. He is a man who gave me an opportunity to research in the field of system software and provided me a lot of insightful advice in our meetings. Not only the direct technical advice, but I also learned important things from his attitude toward knowledge, such as how to ask fundamental questions to make uncertain points clear and to push forward the understanding. I am deeply grateful to Professor Shigeyuki Sato for his entire support of the research in my master's course. When I was uncertain about my research direction, he always encouraged me patiently and suggested a hopeful direction. He is a man from whom I learned most about what research is all about, from technical topics to his philosophy. The collaboration with him was an invaluable experience, and I learned specifically how to write a well-written paper with persuasive storytelling. I would like to express my gratitude to the members of NVIDIA, who investigated our problems deeply and gave me beneficial feedback.

I absolutely thank Dr. Wataru Endo, who developed a fabulous DSM library, Menps, which was surely the starting point of my research. I got a lot both from the analysis of his source code and from the fruitful discussion with him. I surely appreciate Shumpei Shiina, with whom I chatted most frequently on a wide range of topics not limited to research and computer science. He was a so excellent student that I was inspired so much by his passion for research and his explanation skills. I should say thank you for his detailed comment on my paper, which definitely made it sophisticated. I thank Qiheng Liu, who kindly told me useful techniques for server management. I am grateful to Yukio Siraichi, who proactively created opportunities for having lunch with members of our lab. I enjoyed it so much and I believe that it certainly helped to bridge the gap between Japanese students and foreign students. I am grateful to our secretary, Sachie Ikeya, who helped me a lot with paperwork with a quick reply to e-mails. I would like to thank other members of Taura Laboratory who helped me through discussion and advice in our meetings.

Finally, I must express my hearty gratitude to my family, who encouraged and supported me every day.

Publications

International Conferences (peer-reviewed)

[1] T. Fukuoka, W. Endo, and K. Taura. An Efficient Inter-Node Communication System with Lightweight-Thread Scheduling. International Conference on High Performance Computing and Communications, HPCC '19, August. 2019.

[2] T. Fukuoka, S. Sato, and K. Taura. Pitfalls of InfiniBand with On-Demand Paging. International Symposium on Performance Analysis of Systems and Software, ISPASS '21, March. 2021. (under submission)

Domestic Conferences

[3] T. Fukuoka, W. Endo, and K. Taura. An Implementation of Inter-node Communication System with Efficient Lightweight-thread Scheduling. The 3rd cross-disciplinary Workshop on Computing Systems, Infrastructures, and Programming, xSIG '19, May. 2019. (**Best Undergraduate Student Award**)

[4] T. Fukuoka, S. Sato, and K. Taura. Analyzing Performance Pitfalls of On-Demand Paging of InfiniBand. 2020年並列／分散／協調処理に関する『福井』サマー・ワークショップ, SWoPP '20, August. 2020. （第150回OS研究会 最優秀若手発表賞）

Domestic Conferences (Poster Presentation)

[5] T. Fukuoka, W. Endo, and K. Taura. An Implementation of Inter-node Communication System with Efficient Lightweight-thread Scheduling.
The 3rd cross-disciplinary Workshop on Computing Systems, Infrastructures, and Programming, xSIG '19, May. 2019.

Bibliography

[1] About ABCI: Computing resources. https://abci.ai/en/about_abci/computing_resource.html.

[2] Introduction to the reedbush supercomputer system. https://www.cc.u-tokyo.ac.jp/en/supercomputer/reedbush/system.php.

[3] Introduction of ITO. https://www.cc.kyushu-u.ac.jp/scp/eng/system/ITO/01_intro.html, 2018.

[4] Intel MPI Library. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/mpi-library.html, 2020.

[5] MPICH. https://www.mpich.org/, 2020.

[6] MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE. https://mvapich.cse.ohio-state.edu/, 2020.

[7] Open MPI: Open Source High Performance Computing. https://www.open-mpi.org/, 2020.

[8] Top500. https://www.top500.org/lists/top500/, 2020.

