Hardware-Supported Remote Persistence for Distributed Persistent Memory

Zhuohui Duan, Haodi Lu, Haikun Liu, Xiaofei Liao, Hai Jin, Yu Zhang, Song Wu
National Engineering Research Center for Big Data Technology and System,
Services Computing Technology and System Lab, Cluster and Grid Computing Lab,
School of Computing Science and Technology, Huazhong University of Science and Technology
Wuhan, 430074, China
{zhduan,haodilu,hkliu,xfliao,hjin,zhyu,wusong}@hust.edu.cn

ABSTRACT

The advent of *Persistent Memory* (PM) necessitates an evolution of *Remote Direct Memory Access* (RDMA) technologies for supporting remote data persistence. Previous software-based solutions require remote CPU intervention and postpone the visibility of remote persistence. In this paper, we design several hardware-supported RDMA primitives to flush data from the volatile cache of *RDMA Network Interface Cards* (RNICs) to the PM. We also propose durable RPCs based on the proposed RDMA `Flush` primitives to support remote data persistence and fast failure recovery. We emulate the performance of RDMA `Flush` primitives through other RDMA primitives, and compare our proposals with several state-of-the-art RPCs in a real testbed equipped with PM and InfiniBand networks. Experimental results show that our proposals can improve the throughput of RPCs by up to 90%, and reduce the 99th percentile latency by up to 49%. The experimental studies also provide instructive guidelines for designing RDMA-based distributed PM systems.

CCS CONCEPTS

• **Networks** → **Network protocol design**; • **Hardware** → **Networking hardware**.

KEYWORDS

RDMA, PM, RPC, Data Persistence

ACM Reference Format:
Zhuohui Duan, Haodi Lu, Haikun Liu, Xiaofei Liao, Hai Jin, Yu Zhang, Song Wu. 2021. Hardware-Supported Remote Persistence for Distributed Persistent Memory. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3458817.3476194

Zhuohui Duan and Haodi Lu contributed equally to this work. Haikun Liu is the corresponding author.

SC '21, November 14–19, 2021, St. Louis, MO, USA

ACM ISBN 978-1-4503-8442-1/21/11…$15.00
https://doi.org/10.1145/3458817.3476194

1 INTRODUCTION

Persistent Memory (PM) technologies, such as *Phase Change Memory* (PCM) [34] and 3D XPoint [23] promise high memory density, low cost per bit, DRAM-like performance , and disk-like durability. Particularly, the byte-addressability and non-volatility features of PM enable durable and recoverable data structures in main memory, and thus have fundamentally changed the way that system software and applications manage persistent data. On the other hand, *Remote Direct Memory Access* (RDMA) technologies can offer extremely low network latency and high bandwidth, and enable low-latency remote memory accesses over networks by bypassing the operating system kernel and eliminating memory copying across buffers (zero-copy). They have been widely utilized to improve networking performance in HPC and data center environments [15] [16] [19] [39]. Recently, the emergence of PMs has inspired a number of studies on RDMA-based distributed PM systems, file systems [24] [25] [45], and data center applications [4] [7] [41].

Some previous studies [37] [38] [45] have demonstrated significant performance benefit from PM and RDMA. However, existing *RDMA Network Interface Cards* (RNICs) have not considered the durability feature of PM. There has been very little understanding on durable RDMA updates to the remote PM. For standard RDMA protocols, an RDMA `write` operation completes once the client has received a *Work Completion* (WC) acknowledgement from the server. However, this acknowledgement does not necessarily imply that the data buffered in the volatile cache of the server's RNIC has been persisted in the PM. Without a validation of remote data persistence, distributed applications may incur data consistency and concurrency problems. For example, the client may release the current write lock before the data is persisted in the remote PM, and thus other clients may read the stale data. Moreover, the invisibility of remote persistence may cause data corruption in case of a system crash or a power failure.

There have been very few studies on the characteristics of RDMA and its implications on designing durable, correct, and efficient RDMA operations for PM systems. A few proposals verify the remote data persistence and correctness using an RDMA read-after-write mechanism [24, 35], i.e., an RDMA read follows an RDMA write to verify the data integrity. However, the effectiveness of these schemes is highly influenced by some hardware features such as *Data Direct I/O* (DDIO) [6]. DDIO allows the RNIC to directly place the received data in the server's on-chip cache, making the verification of remote data persistence fail repetitively. Some other approaches such as Orion [51] involve the receiver's CPU in data persisting, and thus offset the performance benefit of one-sided

RDMA communication. Moreover, many RDMA-based *Remote Procedure Calls* (RPCs) [5] [39] naturally guarantee remote data persistence via a set of RDMA operations. However, those RPCs require the remote server's CPU to process the incoming data, and thus increase CPU load of the remote server, and also postpone the visibility of remote data persistence.

In this paper, we first make a comparative study of previous RDMA-based RPC designs and their impacts on the efficiency of durable RDMA operations. Based on the lessons learnt, we design a set of RNIC hardware-supported RDMA primitives to flush data from the volatile cache of RNICs to the PM. We then implement several durable RPCs based on the proposed RDMA `Flush` primitives to support remote data persistence and fast failure recovery. Since our durable RPCs decouple the data persisting from the RPC processing, the remote data persistence is visible to applications much earlier than traditional RPCs. This offers vast opportunities to improve the performance of applications by overlapping the RDMA transmission and the RPC processing. We also show that it is potential to recover incomplete RPCs at the server side in case of a system crash or a power failure, without resenting data from the client. Moreover, for special hardware features such as DDIO [6], and typical RDMA transmission optimizations such as data batching, we discuss their impacts on remote data persistence and then present our solutions.

We emulate the performance of RDMA `Flush` primitives through other existing RDMA primitives, and compare our proposals with previous state-of-the-art RPCs in a real testbed equipped with Intel Optane DC Persistent Memory Modules and InfiniBand networks. Experimental results show that our RPCs using RDMA `Flush` primitives can significantly improve the throughput of RPCs by up to 90%, and reduce the tail latency by up to 49%. The experimental studies also provide substantial and instructive guidelines for designing high-performance RDMA-based PM systems with remote data persistence and consistency guarantees.

The remaining of this paper is organized as follows. Section 2 presents the background and discusses remote data persistence problems. Section 3 analyzes existing data persisting mechanisms and presents our motivations. Section 4 describes our designs of RDMA `Flush` primitives and durable RPCs. Section 5 presents experimental results. Section 6 introduces related work, and we conclude in Section 7.

2 BACKGROUND

In this section, we introduce the background of distributed PM systems and challenges of remote data persistence.

2.1 Persistent Memory

The advent of Intel Optane *DC Persistent Memory Modules* (DCPMM) [23] [32] has finally made NVM (or PM) commercially available. They have many promising features, such as DRAM-like performance, disk-like durability, byte-addressability, high density, and low cost. This new memory technology could complement and maybe even replace DRAM technologies in the future. NVMs can be directly attached to the memory bus, and used as PM through load/store instructions. The latest OS kernels already support *APP Direct Mode* or *Memory Mode* to use it. As PM can provide extremely large and durable memory space to enable in-memory computing, it can significantly benefit many big data applications.

For mission-critical enterprise applications, it is essential to guarantee data durability and consistency when data is written to PM. The SNIA PM programming model [29] requires applications to explicitly flush data from volatile CPU cache to PM via a set of machine instructions. For example, Intel's *clflush-opt* and *clwb* instructions are used to fence a number of writes and force them to complete in order before handling other writes. In a distributed PM system, it is also necessary to guarantee data persistence and consistency when writing data to the remote PM.

2.2 RDMA

RDMA is a technology that allows an application to directly access memory in a remote machine across the network. It is able to bypass the OS kernel and to eliminate memory copying among buffers [19, 28], and thus achieves low-latency and high-throughput network transmission with low CPU load. Generally, RDMA supports three transmission modes: *reliable connection* (RC), *unreliable connection* (UC), and *unreliable datagram* (UD). The RC mode guarantees lossless data transfer and in-order delivery of messages. UC and UD modes achieve better performance than the RC mode by relinquishing the reliability of data transmission. In the following, the local server that sends RDMA requests is called the sender, while the remote server that receives RDMA requests is called the receiver. As reliability is essential to durable RDMA write/send operations, we use reliable connections in this paper if not specified.

RDMA supports both channel semantics and memory semantics for remote memory accesses, i.e., two-sided operations using RDMA `send/recv` verbs, and one-sided operations using RDMA `read/write` verbs. Two-sided RDMA operations are usually used in a classic I/O channel. They should interrupt the receiver's CPU for processing RDMA messages. In contrast, one-sided operations do not need to involve the receiver's CPUs. The receiver's RNIC exploits a *Direct Memory Access* (DMA) mechanism to process RDMA `read/write` operations. In addition, RDMA *write with immediate data* (RDMA `write-imm`) is a more complicated primitive that can perform an RDMA `write` to the receiver's memory accompanying with a 32-bit *immediate* (IMM) value. It notifies the write completion to the receiver's CPU which then performs other operations with the IMM. These RDMA verbs and communication modes provide a large design space to choose an efficient solution for a given application, and different choices may have a significant impact on the application performance.

