SC20: International Conference for High Performance Computing, Networking, Storage and Analysis | 978-1-7281-9998-6/20/$31.00 ©2020 IEEE | DOI: 10.1109/SC41405.2020.00056

RDMP-KV: Designing Remote Direct Memory Persistence based Key-Value Stores with PMEM

Tianxi Li*, Dipti Shankar*, Shashank Gugnani, and Xiaoyi Lu
Department of Computer Science and Engineering
The Ohio State University
{li.9443, shankar.50, gugnani.2, lu.932}@osu.edu

***Abstract*—Byte-addressable persistent memory (PMEM) can be directly manipulated by Remote Direct Memory Access (RDMA) capable networks. However, existing studies to combine RDMA and PMEM can not deliver the desired performance due to their PMEM-oblivious communication protocols. In this paper, we propose novel PMEM-aware RDMA-based communication protocols for persistent key-value stores, referred to as Remote Direct Memory Persistence based Key-Value stores (RDMP-KV). RDMP-KV employs a hybrid 'server-reply/server-bypass' approach to 'durably' store individual key-value objects on PMEM-equipped servers. RDMP-KV's runtime can easily adapt to existing (server-assisted durability) and emerging (appliance durability) RDMA-capable interconnects, while ensuring server scalability through a lightweight consistency scheme. Performance evaluations show that RDMP-KV can improve the server-side performance with different persistent key-value storage architectures by up to 22x, as compared with PMEM-oblivious RDMA-'Server-Reply' protocols. Our evaluations also show that RDMP-KV outperforms a distributed PMEM-based filesystem by up to 65% and a recent RDMA-to-PMEM framework by up to 71%.**

***Index Terms*—Next Generation Networking, RDMA, Data storage systems, Key-value Stores, Persistent Memory**

I. Introduction

Emerging persistent memory (PMEM) technologies, such as 3D-XPoint [23], phase change memory (PCM) [35], and STT-RAM [21], promise many advanced features, such as durable writes, byte-addressability, and a higher density than DRAM with a latency slowdown of about 2x–5x. These PMEM devices have led to significant re-design of many storage engines and systems, such as key-value stores with persistence support. Persistent key-value stores can leverage the byte-addressability of PMEM to enable durable in-memory structures and data objects, including, Echo [9], Pmem-Redis [3], and Apache Ignite [6].

On the other hand, many recent in-depth studies [12], [19], [20], [24], [28], [29] have been undertaken towards exploiting Remote Direct Memory Access (RDMA) capabilities on modern high-performance networks (e.g., InfiniBand, RoCE) to improve the end-to-end communication performance of volatile in-memory key-value stores. For instance, HERD [20] combines RDMA writes and messaging verbs to design a low latency single-RTT design for key-value store Put/Get operations. With recent research works [5], [17], [22], [27], [37] showing great potentials in combining RDMA and PMEM for designing high-performance distributed storage and file systems, there is significant scope for exploring how to fully utilize these technologies to benefit distributed persistent key-value stores.

A. Motivation and Challenges

As the starting point of these explorations, we ask the following question first: can existing RDMA-based protocols [12], [19], [20], [24], [28], [29] designed for accelerating volatile in-memory key-value stores be directly used to replace the communication engine in persistent in-memory key-value stores? This approach can, of course, be taken, but this simple approach cannot lead to the best utilization of 'RDMA+PMEM'. The reason is that existing RDMA-based protocols [20], [24], [29] for volatile key-value stores are oblivious to PMEM's byte-addressability and to the data durability needs of persistent in-memory key-value stores. Naively replacing the communication engine requires explicitly copying and persisting data from RDMA communication buffers to PMEM (DRAM-to-PMEM staging) in the PUT operation. Similar data staging overhead for GET will happen as well. These protocols diminish the benefits of zero-copy RDMA reads/writes and do not fully leverage the byte-addressability of PMEM. This situation leads to our first challenge: *Can we design PMEM-aware RDMA protocols for persistent in-memory key-value stores that can enable the clients to directly access and 'durably' update key-value pairs in servers' PMEM, to possibly avoid the 'DRAM-PMEM' staging?*

To overcome such noticeable data staging overhead between DRAM and PMEM, some advanced designs have also been proposed in the literature recently to enable PMEM-aware RDMA protocols. For example, Octopus [22] is a recent user-space RDMA-enabled distributed file system built for PMEM systems. Another example system, called Forca [14], comes close to enabling PMEM-aware key-value stores with built-in data consistency RDMA protocols. To our best knowledge, neither of these systems can provide strong data durability and efficient consistency at the same time. This means these systems cannot guarantee durability per key-value pair insert/update request, which is a key requirement for persistent in-memory key-value stores. Some other storage systems like Mojim [37] and Hotpot [27] expose distributed persistent memory to data center applications, and leverage one-sided and two-sided RDMA semantics to enable efficient RPC

*Tianxi Li and Dipti Shankar contributed equally to this work.

SC20, November 9-19, 2020, Is Everywhere We Are
978-1-7281-9998-6/20/$31.00 ©2020 IEEE

mechanisms and high-performance primary backup protocols. These studies provide a great basis for designing persistent storage systems, but their designs are tightly coupled with their system architectures, which cannot be directly leveraged to enable persistence of individual key-value object per request for different key-value store backend designs. Moreover, some of these designs, such as Mojim, are heavily reliant on special hardware support (experimental RDMA Commit capability[1]), which makes it hard for these designs to be adapted in many existing HPC environments.

In contrast to these existing studies, we find that designing high-performance and generic PMEM-aware RDMA protocols for persistent key-value stores is non-trivial. There is no existing work which can be directly used to design an efficient communication engine to adapt to different key-value store backend architectures and cluster environments. This situation further leads to the second important challenge: *Can we design efficient and generic PMEM-aware RDMA communication protocols for different persistent in-memory key-value store architectures, that can enable the system to ensure strong data durability for key-value pair updates while maximizing end-to-end performance?*

Third, since distributed key-value stores are typically accessed and updated by many concurrent clients, it is vital to ensure that any inconsistent updates to the key-value data in PMEM are identifiable, so that data consistency can be ensured during any types of failures. Efficiently achieving this becomes more challenging for our work because we allow clients to access remote server memory and directly update key-value data in the server's backend. In this case, the server backend processing could be bypassed and it may not be able to detect any failures. This problem is exasperated by the volatile cache hierarchy in front of PMEM. The cache hierarchy, through cache-line evictions, may change the order in which stores become persistent. This situation brings a significant challenge: *What kind of schemes can be designed to effectively and efficiently ensure remote data consistency without sacrificing the performance goals discussed above?*

B. Contribution

To address the above challenges, in this paper, we propose a novel PMEM-aware communication runtime for persistent key-value stores over modern RDMA-enabled networks, referred to as Remote Direct Memory Persistence based Key-Value stores (RDMP-KV). The key goal of RDMP-KV is to leverage RDMA to 'durably' store and access key-value pairs in the server's PMEM directly. To achieve this, RDMP-KV employs a hybrid 'server-reply/server-bypass' protocol to address the DRAM-to-PMEM staging overheads and enable remote direct PMEM access.

To enable strong data durability for every key-value pair insert/update (i.e., `PUT(k,v)`), RDMP-KV designs two communication protocols: (a) a two-sided 'Server-Assisted' protocol (referred to as RDMP-SA) for memory persistence via local cache flush operations, and, (b) a one-sided 'Client-Centric' protocol (referred to as RDMP-CC) via RDMA Commit [30], so that RDMP-KV can adapt to both current generation RDMA network adapters and emerging RDMA Commit capable networks.

To support data consistency, RDMP-KV proposes a novel light-weight consistency checking scheme by only introducing two one-byte flags appended to every key-value pair. Through careful and smart manipulation of the flag bytes, our scheme can effectively provide crash consistency with minimal performance overhead.