[9] BAKER, M. B., ADERHOLDT, F., VENKATA, M. G., AND SHAMIS, P. Openshmem-ucx: Evaluation of UCX for implementing openshmem programming model. In *OpenSHMEM and Related Technologies. Enhancing OpenSHMEM for Hybrid Environments - Third Workshop, OpenSHMEM 2016, Revised Selected Papers* (2016), M. G. Venkata, N. Imam, S. Pophale, and T. M. Mintz, Eds., vol. 10007 of *Lecture Notes in Computer Science*, Springer, pp. 114–130.

[10] BELL, C., AND BONACHEA, D. A new DMA registration strategy for pinning-based high performance networks. In *17th International Parallel and Distributed Processing Symposium (IPDPS 2003)* (2003), IPDPS '03, IEEE Computer Society, p. 198.

[11] CAI, Q., GUO, W., ZHANG, H., AGRAWAL, D., CHEN, G., OOI, B. C., TAN, K., TEO, Y. M., AND WANG, S. Efficient distributed memory management with RDMA and caching. *Proc. VLDB Endow. 11*, 11 (2018), 1604–1617.

[12] CHAPMAN, B. M., CURTIS, T., POPHALE, S., POOLE, S. W., KUEHN, J. A., KOELBEL, C., AND SMITH, L. Introducing openshmem: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model, PGAS 2010* (2010), J. E. Moreira, C. Iancu, and V. A. Saraswat, Eds., PGAS '10, ACM, p. 2.

[13] CHEN, Y., WEI, X., SHI, J., CHEN, R., AND CHEN, H. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016* (2016), C. Cadar, P. R. Pietzuch, K. Keeton, and R. Rodrigues, Eds., EuroSys '16, ACM, pp. 26:1–26:17.

[14] DRAGOJEVIC, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014* (2014), R. Mahajan and I. Stoica, Eds., NSDI '14, USENIX Association, pp. 401–414.

[15] ENDO, W. A decentralized implementation of software distributed shared memory. Doctoral dissertation.

[16] FREY, P. W., AND ALONSO, G. Minimizing the hidden cost of RDMA. In *Proceedings of 2009 29th IEEE International Conference on Distributed Computing Systems* (2009), ICDCS '09, IEEE, pp. 553–560.

[17] GRUN, P., HEFTY, S., SUR, S., GOODELL, D., RUSSELL, R. D., PRITCHARD, H., AND SQUYRES, J. M. 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* (2015), HOTI '15, IEEE, pp. 34–39.

[18] GUO, C., WU, H., DENG, Z., SONI, G., YE, J., PADHYE, J., AND LIPSHTEYN, M. RDMA over commodity ethernet at scale. In *Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication* (2016), SIGCOMM '16, ACM, pp. 202–215.

[19] HUDZIA, B. 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.

[20] INFINIBAND TRADE ASSOCIATION. *InfiniBand Architecture Specification Volume 1*, release 1.4 ed., 2020. https://cw.infinibandta.org/document/dl/8567.

[21] JOSE, J., SUBRAMONI, H., KANDALLA, K. C., WASI-UR-RAHMAN, M., WANG, H., NARRAVULA, S., AND PANDA, D. K. Scalable memcached design for infiniband clusters using hybrid transports. In *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2012* (2012), CCGrid '12, IEEE Computer Society, pp. 236–243.

[22] KALÉ, L. V., AND KRISHNAN, S. CHARM++: A portable concurrent object oriented system based on C++. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Eighth Annual Conference, 1993, Proceedings* (1993), T. Babitsky and J. Salmons, Eds., OOPSLA '93, ACM, pp. 91–108.

[23] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using RDMA efficiently for key-value services. In *Proceedings of the 2014 ACM conference on SIGCOMM* (2014), SIGCOMM '14, ACM, pp. 295–306.

[24] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Design guidelines for high performance RDMA systems. In *Proceedings of the 2016 USENIX Annual Technical Conference* (2016), USENIX ATC '16, USENIX, pp. 437–450.

