Fast and Consistent Remote Direct Access to Non-volatile Memory

Jingwen Du
Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology
Wuhan, China
jwdu@hust.edu.cn

Fang Wang*
Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology
Wuhan, China
wangfang@hust.edu.cn

Dan Feng
Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology
Wuhan, China
dfeng@hust.edu.cn

Weiguang Li
Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology
Wuhan, China
wgli@hust.edu.cn

Fan Li
Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology
Wuhan, China
lfhust@hust.edu.cn

ABSTRACT

For performance benefits, recent trends of modern data centers tend to use NVM (Non-volatile Memory) as storage and utilize one-sided primitives of RDMA (Remote Direct Memory Access) to directly access NVM for I/O requests. However, one-sided RDMA doesn't have durability semantics currently, and data may partially exist in the NVM when crashes happen, thus causing unexpected data inconsistency.

Existing solutions sacrifice either read or write performance in exchange for consistency guarantees. In this paper, we introduce a multi-version log-structuring design, named eFactory, which delivers high performance for both read and write, yet also provides data consistency. In eFactory, a multi-version list is built for each object to solve the problem of crash consistency, especially when multiple clients update the same object concurrently. To transfer data directly with RDMA write as well as reduce CRC overhead on the read critical path, a single background thread is assigned to conduct integrity verification and data persisting. Furthermore, eFactory proposes a hybrid read scheme to gain high performance for reads without losing consistency guarantee. With a durability flag embedded in the object, the client can detect the durability status of data and re-read it if necessary. Evaluations show that eFactory outperforms IMM and SAW (solutions that sacrifice write performance) by 0.42x-2.79x and 0.66x-2.85x for write respectively, while maintains comparable read performance with them. In addition, the read throughput of eFactory is 1.3x-1.96x and 1.24x-1.67x of Erda and Forca (solutions that guarantee consistency at the cost of reading performance) respectively.

*Corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPP '21, August 9–12, 2021, Lemont, IL, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9068-2/21/08…$15.00
https://doi.org/10.1145/3472456.3472480

CCS CONCEPTS

• **Computer systems organization → Dependable and fault-tolerant systems and networks**.

KEYWORDS

distributed storage systems, crash consistency, RDMA, non-volatile memory, high performance, remote persistence

ACM Reference Format:
Jingwen Du, Fang Wang, Dan Feng, Weiguang Li, and Fan Li. 2021. Fast and Consistent Remote Direct Access to Non-volatile Memory. In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3472456.3472480

1 INTRODUCTION

Emerging non-volatile memory (NVM), such as phase-change memory (PCM) [20], spin-transfer torque magnetic RAM (STT-RAM) [8], and Intel Optane DC persistent memory [5] provide byte addressability, persistence, and latencies close to that of DRAM. These kinds of memories allow applications to checkpoint fast and recover fast. Due to the higher density, NVM can provide considerable capacity, which may greatly exceed the requirements of a single machine. On the other hand, remote direct memory access (RDMA) technology has narrowed the latency gap between remote access and local access, which makes systems based on remote memory more attractive than decades ago. Furthermore, an increasing number of large-scale data center applications demand fast access to vast amounts of persistent data. Therefore, many efforts aim to integrate RDMA and NVM to improve the performance and reliability of distributed systems [10–13, 15, 21, 22, 27, 31, 36, 37].

Since one-sided RDMA write enables direct remote data access without the participation of target CPUs, prior RDMA-based systems adopt a client-active scheme to write remotely [7, 11, 13, 21, 22, 26, 31, 34, 36]. Compared with RPC (Remote Procedure Call)

in which server handles everything, the server in the client-active scheme is only responsible for space allocating and metadata updating while data writing is offloaded to clients with one-sided RDMA write. Hence, the server can process more requests and high throughput are gained. Unfortunately, when the volatile DRAM is switched to durable NVM, using one-sided RDMA may cause unexpected data inconsistency. This is because RDMA doesn't have persistence semantics currently [10, 36]. When the client receives the acknowledgement of RDMA write, the data are only guaranteed to reach the NIC (Network Interface Card) cache. Therefore, data may incompletely exist in NVM due to a sudden crash.