2.3 DDIO Technologies

Recently, modern processor vendors such as Intel announce an innovative technology called DDIO [6]. It allows incoming RDMA writes to be placed directly in the CPU's L3 cache, rather than *Integrated Memory Controller* (IMC) buffers. DDIO can avoid writing data to main memory and then reading it to the CPU cache for further processing, and thus can significantly improve the system I/O performance. However, DDIO complicates remote data persisting in distributed PM systems. To ensure data persistence, the remote host's CPU needs to flush the incoming data from the LLC to the persist domain. Clearly, DDIO prolongs the data path of a remote

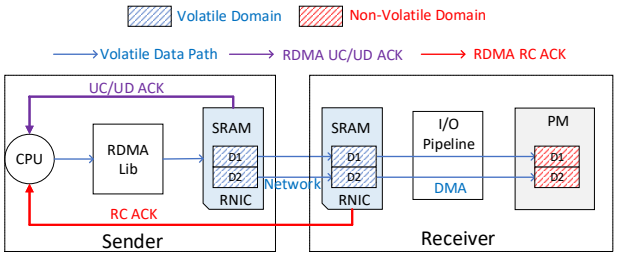

Figure 1: The data path of an RDMA `write`

write, and thus hampers remote data persistence in RDMA-based distributed PM systems.

2.4 Challenges of Remote Data Persisting

Figure 1 shows a typical data path of RDMA `write` in an RDMA-based PM system. In the RDMA RC mode, once the data is received by the receiver's RNIC, it returns a *work completion* (*WC*) event to the sender. In the RDMA UC/UD modes, once the sender's RNIC has sent out the data, it notifies the completion of RDMA operations to applications. Assume T_A represents the time when applications receive the *WC* event of RDMA write operations, and T_B represents the time when the received data has been physically written from the receiver's RNIC buffer to the PM. If T_A is earlier than T_B, a system crash or a power failure may lead to a data inconsistent problem.

Overall, the volatile cache of RNICs is the root cause of data persistence problems in distributed PM systems. The invisibility of data persistence leads to a risk of data corruption. As a result, the SNIA PM programming model advocates an RDMA read-after-write mechanism [35] to guarantee data persistence. Specifically, the sender reads the last byte of the data from the PM to check its correctness once the RDMA write operation completes. Although the RDMA read-after-write mechanism is easy to implement, it has been questioned [13] since some hardware features of the remote server may affect its effectiveness. For example, the DDIO technology allows the receiver's RNIC directly place the incoming data in the volatile L3 cache. The sender is not able to perceive whether and when the data is written to the PM. In this case, the RDMA read-after-write mechanism becomes ineffective because the data read by the sender is actually in the L3 cache. Overall, because the RDMA transmission is decoupled with the data persisting in current RDMA networks, it is often costly to guarantee data persistence and consistency through a software approach solely.

3 MOTIVATIONS

In an RDMA-enabled PM system, the sender should be aware of the time when the data is persisted in the remote PM. This is particularly essential to concurrency control for distributed shared PM systems. To guarantee remote data persistence and consistency, it is necessary to support durable RDMA writes for RDMA networks. Unfortunately, so far there is no commercially available RNIC hardware that supports durable RDMA writes and failure atomicity.

Although most existing RPC systems are designed to facilitate RDMA programming for *Distributed Shared Memory* (DSM), they

Table 1: RDMA based RPC systems

RPC Implementations	RC	UC/UD	DDIO
RDMA Write	Jakiro [40], L5 [11], FaRM [8]	Herd [18]	ScaleRPC [5], [14]
RDMA Send	DaRPC [39], Mojim [54], Hotpot [38], NVFS [15]	FaSST [19]	FileMR [52]
RDMA Write with Imm	LITE [46], Orion [51]	-	-

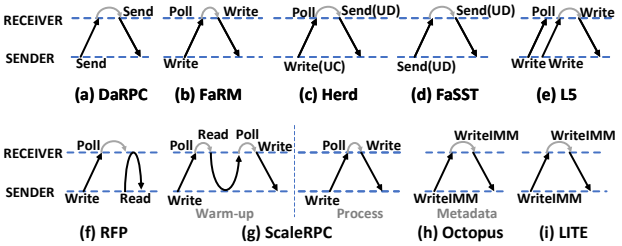

Figure 2: Data transmission modes in different RPC systems

actually support durable RDMA writes. To study previous durable RDMA write mechanisms in existing RPC systems, we classify existing RDMA-based DSM systems according to different RDMA primitives and transmission modes, as shown in Table 1. Since RDMA RC mode ensures reliable data transmission, it has a potential to achieve durable RDMA writes without involving the remote server's CPUs. In contrast, RDMA UC/UD modes must interrupt the remote server's CPUs to verify the data persistence. More importantly, since the DDIO technology complicates data persisting in the remote PM, we discuss its impact on remote data persistence particularly.

Many previous studies use RPCs to optimize RDMA transmission. An RPC often uses a set of RDMA primitives to perform a complicated task with fewer RTTs. Most RPCs require the remote server's CPU to process the written data. The sender has to wait for a completion event from the remote CPU. Although the primary goal of RPC designs is not to guarantee data persistence, they naturally support durable RDMA writes. In the following, we elaborate several typical RDMA-based RPC designs to illustrate how they guarantee remote data persistence. Figure 2 illustrates the transmission modes and RDMA primitives used in different RPC systems.

Figure 2(a) shows a typical RPC design in DaRPC [39]. At first, the sender uses an RDMA `send` primitive to send a message (including data and metadata) to the receiver. The receiver's RNIC receives the message and stores it in a message buffer. It notifies the receiver's CPU to parse data and metadata in the message, and then the data is copied to the target memory space according to the metadata. At last, the receiver uses an RDMA `send` primitive to notify the completion of this RPC to the sender.

Figure 2(b), Figure 2(c), and Figure 2(d) show abstractions of RPCs in FaRM [8], Herd [18], and FaSST [19], respectively. The data transmission model of these RPCs is similar to DaRPC. The only difference is that they use different RDMA primitives to transfer data and acknowledgement (ACK) messages. Specifically, FaRM, Herd, and FaSST use RC-based RDMA `write`, UC-based RDMA `write`, and UD-based RDMA `send`, respectively. The receivers exploit RDMA `recv` primitives or a polling mechanism to store the

received data in the persistent domain. If a sender receives a completion event of the RPC, it can confirm that the data has been persisted in the remote server.

As shown in Figure 2(e), L5 [11] uses two RDMA `writes` to transfer data and a valid message, respectively. The receiver polls the message buffer for incoming RDMA writes, and then returns the processing result using an RDMA `write` primitive. Figure 2(f) illustrates the communication mode of RFP [40]. The sender uses an RDMA `write` to transfer data, and then uses an RDMA `read` to collect the result processed by the receiver. Figure 2(g) shows the work flow of ScaleRPC [5]. The data transfer includes two phases: *warmup* and *process*. In the former phase, the sender uses an RDMA `write` to only send the local address of the data to the receiver's message buffer. The receiver polls the message buffer and uses an RDMA `read` to fetch the requested data from the sender. Once the computation is completed, the receiver notifies the sender of a completion event using an RDMA `write`. Subsequently, ScaleRPC begins the process phase, and the data transmission model is the same as FaRM (Figure 2(b)). Figure 2(h) and Figure 2(i) show the RPC designs in Octopus [25] and LITE [46], respectively. They both use the RDMA `write-IMM` primitive to notify the receiver's CPU for further data processing. Their data transmission models are similar to FaRM, except different RDMA primitives used. Unlike Octopus, LITE implements RPCs in the kernel, and thus data access permissions should be spread to the kernel.

From the above description, we can find that these RPCs all guarantee data persistence naturally. When the RPC is completed, the sender can validate that the data has been persisted by the receiver. However, the costly RPC processing, RDMA networking, and PCIe operations are on the critical path of client applications. They often significantly increase the end-to-end latency compared with a single RDMA primitive. Moreover, the DDIO technology also prolongs the data path of remote data persisting at the receiver side.

To evaluate the impact of different RDMA-based RPC designs on the performance of durable RDMA writes, we design a set of RDMA `Flush` primitives and use them to implement new RPCs for durable RDMA writes. We compare our RPC design with previous RPCs in a real testbed using Intel Optane DCPMMs and InfiniBand networks. Compared with previous works using software-emulated NVMs, our evaluation can reflect the impact of different durable RDMA write mechanisms on application performance more accurately.

4 DURABLE RDMA OPERATIONS

Current RNIC hardware and firmware do not provide enough support for remote data persistence. To embrace the new PM, it is essential to design new RDMA primitives and more efficient RPCs to utilize PM in a distributed environment. Moreover, since previous RPC systems tightly couple the data persisting with the processing of RPCs, they should be also redesigned to allow the remote data persistence visible to the sender earlier.

4.1 RDMA Flush Primitives

To reduce the end-to-end latency of durable RDMA operations, we design new RDMA `Flush` primitives and data persisting models. We classify the proposed RDMA `Flush` primitives into two

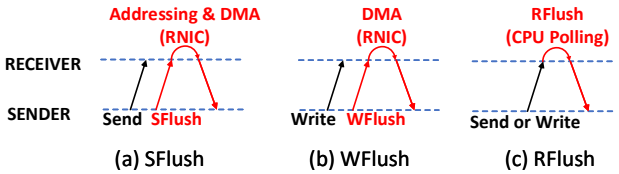

Figure 3: RDMA Flush primitives

categories: sender-initiated and receiver-initiated. Sender-initiated RDMA `Flush` primitives are issued by the sender to verify the remote data persistence, while receiver-initiated Flush primitives are issued by the receiver to notify the completion of data persisting to the sender.

4.1.1 Sender-initiated RDMA Flush Primitives. Sender initiated RDMA `Flush` primitives are designed according to the hardware features of RDMA and PM. One-sided RDMA communication mode bypasses OS kernels, and does not involve the remote server's CPUs. Also, future smart RNICs have a potential to directly flush data from their volatile cache to PM. Thus, it is possible to achieve durable RDMA writes/sends without the involvement of remote CPUs. Sender-initiated RDMA primitives can be used only in the RC mode, because these RDMA primitives have to wait for ACKs from the remote RNIC.