[25] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. 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* (2016), OSDI '16, USENIX, pp. 185–201.

[26] KAXIRAS, S., KLAFTENEGGER, D., NORGREN, M., ROS, A., AND SAGONAS, K. 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* (2015), HPDC '15, ACM, pp. 3–14.

[27] KOOP, M. J., JONES, T., AND PANDA, D. K. Mvapich-aptus: Scalable high-performance multi-transport MPI over infiniband. In *22nd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008* (2008), IPDPS '08, IEEE, pp. 1–12.

[28] KOOP, M. J., KUMAR, R., AND PANDA, D. K. 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* (2008), ICS '08, ACM, pp. 145–154.

[29] KOOP, M. J., SUR, S., GAO, Q., AND PANDA, D. K. High performance MPI design using unreliable datagram for ultra-scale InfiniBand clusters. In *Proceedings of the 21st annual international conference on Supercomputing* (2007), ICS '07, ACM, pp. 180–189.

[30] LESOKHIN, I., ERAN, H., RAINDEL, S., SHAPIRO, G., GRIMBERG, S., LISS, L., BEN-YEHUDA, M., AMIT, N., AND TSAFRIR, D. Page fault support for network controllers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (2017), ASPLOS '17, ACM, pp. 449–466.

[31] LI, D., CAMERON, K. W., NIKOLOPOULOS, D. S., DE SUPINSKI, B. R., AND SCHULZ, M. Scalable memory registration for high performance networks using helper threads. In *Proceedings of the 8th Conference on Computing Frontiers, 2011* (2011), C. Cascaval, P. Trancoso, and V. K. Prasanna, Eds., CF '11, ACM, p. 38.

[32] LI, M., HAMIDOUCHE, K., LU, X., SUBRAMONI, H., ZHANG, J., AND PANDA, D. K. 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* (2016), SC '16, IEEE, pp. 433–443.

[33] LI, M., LU, X., SUBRAMONI, H., AND PANDA, D. K. 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* (2017), HiPC '17, IEEE, pp. 62–71.

[34] LU, Y., SHU, J., CHEN, Y., AND LI, T. Octopus: an RDMA-enabled distributed persistent memory file system. In *Proceedings of the 2017 USENIX Annual Technical Conference* (2017), USENIX ATC '17, USENIX, pp. 773–785.

[35] MACARTHUR, P., LIU, Q., RUSSELL, R. D., MIZERO, F., VEERARAGHAVAN, M., AND DENNIS, J. M. An integrated tutorial on infiniband, verbs, and MPI. *IEEE Commun. Surv. Tutorials 19*, 4 (2017), 2894–2926.

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

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

[38] MIETKE, F., REX, R., BAUMGARTL, R., MEHLAN, T., HOEFLER, T., AND REHM, W. Analysis of the memory registration process in the Mellanox InfiniBand software stack. In *Euro-Par 2006 Parallel Processing* (2006), vol. 4128 of *Lecture Notes in Computer Science*, Springer, pp. 124–133.

[39] MITCHELL, C., GENG, Y., AND LI, J. Using one-sided RDMA reads to build a fast, cpu-efficient key-value store. In *2013 USENIX Annual Technical Conference, 2013* (2013), A. Birrell and E. G. Sirer, Eds., USENIX ATC '13, USENIX Association, pp. 103–114.

[40] MITTAL, R., SHPINER, A., PANDA, A., ZAHAVI, E., KRISHNAMURTHY, A., RATNASAMY, S., AND SHENKER, S. Revisiting network support for RDMA. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), SIGCOMM '18, ACM, pp. 313–326.

[41] NAKAMURA, M. Understanding retransmission control in infiniband. http://www.nminoru.jp/~nminoru/network/infiniband/iba-retransmission.html, 2014. In Japanese.