To solve the problem, the existing design [10] follows RDMA writes with an extra RDMA send, which is issued at the durability point and forces the data to persist explicitly. Another solution [36] is to replace RDMA write with `write_with_imm` so that the server can be aware of the completion status and make data durable immediately. However, we find that writing durably with the above two ways loses the advantage over RPCs, although using the client-active scheme when not considering durability improves performance greatly. To keep using the client-active scheme and maintain performance advantage, some other designs [13, 21] detect data inconsistency by CRC (Cyclic Redundancy Check) [1] when reading. However, the CRC overhead is so large that the read performance will degrade much.

In order to overcome all these limitations, we propose eFactory[1], a multi-version log-structuring design that guarantees data consistency and provides high performance for both read and write. In eFactory, the log-structuring mechanism is adopted to ensure remote atomic updates and keep multiple versions for each object. Versions of each object are linked to constitute a version list, the head of which is indexed by a hash entry. Consequently, we can find the previous intact versions for recovery through traversing the version list. To achieve high performance, durability and CRC overhead should be removed from the critical path as much as possible. Thus, eFactory assigns a single background thread that is in charge of integrity verification and data persisting. In this way, eFactory can take benefits of one-sided RDMA primitives to the greatest extent for both read and write. Specifically, eFactory performs write using the client-active scheme with asynchronous durability. For reads, eFactory proposes a hybrid read scheme, in which the pure RDMA read scheme is used first for high performance, and RPC+RDMA read scheme is used as a supplement in case of the object hasn't been completely persisted by the background thread. By the durability flag embedded in the object, eFactory can detect incomplete objects without calculating CRC on the read path, and then switch to the RPC+RDMA read scheme for consistency guarantee. In summary, our major contributions are listed as follows:

- We identify that existing remote crash consistency schemes either lose write performance advantage over RPCs or maintain write performance at the cost of reading performance.
- We propose a multi-version log-structuring design, named eFactory, which enables remote atomic updates and maintains multiple versions of each object. Hence, a consistent state can be recovered using the previous intact version.
- We propose the background verification and persisting scheme, which supports the client-active scheme without synchronous durability, thus providing high performance for write. Moreover, this scheme also helps to reduce CRC overhead on the critical path for reads.
- We propose the hybrid read scheme, which combines the advantages of a pure RDMA read scheme and an RPC+RDMA read scheme. By using pure RDMA read scheme optimistically and leverage the RPC+RDMA read scheme as a supplement, consistency can be guaranteed without losing high performance for reading.
- We implement and evaluate eFactory. Evaluation results show that eFactory outperforms IMM [36] and SAW [10] by 0.42x-2.79x and 0.66x-2.85x for write respectively, while maintains comparable read performance with them. In addition, the read throughput of eFactory is 1.3x-1.96x and 1.24x-1.67x of Erda [21] and Forca [13] respectively[2].

The rest of this paper is organized as follows. Section 2 describes the background of RDMA networking and non-volatile main memory. In Section 3, we illustrate our motivation to design eFactory. We present our design in Section 4. The experimental methodology and evaluation results are shown in Section 5 and Section 6. We cover related work in Section 7. Finally, we conclude this paper in Section 8.

2 BACKGROUND

2.1 RDMA networking

Remote direct memory access (RDMA) enables accessing remote memory directly via bypassing the kernel and zero memory copy. It is now commonplace within data centers due to its ultra-low latency, high throughput, and decreasing prices [7, 23, 26, 27, 32, 34, 35]. Several distributed systems have been redesigned to take advantage of this high-performance userspace networking technique, including distributed transaction processing [11, 17, 24, 33, 34], object stores [16, 25], and state machine replication [6, 19, 26, 32]. RDMA has two kinds of primitives, i.e. one-sided and two-sided. Two-sided primitives, including send and recv verbs, are similar to socket programming. RDMA recv is posted at the receiver before the sender posts a send request, so as to specify the address of the incoming message. One-sided primitives, such as RDMA read, write, and atomic, are capable of accessing remote memory without the involvement of remote CPUs.

RDMA `write_with_imm` is a special two-sided primitive. It differs from RDMA write in two aspects [22]: (1) it can carry an immediate field (32 bits) within the message, and (2) it notifies the remote side immediately while RDMA write does not. In other words, the request sent by `write_with_imm` primitive gets instantly processing after it arrives.

[1]It is called eFactory (efficient factory) because the system conducts asynchronous durability and durability-aware read, which is just like pipelining processing in factory and quality inspection before serving customers.

[2]For small value (e.g. 64B), since the CRC overhead is relatively small compared to network communication, eFactory and Erda have comparable read performance. To highlight the advantages of eFactory, the improvement ratio of eFactory compared to Erda shown here does not include this situation.

