High-Performance Remote Data Persisting for Key–Value Stores via Persistent Memory Region

Yongping Luo, Peiquan Jin, *Member, IEEE*, Xiaoliang Wang, *Member, IEEE*, Zhaole Chu, Kuankuan Guo, Peng Xu, Jinhui Guo, and Fei Liu

***Abstract*—Key–value stores (KVStores), such as LevelDB and Redis, have been widely used in real-world production environments. To guarantee data durability and availability, traditional KVStores suffer from high write latency, mainly caused by the long network and data-persisting time. To solve this problem, this article presents a novel data-persisting path for KVStores, allowing remote clients to persist data to the KVStore server with μs-level latency. The novelty of this study is threefold. First, we propose PMRDirect, which utilizes a persistent memory region (PMR) in the NVM express standard to construct a direct data-persisting path from the RDMA networking card (NIC) to the PMR region inside an SSD. Second, to showcase PMRDirect in KVStores, we developed a new accessing stack called PMRAccess, enabling remote clients to access existing KVStores and providing durability for each write request. Specifically, we present a low-latency RDMA-based messaging mode and a chunk-based PMR management in PMRAccess to reduce write latency and improve system throughput. Finally, we conducted extensive experiments to evaluate the performance of our proposals. We first compared PMRDirect with a few remote data-persisting paths to show its effectiveness. Then, we evaluated PMRAccess upon two KVStores, including LibCuckoo (an in-memory KVStore) and LevelDB (an in-storage KVStore). The results showed that PMRAccess outperformed the SSD-based accessing stack by up to 6.1× in write throughput and 36× in write tail latency, and it achieved 1.7× higher write throughput and 0.59× lower write tail latency over the PMEM-based accessing stack. Further, we conducted a system-to-system comparison between the PMRAccess-integrated LibCuckoo and Redis, and the results showed our proposal achieved up to 13× higher throughputs and 40× lower write latency than Redis.**

***Index Terms*—Key–value stores (KVStores), persistent memory region (PMR), RDMA, write latency.**

I. Introduction

KEY–VALUE stores (KVStores) have been extensively used to store and manage unstructured data generated from a wide range of data-intensive applications. Large Internet companies like Google, Meta, and ByteDance all employ KVStores at scale. In these KVStores, new data or log entries from clients to the KVStore server are persisted into storage, and the associated write latency mainly includes networking and data-persisting latency. Under traditional TCP/IP-based accessing stacks and SSD-based storage, the write latency per request is at least hundreds of microseconds [1]. For latency-sensitive scenarios such as financial transaction services, it is necessary to reduce the write latency for higher performance.

Recently, some research and industry practices introduced RDMA networking card (NIC) to improve the networking latency [2], [3]. Commodity RDMA NICs can deliver ∼3-μs networking latency and up to 200 GbE networking bandwidth. Because RDMA only supports accessing main memory, RDMA-based KVStores can only store all data in memory and, therefore, do not support data durability. In addition, storing too much data in memory will increase the DRAM cost (about $5.1 per GB). Some other researchers proposed to use PMEM to reduce persistent latency as PMEM offers ∼120-ns persistent latency [4]. Combined with RDMA NICs, PMEM-based KVStores [5], [6] can reduce the write latency to μs-level while providing data durability. However, commercial PMEM products such as Optane PMEM (about $4.2 per GB) are much more expensive than NVMe SSDs (about $0.2 per GB) [7] and rely on specific costly CPUs. When users deploy PMEM-based KVStores at scale, the total cost is usually unacceptable for many companies. On the other hand, Intel has announced that the original Optane production line is closed, meaning that PMEM-based KVStores will become impractical because of the shortage of PMEM.

In this article, we propose to leverage an overlooked feature in the NVMe standard called persistent memory region (PMR) [10] to reduce the latency of write requests from remote clients. PMR is a PMR inside NVMe SSDs. Many commodity NVMe SSDs, such as Starblaze SSD [11] and DapuStor Haishen5 [12], have already supported the PMR technology, e.g., by adding power protection for its control memory buffer (CMB) at low cost. Unfortunately, PMR is not typically exposed and manipulated from userspace with portable interfaces. Although a few works exploited CPU's functions for accessing PMR [13], [14], no study proposed RDMA-based access to PMR. Unlike existing studies, this article proposes to build a new data-persisting path from RDMA NICs to PMR directly and develop a low-latency accessing stack for KVStores to ensure data durability.

Manuscript received 30 July 2024; accepted 30 July 2024. This work was supported in part by the National Science Foundation of China under Grant 62072419, and in part by ByteDance Inc. This article was presented at the International Conference on Hardware/Software Codesign and System Synthesis (CODES + ISSS) 2024 and appeared as part of the ESWEEK-TCAD Special Issue. This article was recommended by Associate Editor S. Dailey. *(Corresponding authors: Peiquan Jin; Fei Liu.)*

Yongping Luo, Peiquan Jin, Xiaoliang Wang, and Zhaole Chu are with the School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China (e-mail: ypluo@mail.ustc.edu.cn; jpq@ustc.edu.cn; wxl147@mail.ustc.edu.cn; czle@mail.ustc.edu.cn).

Kuankuan Guo, Peng Xu, Jinhui Guo, and Fei Liu are with the Applied Research Center (Storage), ByteDance Inc., Beijing 100800, China (e-mail: guokuankuan@bytedance.com; peng.xu@bytedance.com; guojinhui.liam@bytedance.com; fei.liu@bytedance.com).

Digital Object Identifier 10.1109/TCAD.2024.3442992

1937-4151 © 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

85 On the other hand, it is not trivial to establish a fast data-
86 persisting path from RDMA to PMR. The main challenges
87 are: 1) no existing mechanism supports accessing PMR from
88 local and remote CPUs simultaneously; 2) it lacks a high-
89 performance PMR-based accessing stack for KVStores, which
90 can ensure low latency, high throughput, and data durability
91 when serving multiple clients simultaneously; and 3) PMR
92 is capacity-constrained in commodity NVMe SSDs, and an
93 efficient management scheme for PMR is needed.

94 To address the above challenges, we modify the Linux
95 NVMe driver to expose PMR to the kernel space and
96 userspace. Users can access PMR from remote CPUs through
97 RDMA *Write* verbs and local CPUs through memory mapped
98 I/O (MMIO). To the best of our knowledge, this is the first
99 effort to build a direct high-performance data-persisting path
100 from RDMA NICs to the PMR region inside NVMe SSDs.
101 Briefly, we make the following contributions in this article.

102 1) We develop a new high-performance data-persisting path
103 called PMRDirect for remote clients to directly write
104 data from an RDMA NIC to the PMR region inside an
105 NVMe SSD. We demonstrate that such a data-persisting
106 path can offer much lower latency than existing data
107 paths (Section III).

108 2) We present a new accessing stack called PMRAccess
109 for client/server scenarios that utilizes PMRDirect.
110 PMRAccess provides low latency and data durability for
111 write requests from remote clients to the KVStore server.
112 Specifically, we implemented an RDMA-based messag-
113 ing mode and a new chunk-based PMR management
114 scheme in PMRAccess (Section IV).

115 3) We conduct extensive experiments with real RDMA
116 NICs and a PMR-enabled NVMe SSD to evalu-
117 ate the performance of PMRDirect and PMRAccess.
118 PMRAccess offers 0.59×–36× lower latency and
119 1.7×–6.8× higher write throughput than alterna-
120 tive solutions, including SSD-based and PMEM-based
121 accessing stacks. We further conduct a system-to-
122 system comparison between the PMRAccess-integrated
123 KVStore and Redis, and the results also suggest the
124 efficiency of our proposal (Section V).

125 II. Background and Motivation

126 A. KVStores and Its Write Latency

127 Traditional TCP/IP-based KVStores can offer 20–60-μs
128 access latency between a client and a KVStore server [15]. The
129 high networking latency of TCP/IP-based KVStore accessing
130 stacks is owing to three aspects: 1) TCP/IP-based networking
131 stacks rely on the CPU to perform networking computation;
132 2) KVStores may require several networking round trips to
133 finish one request; and 3) the capability of traditional Ethernet
134 cards is limited.

135 The latency of persisting a write to storage depends mainly
136 on devices; the latency value is several milliseconds for HDDs,
137 ∼100 μs for SATA SSDs, and 10–60 μs for NVMe SSDs.
138 Except for device type, the persistent latency differs slightly
139 according to how we achieve durability. Users can invoke
140 `fsync` or `fdatasync` system calls to flush data to storage
141 after the *write* system call that moves data to the page cache.