[42] NELSON, J., HOLT, B., MYERS, B., BRIGGS, P., CEZE, L., KAHAN, S., AND OSKIN, M. Latency-tolerant software distributed shared memory. In *2015 USENIX Annual Technical Conference, USENIX ATC '15* (2015), S. Lu and E. Riedel, Eds., USENIX ATC '15, USENIX Association, pp. 291–305.

[43] NOVAKOVIC, S., SHAN, Y., KOLLI, A., CUI, M., ZHANG, Y., ERAN, H., PISMENNY, B., LISS, L., WEI, M., TSAFRIR, D., AND AGUILERA, M. K. Storm: a fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage, SYSTOR 2019, Haifa, Israel, June 3-5, 2019* (2019), M. Hershcovitch, A. Goel, and A. Morrison, Eds., SYSTOR '19, ACM, pp. 97–108.

[44] OU, L., HE, X., AND HAN, J. 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* (2006), MSST '06.

[45] OU, L., HE, X., AND HAN, J. An efficient design for fast memory registration in RDMA. *Journal of Network and Computer Applications 32*, 3 (2009), 642–651.

[46] PRITCHARD, H. P. J. UCX in a nutshell and comparison to libfabric. Report, 2015.

[47] PRITCHARD, H. P. J. Comparison of Open UCX and OFI Libfabric. Open UCX Workshop 2016, 2016.

[48] RAFFENETTI, K., AMER, A., ODEN, L., ARCHER, C., BLAND, W., FUJITA, H., GUO, Y., JANJUSIC, T., DURNOV, D., BLOCKSOME, M., SI, M., SEO, S., LANGER, A., ZHENG, G., TAKAGI, M., COFFMAN, P. K., JOSE, J., SUR, S., SANNIKOV, A., OBLOMOV, S., CHUVELEV, M., HATANAKA, M., ZHAO, X., FISCHER, P. F., RATHNAYAKE, T., OTTEN, M., MIN, M.,

AND BALAJI, P. Why is MPI so slow?: analyzing the fundamental limits in implementing MPI-3.1. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017* (2017), B. Mohr and P. Raghavan, Eds., SC '17, ACM, pp. 62:1–62:12.

[49] RUDENKO, P. Sparkucx – rdma acceleration plugin for spark. 2020 Virtual OFA Workshop, 2020.

[50] SHAMIS, P., VENKATA, M. G., LOPEZ, M. G., BAKER, M. B., HERNANDEZ, O., ITIGIN, Y., DUBMAN, M., SHAINER, G., GRAHAM, R. L., LISS, L., SHAHAR, Y., POTLURI, S., ROSSETTI, D., BECKER, D., POOLE, D., LAMB, C., KUMAR, S., STUNKEL, C., BOSILCA, G., AND BOUTEILLER, A. UCX: An open source framework for HPC network APIs and beyond. In *Proceedings of 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects* (2015), HOTI '15, IEEE, pp. 40–43.

[51] SORIN, D. J., HILL, M. D., AND WOOD, D. A. A Primer on Memory Consistency and Cache Coherence. Book, 2012.

[52] TECHNOLOGIES, M. Introduction to InfiniBand. *Technical Report* (2003), 1–20.

[53] TEZUKA, H., O'CARROLL, F., HORI, A., AND ISHIKAWA, Y. 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* (1998), IPPS/SPDP '98, IEEE, pp. 308–314.

[54] WEI, X., DONG, Z., CHEN, R., AND CHEN, H. Deconstructing RDMA-enabled distributed transactions: Hybrid is better! In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation* (2018), OSDI '18, USENIX, pp. 233–251.

[55] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015* (2015), E. L. Miller and S. Hand, Eds., SOSP '15, ACM, pp. 87–104.

[56] WU, J., WYCKOFF, P., PANDA, D., AND ROSS, R. 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* (2004), CCGrid '04, IEEE, pp. 523–530.

[57] WU, J., WYCKOFF, P., AND PANDA, D. K. PVFS over infiniband: Design and performance evaluation. In *32nd International Conference on Parallel Processing (ICPP 2003), 6-9 October 2003, Kaohsiung, Taiwan* (2003), ICPP '03, IEEE Computer Society, pp. 125–132.