Overall, this paper makes the following novel contributions:

- We introduce RDMP-KV protocols and runtime for designing RDMA-aware persistent key-value stores for PMEM systems, that can avoid data staging overhead, while ensuring strong data durability, crash-consistency, and atomicity for every key-value pair stored. Unlike prior work, the proposed protocols allow clients the direct access to data on the servers using a novel two-flag consistency scheme to enable inline durability and consistency with minimal performance impact.
- RDMP-KV's communication protocols can adapt to current and emerging high-performance RDMA-capable networks. It is generic enough to support different PMEM-based persistent key-value store backend architectures. The proposed protocols have been carefully tailored to work with both currently available commodity RDMA NICs and future generation RDMA NICs. We showcase RDMP-KV's generality across three different persistent key-value store architectures based on a widely used key-value store, Pmem-Redis [2], [3].
- RDMP-KV protocols consider a holistic approach to designing RDMA- and PMEM-aware protocols to enable the server to scale with client-centric data processing. We demonstrate RDMP-KV's performance benefits over existing PMEM-oblivious RDMA protocols (i.e., HERD [20]) with extensive microbenchmark evaluations and the YCSB benchmark. We also show the benefits of RDMP-KV by comparing it with two state-of-the-art persistent memory storage solutions, Octopus [22] and Forca [14].

Performance evaluations on an InfiniBand EDR (100 Gbps) cluster show that our proposed RDMP-KV designs can improve the server-side performance with different persistent key-value storage architectures by 1.7x–22x for throughput and latency by up to 88%, as compared with the protocols used in HERD for PUT. We also observe a boost over HERD of 1.1x–9.5x in terms of throughput at server side and up to 57% improvement in terms of end-to-end latency for GET. In addition, we demonstrate an end-to-end performance gain of up to 79% for durable PUTs. For throughput, RDMP-SA and RDMP-CC gain up to 1.7x and 4.9x boost over RDMA-HERD for PUT and 1.7x for GET, with both update-heavy (50:50) and read-heavy (95:5) YCSB workloads running over 96 clients. Our evaluations also show that RDMP-KV

[1]RDMA Commit is not mature enough to be deployed on RDMA networks yet

outperforms Octopus FS and the Forca RDMA-to-PMEM framework by up to 65% and 71%, respectively, in terms of latency for PUT operations.

The rest of the paper is organized as follows. Section II presents our characterization and analysis of existing PMEM-based persistent key-value storage and PMEM-oblivious RDMA-based protocols. Section III presents the RDMP-KV communication runtime and our key-value store prototype based on Pmem-Redis. Section IV describes detailed evaluations. Section V discusses related works and Section VI presents our conclusion and future work.

II. Characterization and Analysis

As discussed in Section I, naively combining PMEM-oblivious RDMA-based protocols with PMEM-aware persistent key-value stores has overheads, such as DRAM-PMEM staging and data persistence. In this section, we use Pmem-Redis [3] as an example to quantitatively discuss these performance bottlenecks.

A. Pmem-Redis Design Overview

Persistent in-memory key-value stores are typically a combination of three main components: (1) an in-memory index such as a hash table or a B+-tree (Index), (2) memory pools of key-value pair data (KVData), and, (3) an append-only write-ahead log (WAL) [15] on the persistent layer which stores key-value data for recoverability. Every key-value pair insertion or update involves a PUT operation that allocates and writes a new key-value object into the 'KVData' pool, appends the PUT operation to the 'WAL', and flushes all PMEM-resident data for durability. Then, it updates the Index to point to the new key-value object. Every GET operation accesses the 'Index' to locate the key-value object and returns the value to the client, if any.

To give a concrete example, Figure 1 presents a typical single round trip protocol for durable PUT and GET operations in Pmem-Redis [3], which is an PMEM-aware persistent in-memory key-value store based on Redis [2]. In Pmem-Redis, 'KVData' is PMEM-resident and is accessed via a fast DRAM-resident 'Index'. An PMEM-based WAL compactly stores write requests by requiring only metadata and persistent key-value object pointer per log entry. Pmem-Redis employs Intel's Persistent Memory Development Kit (PMDK) [17] to allocate and manage persistent memory key-value objects, and does not rely on in-place updates. We use this as the basis for our study.

Due to the asymmetric read/write performance as compared to DRAM, PMEM mainly affects write operations. Hence, we focus our analysis on persistent key-value store PUT operations. Since Pmem-Redis does not have RDMA support by default, we first implement RDMA-based HERD protocol (HERD) [20] for it. We choose HERD's RDMA protocols because [20], [29] show that this approach ensures optimal RDMA usage with a single round-trip per PUT operation. Hence, we use 'Pmem-Redis+HERD' as the comparison baseline to represent PMEM-oblivious RDMA-based persistent

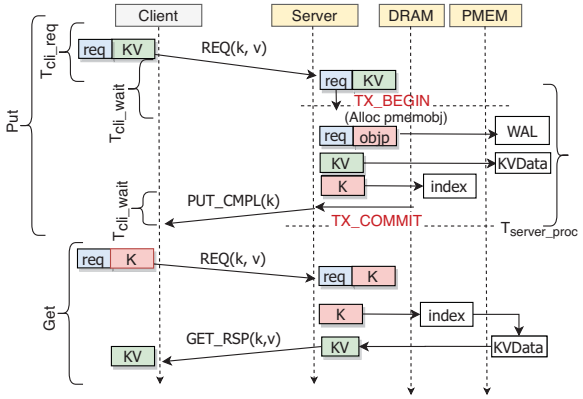


Fig. 1. PUT and GET in Pmem-Redis

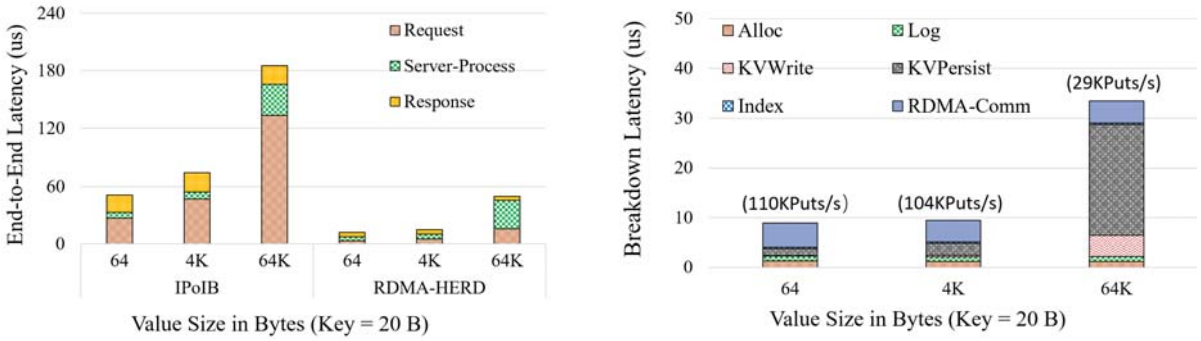
key-value stores. We believe this is a reasonably good baseline for the study.

Pmem-Redis over HERD first employs the one-sided RDMA-Write-with-Immediate (RWImm) operation to send the PUT and GET requests, i.e., `REQ(k, v)`, which is placed directly into a pre-allocated DRAM-resident RDMA buffer in the server communication engine (HERD). Then, the RWImm operation triggers the Pmem-Redis engine to process the request. Here, the key-value object data in DRAM-resident RDMA buffer needs to be staged into PMEM for PUT requests. Based on how the persistent key-value store backend is designed, different kinds of data and structures need to be updated in DRAM or PMEM. Typically, these updates need the transaction support to guarantee strong data durability and consistency, as shown in Figure 1 (i.e., TX_BEGIN and TX_END/COMMIT). Once the server process on the PUT request is complete, the server uses a two-sided RDMA SEND operation to reply to the client with the `PUT_CMPL(k)` message. For GET requests, the response including the data requested is sent using a two-sided RDMA SEND.

B. Performance Characterization

Based on Pmem-Redis over HERD described above, we first evaluate its performance with the write-only PUT workload using the `redis-benchmark` [26] utility. These experiments are performed on two nodes on Cluster A (described in Section IV-A) that is equipped with InfiniBand EDR (100 Gbps). We emulate PMEM over DRAM, as detailed in Section IV-A. A single Pmem-Redis server instance and a remote Pmem-Redis client are employed, with typical KV pair sizes [8] (value sizes 64 B, 4 KB, 64 KB and a fixed key of 20 B).