TABLE I
Different Accessing Technologies for KVStores Regarding Durability, Tail Latency (P90), and Storage Cost. The Data Is From the Experimental Results in Section V-C and Previous Literature [7]

Technology	Durability	Tail Latency (P90)	Storage Cost
Traditional [8]	✔	74.37 μs	$0.2 per GB
RDMA+DRAM [2], [3], [9]	✖	2.75 μs	$5.1 per GB
RDMA+PMEM [5], [6]	✔	6.55 μs	$4.2 per GB
PMRAccess (Our proposal)	✔	2.75 μs	$0.2 per GB

Users can also write data to storage directly by `DIRECT_IO` 142
in POSIX interfaces. `DIRECT_IO` incurs less data copy which 143
makes it faster than `fsync`, at the cost of page alignment 144
constraint. `fdatasync` reduces the persistent cost of meta- 145
data in the file system and offers lower latency than `fsync`. 146

B. RDMA NIC and PMEM-Based KVStores 147

To reduce the high write latency in KVStores, researchers 148
propose many new hardware-oriented techniques that offer 149
tremendous opportunities for low latency and higher through- 150
put. Table I summarizes these techniques in terms of 151
durability, latency (P90), and storage cost. 152

RDMA-Based KVStores: One line of research is RDMA- 153
based networking latency reduction. They leverage the RDMA 154
technique, namely, accessing remote memory directly, to 155
enhance in-memory KVStore accessing. Distributed clients 156
can send requests to the server through RDMA verbs (such as 157
two-sided *Send/Recv* verbs and one-sided *Read/Write/Atomic* 158
verbs). Owing to the high networking bandwidth (10–200 159
GbE) and ultralow accessing latency (−3 μs) of RDMA 160
NICs, they can reduce networking latency for two orders 161
of magnitude and improve CPU consumption for 20× [9]. 162
However, RDMA-based in-memory KVStores do not preserve 163
data durability. Since all data are stored in main memory 164
and accessed through RDMA, they offer high performance 165
at the risk of data loss. Nowadays, many real-world storage 166
scenarios, such as financial and healthcare applications, require 167
no data loss after a system crash or power loss event. Besides, 168
storing all data in memory exaggerates the total cost of 169
ownership (TCO), which is crucial for commercial companies. 170

PMEM-Based KVStores: Another line of research is 171
PMEM-based persistent latency reduction. They leverage 172
PMEM as storage-class memory to employ KVStores. 173
Commodity Optane PMEM modules sit on the memory bus. 174
Users can issue memory load and store toward PMEM, similar 175
to DRAM. When users explicitly issue `CLFLUSH` instructions 176
on a dirty cache line associated with the PMEM address, 177
it can force data into PMEM persistently. The persistent 178
latency of PMEM is roughly 120 ns, which brings KVStore 179
memory-level performance while preserving data durability. 180
Besides, some distributed systems also combine PMEM and 181
RDMA, which can reduce networking latency and persistent 182
latency. However, it is uneconomical to employ PMEM-based 183
KVStores at scale. Building a system equipped with Optane 184
DC PMEM needs a vendor-specific CPU and motherboard, 185
which costs $547 for one 128-GB Optane DIMM and $2517 186
for one Intel Xeon Gold 6242R CPU. As the first pioneer of 187

PMEM, the Intel Optane production line will be closed soon due to cost-benefit considerations.

C. NVMe PMR and Its Opportunity

The NVM express (NVMe) 1.4 introduces the PMR concept in 2019 [10]. PMR represents a piece of nonvolatile memory located on the NVMe SSD devices. Users can map PMR into virtual address space, and then the CPU can issue memory read and write requests to PMR through MMIO. So far, many commodity NVMe SSDs, such as Starblaze SSD [11] and DapuStor SSD [12], have already supported the PMR technology, which has received much attention from the academia and industry [13], [14], [16], [17]. We can also implement PMR by adding power protection to the DRAM buffer inside SSDs. Introducing a 128-MB PMR to an 8-TB NVMe SSD costs only 0.1% of the SSD price, according to an internal estimation at ByteDance. As CMB and PMR are functionally equivalent and we can implement PMR by CMB, we refer to them only by PMR, as with previous works on PMR [13], [14].

In this article, we propose to leverage PMR to realize a low latency, low cost, and portable data-persisting path named PMRDirect. In PMRDirect, clients write data directly from RDMA NIC to the server-side PMR. As RDMA offers low networking latency and PMR guarantees data durability, PMRDirect can fulfill our goal.

III. PMRDirect: New Remote Persisting Path

A. Overview of PMRDirect

Enabling the direct path from RDMA NIC to PMR can reduce the networking latency and persistent latency of distributed write accesses, but it is a nontrivial task. Though SPDK [18] provides a userspace driver to access PMR from local CPUs and RDMA provides verbs to access remote memory directly, users cannot issue RDMA verbs to the PMR region due to lacking driver support so far. Specifically, the NVMe driver needs to expose the PMR region to the kernel space and provide support for pinning the PMR address and preparing DMA mappings upon DMA operations; the RDMA NIC driver should be able to identify the bus address of PMR. In a word, Remote DMA operations toward PMR require driver support from both the initiator and target sides.

We address the above challenges with the following contributions. First, we resort to a long-standing framework in Linux kernel called *dma-buf* [19]. *Dma-buf* enables sharing buffers for hardware DMA access across multiple devices and synchronizing asynchronous hardware accesses. Second, we add a patch to the Linux NVMe driver (438 LoC) to expose PMR as *dma-buf* objects. Afterward, we leverage an interface in the RDMA library (`ibv_reg_dmabuf_mr`) to register *dma-buf* objects as remote memory for clients (no LoC to the RDMA NIC driver). Finally, remote clients can post RDMA verbs to registered PMR buffers like normal RDMA. Besides, users can map a *dma-buf* object into virtual address space through `mmap`, allowing userspace access to PMR.

The NVMe driver acts as an exporter of *dma-buf* object related to PMR. To operate on a *dma-buf* object, the NVMe driver implements three pairs of kernel functions: {`pmr_attach`, `pmr_detach`}, {`pmr_map_dmabuf`, `pmr_unmap_dmabuf`}, and {`pmr_dmabuf_mmap`, `pmr_dmabuf_release`}. The `pmr_attach` function is called when other devices (such as RDMA NICs) want to attach the *dma-buf* object. It exposes PMR memory as a DMA memory pool for later DMA operations. The `pmr_map_dmabuf` function is called upon DMA operations to pin the buffer for RDMA NIC accessing and prepares the DMA mapping. The `pmr_dmabuf_mmap` is called when users map this *dma-buf* object to virtual address space, so it executes corresponding virtual memory mappings. The other three functions are reverse operations, respectively. By adding a new control operation into struct *nvme_dev_ioctl*, users can allocate a specific size of PMR as a *dma-buf* object through `ioctl` system call. Users pass two arguments to `ioctl`, i.e., a file descriptor (related to the NVMe SSD device) and an allocation size. The control operation allocates the new *dma-buf* object and returns a file descriptor (related to the *dma-buf* object) to the users. The latter can be used to register the PMR region for RDMA or `mmap` it to virtual address space.

B. Comparison With Other Remote Persisting Paths

Fig. 1 shows other remote data-persisting paths. Fig. 1(a) shows a typical data-persisting path in classical KVStores, which is adopted by NVMe-oF, Redis, and Memcached. A remote client sends data writes to server-side main memory through networking interfaces, such as TCP/IP or RDMA-based networking. The server is responsible for persisting data from memory to SSD. If the SSD supports PMR, an alternative destination of the persisting can be PMR [13], which results in Fig. 1(b). We refer to the remote data path in Fig. 1(a) and (b) as SSDSync and PMRSync. After introducing PMEM as the storage device, we can exploit a data-persisting path from RDMA NIC to PMEM as in Fig. 1(c), which we refer to as PMEMDirect [5], [6], [20]. With DDIO disable,[1] RDMA *Write* verbs upon PMEM address can guarantee durability. In this article, we present PMRDirect as a new remote data-persisting path that offers μs-level latency, as in Fig. 1(d).

Advantage of PMRDirect: The advantage of PMRDirect over other remote data-persisting paths lies in three aspects.

1) Compared to SSDSync, it offers μs-level low latency and provides data durability.
2) Compared to PMEMSync, it reduces the high cost of building a storage cluster that supports PMEM. Evaluation in Section 5 shows that the latency of PMRDirect is lower than PMEMSync under larger than 256-byte writes.
3) PMRDirect exploits peer-to-peer DMA between RDMA NIC and PMR-enabled NVMe SSD, which liberates memory bandwidth to applications. If a node is equipped with multiple NVMe SSDs and RDMA ports, PMRDirect is also advantageous for extending multiple highly isolated direct data paths.