[58] WU, J., WYCKOFF, P., AND PANDA, D. K. Supporting efficient noncontiguous access in PVFS over infiniband. In *2003 IEEE International Conference on Cluster Computing (CLUSTER 2003), 1-4 December 2003, Kowloon, Hong Kong, China* (2003), CLUSTER '03, IEEE Computer Society, p. 344.

[59] XUE, J., MIAO, Y., CHEN, C., WU, M., ZHANG, L., AND ZHOU, L. Fast distributed deep learning over RDMA. In *Proceedings of the 14th EuroSys Conference 2019* (2019), EuroSys '19, ACM, pp. 1–14.

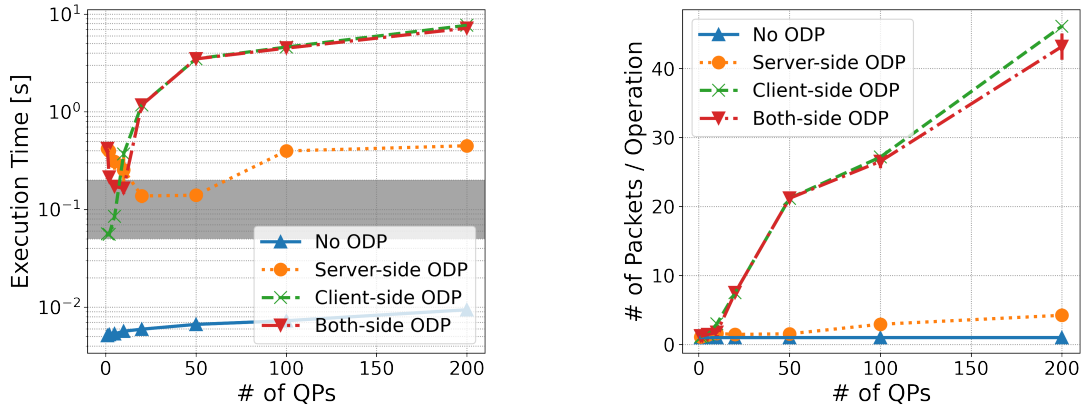
[60] ZHOU, Y., LI, K., BILAS, A., JAGANNATHAN, S., DUBNICKI, C., AND PHILBIN, J. Experiences with VI communication for database storage. In *29th International Symposium on Computer Architecture (ISCA 2002), 25-29 May 2002, Anchorage, AK, USA* (2002), Y. N. Patt, D. Grunwald, and K. Skadron, Eds., ISCA '02, IEEE Computer Society, p. 257.

[61] ZHU, Y., ERAN, H., FIRESTONE, D., GUO, C., LIPSHTEYN, M., LIRON, Y., PADHYE, J., RAINDEL, S., HAJ YAHIA, M., AND ZHANG, M. Congestion control for large-scale RDMA deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), SIGCOMM '15, ACM, pp. 323–536.

Appendix A

Performance Analysis with the Latest InfiniBand Adapter

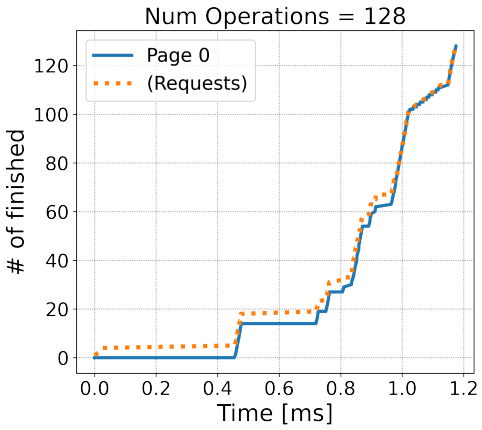
In Section 5.2, we have described when and why packet flood happens using ConnectX-4 adapters. This chapter shows packet flood still happens using the latest ConnectX-6 adapters. The machines used in experiments in this chapter is KNL-6 as Table 4.2 shows. The experimental conditions were basically the same as introduced in Section 5.2. We recommend that this chapter is read in light of the experimental results of ConnectX-4 adapters with the same experimental conditions.