We design two RDMA `Flush` primitives (i.e., `SFlush` and `WFlush`) for RDMA `send` and RDMA `write` operations, respectively. These RDMA `Flush` primitives can flush data from volatile cache (SRAM) to the PM by the RNIC hardware. Once the data has been stored in the PM, an ACK of the RDMA `Flush` is issued by the receiver's RNIC to notify the completion of data persisting to the sender. Both `SFlush` and `WFlush` do not involve the receiver's CPUs.

A `SFlush` should be accompanied with an RDMA `send` primitive. Because RDMA `send` primitives rely on the receiver's CPU to obtain the destination memory address and perform DMA operations, the CPU intervention increases software overhead. We believe future smart RNICs can do the same job itself with our proposed `SFlush` primitive. When the receiver's RNIC receives a `SFlush` primitive, it should first parse the received data packet to get the remote memory address, and then performs the DMA operation to persist the data, as shown in Figure 3(a). In this way, we can shorten the latency that the data persistence is visible to the sender. Similarly, a `WFlush` should be accompanied with an RDMA `write` primitive, as shown in Figure 3(b). It is aware of the target address of the received data because the RDMA `write` primitive contains the remote memory address.

4.1.2 Receiver-initiated RDMA Flush Primitive. Unlike sender initiated RDMA `Flush` primitives, the receiver initiated RDMA `Flush` primitive (i.e., `RFlush`) should flush the received data to PM by the receiver's RNIC itself, without any involvement of the receiver's CPU. However, since current RNIC hardware does not support active data flushing, we rely on the receiver's CPU to emulate RDMA `RFlush` primitives. Figure 3(c) demonstrates durable RDMA send/write operations using `RFlush`. The receiver's CPU polls the message buffer for incoming RDMA `send/write` operations, and issues an RDMA `RFlush` primitive to flush data to the PM. Once the data has been persisted, the receiver's CPU notifies the completion

Figure 4: Durable RPCs based on RDMA Flush primitives

of data persisting to the sender immediately. Although RFlush leads to a moderate CPU load, it can be used in all RDMA connection modes, such as UC/UD.

4.1.3 Emulation of RDMA Flush Primitives. Since RDMA Flush primitive is not supported by current RNIC hardware, we use other existing RDMA primitives to emulate the performance of RDMA Flush primitives. Because a WFlush follows an RDMA write, we emulate RDMA WFlush primitives by reading the last byte of the data via an RDMA read primitive, which forces the receiver's RNIC to flush the data to the PM via DMA immediately. For RDMA send primitives, the RDMA SFlush primitive should consult the receiver's RNIC to get the destination address of the data. To emulate the RDMA SFlush, we first wait for a while to simulate the time spent in looking up the destination address, and then perform an RDMA read operation to force the data flushing. Since the address lookup is performed by RNIC hardware, we use a function *sleep(0)* to simulate the latency (about 7 us) of addressing conservatively. For receiver-initiated RDMA RFlush primitive, we rely on the receiver's CPU to emulate its functionality. The CPU detects incoming RDMA requests via busy polling. Once the data is flushed to the PM, it notifies the completion of data persisting to the sender immediately.

4.2 Durable RPCs for Failure Recovery

Failure recovery is important for mission-critical enterprise applications. In an RDMA-based RPC system, a failure at the server side may occur during the execution of the RPC. We exploit redo logging to achieve fast recovery of incomplete RPCs, without re-sending the data from the client.

We design durable RPCs using the aforementioned RDMA Flush primitives, as shown in Figure 4. Unlike previous RPC designs, our durable RPCs can guarantee that the received data is stored in the receiver's PM when the sender has received the ACK of RDMA Flush primitives. At this time, if the receiver suffers a system crash or a power failure, we can recover the incomplete RPCs using the redo log stored in the PM. Once the receiver is recovered from a failure, the RPC can be re-executed with the operation log, without re-sending the data from the client. However, if the received data has not yet persisted to the remote PM, the data in the RNIC's volatile cache would be lost upon a system crash or a power failure. In this case, the RDMA connection should be re-established and the data should be re-sent to the receiver. This process is usually very costly.

Figure 5 shows how durable RPCs are processed at the receiver side. When the receiver's RNIC receives RDMA requests, the data packets are cached in the volatile RNIC buffer (SRAM). We maintain a ring buffer in the PM to store the redo log. For each RDMA connection, the corresponding connection information is recorded in the log header. A log entry contains the RPC operator and the

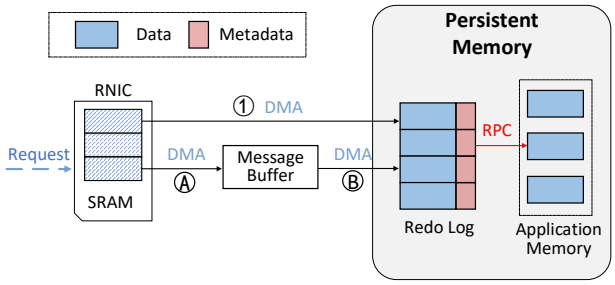

Figure 5: Failure recovery mechanism using durable RPCs

data. To guarantee failure atomicity, the data is always persisted before the RPC operator in the logging buffer. Because the size of the RPC operator is very small, it can be persisted with an atomic write. Once an RPC is successfully processed, the corresponding log entry is removed from the ring buffer. For an RDMA WFlush-based RPC, the RNIC performs a DMA operation (①) to store the data and the RPC operator in the log, and then notifies the completion of remote data persisting to the sender. At this time, a thread is created to handle the RPC requests. Meanwhile, the sender can initiate another RPC request earlier without waiting for the completion event of the former RPC, because our design guarantees that the durable RPC can be processed eventually with the redo log even when a system failure occurs. For an RDMA SFlush-based RPC, the data packet should be first stored in the message buffer for further processing (Ⓐ) because the RDMA send primitive does not contain the remote memory address. With the RDMA SFlush primitive, we can immediately flush the data in the message buffer to the redo log via another DMA (Ⓑ).

Since our durable RPC design decouples the data persisting from the RPC processing via RDMA Flush primitives, the remote data persistence is visible to the sender much earlier than traditional RPCs. With the redo logging mechanism, the sender can issue other RPC requests without waiting for the completion event of the RPC that is still being processed. Even when the following RPC requests would process the same data, they will be queued in the logging buffer and then processed in a FIFO order. The redo logging mechanism not only guarantees failure atomicity during the processing of RPC, but also provides the ordering guarantee of RPC requests for data concurrency. On the other hand, the sender may send too many requests even when the receiver is under a high load. Thus, when the incomplete RDMA requests accumulated at the receiver side become larger than a given threshold, the receiver should notify the sender to slow down the speed of data transmission. The sender can simply throttle new RPC requests for a short while.

4.3 Remote Data Persisting for Batching

A number of RPC systems such as DaRPC [39], FaSST [19], and ScaleRPC [5] perform a set of RDMA operations in a batch to improve the network throughput. Our RDMA Flush primitives are also applicable for batched RDMA operations. Taking RDMA WFlush as an example, Figure 6 shows the batched RDMA writes combining with RDMA WFlush primitives. In a batching-disabled case, an RDMA WFlush is usually accompanied with an RDMA write operation to persist the data, as shown in Figure 6(a). When

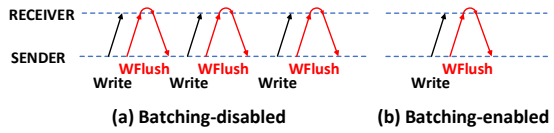

Figure 6: RDMA `WFlush` for batched data transmission

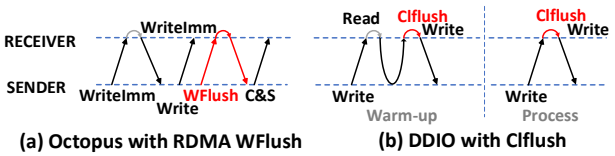

Figure 7: Remote data persistence guarantee in Octopus and a DDIO-enabled PM system

the batching mechanism is enabled, the sender transfers a large message accompanying with an RDMA `WFlush` at a time, as shown in Figure 6(b).

4.4 Case Studies of Using RDMA Flush Primitives

In this section, we illustrate how previous RDMA systems and the DDIO setup can support remote data persistence by using our RDMA `Flush` primitives.

4.4.1 Remote Data Persisting for Existing RDMA Systems. A few previous proposals such as Tailwind [42], Octopus [25], and Clover [45] store data to remote memory via RDMA `write` primitives. Unfortunately, these systems do not guarantee data persistence for RDMA `write` operations.

Taking Octopus [25] as an example, we illustrate how these RDMA-enabled PM systems can exploit the RDMA `WFlush` primitive to guarantee remote data persistence effectively, as shown in Figure 7(a). In Octopus, the sender first obtains the destination memory address using a RPC implemented with RDMA `write-IMM`. It then uses an RDMA `write` primitive to write the data to the PM. To make the RPC in Octopus durable, we can use an RDMA `WFlush` primitive following the RDMA `Write` primitive. The RDMA `WFlush` primitive guarantees that the received data is flushed to the PM when the sender receives the ACK of the RDMA `WFlush`.

4.4.2 Remote Data Persistence with DDIO Technologies. The DDIO technology changes the destination of RDMA data path from the PM to the volatile on-chip cache. To guarantee data persistence, the receiver's CPUs should perform a *clflush* instruction to flush the data from the L3 cache to the PM.

We take ScaleRPC [5] as an example to illustrate our remote data persisting mechanism in a DDIO-enabled system. Figure 7(b) illustrates the RPC in ScaleRPC. In the *warmup* phase, the sender uses an RDMA `write` to send the local address of the data to the receiver's message buffer. The receiver parses the message and uses an RDMA `read` to fetch the data from the sender. To guarantee data persistence if the DDIO technology is enabled, the receiver has to actively perform a *clflush* instruction to flush the data from the L3 cache to the PM. The receiver then uses an RDMA `write` to notify the completion of data persisting to the sender. In the following, ScaleRPC begins the *process* phase, and then the sender uses RDMA `write` primitives to transfer data to the receiver. Again, the receiver needs to flush data from the L3 cache to the PM via a *clflush* instruction. Then, the sender can validate the remote data persistence once it receives the completion event of *clflush* via an RDMA `write` primitive.