[1] Intel DDIO is a feature that makes the processor cache the primary destination and source of I/O data rather than access main memory.

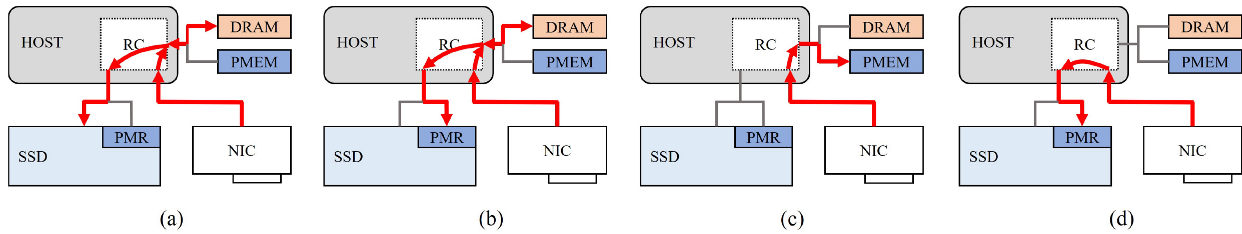


Fig. 1. Illustration of PMRDirect and other remote persisting paths. (a) SSDSync. (b) PMRSync. (c) PMEMDirect. (d) PMRDirect.

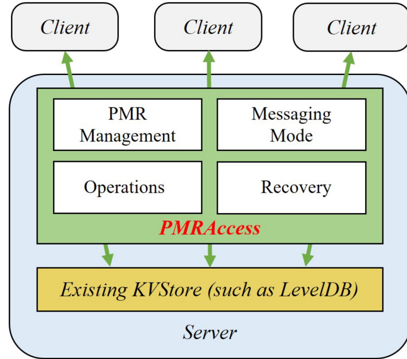


Fig. 2. Structure of PMRAccess. PMRAccess provides basic key–value operations for KVStores under client/server architecture.

Potential of PMRDirect: We can utilize PMRDirect to reduce the write latency of distributed systems that serve latency-sensitive applications such as financial transactions. Taking the chunk-based storage system at ByteDance as an example, when a client writes a piece of data to a single-node ChunkServer, the ChunkServer needs to persist data into storage before acknowledging it to the client. PMRDirect is promising for reducing this latency to μs-level, improving upstream service quality. A second use case of PMRDirect can be optimizing the well-known Raft consensus protocol [21]. Some KVStores require the Raft consensus protocol to ensure consistency. A log entry in the Raft protocol is deemed committed only if most nodes have persisted it in their local log file. Therefore, the latency of remote data-persisting paths reflects the write latency of committing a new state in a distributed system. PMRDirect is promising to reduce the write latency of Raft-based distributed systems.

IV. PMRAccess: PMRDirect-Based Accessing Stack for KVStores

To showcase the capability of PMRDirect, we propose to use PMRDirect to improve the accessing latency of existing KVStores, which results in our PMRAccess. Consider a KVStore server that serves a certain number of clients on different nodes. Clients send PUT/UPDATE/GET/DELETE requests to the server regarding any individual key–value pair, and the server then processes requests and sends results back to clients. If clients require write requests to be durable upon acknowledgment, the long write latency is always unavoidable. We present a PMRDirect-enabled KVStore accessing stack called PMRAccess to address this problem. The overview structure of PMRAccess is shown in Fig. 2. PMRAccess is KVStore-independent, which means it can provide data durability and low access latency for both in-memory KVStore and persistent KVStore without durability requirements from the back-end KVStore.

A. Challenges of PMRAccess

PMRAccess leverages the PMRDirect remote data-persisting path to transport data directly from client-side RDMA NIC to server-side PMR, avoiding the long persistent latency exposed by NVMe SSD. However, several challenges should be overcome before PMRAccess becomes a full-fledged KVStore accessing stack.

Challenge 1: How to reduce latency involved with KVStore requests with more efficient one-sided RDMA Write verbs? Supporting requests in accessing persistent KVStore requires two-sided primitives. A complete KVStore request can be divided into three phases: 1) the client posts a request to the server; 2) the server accesses KVStore (might generate I/O to retrieve or persist data); and 3) the server acknowledges the client. The three phases are strictly ordered and require the involvement of both sides. Thus, it is straightforward to implement two-sided RDMA verbs. However, two-sided RDMA verbs exhibit higher latency than one-sided RDMA verbs. In addition, using two-sided RDMA verbs requires two round trips for each request from the client's perspective, one for sending the request and the other for receiving its acknowledgment. Therefore, the first challenge is how to use one-sided RDMA *Write* verbs to reduce write request latency.

Challenge 2: How to manage memory effectively for PMR with limited capacity under multiple clients? Currently, the PMR capacity is constrained. Therefore, the PMR space left for storing persistent data is limited to MB-level. If multiple clients write data to PMR simultaneously, one should be concerned about write stall due to high write contention of RDMA writes and PMR capacity shortage. Thus, PMRAccess should overcome this challenge. First, it has to persist data in PMR to SSD asynchronously and reclaim them for later usage. Second, it has to reduce the write contention of multiple clients while maximizing PMR utilization. Finally, it can recover from system crashes or power loss events and guarantee the durability of data writes.

B. RDMA Write-Based Messaging Mode

Previous works have proposed customized messaging modes for client/server scenarios, such as FaSST [22] and CatFish [23]. FaSST relies on the RDMA Send-based messaging mode, which incurs two network round trips per

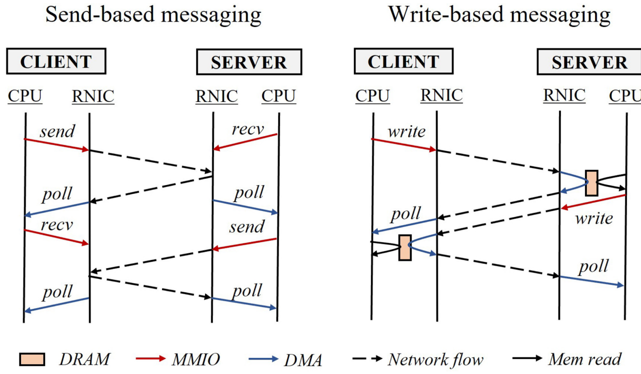
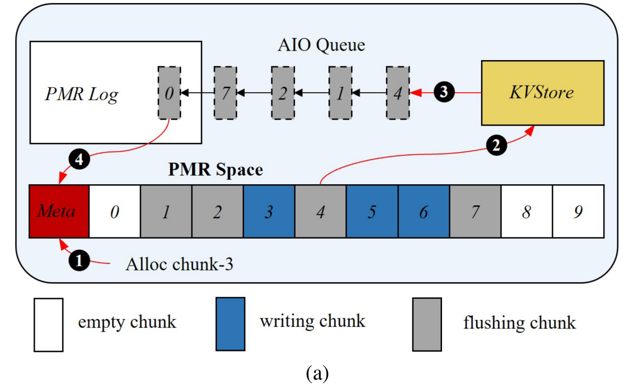


Fig. 3. Send-based messaging versus write-based messaging in RDMA.



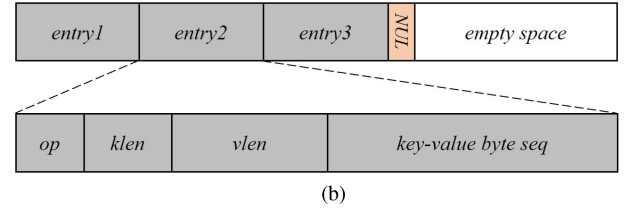


Fig. 4. PMR management scheme in PMRAccess. (a) Chunk-based PMR management. (b) Structure of a writing chunk.

request. CatFish proposes using one-sided RDMA verbs, but it maintains a central messaging queue to serve all clients and processes each request asynchronously, which increases the request latency. In contrast, we propose using RDMA Write verbs to reduce the network round trips to one per request, and the server processes request synchronously.

In the RDMA Send-based messaging mode [Fig. 3(a)], the client posts an RDMA *Send* verb to the server-side NIC, and then the client has to block and wait for the verb to complete. After the *Send* verb, the client posts an RDMA *Recv* verb and waits for completion. The *Send* and *Recv* verbs on each side cannot overlap as they will block the CPU and poll for a completion event. In addition, the receiver RDMA NIC has to ensure that a *Recv* verb has been posted before a *Send* verb arrives. Therefore, one request takes two network round trips.