2.2 Non-volatile Main Memory

Non-volatile Main Memory (NVMM) attaches to the CPU over the memory bus. It can be accessed by load/store interface locally and by RDMA read/write remotely. When NVM (Non-volatile Memory) is used as the main memory in computer systems, the volatility-persistence boundary exists between the volatile CPU cache and persistent NVM. Unlike block-based storage devices, the failure atomicity unit of NVM is generally expected to be 8 bytes. Since data keeps within NVM across crashes, data consistency (i.e., data correctness after recovering from unexpected failures) must be ensured at the memory level in the NVMM-based storage system. The key for data consistency is to provide atomic durability. For local writes, data must be flushed from CPU caches to the memory controller by instructions like CLFLUSH. To wait for writes to become persistent, applications initiate a store fence (SFENCE) instruction after one or more CLFLUSH. Also, metadata is updated after persisting data so that only consistent data are exposed.

For remote writes, there are two high-level methods.

One-sided RDMA. It's popular to access NVMM directly from NICs via RDMA write, which has the benefit of reducing or eliminating remote CPU use [7, 11, 22, 26, 34, 36]. Unfortunately, RDMA writes perform like write-back semantics. Therefore, when a crash happens, data may partially exist in the NVMM due to no durability guarantee.

Remote procedure calls. Another method is to treat NVMM as conventional storage that clients access using RPCs: clients send requests to the server's CPU, which copies volatile network buffers to NVMM and sends responses. When using RPCs, the server's CPU is involved, so it's easy to flush data into NVMM and then update metadata.

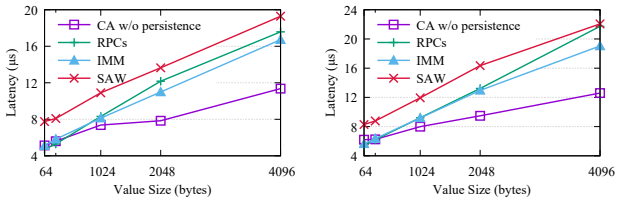
2.3 RDMA with NVM

The RDMA technology has dramatically reduced communication overhead. With its support, distributed memory systems are built to serve modern large-scale data center applications. However, these systems had to checkpoint to the slow persistent media (e.g. $\approx 10\mu$s for NVMe SSDs). Therefore, traditional systems often avoid data synchronization to gain high-speed processing, which sacrificing consistency guarantees. The emergence of NVM broke this limitation. NVM allows fast access and persistence of data. Consequently, combining NVM and RDMA can improve the performance and reliability of distributed systems, so as to meet the needs of modern large-scale data-intensive applications.

3 MOTIVATION

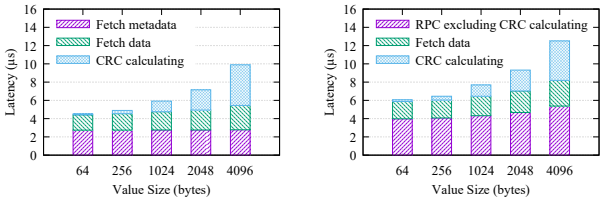
As RDMA write can reduce CPU's involvement of the server, prior researches adopt a client-active scheme to conduct remote write [7, 11, 13, 21, 22, 26, 31, 34, 36]. In this scheme, the server is only responsible for space allocation and metadata updating while data-transferring is offloaded to clients by using RDMA write, so the performance can be largely improved.

However, when the volatile DRAM is replaced by durable NVMM, using one-sided RDMA may threaten data consistency. The reasoning is as follows: RDMA write doesn't have persistence semantics. When the client receives the acknowledgement, the data could be still in the NIC caches, PCIe buffers, or CPU caches, rather than in non-volatile memory. Besides, the server is unaware of the completion status of RDMA write, so unless the client informs the server of write completion, metadata can only be updated before data transfer, causing the incomplete data exposed.



(a) **Median latency** (b) **99th percentile latency**

Figure 1: Latency of writing to remote NVMM with different methods



(a) **Erda** (b) **Forca**

Figure 2: GET latency breakdown

To perform remote write durably with RDMA write, the required sequence of operation is an RDMA write followed by an RDMA send, with DDIO enabled [10]. (We term this scheme **send-after-write**, **SAW**.) The subsequent RDMA send is issued to tell the server about the data to be persisted, and then the data is purposefully flushed by the server. Another solution [36] is to replace RDMA write with `write_with_imm` primitive (**IMM**), which can carry the offset information in the immediate field. Once the server detects the completion of `write_with_imm`, the written data can be forced into NVMM. In these two methods, metadata can be updated after the data has been durable, which ensures data consistency.