We argue it is potential to support remote data persistence for a given application while other applications can still take advantage of DDIO technologies. For example, applications can apply for non-cacheable memory regions at the remote server side, and thus data mapped to the non-cacheable memory region is oblivious to the DDIO. Moreover, fine-grained DDIO control can be achieved by configuring PCIe root ports or PCIe transactions [30], and thus one application can be configured to forego DDIO while other application can still benefit from DDIO.

4.5 Discussion

SmartNICs. Although there is not commercially available RNIC hardware that supports durable RDMA operations currently, we believe these functionalities would be enabled by future smartNICs. They contains considerable programmable elements such as ARM cores and *lookup tables* (LUTs), which are used to offload some CPU-intensive operations. To support RDMA `Flush` primitives in smartNICs, both the firmware of smartNICs and the RDMA library should be modified. For the RDMA `SFlush` primitive, it is possible to use an ARM core to infer the destination memory address directly, without remote CPU intervention. For the RDMA `RFlush` primitive, lookup tables and on-chip memory in smartNICs can be used to configure applications' data persistence requirements, and then the smartNIC itself can issue RDMA `RFlush` primitives to flush data from the RNIC cache into the PM. Moreover, smartNICs can be also programmed to offload some RPC operations that are originally handled by the receivers' CPUs.

Data Persistence with Multiple Replicas. In distributed systems, data replication is widely used to improve system availability and reliability. It can guarantee that an accurate backup exists at all times in case of a permanent hardware failure. However, although the RDMA reliable connection guarantees reliable transmission, it can not guarantee the order of RDMA `Flush` ACKs from multiple distributed replicas. In this scenario, a consensus mechanism is usually required to make a tradeoff between data consistency and application performance. A early work Hyperloop [22] designs new RDMA primitives to offload data replication transactions in storage systems to advance RNICs. However, it still relies on the RDMA read-after-write mechanism to guarantee durable RDMA writes. In this work, we mainly focus on remote data persistence and transient failure recovery in a point-to-point connection. Our work offers foundational capabilities for data replication protocols in distributed storage systems. We believe that the advent of smartNICs offers vast opportunities to offload many CPU operations in data replication protocols, significantly shortening the data path of replication.

Maintaining Cache Coherence. In the data server, since the PM can be accessed by both RNICs and local CPUs concurrently, the RNICs' DMA operations may cause a cache coherency problem. If the DDIO technology is enabled, the received data is directly placed in the L3 cache, and the receiver's CPU guarantees cache

coherence. If the DDIO is disabled, there are two optional solutions to guarantee data consistency between the CPU cache and the PM. First, we can rely on the CPU's memory controller to guarantee the correct snoops/invalidates/atomics in the caching system. Intel's Optane PM also supports directory-based or snoop-based cache coherence protocols. Second, we can use a dedicated buffer to store the received data for each communicating pair. The receiver's CPU then copies the data from the dedicated buffer to the application's memory. In this case, the data consistency is also guaranteed by the receiver's CPU. We note that some HPC networking protocols achieve zero-copy using tag matching [1] [26]. Our `SFlush`-based RPC can also utilize these technologies to improve performance. However, we still use the buffering mechanism in our work because the redo logging mechanism not only guarantees cache coherence, but also guarantees failure atomicity in a PM system.

5 EVALUATION

In this section, we make an extensive comparison between previous RDMA-based RPCs and our failure recoverable RPCs. We evaluate the performance of these systems with micro-benchmarks and real-world applications. The source code and the experiments in this paper are available at Github [33].

5.1 Experimental Setup

We conduct our experiments using servers equipped with two-socket Intel Xeon Gold 6230 2.10 GHz 20-core processors, 128 GB DRAM, 1 TB Intel Optane DC Persistent Memory, and Mellanox ConnectX-4 40/56 GbE network controller. We use Intel Optane DCPMM in *App Direct mode* and manage it via *Direct Access* (DAX) [12]. We disable the DDIO by default in our experiments.

RPC Systems for Comparison. We implement the RPC communication models of L5 [11], RFP [40], FaSST [19], Octopus [25], FaRM [8], ScaleRPC [5], and DaRPC [39], and then compare our durable RPCs with them. We deploy each RPC service in a single server, and use one client on another server to evaluate the end-to-end latency of RPCs. Since the maximum transmission unit of the RDMA UD connection in FaSST is 4KB, we only show experimental results of FaSST for objects smaller than 4KB. For ScaleRPC, we interleave one *warm-up* phase with 100 *process* phases.

For durable RPCs in this paper, we call the RDMA `SFlush`-based RPC as *SFlush-RPC*, the RDMA `WFlush`-based RPC as *WFlush-RPC*, the RDMA `RFlush`-based RPC using RDMA `send` as *S-RFlush-RPC*, and the RDMA `RFlush`-based RPC using RDMA `write` as *W-RFlush-RPC* for simplicity. **We only compare *SFlush-RPC, S-RFlush-RPC* with DaRPC and FaSST because they all use RDMA `send` primitives. In contrast, we only compare *WFlush-RPC* and *W-RFlush-RPC* with L5, RFP, Octopus, FaRM, ScaleRPC because these RPCs all use RDMA `write` primitives.**

Micro-benchmarks. We develop micro-benchmarks [33] to evaluate previous RDMA-based RPCs and our durable RPCs. In our experiments, if not specified otherwise, the sender first generates 50K objects in a remote server, and then reads/writes these objects for 300K times via different RDMA-based RPCs. The default object size is 64 KB. The data access pattern follows a zipfian distribution with a read/write ratio of 1:1. We use 32 B, 1 KB, and 64 KB objects to evaluate the efficiency of our RPCs for typical message-passing systems, K-V stores, and file systems, respectively.

Macro-benchmarks. We use real-world applications such as compute-intensive PageRank [31] and latency-sensitive YCSB [53] benchmarks to evaluate the performance of different RPCs. For the PageRank algorithm, we use different graph datasets as follows. *Word association-2011* [50] contains 10K nodes and 72K edges. *Enron* [49] contains 69K nodes and 276K edges. *Dblp-2010* [48] contains 326K nodes and 1615K edges. We store the graph data in a remote server's PM, and store the intermediate results of PageRank in the main memory of the computation node locally. The graph data are fetched via RPCs by the client node. For YCSB benchmarks, we store 50K objects in a KV store system. The sizes of keys and values are 8 Bytes and 4 KB, respectively. Clients perform RPCs to access KV pairs in the remote PM, and maintain KV indexes in the main memory of clients locally. We perform KV operations 300K times in each test for different workloads. Workload A has 50%-50% update-read ratio. Workload B performs 95% reads and 5% updates (overwrites). Workload C is read-only. Workload D performs 5% inserts and 95% reads for the most recently inserted records. Workload E performs 95% scans (range queries) and 5% inserts. Workload F performs 50% reads and 50% write-modify-reads. The data access patterns of these workloads (except D) all follow a zipfian distribution (99% skewness).

5.2 Micro-benchmark Performance

Figure 8 shows the throughput of different RPCs in the micro-benchmark when the requested object sizes are 32 B, 1 KB, and 64 KB. Since real-world RPCs usually perform more complex operations than just accessing data, we emulate the execution of a real-world RPC at the receiver side by injecting an additional latency of 100 us for data processing, like DaRPC [39]. In this case, we deem that these RPCs lead to heavy load at the receiver side. Also, we assume that light-load RPCs only perform read/write operations.

For the light load case (Figure 8(b)), our durable RPCs achieve moderate performance improvement compared with other RPCs for small objects (32 B and 1 KB). For large objects (64 KB), the throughput of *WFlush-RPC* and *W-RFlush-RPC* is improved by 20%-90% compared with RPCs implemented with RDMA `write` primitives, such as L5, RFP, Octopus, FaRM, and ScaleRPC. Moreover, the throughput of *SFlush-RPC* and *S-RFlush-RPC* is improved by 42% compared with DaRPC which uses RDMA `send` primitives. For the heavy load case (Figure 8(a)), our durable RPCs all achieve the best performance than other RPCs. The throughput of *WFlush-RPC* and *W-RFlush-RPC* is improved by 58%-85% for all objects compared with other RPCs implemented with RDMA `write` primitives. *SFlush-RPC* and *S-RFlush-RPC* also improve the throughput by 43%-69% compared with DaRPC. The performance improvement is mainly attributed to decoupling the remote data persisting from costly RPC processing. Once the sender is aware of the completion of remote data persisting, it can issue other RPC requests earlier even if it has not received the ACK of the prior RPC. In this way, the RPC processing is partially overlapped with the RDMA transmission.