In our RDMA Write-based messaging mode [Fig. 3(b)], the client sends two RDMA writes to the server-side memory addresses: one for the data request and the other to set a server-side flag that marks a data request that reaches the server. The two writes are combined in one RDMA *Write* verb and signal once to reduce network latency. Afterward, the client starts waiting for the server to set its local flag (as shown in `SendReq` in Algorithm 2). When the server finds its local flag is set, it parses and executes the new request. The server returns the result to the client and sets a client-side flag with a single RDMA *Write* verb. Our RDMA Write-based messaging mode overlaps the latency of waiting for a *Write* verb to complete with the server-side acknowledgment to come back. Therefore, it can reduce both the network round trip per request and round trip latency.

C. PMR Management

In PMRAccess, PMR is used to buffer writes for multiple clients. As the capacity of PMR is limited (e.g., only 8 MB in DapuStor Haishen5 [12]), we need to design a PMR management mechanism to reduce write contentions between multiple clients and batch-write data to the SSD efficiently, ensuring high PMR utilization efficiency.

RDMA-based locks are much slower than memory atomic operations [24]. Thus, we need to use a private PMR region for each client to reduce coordination costs. When the private PMR region becomes full, we must flush all its data to SSD and reclaim it, blocking the client until it vacuums its PMR region and incurring high tail latency. Besides, a large private PMR region will limit the maximum number of connected clients, and a small private PMR region will limit the throughput of clients. To avoid this problem, we propose a chunk-based PMR management for multiple clients, as shown in Fig. 4(a). We divide the PMR space into identical small chunks. Initially, all chunks are empty chunks. When a new client connects to the server, it allocates an empty chunk (❶) as a writing chunk. Then, it sends data write requests to this chunk and preserves durability by it. The client maintains the usage of a writing chunk until it contains no more space. Then, the client sends a CHUNKALLOC request to the server to transfer ownership of its writing chunk to the server. The server adds the old chunk into a queue and allocates a new empty one for the client (❶). We store states of all chunks in a concurrent bitmap [Meta in Fig. 4(a)], where 0 represents an empty chunk, and 1 represents a writing chunk or a flush chunk.

The server processes flushing chunks with a background thread. A flushing chunk is first passed to the KVStore for potential usage (some KVStores support writing in a batch, and they can utilize the flushing chunk ❷). Afterward, it is added to an asynchronous I/O (AIO) Queue to persist to an in-SSD log file (❸). When a flushing chunk reaches SSD, the server modifies its state back to an empty chunk (❹), and then it can be allocated again. As we persist chunk to SSD in a background manner, the long persisting latency is hidden from clients. When all empty chunks are used up, clients with no available PMR chunks will wait for the server to release new empty chunks (line 17 in Algorithm 2). In such situations, the throughput of PMRAccess will be limited by the SSD write bandwidth. This behavior is common in many existing systems, such as RocksDB, which also uses rate-limit and write-stall policies to postpone the client's requests.

Algorithm 1: Operations for Client

```
1  Function GET(key):
2      req.op = GET; req.key = key; req.val = NULL;
3      res = SendReq(req, pmrchunk + off);
4      return res;
5  Function PUT(key, val):
6      req.op = PUT; req.key = key; req.val = val;
7      res = SendReq(req, pmrchunk + off);
8      off = off + 8 + key.size + val.size;
9      return;
10 Function CHUNKALLOC():
11     req.op = CHUNKALLOC;
12     req.key = pmrchunk; req.val = NULL;
13     pmrchunk = SendReq(req, pmrchunk + off);
14     off = 0;
15     return pmrchunk;
16 Function SendReq(req, pmr_addr):
17     rdma_post_write(req, req.len, pmr_addr);
18     rdma_post_write_signaled(1, 1, &server_flag);
       // wait for write completion
19     rdma_poll_complete();
20     while client_flag == 0 do
21         nop; // wait for server
22     end
23     client_flag = 0;
24     return parse(client_buf);
```

Algorithm 2: Operations for Server (Serve One Client)

```
1  Function ProcessRequest(db):
2      while True do
3          while server_flag == 0 do
4              nop; // wait for client
5          end
6          server_flag = 0;
7          req = parse(pmrchunk);
8          switch req.op do
9              case GET do
10                 res = db.get(req.key);
11             end
12             case PUT do
13                 res = db.put(req.key, req.val);
14             end
15             case CHUNKALLOC do
16                 add req.key to flushing AIO queue;
17                 pmrchunk = find an empty chunk;
18                 res = pmrchunk;
19             end
20         end
21         rdma_post_write(res, res.len, &client_buf);
22         rdma_post_write_signaled(1, 1, &client_flag);
           // wait for write completion
23         rdma_poll_complete();
24     end
```

We depict the structure of a writing chunk in Fig. 4(b). The write requests in a writing chunk are represented as an entry. Each entry comprises a 1-byte operation field, a 3-byte key-length field, a 4-byte value-length field, and a variable-length byte sequence containing the key–value pair. The last valid write request in a writing chunk has a `NUL` terminator character followed, which separates the entry area and empty area for the server upon recovery. For all write operations, the client will append the request with a terminator right at the position to override the last terminator. For nonwrite requests, the client will not proceed to the entry area to reuse it later.

D. Operations

PMRAccess provides basic database access operations (PUT/GET/UPDATE/DELETE) for clients to access server-side KVStore (Algorithm 1). When a client connects to the server, the server spawns a new thread to serve this client exclusively. This thread runs the `ProcessRequest` function as listed in Algorithm 2. `ProcessRequest` is a dead loop that processes new requests iteratively. Basic KVStore operations invoke interfaces provided by the back-end KVStore. Clients send operations to the server by RDMA Write-based messaging mode in Section IV-B. For write requests, PMRAccess requires no durability assurance from the KVStore itself, as write requests are either stored in PMR chunks or flushed to the in-SSD log file. Therefore, the KVStore is unnecessary to persist log entries or data writes to SSD immediately.

PMRAccess also provides a CHUNKALLOC operation to allocate a free chunk from the PMR address space. Before a client can send basic operations to the server, it should check whether the client owns a writing chunk with enough free space. If not, it sends a CHUNKALLOC request to the server. The old writing chunk address is embedded in the request. The corresponding serving thread on the server processes this request according to Algorithm 2. It works as follows: 1) pass the old chunk to the back-end KVStore for potential usage; 2) add the old chunk to the AIO queue; and 3) allocate a new chunk by changing its state in the Meta array from 0 to 1.

E. Recovery

Because only server-side threads can modify the Meta bitmap, and these writes are all atomic, writes to PMR are failure-atomic. In addition, we leverage *dma-buf* framework to handle the coherent issue of CPU accessing PMR with RDMA NIC, which guarantees that local writes reach PMR space before RDMA NIC accesses it. Writing chunks are only modified by RDMA *Write* verbs, where data goes from RDMA NIC to PMR directly. The semantic of RDMA *Write* verb guarantees that when the terminator character reaches PMR, the data requests are valid.

When encountering power loss or system crash, we need to recover the PMR area. It proceeds as follows: 1) scan the Meta bitmap to gather all writing chunks and flushing chunks; 2) write each flushing chunk to the in-SSD log file and clear their Meta state; 3) scan each writing chunk to locate the first terminator, write all valid data requests to the log file, and clear their bits; and 4) replay the log to the KVStore. To reduce the log file size, we checkpoint the KVStore and then reclaim the in-SSD log file. Note that the log file in PMRAccess differs from the internal WAL of the KVStore. If we use PMRAccess on a persistent KVStore, it may contain such a WAL file that persists logs periodically, but when serving a purely in-memory KVStore, the log file in PMRAccess is all we have for durability.

V. Performance Evaluation

In this section, we conduct experiments to answer the following questions about our design.

1) How does PMRDirect perform when compared with other remote data-persisting paths? Does PMRDirect compare against PMEMDirect? The results are reported in Section V-B.
2) Can PMRAccess reduce the write latency of remote writes on KVStores? Can PMRAccess work efficiently on both in-memory KVStores and in-storage KVStores while providing data durability? Can PMRAccess help reduce the tail latency? The results are reported in Section V-C.
3) Can PMRAccess still work well in a system-to-system comparison? For example, can a PMRAccess-enabled KVStore outperform the well-known Redis? The results are shown in Section V-D.
4) Does the RDMA Write-based messaging mode outperform the classical RDMA Send-based messaging in terms of average latency and write bandwidth? The results are shown in Section V-E.
5) Can PMRAccess fully exploit the PMR capacity, and how does it impact the overall bandwidth? The results are shown in Section V-F.