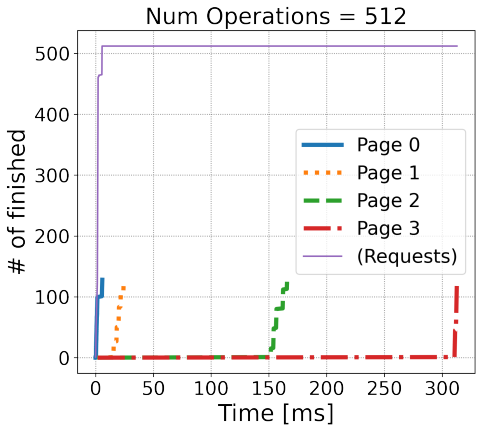
(a) Execution Time (s)

(b) Number of Packets

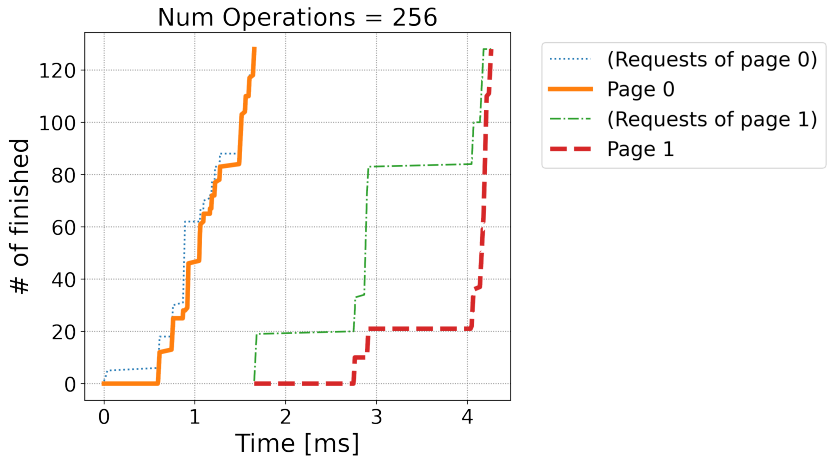
Figure A.1: Effect of varying the number of QPs with ConnectX-6 adapters. The experimental conditions were the same as introduced in Figure 5.7. Compared with the results of ConnectX-4 in Figure 5.7a, the execution time of ConnectX-6 adapters was overall shorter. Especially, the better performance with server-side ODP and more than 100 QPs came from the fact that packet damming did not happen with ConnectX-6 adapters.