Figure 9 shows the tail latency of RPCs when accessing 1 KB and 64 KB objects in the micro-benchmark. *SFlush-RPC* and *S-RFlush-RPC* reduce the tail latency by about 10% compared with DaRPC

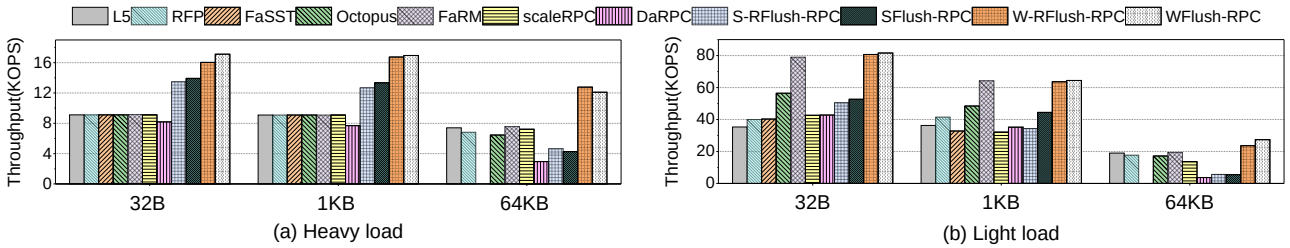

Figure 8: The throughput of different RPCs in micro-benchmarks

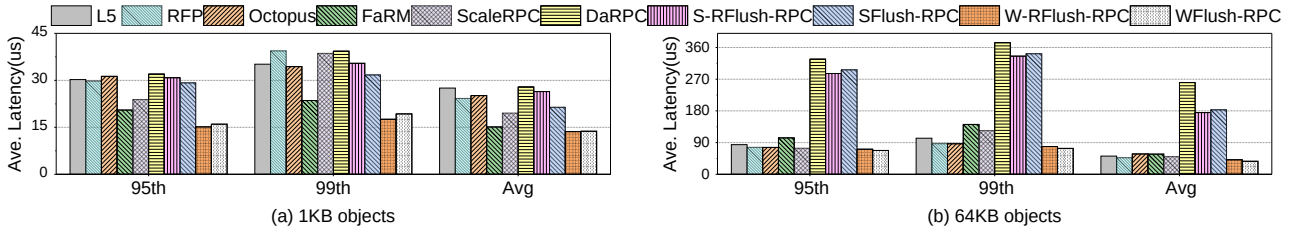

Figure 9: The tail and average latency of different RDMA-based RPCs

because the software overhead of two-side RDMA operations is the dominant factor of high latency. In contrast, compared with RPCs using RDMA `write` primitives, *W-RFlush-RPC* and *WFlush-RPC* reduce the 99th percentile latency by about 49% and 24% for 1 KB and 64 KB objects, respectively. Overall, these experimental results demonstrate that our durable RPCs can offer higher quality of service than previous RPCs for latency-sensitive applications.

Lessons learnt: *1) For small data packets, RPCs using either one-sided or two-sided RDMA primitives achieve similar application performance. However, for large data packets, RPCs using one-sided RDMA primitives (such as* `write`*) are much faster than RPCs using two-sided RDMA primitives (such as* `send`*). 2) Our durable RPCs using RDMA* `Flush` *primitives achieve notable performance improvement for both one-sided and two-sided RDMA primitives compared with previous RPCs. Because our durable RPCs are persistent and recoverable, they allow the remote data persistence visible to the sender much earlier than traditional RPCs. This offers an opportunity to overlap the data transferring with the RPC processing. 3) Our durable RPCs show similar performance whether they use sender-initiated or receiver-initiated RDMA* `Flush` *Primitives.*

5.3 Macro-benchmark Performance

Figure 10 shows the execution time of PageRank using three datasets for different RPC systems. Our RPCs always shows the best performance for all cases. Particularly, *S-RFlush-RPC* and *SFlush-RPC* can reduce the execution time by 8% and 30% compared with DaRPC, respectively. *W-RFlush-RPC* and *WFlush-RPC* can reduce the execution time by 8%-38% compared with other RPCs using RDMA `write` primitives. As Pagerank is a computation-intensive application, the high CPU load at the client side has a non-trivial impact on RDMA transmission. However, our durable RPCs still achieves substantial performance improvement.

Figure 11 shows the average latency of RPCs in different YCSB workloads. Since most YCSB workloads are read-intensive, such as workloads B, C, and D, our durable RPCs show moderate performance improvement compared with other RPCs of the same type.

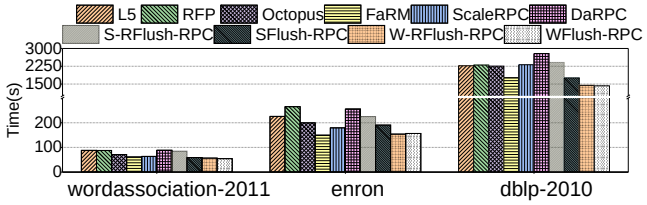

Figure 10: The performance of PageRank using different RPCs

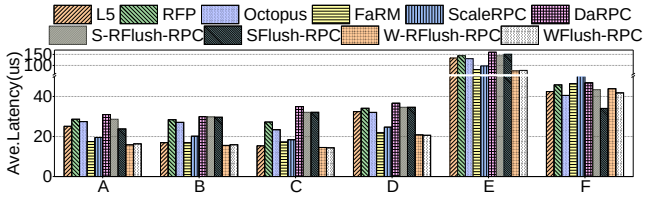

Figure 11: The average latency of RPCs in YCSB workloads

For workloads A and E, *W-RFlush-RPC* and *WFlush-RPC* reduce the average latency of RPCs by up to 50% compared with other RPCs using RDMA `write` primitives. *S-RFlush-RPC* and *SFlush-RPC* also reduce the latency of remote read/write operations by 7% and 23% compared with DaRPC. These results demonstrate that our RDMA `Flush`-based durable RPCs are beneficial for real-world write-intensive workloads.

5.4 Efficiency of Failure Recovery

We evaluate the efficiency of failure recovery for our durable RPCs in a microservice scenario. The RPC services are deployed in light-weight VMs using unikernels, which show extremely short startup latency (about 300 ms) [3]. We simulate unexpected failures for the unikernels with different probabilities of server availability [2] [44]. We set the re-transfer interval of RDMA packets to 100 ms [47]. We use workloads with different read/write ratios to perform RPCs for 10^9 times and measure the total execution time of different workloads. For traditional RPCs, the sender has to re-sent RPC requests

Figure 12: The total execution time of different workloads using our durable RPCs, all normalized to a traditional RPC system in which the sender needs to re-issue RPC requests upon a system failure

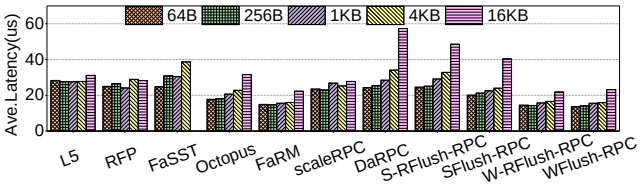

Figure 13: The latency of RPCs varies with object sizes

upon a system failure, and we refer those RPCs as the baseline. Figure 12 shows that our durable RPCs lead to much lower cost in failure recovery than traditional RPCs for all cases. Because our RPCs can decouple the data persisting from the RPC processing, uncompleted RPCs can be re-executed if the data is already persisted in the log buffer, without re-sending the data from clients. Thus, write-intensive workloads benefit more from our durable RPCs than read-intensive workloads. Moreover, our RPCs reduce more recovery latency when the rate of server availability is lower.

5.5 Sensitivity Studies

Object Sizes. Figure 13 shows the average latency of RPCs in the micro-benchmark using different object sizes. Our durable RPCs always achieve less latency than other RPCs of the same type. When the object size increases from 64 B to 4 KB, the micro-benchmark shows a slight growth of latency for all RPC systems. However, when the object size becomes larger than 4 KB, there is a significant growth of latency for all RPCs. This implies that the application performance is mainly determined by the software cost of RDMA communication when the data size is small, and the RDMA network round-trip time for data transferring becomes a dominant factor of application performance when data packets become larger than 4 KB. Moreover, RPCs using RDMA `send` primitives such as DaRPC are more sensitive to the object size. Thus, *we advocate RPCs using one-sided RDMA primitives if the application has to access different sizes of objects.*

Load of RDMA Networks. Figure 14 shows the average latency of RPCs in the micro-benchmark under different RDMA network loads. To simulate a high load of RDMA network between sender and receiver, we exploit a background program to contiguously send small data packets. When the network link is congested with data packets (busy), *S-RFlush-RPC* reduces the average latency by 45% compared with RPCs using RDMA `send` primitives such as DaRPC. *W-RFlush-RPC* reduces the average latency by 43% compared with other RPCs using RDMA `write` primitives, such as L5, RFP, Octopus, FaRM, ScaleRPC. RPCs using receiver-initiated RDMA `Flush` primitives can achieve higher performance

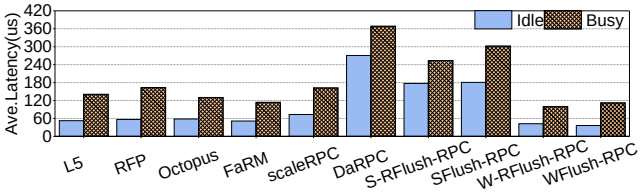

Figure 14: The impact of RDMA network load on the RPC latency

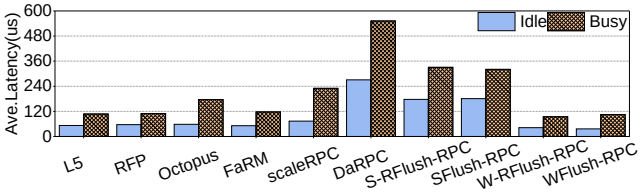

Figure 15: The impact of receivers' CPU load on the RPC latency

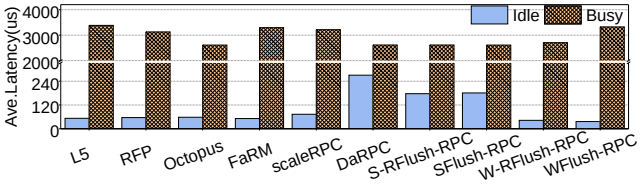

Figure 16: The impact of senders' CPU load on the RPC latency

because there are fewer RDMA primitives in the persistent data path than RPCs using sender-initiated RDMA `Flush` primitives, such as *SFlush-RPC* and *WFlush-RPC*. The receiver-initiated RDMA flushing mechanism can reduce the load of RDMA network because the RNIC at the receiver side issues RDMA `Flush` primitives itself. Moreover, RPCs using RDMA `write` primitives get more performance lose than RPCs using RDMA `send` primitives when the load of RDMA networks becomes high. *This implies RDMA* `writes` *are highly sensitive to the network load.*