A. Settings

We conduct all experiments on two physical machines: one node serves as the storage server node, and the other is the client node. Each node has two Intel Xeon Gold 6240 CPUs. Each CPU has 18 cores and a shared 24-MB L3 Cache. There are 32-kB L1I cache, a 32-kB L1D cache, and a 1-MB L2 cache per core. For networking, each node is equipped with a single-port 100 GbE Mellanox MCX515A-CCAT RDMA NIC to connect the server and client node back to back. The server node also has four 128-GB Optane PMEM DIMMs on each socket and one 3.20-TB DapuStor NVMe SSD [12] with an 8-MB on-chip PMR region.

Each node is running Linux with a 5.14.0 kernel. The Optane PMEM modules are configured into the `app-direct` mode and exposed as particular Device DAX (`devdax`) character devices. The DapuStor NVMe SSD is exposed to the users as an NVMe block device, and we install the ext4 filesystem on it. For accessing PMEM from remote RDMA NICs, we map the `devdax` devices into virtual memory and register it for RDMA. For accessing PMR, we obtain the file descriptor through `ioctl` to the NVMe device. We map the file descriptor into virtual memory using `mmap` to support local accesses. For RDMA writes to PMR, we register PMR memory by the `ibv_reg_dmabuf_mr` interface and then issue RDMA verbs.

B. Performance of PMRDirect

This experiment aims to demonstrate the performance upper bound of our PMRDirect data path. We compare four remote data-persisting paths as elaborated in Section III-B. As we need not manipulate data, we reuse target address space as we care more about the latency and bandwidth metrics.

Fig. 5(a) shows the client-side request latency. Under small requests, PMRDirect and PMEMDirect exhibit similar write latency, which is 1.63 μs per 64-byte request. PMRSync shows 56% more write latency than PMRDirect because it needs to persist data from host memory to PMR, twice as much as PCIe transactions as in PMRDirect. SSDSync exhibits 16-μs write latency to persist data into SSD, 10× that of PMRDirect. As the request size increases, PMRDirect achieves the lowest write latency. As the write request size increases 64 times, the write latency only increases for 94%, which is 3.1 μs at a 4-kB write request. PMRSync shows roughly 1 μs more write latency than PMRDirect. The latency of PMEMDirect starts rising after 256 B. As the write requests increase to 4 kB, it needs 13 μs to directly finish a write request from RDMA to PMEM. The write latency does not increase because write requests to NVMe SSD are amplified to 4 kB due to filesystem block size.

We measure the maximum write bandwidth of four competitors under 64 B, 256 B, 1 kB, and 4 kB report results in Fig. 5(b). In practice, it needs 32/16/12/32 clients to saturate the max write bandwidth for PMRDirect/PMEMDirect/SSDSync/PMRSync, respectively. PMRDirect achieves the highest write bandwidth under different request sizes. As the request size increases to 256 B, the maximum write bandwidth increases by 2.7×. The value reaches 2.6 GB for requests larger than 1 kB. Following PMRDirect, PMRSync exhibits a 33% lower write bandwidth than PMRDirect as a penalty for copying data up and down in terms of PCIe transactions. The max write bandwidth in PMEMDirect is only half of PMRDirect because of the low write bandwidth of commodity Optane PMEM. While for SSDSync, its max write bandwidth increases linearly as the request size. It exhibits 30-MB write bandwidth at 64-B write requests. This bandwidth reaches 1.2 GB at 4 kB requests, which is almost the peak write bandwidth of our NVMe SSD.

PMRDirect is a direct data path from RDMA NIC to PMR and will not occupy the memory bus. Therefore, it cannot be affected by other processes that occupy the memory bus bandwidth. To verify that, we measure the max write bandwidth of four data paths in the shadow of a bandwidth-stealing process that occupies server-side memory bandwidth (16 threads do random writes concurrently). We report the bandwidth loss compared with Fig. 5(b) in (c). It reveals that remote data-persisting paths, except PMRDirect, are more or less affected by the bandwidth-stealing process. The max write bandwidth of PMEMDirect/SSDSync/PMRSync drops by 23%/3%/13%, respectively. Fig. 5(c) shows that PMRDirect occupies no memory bandwidth and can scale the overall bandwidth linearly with more PMR-enabled NVMe SSDs.

C. Performance of PMRAccess

PMRAccess is a KVStore accessing stack that provides low latency and data durability for accessing any existing KVStores from distributed clients. We use PMRAccess to provide primary accesses for two types of KVStores, including

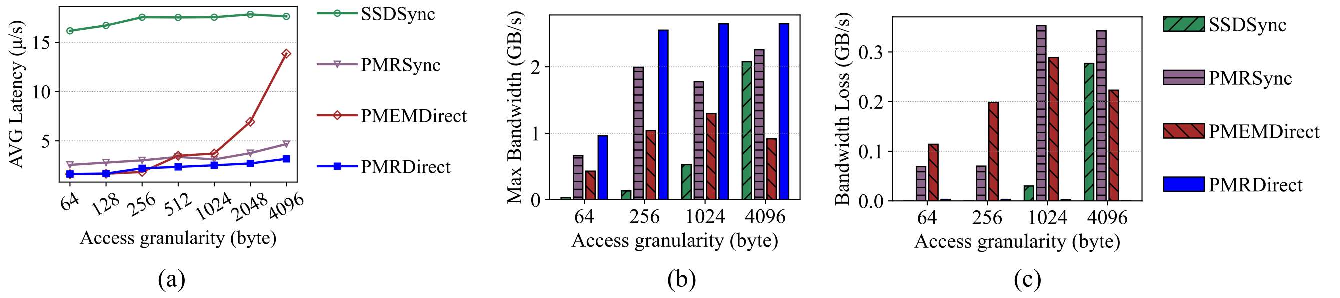


Fig. 5. Performance comparison between PMRDirect and three remote data-persisting paths: SSDSync, PMRSync, and PMEMDirect. (a) Average latency. (b) Max bandwidth. (c) Bandwidth loss.

LibCuckoo and LevelDB. We aim to demonstrate that our PMRDirect data path is effective in optimizing KVStores.

Competitors: We compare the following KVStore accessing stacks, including SyncAccess, GroupAccess, PMEMAccess, and PMRAccess, all providing data durability for each write request. Though they differ in how to guarantee durability when sending write requests, these accessing stacks adopt identical messaging modes and provide identical functionalities for accessing KVStores. In a word, the comparison is fair in terms of network latency.

1) *PMRAccess:* Using RDMA Write-based messaging mode and PMRDirect data path, clients send write requests to the server while preserving WAL durability by PMR. WAL records in PMR are persisted to SSD at the unit of a Chunk.
2) *SyncAccess:* Using RDMA Write-based messaging mode and SSDSync data path, clients send write requests to the server while preserving WAL durability by persisting data to SSD before acknowledging clients (adopted by Redis [8]).
3) *GroupAccess:* WAL records from different clients are grouped and persist to SSD in a batch as in LevelDB. GroupAccess can reduce write amplification under multiple-client scenarios (adopted by LevelDB [25]).
4) *PMEMAccess:* Using RDMA Write-based messaging mode and PMEMDirect data path, clients send write requests to the server while preserving WAL durability by PMEM (adopted by FlatStore [5]).

Back-End KVStores: We compare these four accessing stacks under two types of KVStores: 1) LibCuckoo and 2) LevelDB. It includes both in-memory KVStore and in-storage KVStore to make the comparison diverse. LibCuckoo[2] is a high-performance, concurrent hash table. LibCuckoo stores key–value pairs in memory and guarantees no durability. LevelDB[3] is a fast key–value storage library written at Google that provides an ordered mapping from string keys to string values. It stores primary data in storage devices and guarantees durability through log files. LevelDB can control the durability of each write operation by setting its write option to persist a WAL record. However, the performance is poor when providing data durability for each write.

Workloads: Because PMRAccess focuses on write operations, we choose the Load Only, YCSB-A, and YCSB-B query workload, which stands for write-intensive, read–write balanced, and read-intensive workloads. Key–value pairs are 64-byte, composed of 16-byte string key and 48-byte value. For larger value sizes, readers can refer to Section V-D for detailed comparison under various value sizes. We populate each KVStore with 128M key–value pairs. Then, we measure the overall throughput and latency for each workload.

1) *Load Only:* This workload consists of 100% PUT operations with randomized new keys.
2) *YCSB-A:* This workload consists of 50% GET operations and 50% UPDATE operations. All keys follow the Zipfian distribution. The skewness value of Zipfian is 0.9.
3) *YCSB-B:* This workload consists of 95% GET operations and 5% UPDATE operations. Similarly, all keys followed the Zipfian distribution.

Performance on LibCuckoo: This experiment evaluates PMRAccess in LibCuckoo, a purely in-memory Data Store that supports no durability. All accesses to KVStore come to memory, but WAL records for each write request are persisted by each accessing stacks with their strategies.