First, we measure the time-wise breakdown of End-to-End PUT latency in Pmem-Redis over both IPoIB and HERD protocols. Figure 2(a) presents the detailed breakdown results. From this figure, we can clearly see that the HERD-based RDMA protocol outperforms the IPoIB protocol by 73% – 79% for latency. This is a consequence of the impact of RDMA on improving communication performance. From Figure 2(a), we also find that for RDMA-HERD, the server-side processing time in PUT constitutes a large portion (33% – 59%) of



(a) Breakdown of PUT Latency

(b) Breakdown of Server Processing in PUT (with throughput)

Fig. 2. Pmem-Redis PUT Performance with PMEM-oblivious IPoIB and RDMA protocols

the PUT latency. Compared with IPoIB that spends most time in communication, server-side processing becomes the major bottleneck of RDMA-HERD after exploiting RDMA protocols.

we further breakdown the server processing per-PUT operation into five different major operations as shown in Figure 2(b), which are memory allocation in PMEM (Alloc), writing key-value data to PMEM-based memory (KVWrite), flushing key-value data from CPU cache to PMEM (KVPersist), WAL-based logging (Log) and index updating (Index). We also add server's response time (RDMA-Comm) for comparison.

From Figure 2(b), we observe that the costs of KVWrite and KVPersist keep increasing with key-value pair sizes and adversely affect the server throughput significantly (measured as KPuts/s as shown on top of the bars), especially for large key-value sizes. Also, the PMEM persist times are significantly larger than the RDMA communication time.

C. Analysis Summary

Through the above performance characterizations, we find that even though RDMA can reduce the communication overhead compared with IPoIB, the server-side processing time still poses a major bottleneck to the end-to-end performance. This makes it clear that, rather than exploiting RDMA only for communication, it helps to leverage RDMA to cut short the server side DRAM-to-PMEM staging that drastically limits the system's scalability.

Besides, data persistence time could take more than half of the processing time at large message sizes (e.g., 64 KB) that adversely affects scalability of server. By contrast, the memory allocation and index updating phases only account for a small and constant overhead. Therefore, it can be inferred that the overhead of ensuring PMEM persistence affects the server's performance. This implies we need a new scheme to reduce the persistence bottleneck.

In summary, we find that the widely adopted techniques in the RDMA community, e.g., RDMA-HERD protocols, designed for volatile key-value stores cannot ensure both low latency and high scalability in persistent key-value stores. We also pinpoint the bottleneck of staging and persistence. Therefore, it is vital to design PMEM-aware RDMA communication protocols that can alleviate server-side processing overheads due to durable PMEM writes. With this as the basis, we present our RDMP-KV designs in the following section.

III. RDMP-KV Design

In this section, we present the detailed design of the RDMP-KV runtime and its corresponding RDMP protocols. We also present an overview of adapting RDMP-KV with different key-value store backends through co-designs.

Figure 3 provides an overview of RDMP-KV's runtime architecture. RDMP-KV's runtime consists of three main components to manage and achieve the three goals laid out in Section I: (a) durability, (b) data consistency and recoverability, and, (c) metadata engine for log, index, and memory management. We discuss the functionality and design of each of these components below.

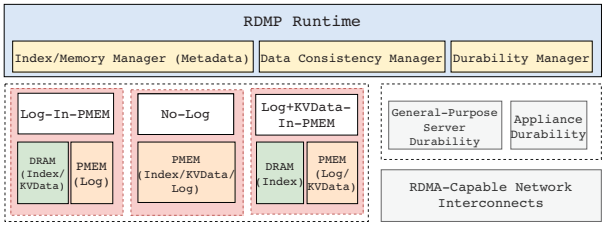


Fig. 3. Overview of RDMP-KV Runtime Architecture

A. Remote Direct PMEM Access

To avoid the 'DRAM-to-PMEM' staging overheads of PMEM-oblivious RDMA protocols, we need to enable persistent in-memory key-value stores to leverage RDMA to durably store and access key-value pairs directly from the server's PMEM (i.e., server bypass for data access). This requires the client to know the PMEM address of the key-value object that needs to be stored or retrieved. In order to enable this, RDMP-KV employs a hybrid two phase 'Server-Reply/Server-Bypass' approach to separate the data and metadata (log, index, and memory) management involved to store or retrieve a key-value

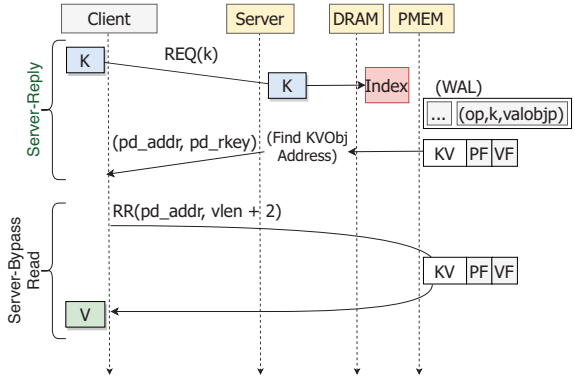


Fig. 4. RDMP-KV Protocol for GET ('RR' means RDMA-Read)

pair in PMEM. We choose a two phase approach as opposed to a one phase (single RTT) approach for offloading processing from the server to the client and increase server throughput. The end-to-end communication protocol we propose involves two phases:

(1) Server-Reply Metadata Phase: In this phase, a two-sided 'Server-Reply' (RPC-like) protocol is employed to perform metadata management at the server and to retrieve the PMEM key-value object address from the server (client request and server response are posted using two-sided RDMA-Sends). For PUT(k,v), metadata management at the server involves updating local DRAM and PMEM-resident data structures, including the allocation from PMEM pool, recording key and meta information like value size, persistently appending the request to the write-ahead log and updating the index prior to reply to the client with kv pair address. Server will also save the allocated KV data address and length in a data structure for the 'Server-Bypass' phase. For GET(k)[2], metadata management involves index lookups to locate the key-value object and consistency check. Figure 4 shows the Get protocol in detail.

(2) Server-Bypass Read/Write Phase: In this phase, the RDMP-KV enables the client to read or write directly from PMEM, with the address received from the 'Server-Reply' phase. For PUT requests, RDMP-KV also initiates remote durability steps from the client-side, to ensure durability for every key-value object (detailed in Section III-B).

An overview of the end-to-end RDMP-KV protocols is presented in Figure 5. With the above hybrid approach, RDMP-KV can be leveraged on both current and emerging RDMA-capable networks by enabling remote durability support via two mechanisms discussed in the literature [11]:

(1) General Purpose Server Durability (GPSD), wherein, the host performs an RDMA Write into remote side's PMEM to explicitly notify a dedicated process to: (a) flush all cache-lines associated with the specified data and invoke a persistent store barrier, and, (b) notify the initiator on completion of persistence.

[2]We refer to RDMP-KV Get Protocol as "RDMP-GET" for our evaluations

(2) Appliance Durability (AD), wherein, hardware-assisted commands like **RDMA Commit** [30] can be leveraged. Its semantic is similar to that of a one-sided RDMA Read operation, but it initiates the RDMA-capable network adapter on the remote target to force data to be flushed to the PMEM durably, directly via PCIe.

B. Remote Data Durability

The 'Server-Bypass Write' phase discussed above typically involves two steps: (a) using RDMA to write key-value objects directly into PMEM without involving the server, and, (b) ensuring remote durability. As presented in Figure 5, RDMP-KV can ensure durability for PUT operations via either a 'Server-Assisted' or a 'Client-Centric' approach, described in detail below.

1) Server-Assisted RDMP Protocol (RDMP-SA): As the starting point, we leverage the GPSD mechanism for durability. We refer to this persistence protocol as 'Server-Assisted RDMP Protocol' (**`RDMP-SA`**) and can be easily enabled on current-generation RDMA-capable networks. Figure 5(a) illustrates the RDMP-SA communication protocol, and its interaction with the PMEM-based backend. As seen in this figure, ensuring durability per PUT requires an additional round trip. It involves a request/response protocol using the one-sided RDMA-Write-with-Immediate, wherein, the client specifies the index of server's saved data structure that has the PMEM-resident object's address and the length to be 'flushed-for-persistence'. At last, the server responds with the status of the completed flush.