Receivers' CPU Load. Figure 15 shows the impact of CPU load at the receiver side on the average latency of RPCs for the micro-benchmark. We increase the receivers' CPU load by running a computation-intensive program in the background. When the CPU load becomes high (busy), all RPCs show a significant increase of latency. RPCs using two-sided RDMA primitives such as DaRPC, *SFlush-RPC*, and *S-RFlush-RPC* show a larger growth of latency than other RPCs using one-sided RDMA primitives because they require more CPU resource for polling the message buffer. However, RPCs using one-sided RDMA primitives get higher performance slowdown relative to the idle case, such as Octopus and scaleRPC. *This implies the performance of RPCs using one-sided RDMA primitives is more sensitive to the receivers' CPU load.*

Senders' CPU Load. Figure 16 shows how the CPU load of the sender can effect the average latency of RPCs for the micro-benchmark. When the sender is under a high CPU load, the average latency of RPCs all increases significantly. *This implies that the performance of RPCs is highly sensitive to the sender's CPU load.*

The Number of Concurrent Senders. Figure 17 shows the average latency of RPCs for the micro-benchmark which uses multiple senders to communicate with a single receiver concurrently.

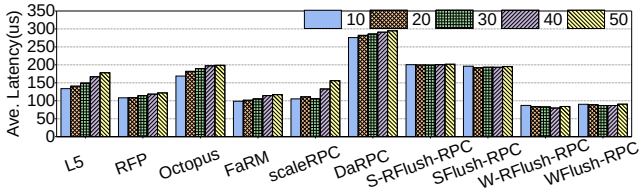

Figure 17: The impact of concurrent senders on the RPC latency

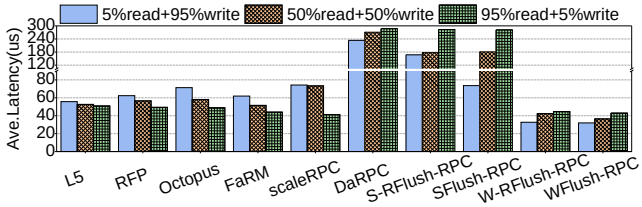

Figure 18: The impact of data access patterns on the RPC latency

Each sender requests objects for 30K times. We find that the average latency of previous RPCs increases sightly with the number of concurrent senders. In contrast, the average latency of our durable RPCs remains stable regardless of the number of concurrent senders. This implies that *our durable RPCs offer higher scalability than traditional RPCs*. The reason is that our RPCs require less remote CPU intervention for RDMA network transmission.

Data Access Patterns. Figure 18 shows the average latency of RPCs for the micro-benchmark with different read/write ratios. For the read-intensive workload (95% read + 5% write), our durable RPCs achieve similar performance to other RPCs because RDMA read operations are on the critical path of applications' execution, and RDMA `Flush` primitives are only needed for a small portion of RDMA write operations. For the write-intensive workload (5% read + 95% write), *W-RFlush-RPC* and *WFlush-RPC* can reduce the average latency by 63%-71% compared with other RPCs using RDMA `write` primitives. Compared to DaRPC, *SFlush-RPC* and *S-RFlush-RPC* even reduce the latency by 1.3 and 3.1 times, respectively. *These results demonstrate that our durable RPCs can achieve significant performance improvement relative to other RPCs for write-intensive workloads by overlapping the data transmission and the RPC processing.*

Batch Sizes. Figure 19 shows the total execution time of micro-benchmark when multiple RDMA requests are batched into one RPC. The batching mechanism proposed in DaRPC and ScaleRPC can significantly reduce the software overhead of RDMA transmission. However, for RPCs using RDMA `send` primitives such as DaRPC, we find that the performance improvement is not that significant when the batch size increases because the software cost of RDMA `send` primitives is sensitive to the data size. In contrast, the execution time of *W-RFlush-RPC* and *WFlush-RPC* is significantly reduced when the batch size increases. *This implies that RDMA flushing integrated with batching can significantly improve the performance of RPCs using RDMA* `write` *primitives.*

5.6 Software Overhead

Figure 20 shows the breakdown of different RPC latencies for YCSB workload A. The total latency includes software overhead at the

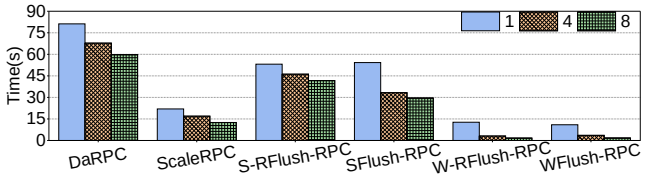

Figure 19: The impact of batch sizes on the RPC performance

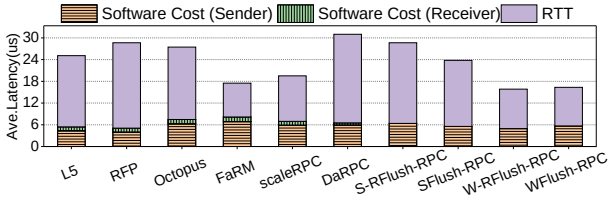

Figure 20: The hardware/software overhead using different RPCs

Table 2: Summary of RPCs using different RDMA Flush primitives

	SRFlush	SFlush	WRFlush	WFlush	Other RPCs
Network Load Sensitivity	High	Medium	High	Medium	Medium
Receiver CPU Requirement	Medium		Low		High
QoS	High				Medium
Tail Latency	Low				Medium
Scalability	Good				Medium
Data Persistence	Proactive, Decoupled with RPCs				Passive
Application Scenarios	Messages, KVs, Objects, Files				Small Messages

sender side, network RTTs (hardware), and software overhead at the receiver side (including RPC processing cost and data persisting cost). We can find that the RPC latency is dominated by the network RTTs. For different RPC designs, the difference of hardware cost is rather significant. For example, the RTTs of DaRPC are almost twice higher than that of FaRM. The software costs at the sender side are almost the same for all RPCs. Our PRCs using RDMA `Flush` primitives can hide the RPC processing time by overlapping it with RDMA transmission. Overall, the total software overhead of our RPCs is no more than 7%.

5.7 Summary

Based on our experimental results, we summarize the features of different RPCs, as shown in Table 2. Our durable RPCs can decouple the data persisting from the RPC processing, and thus provide opportunities to overlap the RDMA transmission and the costly RPC processing. In this way, our durable RPCs can significantly improve the RPC performance compared with traditional RPCs, particularly for write-intensive applications. We also find that RPCs implemented with sender-initiated RDMA `Flush` primitives have a greater impact on the RDMA network load because the sender has to send an additional RDMA `Flush` primitive over the network. As a result, we advocate receiver-initiated RDMA `Flush` primitives for the scenario of high RDMA network load. Moreover, we advocate one-sided RDMA primitives rather than two-sided ones for higher throughput and lower latency.

6 RELATED WORK

We describe the related work in software/hardware categories.

Software Approaches to Remote Persistence. A few proposals such as Intel's Appliance Remote Replication [36] and Erda [24] exploit the read-after-write mechanism [29] to ensure remote data persistence without involving the remote server's CPU. Although these approaches do not involve the remote CPU, the sender needs to issue additional RDMA read operations for remote data persisting, and thus increases the load of RDMA networks. Moreover, the effectiveness of these approaches is highly influenced by the DDIO hardware feature. A few proposals rely on remote CPUs to guarantee remote data persistence. Hotpot [38] exploits the the data servers' CPUs to conduct a multi-phase commit protocol. Orion [51] uses RDMA `write-IMM` primitives to notify the memory address of written data to the remote server's CPU. Consequently, the CPU stores the data to PM. Intel also proposes another remote data persisting mechanism called *General-purpose Remote Replication* [27], which needs the client to issue an RDMA `send` verb to the remote server after a set of RDMA `write` operations. The remote server's CPU has to notify the completion of data persisting to the client. These approaches all get the remote CPU involved in data persisting, and thus increase latency on the critical path of data transmission, and also offset the performance benefit of one-sided RDMA communication. In contrast, our proposals exploits RNIC hardware functionalities to flush data from the RNIC cache to the PM actively, and guarantees remote data persistence without involving the remote server's CPU.

RPCs de facto provide another way to guarantee remote data persistence. There have been a number of RPC systems designed for distributed PM, such as DaRPC [39], FaRM [8], Herd [18], FaSST [19], L5 [11], RFP [40], ScaleRPC [5], Octopus [25], and LITE [46]. They actually support durable RDMA operations. Kalia *et. al.* analyze the challenges of remote data persistence introduced by RNIC and DDIO technologies [17], and propose state machine replication using optimized eRPCs [20] for durable RDMA operations. However, these RPCs tightly couple the remote data persisting with the costly RPC processing, and thus postpone the visibility of remote data persistence to client applications. In contrast, our work allows the remote data persistence visible earlier by decoupling data persisting and RPC processing. Thus, our design can achieve higher performance than previous RPCs when the latency of RPC processing is high.

Hardware Approaches to Remote Persistence. A number of studies have discussed the data persistence problem in RDMA-based PM systems. They expect that the future RDMA network device can provide flushing operations to guarantee data persistence, for example, novel I/O flows for remote durability [25], and new RDMA primitives like `RDMA Commit`. Tailwind [42] argues that hardware and firmware modifications and non-portable solutions are required to guarantee remote data persistence. Tsai *et. al.* [45] believe that the durable RDMA write commit takes one network round trip [9], and thus use RDMA `write` to emulate the performance of durable RDMA write directly.