We evaluate the latency of different methods for writing to NVMM durably, including the above two schemes and RPC. As a baseline, we also measure the latency of the client-active scheme without a persistence guarantee (**CA w/o persistence**). Figure 1 demonstrates the median and 99th percentile latency of these methods.

As shown in Figure 1, when the persistence is not taken into consideration, using the client-active scheme can greatly improve the performance (36%). Unfortunately, the client-active scheme loses most of its latency advantage over RPCs for writing remote NVMM durably. Specifically, SAW performs worse than RPC for all data sizes that we set. Furthermore, IMM achieves slightly better performance (5%) than RPC.

In order to achieve high performance for write, some other works [13, 21] keep using the client-active scheme without making data durable immediately. Instead, integrity verification is performed

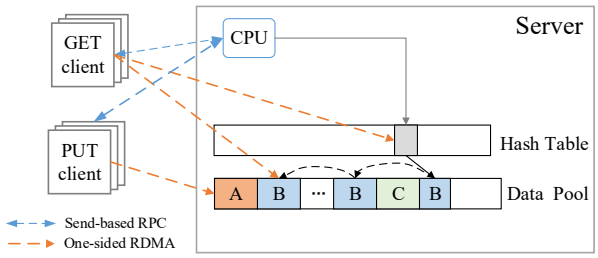


Figure 3: The architecture of eFactory

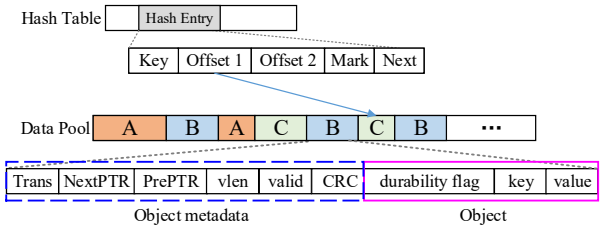


Figure 4: Data and metadata structures

when conducting read, so data consistency can be handled. For example, Erda [21] packs the address offset of the latest two versions in an 8-byte region in the hash entry. The client in Erda calculates the CRC to check data integrity. Once the incomplete data is detected, the client re-reads the previous version of the data using the offset in the 8-byte region. Different from Erda, Forca [13] performs integrity verification by the server and conducts persisting on the read path.

To evaluate the added read latency by CRC, we measure the read latency of Erda and Forca. Figure 2 shows the results of latency breakdown. We find that when value size increases, the overhead caused by CRC calculating is so large that seriously degrades the read performance. For instance, it takes about 4.4μs to verify a 4KB object, which accounts for 45% and 35% of the read latency for Erda and Forca respectively.

Therefore, we strive for guaranteeing data consistency as well as providing high performance for both read and write.

4 DESIGN

4.1 Overview

eFactory is a multi-version log-structuring zero-copy design that guarantees data consistency while providing high performance for both read and write. Figure 3 illustrates the overall architecture of eFactory. It leverages a log-structuring mechanism to provide remote atomic updates and keep multiple versions of objects. Multiple versions of each object constitute a linked list. The head of each linked list, which is the latest version, is pointed by a hash entry. Consequently, we can find the latest version by searching the hash table. Moreover, when the latest version happens to be incomplete due to a read-write race or a crash, previous intact objects can be found by traversing the version list. To provide high performance, CRC and durability overhead are removed from the critical path as much as possible. Specifically, we run a background thread to conduct integrity verification and data persisting. This allows eFactory to perform write using the client-active scheme without immediate persistence. For reading, eFactory adopts a hybrid read scheme. In the hybrid read scheme, objects are firstly fetched by a pure RDMA read scheme optimistically. If the obtained object hasn't been completely durable in NVMM, eFactory will re-read it with RPC+RDMA read scheme.

The remainder of this section provides details of how eFactory organizes and manages metadata and data (Section 4.2), client-active scheme with asynchronous durability (Section 4.3.1), hybrid read scheme (Section 4.3.3), and how the background thread operates to support these schemes (Section 4.3.2).