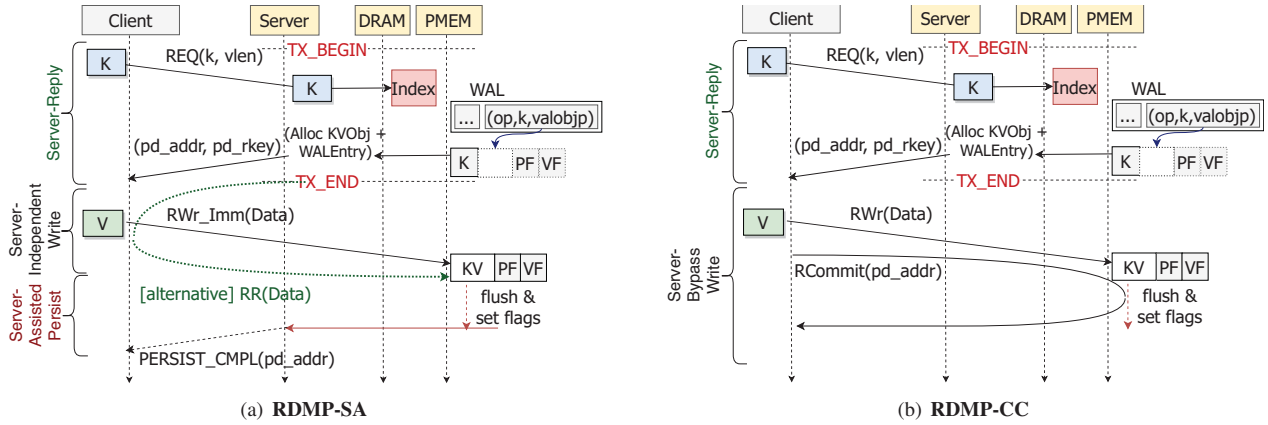
RDMP-SA enables offloading RDMA-based PMEM writes either: (a) at the client's RDMP communication engine via RDMA-Write, or, (b) at the server's RDMP engine via RDMA-Read from the client's communication buffers. Since we are focused on scalable server designs, we choose to offload the 'Server-Bypass Write' phase to the client's RDMP runtime(i.e., posting RDMA writes to client's network adapter).

2) Client-Centric RDMP Protocol (RDMP-CC): In order to maximize server's scalability, it is vital to alleviate the overhead of persistence incurred at the server processes. This can be enabled via a one-sided server-bypass approach employed by the AD mechanism, via hardware-assisted commands like RDMA Commit [30].

Figure 5(b) illustrates the RDMP-CC protocol, and its interaction with the PMEM-based backend. As seen in this figure, RDMP-CC enables a truly one-sided 'Server-Bypass Write Phase'. The client-side RDMP-KV runtime posts an RDMA Commit to the underlying RDMA-capable network adapter, and the server-side network hardware can directly flush the PMEM-resident object to persistence via the PCIe. Based on these, RDMP-CC will enable the server to offload both data writes and durability duties to the client.

C. Data Consistency and Recoverability

RDMP-KV separates the metadata and data update phases (as discussed in Section III-A) for enabling direct PMEM



(a) **RDMP-SA** (b) **RDMP-CC**

Fig. 5. RDMP-KV Protocols for PUT ('RWr': RDMA-Write, 'RWr_Imm': RDMA-Write-with-Immediate)

access. Hence, we need an explicit mechanism to ensure data consistency during the 'Server-Bypass Read/Write' phase, for managing concurrent reads/updates and for the correctness of data recovery.

Both PMEM persistence instructions (e.g., `clwb`) and one-sided RDMA operations guarantee atomicity only at the granularity of a word, i.e., 8 B [7]. This is because spurious cache-line eviction can change the order of stores to PMEM. Therefore, we have the following issues that will result in inconsistency in terms of a single KV operation:

- Once the data arrives at the remote side (i.e., the server), any cache lines can be written back to persistent memory implicitly in an out of order manner. This can happen before the server initiates persistence.
- Even in persistence function calls, there is no guarantee that data will be flushed to the PMEM device in order, i.e., segments can be re-ordered by the cache hierarchy. In case the KV pair data buffer spans multiple cache-lines, (i.e, KV size >64 B), `clflushopt`, `clwb`, etc., do not guarantee in order data flushing either.

In order to maintain the consistency of the KV store system, we exploit the basic 8 B atomicity capabilities of the underlying RDMA hardware to design a light-weight, efficient scheme that ensures data consistency for updates via RDMA Write. We employ **two additional one-byte flags per KV pair: persist flag and valid flag**. The additional bytes, referred to as **'flag bytes'**, are allocated after KV pair from the PMEM-based memory pools. We elaborate and illustrate this mechanism in Figure 6, which presents a snapshot of the DRAM and PMEM data placement in RDMP-KV with the two flags, for consistent updates to KV pair objects in PMEM below.

During **'Server-Reply Metadata'** phase, the allocated 'flag bytes' are persistently set to zero before the PMEM-resident object address is returned by the server's RDMP-KV runtime (i.e., `kvlen+2` per KV pair). Since these metadata-related PMEM-resident structures can be transactionally updated by the server (e.g., PMDK's pmemobj [1]), any failures can be rolled back or undone. For instance, Case ① in Figure 6 illustrates a failure during this phase for PUT(K3, V3). This request does not get persisted to the WAL, and the failure is returned to the KVS client (instead of a PMEM address location).

For **`RDMP-SA`**, once the server receives a request to persist a key-value pair, it first persists (flush + store fence) all cache-lines except for the one containing the persist flag (PF). Once the cache-lines are persisted, the PF is set and the cache-line containing it is persisted in an atomic manner. Finally, the valid flag (VF) is set to indicate that the key-value pair is persistent and is safe to read. The VF is required to guarantee the consistency in scenarios involving concurrent PUT and GET operations to the same key. If the system only has PF, at the point when PF is set but yet persisted, a GET operation will only check PF and assume the data is valid. However, if the system crashes at this moment, on recovery the KV pair will be aborted since PF is unset but a client has already fetched the non-existent data. Therefore, checking the PF does not guarantee that all data has been persisted. It is important to note that the VF is not required to be persisted explicitly because it is only used by GET requests to ensure that the value read is already persistent. On recovery, only the PF needs to be checked because the PF is only set and atomically flushed once all other data cache-lines have been flushed.

For **`RDMP-SA`**, as the server is involved, the flags can be set once the server flushes the KV object to PMEM. However, for **`RDMP-CC`**, we need to leverage additional **`RDMP Commit`** semantic extensions recently proposed for emerging RDMA-capable networks [31]. We discuss these in Section III-D. With 'no in-place updates', there is only one KVS client updating the 'flag bytes'. Since 8 B atomicity is guaranteed for x86 cache-line flushes, setting these flag bytes can clearly indicate that the data has been consistently written and persisted by a remote PM-KVS client. For instance, Case ② in Figure 6 illustrates an ongoing 'Server-Bypass Write' phase for PUT(K2, V2), as the corresponding VF is still zero, i.e., a PUT-in-progress.

During any subsequent **'Server-Bypass Read' phase** (for GET), as shown in Figure 4, the client RDMP-KV runtime posts the RDMA Read of size (`kvlen+2`) and checks the

trailing byte, i.e., the VF, to ensure that it reads a consistently updated KV pair.

If the VF is zero (as in Case ② in Figure 6), then we can infer that the 'Server-Bypass Write' phase of the same KV pair may not be successfully completed (i.e., GET(k) is considered as failed). The PM-KVS server is queried again via RDMA Read to retrieve the correctly updated data.

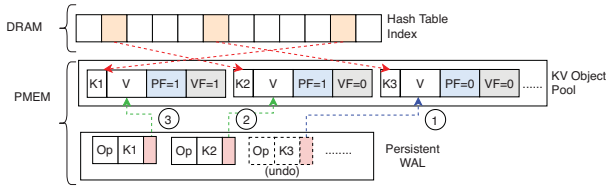


Fig. 6. RDMP-KV Memory Layout and Data Consistency Example: illustrated with three PUT(Kx,Vx) operations (PF = 'persist flag', VF = 'valid flag')

For data recoverability, in case of node or process failures, the write-ahead log in PMEM (if any) is typically replayed. During this recovery process, only those PMEM-resident objects with valid PF are retrieved and used to reconstruct the DRAM-resident index. For instance, from Figure 6, Cases ② and ③ illustrate that PUT(K1, V1) and PUT(K2, V2) have successfully persisted and are replayed. Note that PUT(K3, V3) was incomplete in 'Server-Bypass Write' phase, and is hence rolled back via undo from WAL (represented with dotted-lines in Case ①). An entry for such PUT operations will be non-existent in the PMEM-based WAL when replayed in case of failures.