Because DDIO technologies [6] change the data path of RDMA data transmission, some studies discuss its impact on the remote data persistence. Yang [52] *et. al.* deem that DDIO makes remote data persistence more costly and difficult. Hu [14] *et.al.*, exploit advanced network controller and self-developed memory controller to send the ACK of data persisting to the client's RNIC, and thus eliminate the impact of DDIO on the remote data persistence. However, they do not provide details about the advanced network controller.

Recently, there have been some discussions on hardware-assisted RDMA primitives for durable RDMA transmission. Tavakkol [43] *et. al.* analyze multiple possible data paths when data is evicted from the volatile cache of the remote RNIC, and propose a number of RDMA primitives to improve the efficiency of persistent memory replication. However, they just evaluate the proposed primitives through simulation which may not accurately reflect the RDMA performance in a real testbed. Kashyap [21] *et. al.* analyze the impact of different system configurations on the performance of remote data persisting, including the range of persistent domain, DDIO technologies, and the durability of RDMA QPs. Their results show that the correct and fast remote persisting solutions differ widely depending on the system configurations at the server side, compelling programmers to make trade-offs among different design goals. IBTA [10] also defines the InfiniBand specification for RDMA `Flush` primitives, including their functionalities, packet formats, ordering rules, memory region selectivity levels, and so on. However, this proposal only specifies the design of sender-initiated RDMA `Flush` primitives. In contrast, we further propose another optional design, i.e., receiver-initiated RDMA `Flush` primitives. Although receiver-initiated primitives may require an extra RTT to notify the completion of an RDMA `Flush` operation, the latency is not on the critical path of applications. Thus, the network costs of sender-initiated and receiver-initiated primitives are similar.

Inspired by the above studies, we further propose several potential implementations of RDMA `Flush` primitives, and design durable and recoverable RPCs based on the RDMA `Flush` primitives. Our work is complementary to previous proposals for remote data persistence. Moreover, through extensive experimental studies, we provide instructive guidelines to support remote persistence in different scenarios.

7 CONCLUSION

In this paper, we made an extensive study of existing RDMA-based RPCs. Based on the lessons learnt, we designed several hardware-supported RDMA `Flush` primitives to guarantee the persistence of RDMA update operations. We also proposed durable RPCs based on the new RDMA `Flush` primitives for fast failure recovery. We compared our proposals with several state-of-the-art RPCs in a real testbed equipped with Intel Optane DCPMM and InfiniBand networks. Experimental results show that the proposed RDMA `Flush` primitives and the corresponding RPCs can significantly improve the throughput and latency of durable RDMA transmission.

ACKNOWLEDGMENTS

We appreciate the anonymous reviewers' insightful comments for improving this paper. This work is supported by National Key Research and Development Program of China under grant No. 2017YFB1001603, and National Natural Science Foundation of China under grants No.62072198, 61732010, 61825202, 62032008.

REFERENCES

[1] Mohammadreza Bayatpour, Seyedeh Mahdieh Ghazimirsaeed, Shulei Xu, Hari Subramoni, and Dhabaleswar K. Panda. 2020. Design and Characterization of InfiniBand Hardware Tag Matching in MPI. In *Proceedings of the 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computin (CCGRID' 20)*. 101–110.

[2] Antonia Bertolino, Guglielmo De Angelis, Micael Gallego, Boni García, Francisco Gortázar, Francesca Lonetti, and Eda Marchetti. 2019. A Systematic Review on Cloud Testing. *ACM Comput. Surv.* 52, 5, Article 93 (Sept. 2019), 42 pages.

[3] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E. Engelstad, and Kyrre M. Begnum. 2015. IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services. In *Proceedings of the 7th IEEE International Conference on Cloud Computing Technology and Science (CloudCom' 15)*. 250–257.

[4] Xianzhang Chen, Edwin Hsing-Mean Sha, Ahmad Abdullah, Qingfeng Zhuge, Lin Wu, Chaoshu Yang, and Weiwen Jiang. 2017. UDORN: A Design Framework of Persistent In-Memory Key-value Database for NVM. In *Proceedings of the IEEE 6th Non-Volatile Memory Systems and Applications Symposium (NVMSA' 17)*. 1–6.

[5] Youmin Chen, Youyou Lu, and Jiwu Shu. 2019. Scalable RDMA RPC on Reliable Connection with Efficient Resource Sharing. In *Proceedings of the 14th ACM European Conference on Computer Systems (EuroSys' 19)*. 19:1–19:14.

[6] Intel Data Direct I/O Technology (Intel DDIO). 2012. https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf.

[7] Kangping Dong, Linpeng Huang, and Yanmin Zhu. 2017. Exploiting RDMA for Distributed Low-Latency Key/Value Store on Non-volatile Main Memory. In *Proceedings of the 23rd IEEE International Conference on Parallel and Distributed Systems (ICPADS' 17)*. 225–231.

[8] Aleksandar Dragojevi, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI' 14)*. 401–414.

[9] RDMA durable write commit. 2016. https://tools.ietf.org/html/draft-talpey-rdma-commit-00.

[10] Memory Placement Extensions. 2020. https://cw.infinibandta.org/document/dl/8594.

[11] Philipp Fent, Alexander van Renen, Andreas Kipf, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2020. Low-Latency Communication for Fast DBMS Using RDMA and Shared Memory. In *Proceedings of the 36th IEEE International Conference on Data Engineering (ICDE' 20)*. 1477–1488.

[12] Utility Library for Managing the libnvdimm (non-volatile memory device) Subsystem in the Linux Kernel. 2020. https://github.com/pmem/ndctl.

[13] RDMA Extensions for Remote Persistent Memory Access. 2016. https://openfabrics.org/images/eventpresos/2016presentations/215RDMAforRemPerMem.pdf.

[14] Xing Hu, Matheus Ogleari, Jishen Zhao, Shuangchen Li, Abanti Basak, and Yuan Xie. 2018. Persistence Parallelism Optimization: A Holistic Approach from Memory Bus to RDMA Network. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO' 18)*. 494–506.

[15] Nusrat Sharmin Islam, Md. Wasi-ur-Rahman, Xiaoyi Lu, and Dhabaleswar K. Panda. 2016. High Performance Design for HDFS with Byte-addressability of NVM and RDMA. In *Proceedings of the 30th ACM International Conference on Supercomputing (ICS' 16)*. 8:1–8:14.

[16] Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md. Wasi-ur-Rahman, Nusrat S. Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, and Dhabaleswar K. Panda. 2011. Memcached Design on High Performance RDMA Capable Interconnects. In *Proceedings of International Conference on Parallel Processing (ICPP' 11)*. 743–752.

[17] Anuj Kalia, David G. Andersen, and Michael Kaminsky. 2020. Challenges and solutions for fast remote persistent memory access. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC' 20)*. 169–182.

[18] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 Conference of ACM Special Interest Group on Data Communication (SIGCOMM' 14)*. 295–306.

[19] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (OSDI' 16)*. 185–201.

[20] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2019. Datacenter RPCs can be General and Fast. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI' 19)*. 1–16.

[21] Sanidhya Kashyap, Dai Qin, Steve Byan, Virendra J. Marathe, and Sanketh Nalli. 2019. Correct, Fast Remote Persistence. *arXiv preprint arXiv:1909.02092* (2019).

[22] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. 2018. Hyperloop: Group-based NIC-offloading to Accelerate Replicated Transactions in Multi-tenant Storage Systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM' 18)*. 297–312.

[23] Intel Optane DIMM latency. 2019. https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html.

[24] Xinxin Liu, Yu Hua, Xuan Li, and Qifan Liu. 2019. Write-Optimized and Consistent RDMA-based NVM Systems. *arXiv preprint arXiv:1906.08173* (2019).

[25] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: An RDMA-enabled Distributed Persistent Memory File System. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC' 17)*. 773–785.

[26] W. Pepper Marts, Matthew G. F. Dosanjh, Whit Schonbein, Ryan E. Grant, and Patrick G. Bridges. 2019. MPI tag matching performance on ConnectX and ARM. In *Proceedings of the 26th European MPI Users' Group Meeting (EuroMPI' 19)*. 13:1–13:10.

[27] Persistent Memory Replication Over Traditional RDMA Part 1: Understanding Remote Persistent Memory. 2020. https://software.intel.com/content/www/us/en/develop/articles/persistent-memory-replication-over-traditional-rdma-part-1-understanding-remote-persistent.html.

[28] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build A Fast, CPU-Efficient Key-Value Store. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC' 13)*. 103–114.

[29] NVM Programming Model (NPM). 2020. https://www.snia.org/tech_activities/standards/curr_standards/npm.

[30] DPDK: Hardware-Level Performance Analysis of Platform I/O. 2018. https://www.dpdk.org/wp-content/uploads/sites/35/2018/09/Roman-Sudarikov-DPDK_PRC_Summit_Sudarikov.pptx.

[31] PageRank. 2020. http://pr.efactory.de/.

[32] Ivy Bo Peng, Maya B. Gokhale, and Eric W. Green. 2019. System evaluation of the Intel optane byte-addressable NVM. In *Proceedings of the International Symposium on Memory Systems (MEMSYS' 19)*. 304–315.

[33] PRDMA. 2021. https://github.com/CGCL-codes/PRDMA.

[34] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable High Performance Main Memory System Using Phase-change Memory Technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA' 09)*. 24–33.

[35] Software mechanisms for enabling access to remote persistent memory RDMA with PMEM. 2015. http://www.snia.org/sites/default/files/SDC15_presentations/persistant_mem/ChetDouglas_RDMA_with_PM.pdf.

[36] Remote Persistent Memory Access (RPMA). 2021. https://pmem.io/rpma/manpages/master/librpma.7.html.