4.2 Data and Metadata Structures

4.2.1 Data Management. **Object**. The object is the basic access unit. As illustrated in Figure 4, it consists of a key-value pair and a durability flag. We use the durability flag for checking whether the object has been completely persisted in NVMM. It's useful for avoiding redundant background verification and data persistence operations, as well as supporting the hybrid read scheme for high performance.

Data Pool. As shown in Figure 4, we adopt a log-structured manner to store and manage objects. In this way, data are updated out-of-place, which is not only conducive to concurrent access, but also to avoid inconsistent data (neither old nor new) in the NVM after a crash. Moreover, thanks to updating out-of-place, we can naturally keep multiple versions of each object for consistency recovery. Specifically, multiple versions of each object are linked to construct a linked list. Therefore, we can traverse the version list to find the previous intact object.

4.2.2 Metadata Management. **Object Metadata**. Object metadata includes value size (vlen), a pointer that refers to the previous version of the object (PrePTR), a valid bit that indicates whether each version is valid or not, and a 32-bit CRC checksum of the value for checking data integrity. In addition, to support log cleaning, the object metadata also contains a pointer to the next version of the object (NextPTR) and a transfer identifier (Trans). The former is used to find the next version of the migrated current version during log cleaning. Accordingly, the pointer (PrePTR) in it can be modified to point to the new address. And then the transfer identifier (Trans) in it is also updated to indicate the previous version has been transferred to the new data pool. These metadata can be co-located with each object in the data pool or be resided in the independent memory pool. We choose the former in our implementation.

Hash Table. Each hash entry contains the key and the object's offset address in the data pool. To record the destination address of the latest version during log cleaning, an additional offset is included in the hash entry. Usually, only one offset is valid. eFactory leverages a mark bit to indicate which offset is related to the current working data pool.

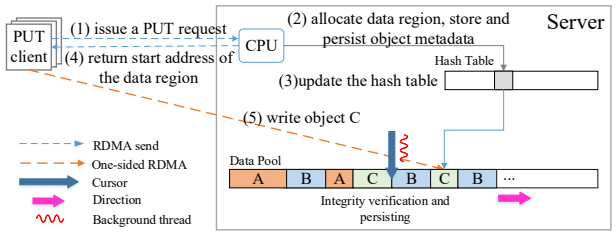


Figure 5: PUT with Asynchronous Durability

4.3 Data Workflow

To allow RDMA access from clients, the server registers the memory regions of the hash table and data pool with RNIC (RDMA enabled NIC) at initialization. And then, when clients first establish connections to the server, they obtain the addresses and corresponding registration keys of these memory regions. With these keys and base addresses, the client can directly access data and metadata on the server.

4.3.1 PUT. We use a client-active scheme without an immediate persistence guarantee. Specifically, as shown in Figure 5, the client sends a PUT request to the server via RDMA send (step (1)). After receiving the PUT request, the server allocates a data region of the requested size in the data pool in a log-structured way. Also, space is allocated to record the object metadata. Then, the server fills the object metadata with the value size and CRC included in the request, and the pointer in the object metadata (PrePTR) is updated to referring the previous version (step (2)). Next, the server updates the hash index so that the newly allocated data region can be found through the hash entry (step (3)). After all the metadata has been updated and persisted into the NVMM, the start address of the allocated data region is returned to the client (step (4)). Finally, the client uses RDMA write to directly transfer the data to the target region, which the returned offset indicates (step (5)).

Compared with IMM and SAW, this method reduces CPU involvement and extra network roundtrips for persistence, thus achieving high performance for write.

4.3.2 Background Verification and Durability. To reduce the CPU involvement and CRC overhead on the critical path as much as possible, we assign a single background thread to handle integrity verification and data persisting. Specifically, this background thread begins from the head of the data pool and operates each object one by one. It calculates the CRC over the object value and then compares the result with the recorded CRC. If they match with each other, persist the object and set the durability flag. Otherwise, check whether the object has timed out, i.e., check whether the elapsed time since the server received the write request has exceeded the predefined timeout period. If it turns out to be timed out, mark it as invalid. The space for invalid objects will be reclaimed during log cleaning.