(a) 128 operations

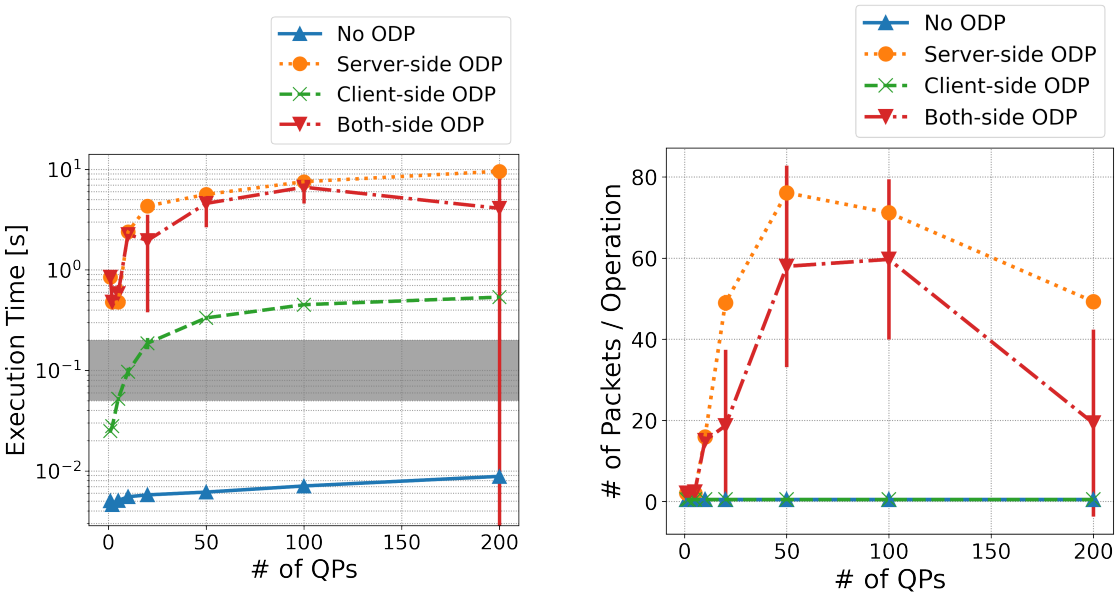


(b) 512 operations



(c) 256 operations

Figure A.2: Number of completed operations per page with ConnectX-6 adapters. The experimental conditions were the same as introduced in Figure 5.9. When 128 operations were issued, the performance did not degrade nor update failure of page status happen unlike the ConnectX-4 described in Figure 5.9a. According to the behavior of the requests, we can deduce that the reason is that ConnectX-6 adapters are equipped with the mechanism to control the number of outstanding requests across all QPs. Meanwhile, when 512 operations were issued, it took about 300 ms for all the operations to finish, which represented the occurrence of performance degradation. To investigate further, we additionally conducted another experiment using 256 operations and found that the update failure of page status certainly happened with ConnectX-6 adapters as the result of page 1 in the figure shows.



(a) Execution Time (s)

(b) Number of Packets

Figure A.3: Effect of varying the number of QPs with the micro-benchmark in Figure 5.1 with WRITE operations and ConnectX-6 adapters. The experimental conditions were the same as introduced in Figure 5.10. The result shows that the packet flood of WRITE operations with only one QP also happened with ConnectX-6 adapters.

Appendix B

Code snippets of the micro-benchmark

The micro-benchmark is basically based on https://github.com/jcxue/RDMA-Tutorial (Apache License 2.0). For the convenience of our experiments, significant modifications are conducted to it. Since it is not possible to include everything, we focus on reproducibility and provide several pieces of a code snippet containing miscellaneous parameters related to QP initialization.

```
1  struct ConfigInfo {
2    bool is_server;
3    int msg_size;
4    int sleep_time;
5    int timeout;
6    int retry_cnt;
7    int rnr_timer;
8    int odp_in_server;
9    int odp_in_receiver;
10 }
11 __attribute__((aligned(64)))
```