D. Remote Data Consistency Extensions for RDMP-CC

Based on the **`RDMA Commit`** extensions proposed in [30], [31], we have the following options to enable remote consistency via the 'flag bytes' for RDMP-CC protocol:

1) **`RDMP-CC-SA`**: Similar to RDMP-SA, client can initiate the update of the 'flag bytes' per KV object by explicitly sending a request to the KV server. Server will return the status via a two-sided RDMA-Send. We refer to this as RDMP-CC Server-Assisted or RDMP-CC-SA. Evidently, this requires an additional RTT for end-to-end PUT completion.
2) **`RDMP-CC-Imm`**: Like RDMA-Write-with-Immediate, 'immediate data' can be optionally provided to signal the upper layer on the remote peer (i.e, the KV server) when RDMA Commit completes. This signal from the RDMA completion queue can be used to set the flags atomically. Likewise, the status will be returned from the server.
3) **`RDMP-CC-ADP`**: The proposed extensions also suggest an additional optional payload (ADP) of 8 B to be placed and made durable in an atomic fashion after the requested commit. This is ideal for our 'flag bytes'-based remote consistency protocol. We emulate RDMA COMMIT in this mode by issuing an RDMA-Read after RDMA-Write the value with the additional 'flag bytes'. Therefore, for our evaluation we use this method.

By studying this futuristic **`RDMA Commit`** protocol and its extensions, we enable RDMP-KV designs to work optimally with current and emerging PMEM and RDMA-capable hardware.

E. Pmem-Redis Co-Design with RDMP-KV

To demonstrate RDMP-KV's effectiveness and backend-agnostic capabilities, we enable three candidate PMEM-aware backend architectures, as shown in Figure 3, by implementing our prototype based on Pmem-Redis (Redis+PMDK) [3] as follows:

(1) **Log+KVData-In-PMEM**: This approach (default in Pmem-Redis) places both the WAL (undo-based log) and key-value pairs in PMEM, and accesses them via a DRAM-resident hash table. The log will only track the addr of kv pair in PMEM instead of the value.

(2) **Log-In-PMEM**: To maximize DRAM utilization (as in traditional disk-based persistent data stores), this approach extends the PMEM-resident WAL in Pmem-Redis to include the actual key-value pairs, and maintains key-value object pools in DRAM along with the index. We tweak RDMP-KV to update both the log entry in PMEM and the key-value data in DRAM via RDMA-Writes.

(3) **No-Log**: To maximize PMEM utilization, the DRAM-resident index (e.g, Pmem-Redis hash table) is moved to PMEM [4]. Since the index can be persisted during the 'Server-Reply Metadata' phase, no WAL is required to ensure data durability and recovery. RDMP-KV can still be leveraged to read/write key-value objects directly from the PMEM.

To enable the 'Server-Reply Metadata' phase, we register callback functions with the RDMP-KV runtime for: (1) metadata management (allocation + index updates) for PUTs, and (2) index lookup for GETs. For the RDMP-SA protocol, a callback is provided for invoking local 'flush-for-persistence' operations.

In order to boost server processing, after system analysis, we manage to take advantage of the benefits from RMDA-HERD on our server assisted design RDMP-SA. In the hybrid RDMP-SA design, we enable server to handle PUTs in the RDMA-HERD mode and reduce communication to a single RTT under a certain value size threshold (e.g., 16 KB). Thus, RDMP-SA can achieve similar performance as RDMA-HERD when KV sizes are small. PMEM-resident object pools allocated via PMDK are registered with the RDMA network adapter on a per-pool basis at server initialization phase. Based on this prototype, we present a detailed evaluation in the following section.

IV. Performance Evaluation

We present the results of our in-depth analysis of the proposed RDMP communication runtime and protocols for PMEM-aware RDMP-KV (described in Section III-E), and contrast it with HERD's one-RTT PMEM-oblivious RDMA-'Server-Reply' protocol over Pmem-Redis. We divide our

evaluations into the following sections:
(1) Evaluations with Redis PUT/GET Micro-benchmarks.
(2) Evaluations with YCSB benchmark.
(3) Comparison with other RDMA-over-PMEM works in the literature and case study to demonstrate RDMP-KV's backend agnostic capabilities

A. Experimental Setup

In this sub-section, we present the HPC cluster configurations used for our experiments, along with the configuration of the PMEM-over-RAMDisk emulation employed.

Cluster Setup:
(1) For most benchmarks, we use a testbed (Cluster A) equipped with two Linux servers for our experiments. Each node is provisioned with a 28 core Intel Skylake processor (Gold6132@2.6GHz), 192 GB DRAM and connected via EDR InfiniBand (100 Gbps). We dedicate 64 GB RAMDisk for PMEM emulation on the server node.
(2) For YCSB benchmark, due to hardware limitation in Cluster A, we set up another cluster (Cluster B) with six nodes. Each node is equipped with a 28 core Intel Broadwell processor (E5-2680v4@2.4GHz) and 128 GB DRAM. The cluster is interconnected with EDR InfiniBand (100 Gbps). Besides, 32 GB RAMDisk is reserved for PMEM emulation on each server node.

Emulating PMEM: We use Intel PMDK 1.8 over DRAM via RAMDisk, to create and maintain byte-addressable persistent memory pools in PMEM. To emulate the slower-than-DRAM writes, we employ the same emulation technique used in the widely accepted work on persistent memory, mnemosyne [32] for our experiments. For our emulation:
(1) We introduce an additional delay after each write into PMEM, using hardware access macros (`clwb`). We also insert the delay after each memory fence (`sfence`) to account for the wait involved for persisting outstanding writes to PMEM. We model the write bandwidth by inserting appropriate delays after every write sequence completes, to limit the effective bandwidth.
(2) As in [32], we add 150 ns of extra latency and limit the write bandwidth to 4GB/s unless otherwise noted. Since PMEM reads are likely to be serviced from the cache, we do not account for additional latency during reads.
(3) **`RDMA Commit`** is emulated by first issuing an RDMA Read at the address to be persisted, to the NIC at the server node and following operations based on different emulation variations (detailed in Section III-D).

We only add the additional delay to emulate slower-than-DRAM latencies persistent objects in PMEM, that are accessed via the PMDK library (e.g., WAL entries and KV pairs in default Pmem-Redis; Redis's DRAM-resident hash table is not affected). Unless otherwise specified, we use RDMP-KV runtime backed by default Pmem-Redis [3], that employs the 'Log+KVData-In-PMEM' backend design (detailed in Section III-E), to contrast the performance of RDMP-KV protocols with the state-of-the-art PMEM-oblivious RDMA-'Server-Reply' protocol, HERD [20]. In contrast to RDMP-KV, HERD employs an RDMA-enabled single round-trip design for both PUT and GET.

RDMP-KV protocols are designed to ensure inline persistence for every PUT. Hence, all of our experiments involving PUTs demonstrate persistence. Additionally, since recovery from persistent memory is not the focus of this paper, we mainly focus on the performance of persistent PUTs and server scalability.

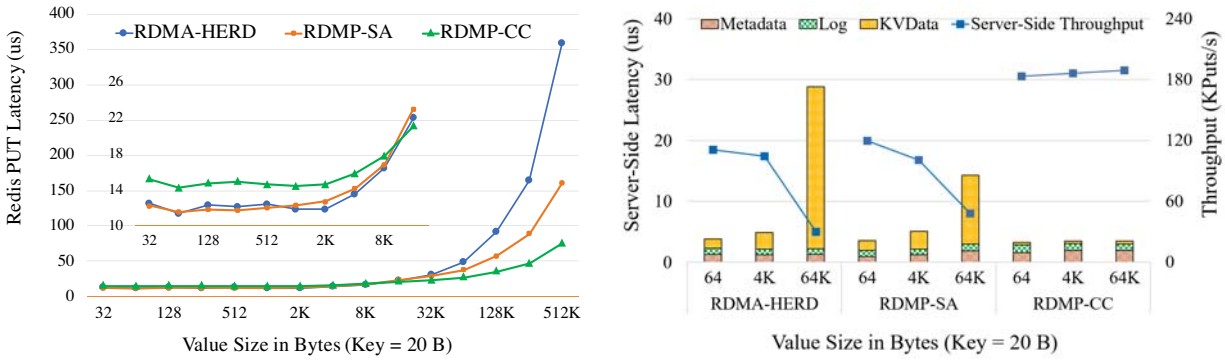
B. Redis Micro-benchmark Studies

First, we extend the evaluations in Figure 2(a), to study the proposed RDMP-KV designs. We contrast RDMP-KV's persistent PUT protocols (RDMP-[SA/CC]) with the PMEM-oblivious RDMA-based HERD protocol [20] (RDMA-HERD).