[37] Timo Schneider, James Dinan, Mario Flajslik, Keith D. Underwood, and Torsten Hoefler. 2017. Fast Networks and Slow Memories: A Mechanism for Mitigating Bandwidth Mismatches. In *Proceedings of the 25th IEEE Annual Symposium on High-Performance Interconnects (HOTI' 17)*. 17–24.

[38] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed Shared Persistent Memory. In *Proceedings of the 8th ACM Symposium on Cloud Computing (SoCC' 17)*. 323–337.

[39] Patrick Stuedi, Animesh Trivedi, Bernard Metzler, and Jonas Pfefferle. 2014. DaRPC: Data Center RPC. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC' 14)*. 1–13.

[40] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. 2017. RFP: When RPC is Faster than Server-Bypass with RDMA. In *Proceedings of the 12th ACM European Conference on Computer Systems (EuroSys' 17)*. 1–15.

[41] Kosuke Suzuki and Steven Swanson. 2015. The Non-Volatile Memory Technology Database (NVMDB). *UCSD-CSE Techreport CS 2015-1011* (2015).

[42] Yacine Taleb, Ryan Stutsman, Gabriel Antoniu, and Toni Cortes. 2018. Tailwind: Fast and Atomic RDMA-based Replication. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC' 18)*. 851–863.

[43] Arash Tavakkol, Aasheesh Kolli, Stanko Novakovic, Kaveh Razavi, Juan Gómez-Luna, Hasan Hassan, Claude Barthels, Yaohua Wang, Mohammad Sadrosadati, Saugata Ghose, Ankit Singla, Pratap Subrahmanyam, and Onur Mutlu. 2018. Enabling Efficient RDMA-based Synchronous Mirroring of Persistent Memory Transactions. *arXiv preprint arXiv:1810.09360* (2018).

[44] Maria Toeroe, Neha Pawar, and Ferhat Khendek. 2014. Managing application level elasticity and availability. In *Proceedings of the 10th International Conference on Network and Service Management (CNSM' 14)*. 348–351.

[45] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC' 20)*. 33–48.

[46] Shin-Yeh Tsai and Yiying Zhang. 2017. Lite Kernel RDMA Support for Datacenter Applications. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP' 17)*. 306–324.

[47] Yi Wang, Kexin Liu, Chen Tian, Bo Bai, and Gong Zhang. 2019. Error Recovery of RDMA Packets in Data Center Networks. In *Proceedings of the 28th International Conference on Computer Communication and Networks (ICCCN' 19)*. 1–8.

[48] The webdata of dblp. 2010. http://law.di.unimi.it/webdata/dblp-2010/.

[49] The webdata of enron. 2020. http://law.di.unimi.it/webdata/enron/.

[50] The webdata of wordassociation. 2011. http://law.di.unimi.it/webdata/wordassociation-2011/.

[51] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2019. Orion: A Distributed File System for Non-volatile Main Memory and RDMA-capable Networks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST' 19)*. 221–234.

[52] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2020. FileMR: Rethinking RDMA Networking for Scalable Persistent Memory. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI' 20)*. 111–125.

[53] YCSB. 2020. https://ycsb.site.

[54] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS' 15)*. 3–18.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We first implement the RPC communication models of L5, RFP, FaSST, Octopus, FaRM, ScaleRPC, and DaRPC, and then compare our durable RPCs with them. We evaluate these RPCs with our self-developed micro-benchmarks and real-world applications. In the micro-benchmarks, the client first generates 50K objects in the remote server, and then reads/writes these objects for 300K times via different RPCs. The data access pattern follows a zipfian distribution with a read/write ratio of 1:1, as described in the paper. The real-world workloads include compute-intensive PageRank and latency-sensitive YCSB benchmarks. More details about the workload configuration are described in the paper.

More detailed instructions on code compiling, running, and testing are elaborated in the following.

1. Compiling

To run pRDMA, the entire project needs to be compiled. The project requires some hardware and software supports and configurations, such as the network setup and NVM use modules.

1.1 Configurations

First, the `config.xml` need to be configured according to the run-time environment.

```
[server/client node] cd bin
[server/client node] vim config.xml
```

In the `config.xml`, the following information should be configured on the client and server sides.

```
<server>
    <nic_name>ib0</nic_name>
    //RDMA Card Name by ifconfig command
    <addr>xxx.xxx.xxx.xxx</addr>
    //Server's RDMA Card Address
    <port>39300</port>
    //Server's listen port
</server>
```

1.2 Compiling

The following commands are used to compile the entire project:

```
[server/client node] mkdir build && cd build
[server/client node] cmake .. && make
```

The executable server and client programs are generated in the folder *bin/*, and the server and client projects are packaged separately for convenience.

2. Running

The executable files can be found in the directories *pRDMA_client/bin* and *pRDMA_server/bin*. Like other RPC systems, the pRDMA server should start at first to serve the client's requests.

```
[server node] cd bin
[server node] ./server
```

Then, a client can start and issue RPC requests. To run micro-benchmarks, the commands at the client are listed as follows:

```
[client node] cd bin
[client node] ./client [Object Size in Byte] [Number of Accesses]
```

For example, the following show the commands to request objects of 64 Byte and access them 500000 times.

```
[client node] ./client 64 500000
```

3. Performance Testing

3.1 Macro-benchmark

pRDMA can run PageRank and YCSB as Macro-benchmarks to evaluate the performance of pRDMA. Since PageRank and YCSB need to be executed with a database, pRDMA extends the native PageRank and YCSB benchmarks to execute them in a RDMA-based distributed persist memory system.

3.1.1 PageRank

pRDMA runs PageRank as compute-intensive application. pRDMA places the graph data in the remote node with NVM and places the generated intermediate data in the client locally.

For the PageRank algorithm, pRDMA uses different graph datasets as follows. Word association-2011 contains 10K nodes and 72K edges. Enron contains 69K nodes and 276K edges. Dblp-2010 contains 326K nodes and 1615K edges.

The command to run PageRank:

```
[client node] ./dblp //Dblp-2010 Dataset
[client node] ./enron //Enron Dataset
[client node] ./word //Word Association-2011 Dataset
```

3.1.2 YCSB

pRDMA runs YCSB as a latency-sensitive application, and uses time33 algorithm as the hash function. We extend YCSB-C, a branch to implement RPC communication interfaces.

The following commands are used to run different workloads of YCSB:

[client node] ./ycsb [workload A-F] [Object Size in Byte] [Number of Accesses]

For example, the following command performs 300000 accesses to objects of 256 Byte for workload A:

[client node] ./ycsb A 256 300000

3.2 Micro-benchmark

The experiments of pRDMA for micro-benchmark show the characteristics of diffident RPCs under different runtime setups and RDMA communication models.

3.2.1 Multiple RPC transmission Models

pRDMA implements several RPC transmission models of state-of-the-art RPC systems for comparison. To test other transmission models, some macro definitions in file *include/dhmp.h* need to be specified. The following list shows the macro definitions and the corresponding RPC models.

(1) "#define L5_MODEL" corresponds to L5 model
(2) "#define RFP_MODEL" corresponds to RFP model
(3) "#define OCTOPUS_MODEL" corresponds to octopus model
(4) "#define SCALERPC_MODEL" corresponds to scaleRPC model
(5) "#define DARPC_MODEL" corresponds to DaRPC model
(6) "#define SRFLUSHRPC_MODEL" corresponds to SRFlush-RPC model
(7) "#define SFLUSHRPC_MODEL" corresponds to SFlush-RPC model
(8) "#define WRFLUSHRPC_MODEL" corresponds to WRFlush-RPC model
(9) "#define WFLUSHRPC_MODEL" corresponds to WFlush-RP model

3.2.2 Load of RDMA Networks

To evaluate the performance of RPC models under different RDMA network load, the following macro definitions in *pRDMA_client/include/dhmp.h* need to be set.

(1) "#define RDMA_STRESS" is used to configure the RDMA network load
(2) "#define STRESS_NUM 5" can adjust the level of RDMA network load

3.2.3 CPU Load

To evaluate the impact of CPU load on performance of RPC models, user needs to run two programs, one is used to generate the background CPU load, and the other runs the RPC client.

For example, the following commands are used to evaluate the performance of RPC with high CPU load at the client node:

[client node, window 1] numactl –physcpubind=1 ./cpu_client
[client node, window 2] numactl –physcpubind=1 ./client 64 500000

3.2.4 The Number of Concurrent Senders

To evaluate the performance of RPC models which use multiple senders to communicate with a single receiver concurrently, the *pRDMA_Client/bin/Nclient.c* file should be used after *pRDMA_Client/bin/client* has been generated.

In order to evaluate the capability of concurrent accessing to a single server, the following commands are executed:

[client node] vim Nclient.c

```
<Nclient.c>
    <#define NUM 10>
    //Modify this to adjust the number of concurrent clients
    <execl("./client","./client","65536","5",NULL);>
    //Modify this to modify execution commands at the client
<Nclient.c>
```

[client node] gcc -o NUM_client Nclient.c
[client node] ./NUM_client

Author-Created or Modified Artifacts:

```
Persistent ID: https://doi.org/10.5281/zenodo.5162688
Artifact name: pRDMA
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Intel Xeon Gold 6230, Intel Optane DC Persistent Memory, and Mellanox ConnectX-4 40/56 GbE Network Controller

Operating systems and versions: CentOS 7.9.2009 running Linux kernel 5.1.1

Compilers and versions: Compile with cmake version 3.18.1 and gcc version 4.8.5

Applications and versions: PageRank, YCSB-C (v0.17.0)

Libraries and versions: ndctl v65, daxctl v65, MLNX_OFED_LINUX-4.7-1.0.0.1-rhel7.6

Key algorithms: PageRank

Input datasets and versions: word association-2011, enron, and dblp-2010