The throughputs of different accessing stacks on LibCuckoo are shown in Fig. 6. We report the Load Only result in Fig. 6(a). Under a single client, PMEMAccess outperforms PMRAccess by 34%. As we increase the client number, PMRAccess outperforms PMEMAccess and reaches 1.059 Mops/s, 2.3×/14×/27× that of PMEMAccess, SyncAccess, and GroupAccess. Under the YCSB-A workload, the throughputs are higher than the load-only workload. Under 32 clients, PMRAccess is 3.2×/47×/23× better than PMEMAccess, SyncAccess, and GroupAccess. Under YCSB-B workloads, the gap between different competitors starts to narrow down.

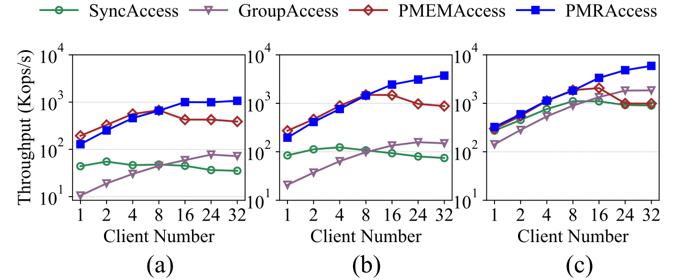


Fig. 6. Throughputs of different accessing stacks on LibCuckoo. (a) Load only. (b) YCSB-A. (c) YCSB-B.

[2] LibCuckoo: https://github.com/efficient/libcuckoo
[3] LevelDB: https://github.com/google/leveldb

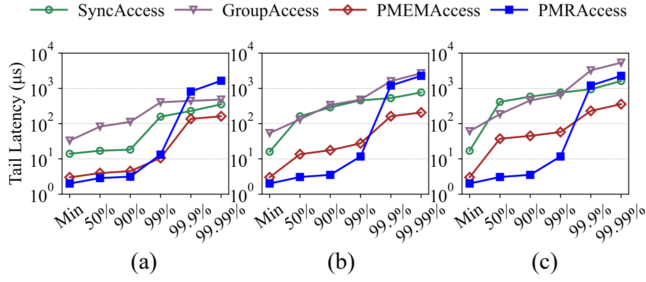


Fig. 7. Tail latency of different accessing stacks on LibCuckoo. (a) One client. (b) Eight clients. (c) Sixteen clients.

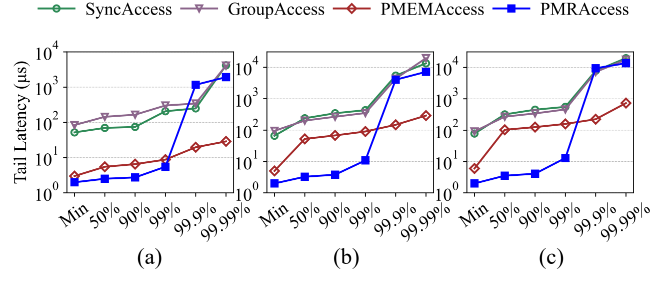


Fig. 9. Tail latency of different accessing stacks on LevelDB. (a) One client. (b) Eight clients. (c) Sixteen clients.

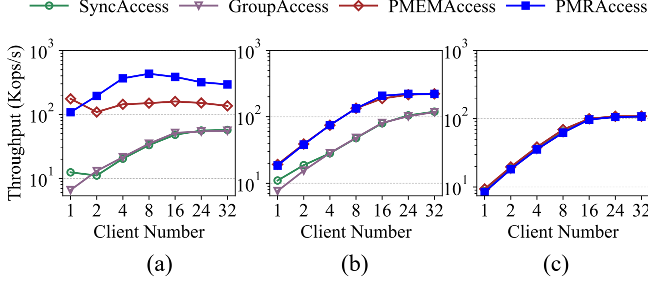


Fig. 8. Throughput of different accessing stacks on LevelDB. (a) Load only. (b) YCSB-A. (c) YCSB-B.

Both PMEMAccess and SyncAccess suffer from multiple-client contention on hardware resources. Under 32 clients, PMRAccess achieves 5.9×/6.5×/3.8× total throughput of PMEMAccess, SyncAccess, and GroupAccess, respectively.

The tail latency of PUT operations with different accessing stacks on LibCuckoo is in Fig. 7. We report the tail latency of four accessing stacks under 1 client [Fig. 7(a)], 8 clients [Fig. 7(b)], and 16 clients [Fig. 7(c)]. PMRAccess maintains the best Min latency, 50%, 90%, and 99% tail latency. Under one client, PMRAccess reduces latency to 3 μs, followed by PMEMAccess, which is 4 μs. Meanwhile, the PUT latency of SyncAccess is roughly 16 and 80 μs for GroupAccess. PMEMAccess maintains the best 99.9% tail latency at about 136 us, which is 80% less than PMRAccess, by storing all WAL records in PMEM. As we increase the client number, PMRAccess maintains its advantage at 99% tail latency. Under 16 clients, the 50% latency of PMRDirect is only 8%, 0.7%, and 1.6% of PMEMAccess, SyncAccess, and GroupAccess, respectively.

Performance on LevelDB: To evaluate PMRAccess in LevelDB, we choose the following system configurations. All competitors follow identical configurations to make the comparison fair. Each competitor is allocated with fixed memory and set with CGroups. The memory size is 4 GB, a third of the dataset, to ensure that requests are served from memory and persistent storage. The block cache size is 256 MB, and the Memtable size is 64 MB. In addition, for each write operation, the *sync* option is set to false, which means that LevelDB does not explicitly guarantee durability.

The throughputs of different accessing stacks on LevelDB are shown in Fig. 8. The overall throughput of accessing LevelDB is far less than that of LibCuckoo, as most of LevelDB's data resides in SSD. In the *Load Only* workload, PMRAccess achieves the highest throughput, which is 120 Kops/s under one client. When we increase the client number, PMRDirect's throughput increases to 430 Kops/s. The peak throughput of PMRAccess is 2.68×/7.1×/7.8× that of PMEMAccess, SyncAccess, and GroupAccess, respectively.

The tail latency of PUT operations with different accessing stacks on LevelDB is shown in Fig. 9. Similar to the tail-latency results on LibCuckoo, PMRAccess is the best among all competitors in terms of the Min, 50%, 90%, and 99% tail latency. Specifically, under one client, the 50% tail latency of PMRAccess is 2.5 μs, which is only 45%/3.8%/1.7% of PMEMAccess, SyncAccess, and GroupAccess, respectively. In addition, PMRAccess keeps the 90% latency less than 3 μs and 99% latency less than 6 μs, which is 36× faster than SyncAccess and even 0.5× faster than PMEMAccess. When we increase the client number to 16, PMRAccess's write latency increases slightly within 1 μs, which proves that PMRAccess has good performance isolation between multiple clients. SyncAccess and GroupAccess are bad in LevelDB as they waste CPU resources and SSD bandwidth on persisting WAL.

To sum up, PMRAccess provides extremely low write latency to 3 μs per remote write request, which is up to 36× better than SyncAccess/GroupAccess and 4× better than PMEMAccess. It can also maintain the best 50%, 90%, and 99% tail latency for both LibCuckoo and LevelDB. Regarding the throughput, PMRAccess achieves a high 1-Mops/s throughput on LibCuckoo and 430-Kops/s throughput on LevelDB and guarantees each remote write-request is durable. It achieves up to 18× higher throughput than SyncAccess/GroupAccess and a 1.5× higher throughput than PMEMAccess. Moreover, PMRAccess is also advantageous for performance isolation between clients and can keep high scalability when the number of remote clients increases.

D. System-to-System Comparison

Further, we conduct a system-to-system comparison to demonstrate the advantage of our proposal. We integrate PMRAccess into LibCuckoo and compare it with Redis [8] with two durability levels, including Redis-Strong and Redis-Everysec. Redis-Strong represents Redis with a durability guarantee for each write request; Redis-Everysec represents Redis with a durability guarantee for every second. We allocate identical hardware resources for each system, such as thread number and NIC resources.