Figure B.1: List of configurations which we changed depending on the experiments in this thesis.

```
1  void main() {
2    // --snip--
3
4    /* create CQ */
5    ib_res.cq = ibv_create_cq(ib_res.ctx, 131072, NULL, NULL, 0);
6
7    /* create QP */
8    struct ibv_qp_init_attr qp_init_attr = {
9      .send_cq = ib_res.cq,
10     .recv_cq = ib_res.cq,
11     .cap = {
12       .max_send_wr = 8192,
13       .max_recv_wr = 8192,
14       .max_send_sge = 3,
15       .max_recv_sge = 3,
16     },
17     .qp_type = IBV_QPT_RC,
18   };
19
20   for (i = 0; i < ib_res.qp_num; i++) {
21     ib_res.qps[i] = ibv_create_qp(ib_res.pd, &qp_init_attr);
22
23     /* connect QP */
24     ret = connect_qp(ib_res.qps[i]);
25   }
26
27   // --snip--
28 }
29
30 int connect_qp(struct ibv_qp* qp) {
31   int ret = 0, n = 0;
32   struct QPInfo local_qp_info, remote_qp_info;
33
34   local_qp_info.lid = ib_res.port_attr.lid;
35   local_qp_info.qp_num = qp -> qp_num;
36   local_qp_info.rkey = ib_res.mr -> rkey;
37   local_qp_info.raddr = (uintptr_t) ib_res.ib_buf;
38
39   if (config_info.is_server) {
40     MPI_Send(&local_qp_info, sizeof(struct QPInfo), MPI_BYTE, 1, 0,
         MPI_COMM_WORLD);
41     MPI_Recv(&remote_qp_info, sizeof(struct QPInfo), MPI_BYTE, 1, 0,
         MPI_COMM_WORLD, MPI_STATUS_IGNORE);
42   } else {
43     MPI_Recv(&remote_qp_info, sizeof(struct QPInfo), MPI_BYTE, 0, 0,
         MPI_COMM_WORLD, MPI_STATUS_IGNORE);
44     MPI_Send(&local_qp_info, sizeof(struct QPInfo), MPI_BYTE, 0, 0,
         MPI_COMM_WORLD);
45   }
46
47   /* store rkey and raddr info */
48   ib_res.rkey = remote_qp_info.rkey;
49   ib_res.raddr = remote_qp_info.raddr;
50
51   /* change QP state to RTS */
52   ret = modify_qp_to_rts(qp, remote_qp_info.qp_num,
53     remote_qp_info.lid);
54
55   return 0;
56 }
```

```c
#define IB_PORT 1
#define IB_SL 0

int modify_qp_to_rts(struct ibv_qp * qp, uint32_t target_qp_num, uint16_t
    target_lid) {
  int ret = 0;

  /* Change QP state to INIT */
  {
    struct ibv_qp_attr qp_attr = {
      .qp_state = IBV_QPS_INIT,
      .pkey_index = 0,
      .port_num = IB_PORT,
      .qp_access_flags = IBV_ACCESS_LOCAL_WRITE |
      IBV_ACCESS_REMOTE_READ |
      IBV_ACCESS_REMOTE_ATOMIC |
      IBV_ACCESS_REMOTE_WRITE,
    };

    ret = ibv_modify_qp(qp, & qp_attr,
      IBV_QP_STATE | IBV_QP_PKEY_INDEX |
      IBV_QP_PORT | IBV_QP_ACCESS_FLAGS);
  }

  /* Change QP state to RTR */
  {
    struct ibv_qp_attr qp_attr = {
      .qp_state = IBV_QPS_RTR,
      .dest_qp_num = target_qp_num,
      .rq_psn = 0,
      .path_mtu = IBV_MTU_4096,
      .max_dest_rd_atomic = 4,
      .min_rnr_timer = config_info.rnr_timer,
      .ah_attr.dlid = target_lid,
      .ah_attr.sl = IB_SL,
      .ah_attr.src_path_bits = 0,
      .ah_attr.port_num = IB_PORT,
    };

    ret = ibv_modify_qp(qp, & qp_attr,
      IBV_QP_STATE | IBV_QP_AV |
      IBV_QP_PATH_MTU | IBV_QP_DEST_QPN |
      IBV_QP_RQ_PSN | IBV_QP_MAX_DEST_RD_ATOMIC |
      IBV_QP_MIN_RNR_TIMER);
  }

  /* Change QP state to RTS */
  {
    struct ibv_qp_attr qp_attr = {
      .qp_state = IBV_QPS_RTS,
      .sq_psn = 0,
      .timeout = config_info.timeout,
      .rnr_retry = 7,
      .retry_cnt = config_info.retry_cnt,
      .max_rd_atomic = 4,
    };

    ret = ibv_modify_qp(qp, & qp_attr,
      IBV_QP_STATE | IBV_QP_TIMEOUT |
      IBV_QP_RETRY_CNT | IBV_QP_RNR_RETRY |
      IBV_QP_SQ_PSN | IBV_QP_MAX_QP_RD_ATOMIC);
  }

  return 0;
}
```