1) End-to-end PUT Latency: For this test, KV pairs are generated and loaded into the persistent in-memory key-value store by PUT; and the average end-to-end (p2p) latency is measured. From Figure 7(a), we observe that for KV sizes less than 16 KB, our proposed hybrid RDMP-SA design can achieve similar PUT performance to that of RDMA-HERD, while RDMP-CC has an overhead from 8% to 24% which results from an additional round trip of communication. For larger KV pairs ($\geq$16 KB), we can observe that as compared with RDMA-HERD, RDMP-SA improves PUT latencies by up to 55% and RDMP-CC by up to 79%, for Pmem-Redis. Upon further analysis, we observe that replacing DRAM-to-PMEM staging with communication-involved hybrid 'Server-Reply/Server-Bypass' schemes does not incur additional overheads to client-side latencies. This is because avoiding staging leads to significant latency reduction despite the need for another round trip, as we observe from Figure 2(b). This design helps lower latency for large values and increase throughput for all value sizes.

2) Server-Side Performance Analysis: To study the server-side performance, similar to Figure 2(b), we measure the Redis server per-instance throughput for the PUTs during the above experiment. From Figure 7(b) (line-graph), we observe that the RDMP-SA design can achieve 1.6x speedup in PUT throughput over RDMA-based HERD protocol at value size of 64KB, by offloading key-value data updates to the client. Most importantly, we observe that RDMP-CC can achieve 1.7x–6.4x speedup over HERD protocol and 1.5x–3.9x speedup over the server-assisted RDMP-SA protocol, by offloading both key-value data updates and durability steps to the client. We further extend our evaluation to the value size of 256KB and observe that RDMP-CC can maintain the throughput of $\sim$185 KPUTs/s while the throughput of RDMA-HERD continues to drop to 9 KPUTs/s. Hence, the speedup of RDMP-CC over RDMA-HERD can reach up to 22x for throughput. To clearly show the breakdown numbers in Figure 7(b) for the cases of small value sizes (e.g, 64B), we choose to not include these results in the figure.

For further clarity, we present the breakdown of the server-side processing time (similar to Figure 2(b)) into: (1) Metadata (allocating from PMEM pool with flag bytes set to zero + hash table updates), (2) Log (updating WAL persistently),



(a) End-to-End Latency

(b) Server-Side Throughput and Latency Breakdown

Fig. 7. End-to-End and Server-Side Performance for PUT

and, (3) KVData (write to key-value pair objects in PMEM persistently) phases. This is also presented (by the stacked-bars) in Figure 7(b). From this figure, we can observe that both RDMA-HERD and RDMP-SA designs are affected by the overhead of persistence at the server, which increases along with the increasing KV pair sizes. However, RDMP-SA attains benefits in terms of throughput especially for large key-value sizes, as the DRAM-to-PMEM staging overheads increase considerably for PMEM-oblivious protocols like HERD. Also, as shown in Figure 7(b), the numbers of 'Metadata' show RDMP-SA/-CC protocols incur a smaller overhead as compared to RDMA-HERD, as RDMP-KV's 'Server-Reply Metadata' phase involves allocating and persisting PMEM-resident 'flag bytes' for ensuring consistent updates and saving KV pair's address in PMEM for persistence in the 'Server-Bypass' phase. However, in RDMP-CC protocol, server only copies key and meta information. Thus, the major bottleneck of 'KVData' in RDMA-HERD and RDMP-SA at server side becomes negligible in RDMP-CC. Hence, RDMP-CC based RDMP-KV design for Pmem-Redis can maintain a constant server-side PUT throughput (∼185 KPuts/s with little overhead from 'KVData' phase).

3) End-to-end GET Latency: For completeness of micro-benchmark studies, we study the key-value store GET performance. We contrast 'RDMP-GET' in RDMP-KV design with the PMEM-to-DRAM staging-based GET in RDMA-HERD (over default Pmem-Redis). We load the Redis instance with KV pairs and run the `redis-benchmark` [26] GET workload. Figure 8 presents the GET latencies for varying value sizes (key size fixed to 20 B). From Figure 8, we observe that RDMP-KV performs similar to HERD for small value sizes (<16 KB), and improves GET latencies by up to 57% for value sizes ≥16 KB. To further understand this, from the server-side GET throughput in Figure 8, we can see that with the RDMP-KV GET protocol, the throughput remains steady around 225 KGets/s, irrespective of KV sizes. In contrast, the performance of RDMA-HERD deteriorates from 225 KGets/s to 24 KGets/s as the KV sizes increase, due to the need for PMEM-to-DRAM staging. RDMA-HERD is based on a one round-trip, server-centric design which incurs PMEM-to-DRAM staging overhead. Our design, in contrast, is based on a two round-trip, client-centric data fetching protocol which avoids the staging overhead. Therefore, our design may incur a small overhead from communication for small messages, which is why HERD marginally outperforms RDMP-KV in latency for <16 KB value sizes. However, note that RDMP-KV has higher throughput even for small message sizes by reducing server side processing.

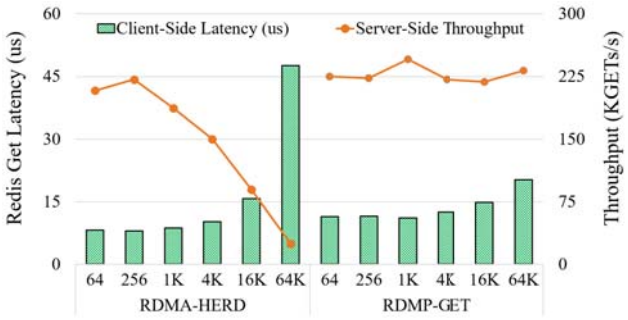


Fig. 8. End-to-End Performance for GET

C. Evaluations with YCSB

To illustrate the applicability for online data processing workloads, we study the performance of RDMP-KV with the YCSB benchmark [10]. We use Cluster B to conduct the experiments and employ two nodes to launch 24 server instances and 4 nodes to run a total of 96 clients. Each server instance reserves 4 GB DRAM and 4 GB of PMEM. Using the YCSB benchmark suite, we generate and load KV pairs into the Redis server instances. Analysis of real-world workloads [8] shows that most values are in the range of 64 B–64 KB. Based on this observation, we choose to generate PUT and GET requests with variable KV pair sizes. Value sizes are chosen from 64 B, 1 KB, 4 KB, or 64 KB with a uniform distribution. And key sizes are selected evenly as either 12 B, 16 B, or 20 B.

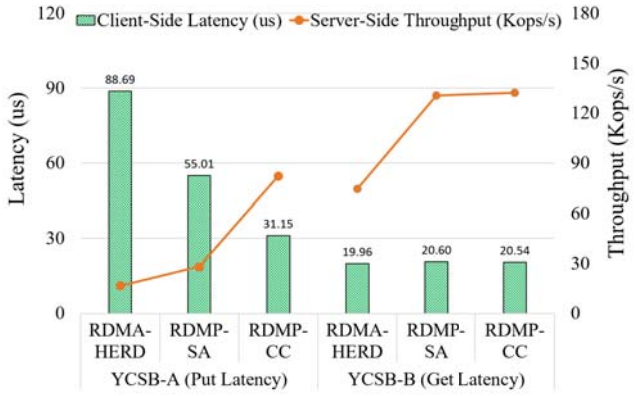


Fig. 9. Performance with YCSB Workloads A (50:50 read:write) and B (95:5 read:write). Contrasting RDMP-KV vs. RDMA-HERD over Pmem-Redis running 96 Clients over 24 server instances

Figure 9 presents the average end-to-end PUT/GET latencies with YCSB-A (write-heavy) and YCSB-B (read-heavy) workloads. We can see that RDMP-SA and RDMP-CC RDMP-KV designs improve the PUT latencies by about 38% and 65% over RDMA-HERD, respectively, with YCSB-A workload while deliver similar performance in terms of GET latency with YCSB-B workload. From Figure 9, it can also be seen that RDMP-KV can maintain higher total PUT/GET throughput, as compared with RDMA-HERD even in a multi-client environment. For PUT operation, the total throughput of RDMP-SA and RDMP-CC exceeds RDMA-HERD by about 1.7x and 4.9x, respectively. With YCSB-B, both RDMP-SA and RDMP-CC deliver similar performance for latency as RDMA-HERD and both have enabled an improvement of ∼1.7x for throughput by enabling the client to fetch the value through an RDMA-Read. In this manner, the PMEM-to-DRAM staging at the server side can be avoided.