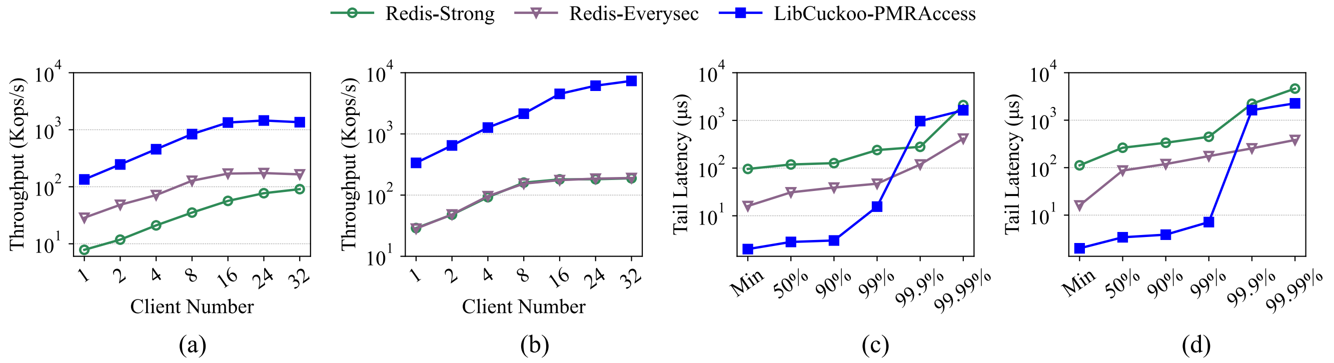


Fig. 10. Throughput and latency comparison between our system (LibCuckoo-PMRAccess) and Redis under different clients. (a) PUT. (b) GET. (c) One client. (d) Sixteen clients.

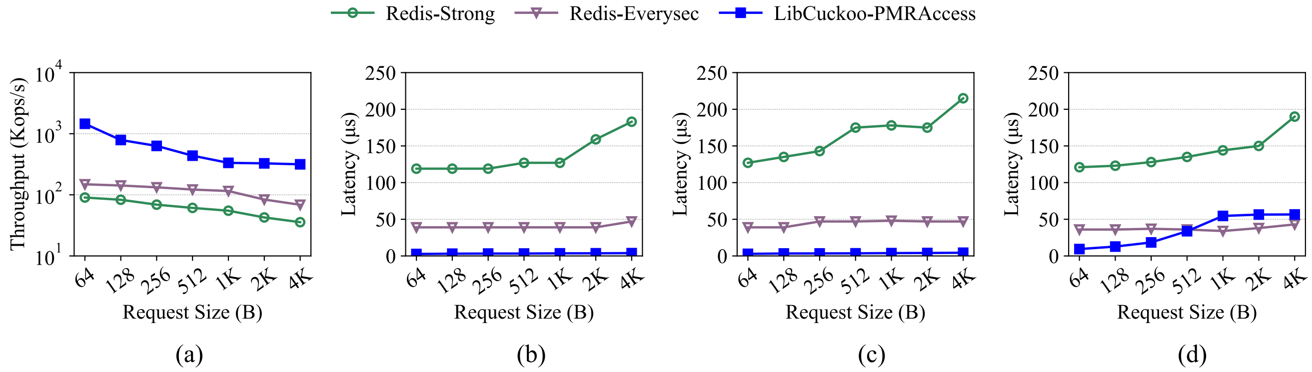


Fig. 11. Throughput and latency comparison between our system (LibCuckoo-PMRAccess) and Redis with different request sizes. (a) PUT IOPS. (b) 50% tail latency. (c) 90% tail latency. (d) AVG latency.

Fig. 10 shows the throughput and latency of three persistent KVStores under multiple clients. As Fig. 10(a) shows, under the PUT workload, LibCuckoo-PMRAccess achieves a 13.9× higher throughput than Redis-Strong and a 7.2× higher throughput than Redis-Everysec with 32 clients. For the GET workload, LibCuckoo-PMRAccess achieves 7 Mops/s, about 38× higher than Redis-Strong/Redis-Everysec. The tail latency of LibCuckoo-PMRAccess is reported in Fig. 10(c) and (d). LibCuckoo-PMRAccess achieves the best Min latency and the 50%/90%/99% tail latency. Specifically, the 90% tail latency of LibCuckoo-PMRAccess is 2.3%/7.7% of Redis-Strong/Redis-Everysec. LibCuckoo-PMRAccess exhibits 3-μs 90% tail latency, almost as fast as one RDMA *Write* verb. LibCuckoo-PMRAccess's 99.9% tail latency is worse than Redis-Everysec as it puts more pressure on SSDs.

Fig. 11 shows the throughput and latency of three persistent KVStore under request size varied from 64 B to 4 kB. At a small request size, LibCuckoo-PMRAccess is 15× better than Redis. As the request size increases to 1 kB, the PUT throughput of LibCuckoo-PMRAccess is 2.89× and 6.06× of Redis-Everysec and Redis-Strong throughput. Under 4 kB requests, LibCuckoo-PMRAccess's PUT throughput is 4.5× and 8.8× that of Redis-Everysec and Redis-Strong. For Tail latency, LibCuckoo-PMRAccess maintains lower than 3.8-μs 50% latency and 4.3-μs 90% tail latency, which is 48× lower than that of Redis-Strong and 9× lower than that of Redis-Everysec. In addition, LibCuckoo-PMRAccess's 50%/90% tail latency increases slightly with the request size, and the 50% tail latency of Redis-Strong increases from 127 to 234 μs. Fig. 11(d) reports the average latency of the three KVStores. The average latency of Redis-Strong is 3.36× and 12.8× higher than that of Redis-Everysec and LibCuckoo-PMRAccess under small request sizes.

In summary, the PMRAccess-enabled KVStore, LibCuckoo-PMRAccess, achieves up to 13× higher throughputs and 40× lower 50%/90% tail latency than Redis under small request sizes. Although the throughput improvement of LibCuckoo-PMRAccess decreases with the increase of the request size, LibCuckoo-PMRAccess still outperforms Redis.

E. Benefits of the RDMA Write-Based Messaging Mode

In this experiment, we compare our RDMA Write-based client/server messaging mode with the RDMA Send-based messaging mode to demonstrate its efficiency in reducing the request latency for remote clients.

Our RDMA Write-based messaging mode reduces the average latency of client/server requests to 2.1–3.5 μs, which is only 26% of the latency observed with the RDMA Send-based messaging mode. Additionally, it can improve the multiclient write bandwidth by at least 3.1×. As the size of the write request increases, our RDMA Write-based messaging can maintain low request latency under 3.6 μs and achieve up to 9-GB/s write bandwidth. Given that one RDMA *Send* verb takes about 4 μs and one RDMA *Write* verb takes about 2 μs, the results suggest that our messaging mode can reduce both

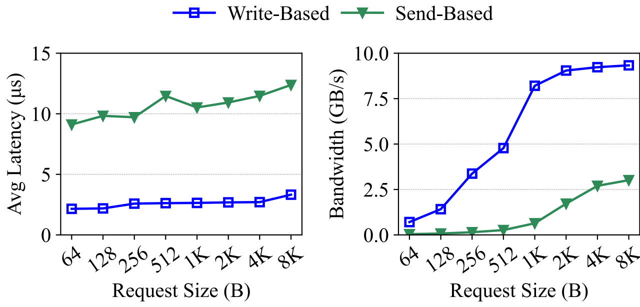


Fig. 12. Performance of different messaging modes: (a) average write latency and (b) total write bandwidth.

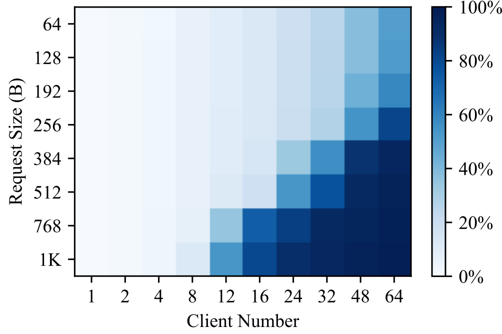


Fig. 13. Average PMR utilization ratio of PMRAccess under different configurations (darker colors mean higher utilization ratios).

the number of network round trips and the latency of each round trip between the clients and the server.

F. PMR Utilization and Write Bandwidth

To uncover the PMR utilization of PMRAccess, we measure the real-time utilization ratio of PMR chunks under write-intensive workloads. We use LibCuckoo as the back-end KVStore engine and report the PMR utilization on the server side and the total write bandwidth on the client side. We evaluated various configurations, including the number of parallel clients and key–value request size, with the number of clients ranging from 1 to 64 and the request size ranging from 64 B to 1 kB.

Fig. 13 shows the average PMR utilization ratio under different configurations. With a small request size and a low number of clients, the PMR utilization ratio remains below 10%. As the number of clients increases to 64, the PMR utilization rises to between 50% and 80%, even with a small request size. When the number of clients exceeds 24, and the request size is larger than 512 B, the real-time PMR utilization ratio reaches 100% most of the time. Fig. 14 shows the write bandwidth of all clients under various configurations. The write bandwidth increases accordingly as the client number and request size grow to a certain extent. However, once the average PMR utilization reaches about 80%, the write bandwidth either plateaus or slightly decreases. To sum up, large write requests or excessive parallel clients tend to exhaust the PMR and eventually bottleneck the overall bandwidth.