D. Comparison with other RDMA-over-PMEM frameworks

To better understand the need for RDMP-KV to enable RDMA-accelerated persistent in-memory key-value storage, we contrast RDMP-KV with two recent RDMA-enable persistent memory storage systems that provide user-space abstractions to store and manage PMEM-resident data via RDMA. Similar to RDMP-KV, these frameworks use 'Server-Reply'-like (i.e., RPC-based) mechanism to exchange PMEM addresses and update metadata information, and use RDMA to read from and write to PMEM directly. To compare RDMP-KV with an RDMA-over-PMEM system that does not provide strong durability guarantees per KV pair and efficient data consistency, we employ Octopus [22], a recent RDMA-enabled distributed PMEM-aware file system. It provides open/close APIs for managing metadata via RDMA-based RPC and write/read file APIs to directly access PMEM on server nodes. We mimic the behavior of a key-value store by storing key-value pairs in individual Octopus files identified by key as the filename. By default, Octopus does not provide inline persistence guarantees for every update of KV pair (denoted as 'Ocotpus (Default)'). So, we explicitly add a persistence barrier to ensure data is written through for each PUT and measure its latency (denoted as 'Octopus (Persistent Put)'). To compare RDMP-KV with a recent RDMA-over-PMEM framework that provides data consistency guarantees but in a different manner, we use Forca [14]. Forca uses self-verifying CRC checksums to verify if the key-value pairs in PMEM are consistent. To enable consistent PUT and GET operations: (a) a checksum is calculated at the client and written into server memory via RDMA for PUTs, and, (b) a checksum is calculated by the server for every GET prior to returning a key-value pair address. We implement Forca in Pmem-Redis with the same checksum consistency guarantee. Thus, we can have a fair comparison between Forca and RDMP-KV protocols.

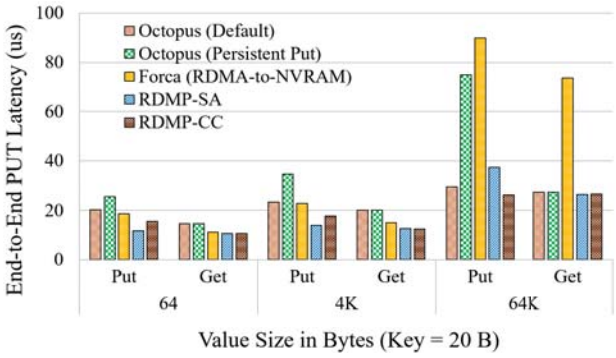


Fig. 10. Comparison with state-of-the-art RDMA-enable persistent memory systems. Contrasting end-to-end PUT/GET latency RDMP-KV with Octopus FS [22] and Forca [14]

For this experiment, we measure the end-to-end latencies for PUT and GET operations, between a single client and server. For RDMP-KV, we present the average PUT/GET latencies for both RDMP-SA and RDMP-CC modes. From Figure 10, we observe that for sizes of 64B and 4KB, RDMP-SA and RDMP-CC can improve PUT latency by about 40% and 24%, respectively, over default Octopus despite the need to guarantee data durability and consistency for every PUT request. In addition, both designs enable an improvement of ∼33% for GETs. For 64KB values, RDMP-CC gains an improvement over default Octopus of 11% and 65% over persistent Octopus. Though default Octopus performs better than RDMP-SA for 64KB values but on adding a persistence barrier we can observe that its performance is actually worse by 50% than RDMP-SA. Similar GET latencies of Octopus and RDMP-KV for larger key-value pairs are observed as RDMA communication time dominates the metadata access/update time at value size of 64KB. In comparison to Forca that incurs a checksum computation overhead for every PUT and GET, RDMP-CC can improve PUT latency by up to 71% and GET latency by up to 64% by leveraging the 'flag bytes'-based consistency mechanism with durability guarantees per PUT operation.

E. Backend-Agnostic RDMP-KV Design

Figure 11 extends the end-to-end PUT latency test in Figure 7(a) to include RDMP-KV based Pmem-Redis with Log-

In-PMEM, No-Log, and Log+KVData-In-PMEM backends discussed in Section III-E. From Figure 11, we can observe that:

(a) For small KV pairs ($\leq$4 KB), the RDMP-SA protocol achieves similar PUT latencies compared with RDMA-HERD for all three kinds of backends. RDMP-CC in the 'Log-In-PMEM' mode incurs more overheads in communication at small value size (64B), which is because RDMP-CC needs an additional RDMA-Write to update remote DRAM data. In the meantime, RDMP-CC shows similar performance in terms of latency as RDMA-HERD in both 'No-Log' and 'Log+KVData-In-PMEM' backends.

(b) For larger KV pairs (64KB), we observe that as compared with RDMA-HERD, RDMP-SA improves PUT latencies by up to 30% for all three backends. RDMP-CC can improve end-to-end latency for PUT by up to 52% in 'No-Log' and 'Log+KVData-In-PMEM' modes by minimizing persistence overhead. RDMP-CC can also reduce latency by 44% in 'Log-In-PMEM' mode. For large value sizes, we see much better performance improvement with RDMP-KV designs, which is mainly because RDMP-KV can alleviate the data staging and persistence overhead in RDMA-HERD as discussed in Section IV-B.

The results demonstrate the superiority of RDMP-SA and RDMP-CC protocols over HERD protocol and indicate a similar trend observed in previous evaluation in 'Log+KVData-In-PMEM' backend.

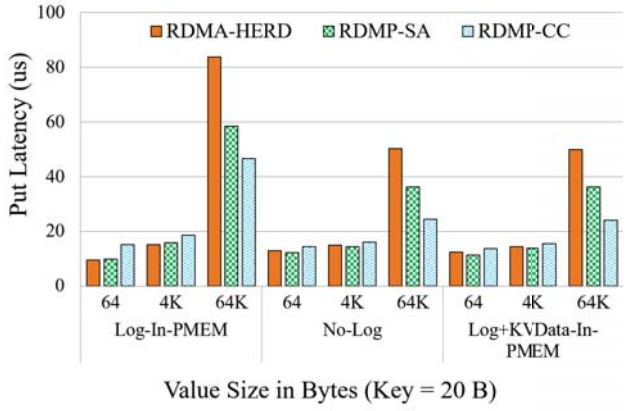


Fig. 11. End-to-End Performance with PMEM-based Backend Alternatives

In addition, by comparing three backends, for RDMA-HERD and RDMP-SA, 'Log-In-PMEM' outperforms the other two backends when message sizes are small (<4KB) because it stores KV data in DRAM with faster access than PMEM. However, the performance deteriorates drastically when value gets larger since it maintains two copies of KV data in DRAM and PMEM as the log. Thus, in RDMA-HERD and RDMP-SA, server needs to do an additional memory copy after client writes values to PMEM log buffer.

'No-Log' and 'Log+KVData-In-PMEM' deliver similar performance. 'No-Log' keeps an individual persistent hashmap [4] and gets rid of WAL since all components are kept persistently. 'Log+KVData-In-PMEM' logs KV pairs' addresses in PMEM and each logging process takes almost constant time when value size changes. It has been evaluated that updating the persistent hashmap in 'No-Log' mode takes slightly longer time than the index update in 'Log+KVData-In-PMEM' mode. On the other hand, 'No-Log' saves time of writing WAL. Hence the test results of 'No-Log' and 'Log+KVData-In-PMEM' are comparable.

Overall, in certain applications where value sizes are mostly small (e.g., 64B), 'Log-In-PMEM' with RDMA-HERD or RDMP-SA would be a better option. In other cases, 'No-Log' and 'Log+KVData-In-PMEM' with RDMP are better candidates.

V. Related Work

Emerging PMEM devices (e.g., PCM, 3D-XPoint, etc.) are paving the way for novel middleware designs on modern HPC clusters. On the other hand, several RDMA-based volatile memory key-value stores have been proposed [12], [19], [20], [24], [28], [29], [34] in the community. We have discussed and compared with several closely related works in Section I and Section IV. In this section, we further discuss more related studies in three categories: PMEM-aware Middleware, RDMA over PMEM, and combining fast DRAM and slower PMEM.

PMEM-aware Middleware: The potential of leveraging PMEM in HPC and data center middleware has come into the spotlight over the recent years [13], [32]. Several research works [15], [25] have been studying the potential of employing PMEM's memory semantics and persistence features to accelerate online transaction processing systems. Bullet [16] is a single-node persistent key-value store which relies on a DRAM frontend to improve performance. These works are indicative of the potential for designing efficient in-memory computing middleware with PMEM. However, these works are single-node systems and do not consider RDMA-based distributed designs, which is the focus of this paper.