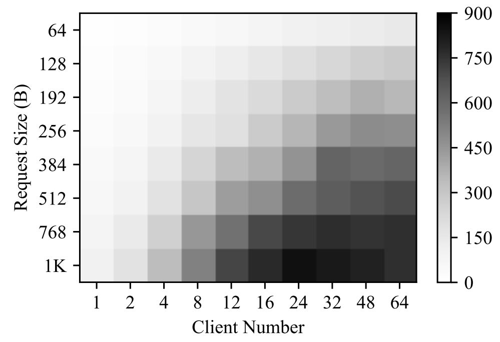


Fig. 14. Total write bandwidth of PMRAccess under different configurations (darker colors mean higher write bandwidths).

VI. Related Work

Traditional KVStores, such as Memcached, Redis [8], and Cassandra, are extensively used by commercial companies. With TCP/IP-based networking stacks and HDD/SSDs as storage devices, the write latency to KVStores for remote clients is usually hundreds of microseconds.

As the development of high RDMA NICs, many pioneers exploit the potential of reducing the networking latency in KVStores [3], [9] and transaction execution [22], [26]. However, RDMA supports accessing memory only; therefore, those RDMA-based KVStores store all data in DRAM and ignore data durability. Besides, the advent of PMEM brings new strategies for the durability of KVStores. Recently, many researchers propose PMEM-based Indexes [27], [28] and PMEM-based KVStore [4], [5]. Most of them exploit the characteristics of commodity PMEM, such as low latency and high bandwidth, to achieve high performance while ensuring data durability. Unfortunately, PMEM is too expensive compared to SSDs, and the Optane production line is closed.

So far, few studies have used PMR to improve storage and file systems. SineKV [13] first used PMR in KVStores to boost the durability of local WAL entries. Horae [14] used PMR to reduce the write order constraints of traditional file systems. X-SSD [16] proposed to use PMR for SSDs and integrated the database logging replication service into SSDs by Nontransparent Bridging networking. Unlike all the above techniques, our proposal opens up a new path that enables remote clients to write data to NVMe PMR directly, ensuring μs-level low latency while preserving data durability with low storage cost.

VII. Conclusion

In this article, we proposed to use NVMe PMR to build a new data-persisting path called PMRDirect to enable low-latency data writes from distributed clients. Further, we designed an accessing stack called PMRAccess for KVStores to reduce the latency of remote data writes. The experiments on real RDMA NICs and a PMR-enabled NVMe SSD showed that PMRDirect achieved the lowest write latency and the highest write bandwidth. Moreover, when evaluated on LevelDB, PMRAccess outperforms the SSD-based accessing stack by up to 6.1× in write throughput and 36× in write latency.

898 # REFERENCES

899 [1] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson, "FaRM:
900 Fast remote memory," in *Proc. 11th USENIX Symp. Netw. Syst. Design*
901 *Implement. (NSDI)*, 2014, pp. 401–414.
902 [2] B. Li et al., "KV-direct: High-performance in-memory key-value store
903 with programmable NIC," in *Proc. 26th SOSP*, 2017, pp. 137–152.
904 [3] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A holistic
905 approach to fast in-memory key-value storage," in *Proc. 11th USENIX*
906 *Symp. Netw. Syst. Design Implement. (NSDI)*, 2014, pp. 429–444.
907 [4] L. Vogel, A. van Renen, S. Imamura, J. Giceva, T. Neumann, and
908 A. Kemper, "Plush: A write-optimized persistent log-structured hash-
909 table," *Proc. VLDB Endowment*, vol. 15, no. 11, pp. 2895–2907, 2022.
910 [5] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. Shu, "FlatStore:
911 An efficient log-structured key-value storage engine for persistent
912 memory," in *Proc. 25th Int. Conf. ASPLOS*, 2020, pp. 1077–1091.
913 [6] T. Li, D. Shankar, S. Gugnani, and X. Lu, "RDMP-KV: Designing
914 remote direct memory persistence based key-value stores with
915 PMEM," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage*
916 *Anal.*, 2020, pp. 1–14.
917 [7] K. Huang, D. Imai, T. Wang, and D. Xie, "SSDs striking back: The
918 storage jungle and its implications to persistent indexes," in *Proc. CIDR*,
919 2022, pp. 9–12.
920 [8] "Redis." 2023. [Online]. Available: https://redis.io/
921 [9] C. Mitchell, Y. Geng, and J. Li, "Using one-sided RDMA reads to build
922 a fast, CPU-efficient key-value store," in *Proc. USENIX ATC*, 2013,
923 pp. 103–114.
924 [10] "Persistent memory region." NVMe. 2023. [Online]. Available: https://
925 nvmexpress.org/specification/nvm-express-base-specification/
926 [11] "Starblaze OC SSD." Starblaze. 2023. [Online]. Available: https://www.
927 starblaze-tech.com/en/lists/content/id/137.html
928 [12] (DapuStor Corp., Shenzhen, China). *Dapustor Haishen5 PCIe*
929 *Gen5 Enterprise SSDs*. (2023). [Online]. Available: https://www.
930 storagereview.com/news/dapustor-haishen5-pcie-gen5-enterprise-ssds-
931 announced
932 [13] F. Li, Y. Lu, Z. Yang, and J. Shu, "SineKV: Decoupled secondary
933 indexing for LSM-based key-value stores," in *Proc. IEEE 40th ICDCS*,
934 2020, pp. 1112–1122.
935 [14] X. Liao, Y. Lu, E. Xu, and J. Shu, "Write dependency disentanglement
936 with HORAE," in *Proc. 14th USENIX Symp. OSDI*, 2020, pp. 549–565.
[15] I. Zhang et al., "The Demikernel datapath OS architecture for 937
microsecond-scale datacenter systems," in *Proc. ACM SOSP*, 2021, 938
pp. 195–211. 939
[16] S. Lee et al., "X-SSD: A storage system with native support for 940
database logging and replication," in *Proc. Int. Conf. Manag. Data*, 2022, 941
pp. 988–1002. 942
[17] X. Liao, Z. Yang, and J. Shu, "RIO: Order-preserving and CPU-efficient 943
remote storage access," in *Proc. 18th Eur. Conf. Comput. Syst. (EuroSys)*, 944
2023, pp. 703–717. 945
[18] "Interfaces to access PMR through SPDK." SPDK. 2023. [Online]. 946
Available: https://spdk.io/doc/nvme_8h.html 947
[19] "Buffer sharing and synchronization." Linux. 2023. [Online]. Available: 948
https://docs.kernel.org/driver-api/dma-buf.html 949
[20] X. Wei, X. Xie, R. Chen, H. Chen, and B. Zang, "Characterizing and 950
optimizing remote persistent memory with RDMA and NVM," in *Proc.* 951
USENIX ATC, 2021, pp. 523–536. 952
[21] D. Ongaro and J. Ousterhout, "In search of an understandable consensus 953
algorithm," in *Proc. USENIX ATC*, 2014, pp. 305–319. 954
[22] A. Kalia, M. Kaminsky, and D. G. Andersen, "FaSST: Fast, scal- 955
able and simple distributed transactions with two-sided RDMA 956
datagram RPCs," in *Proc. 12th USENIX Symp. OSDI*, 2016, 957
pp. 185–201. 958
[23] M. Xiao, H. Wang, L. Geng, R. Lee, and X. Zhang, "Catfish: Adaptive 959
RDMA-enabled R-tree for low latency and high throughput," in *Proc.* 960
IEEE 39th ICDCS, 2019, pp. 164–175. 961
[24] Q. Wang, Y. Lu, and J. Shu, "Sherman: A write-optimized distributed 962
B+tree index on disaggregated memory," in *Proc. SIGMOD*, 2022, 963
pp. 1033–1048. 964
[25] "LevelDB." 2023. [Online]. Available: https://github.com/google/leveldb 965
[26] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design guidelines for 966
high performance RDMA systems," in *Proc. USENIX ATC*, 2016, 967
pp. 437–450. 968
[27] Y. Luo, P. Jin, Q. Zhang, and B. Cheng, "TLBtree: A read/write- 969
optimized tree index for non-volatile memory," in *Proc. IEEE 37th* 970
ICDE, 2021, pp. 1889–1894. 971
[28] Y. Luo, P. Jin, Z. Zhang, J. Zhang, B. Cheng, and Q. Zhang, "Two 972
birds with one stone: Boosting both search and write performance for 973
tree indices on persistent memory," *ACM Trans. Embed. Comput. Syst.*, 974
vol. 20, no. 5s, pp. 1–25, 2021. 975