RDMA over PMEM: To leverage the PMEM technologies in HPC middleware over RDMA-capable interconnects, the benefits of RDMA are discussed in the context of Persistent Memory over Fabrics (PMoF) through collaborations such as SNIA [11]. Similarly, highly available storage services for the OS kernel, such as Mojim [37] and Hotpot [27] have been proposed. NVFS [18] has been proposed to take advantage of RDMA with PMEM to enable high performance data persistency in Hadoop Distributed File System (i.e., HDFS). Crail [5] is an RDMA-enable distributed storage system but is focused on enabling high-performance multi-tiered storage management for in-memory data processing frameworks. On the other hand, PMEM-aware RDMA is being used to design specific systems, such as QueryFresh [33], which is an append-only storage system that guarantees data safety and freshness. FileMR [36] is a recent work which proposes an abstraction that combines PMEM regions and files. It allows direct access to a PMEM-backed file through RDMA.

Combining fast DRAM and slow PMEM: There exists several volatile key-value stores [12], [19], [20], [24], [28],

[29], [34] leveraging RDMA networks. These works either do not provide persistence or have high overhead for persistence because they do not utilize PMEM. So, they should only be used when durability is not a requirement. On the other hand, some systems like Bullet [16] and Crail [5] leverage a combination of DRAM and PMEM/SSD to improve performance and maintain durability. Our work also compares different backend options and demonstrates that utilizing PMEM only to replace disks for WAL ('Log-In-PMEM) can not fully take advantage of PMEM. The hybrid DRAM+PMEM approach that keeps data and logs in PMEM while maintaining index in DRAM for fast update and query ('Log+KVData-In-PMEM') seems to be optimal for memory efficiency and performance.

In contrast to the related work, our work attempts to design a high-performance and generic PMEM-aware RDMA communication runtime for emerging persistent key-value stores, called RDMP-KV. RDMP-KV can leverage 'RDMA+PMEM' in an integrated manner for optimizing the critical path performance of various key-value store backends and support different generations of RDMA networks.

VI. Conclusion

In this paper, we present our proposed designs of Remote Direct Memory Persistence based Key-Value stores with PMEM (RDMP-KV). RDMP-KV can efficiently avoid data staging overhead between DRAM and PMEM and ensure strong data durability, data consistency and recoverability for every key-value object stored or updated. Performance evaluations on InfiniBand EDR clusters with PMEM emulation show that our proposed RDMP-KV designs can improve the server performance with various persistent key-value store backend architectures by 1.7x–22x, as compared with existing PMEM-oblivious RDMA protocols in HERD that require DRAM-PMEM staging; enabling an end-to-end performance gain of 38%–65% for latency in update-heavy workload and a speedup of ∼1.7x for throughput in read-heavy YCSB workloads. Our evaluations also show that RDMP-KV outperforms Octopus FS [22] and the Forca RDMA-to-PMEM framework [14] by 65% and 71%. In the future, we plan to evaluate RDMP-KV over real PMEM hardware and NICs with RDMA Commit capabilities, and study with more online/offline data analytical workloads.

Acknowledgment

We would like to sincerely thank Arjun Kashyap from The Ohio State University for his help in conducting some of the experiments. We also want to thank the anonymous reviewers for their insightful comments and suggestions. This work was supported in part by NSF research grant CCF #1822987.

References

[1] An introduction to pmemobj (part 1) - accessing the persistent memory. http://pmem.io/2015/06/13/accessing-pmem.html, 2015.
[2] Redis. https://redis.io/, 2017.
[3] Redis, enhanced to use PMDK's libpmemobj. https://github.com/pmem/redis, 2017.
[4] pmemkv. https://github.com/pmem/pmemkv, 2018.
[5] Apache Crail. http://crail.incubator.apache.org/, 2019.
[6] Apache Ignite. https://ignite.apache.org/, 2019.
[7] Persistent Memory Replication Over Traditional RDMA. https://software.intel.com/content/www/us/en/develop/articles/persistent-memory-replication-over-traditional-rdma-part-1-understanding-remote-persistent.html, 2019.
[8] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. *SIGMETRICS Perform. Eval. Rev.*, 40(1):53–64, June 2012.
[9] K. A. Bailey, P. Hornyack, L. Ceze, S. D. Gribble, and H. M. Levy. Exploring Storage Class Memory with Key Value Stores. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, page 4. ACM, 2013.
[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *The Proceedings of the ACM Symposium on Cloud Computing (SoCC '10)*, Indianapolis, Indiana, June 2010.
[11] C. Douglas. RDMA with PMEM. http://www.snia.org/sites/default/files/SDC15_presentations/persistant_mem/ChetDouglas_RDMA_with_PM.pdf, 2015.
[12] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, 2014. USENIX Association.
[13] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, page 15. ACM, 2014.
[14] H. Huang, K. Huang, L. You, and L. Huang. Forca: Fast and Atomic Remote Direct Access to Persistent Memory. *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 246–249, 2018.
[15] J. Huang, K. Schwan, and M. K. Qureshi. NVRAM-aware Logging in Transaction Systems. *Proceedings of the VLDB Endowment*, 8(4), 2014.
[16] Y. Huang, M. Pavlovic, V. Marathe, M. Seltzer, T. Harris, and S. Byan. Closing the Performance Gap Between Volatile and Persistent Key-Value Stores Using Cross-Referencing Logs. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 967–979, 2018.
[17] Intel. PMDK: Persistent Memory Development Kit. https://github.com/pmem/pmdk/, 2019.
[18] N. S. Islam, M. Wasi-ur Rahman, X. Lu, and D. K. Panda. High Performance Design for HDFS with Byte-Addressability of NVM and RDMA. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 8:1–8:14, New York, NY, USA, 2016. ACM.
[19] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda. Memcached Design on High Performance RDMA Capable Interconnects. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP, Washington, DC, USA, 2011.
[20] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-Value Services. In *Proceeding of SIGCOMM '14*, Aug 2014.
[21] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 256–267. IEEE, 2013.
[22] Y. Lu, J. Shu, Y. Chen, and T. Li. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 773–785, Santa Clara, CA, 2017. USENIX Association.
[23] Micron. 3D XPoint technology. 2019.
[24] C. Mitchell, Y. Geng, and J. Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Proceeding of USENIX Annual Technical Conference (USENIX ATC '13)*, Jun 2013.
[25] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge. Storage management in the NVRAM era. *Proceedings of the VLDB Endowment*, 7(2):121–132, 2013.
[26] redis. How fast is Redis? https://redis.io/topics/benchmarks.
[27] Y. Shan, S.-Y. Tsai, and Y. Zhang. Distributed Shared Persistent Memory. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 323–337. ACM, 2017.
[28] D. Shankar, X. Lu, N. Islam, M. Wasi-Ur-Rahman, and D. K. Panda. High-Performance Hybrid Key-Value Store on Modern Clusters with RDMA Interconnects and SSDs: Non-blocking Extensions, Designs, and Benefits. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 393–402, May 2016.

[29] M. Su, M. Zhang, K. Chen, Z. Guo, and Y. Wu. RFP: When RPC is Faster Than Server-Bypass with RDMA. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 1–15, 2017.

[30] T. Talpey. Remote Access to Ultra-Low-Latency Storage - SNIA. https://www.snia.org/sites/default/files/SDC15_presentations/persistant_mem/Talpey-Remote_Access_Storage.pdf, 2015.

[31] T. Talpey. RDMA Persistent Memory Extensions. In *15th Annual Open Fabrics Alliance Workshop*, 2019.

[32] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight Persistent Memory. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 91–104. ACM, 2011.

[33] T. Wang, R. Johnson, and I. Pandis. Query Fresh: Log Shipping on Steroids. *Proc. VLDB Endow.*, 11-4:406–419, December 2017.

[34] Y. Wang, L. Zhang, J. Tan, M. Li, Y. Gao, X. Guerin, X. Meng, and S. Meng. HydraDB: A Resilient RDMA-driven Key-value Middleware for In-memory Cluster Computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'15, 2015.

[35] H. S. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson. Phase Change Memory. *Proceedings of the IEEE*, 98(12):2201–2227, Dec 2010.

[36] J. Yang, J. Izraelevitz, and S. Swanson. FileMR: Rethinking RDMA Networking for Scalable Persistent Memory. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 111–125, 2020.

[37] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 3–18. ACM, 2015.