The background thread and the request processing thread run independently, i.e., there is no need for inter-thread synchronization. With the background thread, the objects are asynchronously verified and persisted into NVMM, which leads to high-performance

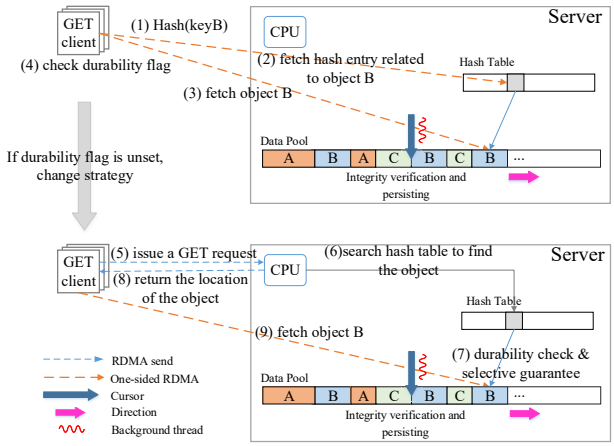


Figure 6: Hybrid Read Scheme for GET

writing. Through the durability flag, the GET request handler can identify the status of the object. If the requested object has not been fully persisted by the background thread, the request handler will first verify and persist the object, set the durability flag, and then return its offset. Since there is no need to wait for the background thread to process from the current position to the accessed position, such requests can be quickly responded to without being blocked. The durability flag in turn helps the background thread determine and skip objects that have been persisted by the request handler.

4.3.3 Hybrid Read Scheme for GET. Basically, to reduce the server's involvement on the critical path as much as possible, GET can be handled in two ways.

RPC+RDMA read. The client uses the RPC to fetch hash entry and then gets the object by one-sided RDMA read. As the server participates in obtaining metadata, this approach is easy to ensure data consistency.

Pure RDMA read. The client uses one-sided RDMA reads to grab the metadata and the object respectively. Compared with the above one, this approach gains higher performance, because RPC consists of two RDMA primitives, which is obviously worse than using RDMA read. However, as RDMA read bypasses the server's CPU, inconsistency is difficult to be ensured. Namely, the client may fetch an incomplete object. This is because data are exposed before it has been completely durable in the NVMM.

To take full advantage of these two methods, we propose a hybrid read scheme: we first optimistically try to read by pure RDMA read method, if it fails to obtain the consistent data, switch to the slower but robust RPC+RDMA read method. Evaluations show that the hybrid read scheme can achieve high performance while providing strong consistency.

To make it clear, the complete process (shown in Figure 6) is as follows: Initially, the client establishes a connection with the server. Then, according to the requested object's key, the client gets the offset address of the metadata, i.e., the hash entry pointing to the requested object, by calculating with the same hash algorithm as that server uses (step (1)). After grabbing the hash entry by one-sided RDMA read (step (2)), the client verifies the received object

key and figures out where the object locates. Next, the client uses another RDMA read to directly fetch the requested object (step (3)). By checking the durability flag in the object (step (4)), the client can determine whether the fetched object is intact and has been persisted in NVM. If the durability flag indicates the object has been completely durable, this GET process ends.

Otherwise, the client re-sends the GET request using RDMA send primitive (step (5)). After receiving the request, the server looks up the hash table to find the location of the requested object (step (6)). Then first check the durability flag to determine whether the object has been completely durable in NVMM (durability check in step (7)). If it has, the start offset of the data region, where the object stores, is returned to the client (step (8)). Otherwise, the server verifies the integrity of the object by checksum. If the current object version is complete, the server flushes the object into NVMM, updates the durability flag (durability guarantee in step (7)), and returns the address offset to the client (step (8)). If not, the server traverses the version list of the object, conducts the same check to the previous versions until finding the intact one (step (7)), and returns the address of it (step (8)). Finally, the client uses a one-sided RDMA read to fetch the object (step (9)).

The key design of the hybrid read scheme is the durability flag embedded in the object. It allows the client to obtain the value and detect data integrity with one RDMA read. More importantly, it helps eliminate the overhead of CRC from the read critical path in most cases, thereby achieving high-performance reading. At the same time, with RPC+RDMA read scheme as a supplement, data consistency is still pledged.

4.4 Log Cleaning

Log cleaning aims at reclaiming the memory of deleted and stale objects. It will be triggered when the reserved space reaches a pre-defined threshold. As shown in Figure 7, it includes two stages, namely, log compressing (Figure 7(a)) and log merging (Figure 7(b)).

In the first stage, the server notifies the client that log cleaning will begin. The client switches to the RPC+RDMA read scheme once it receives the notification. Then a new data pool is allocated at the server end and registered with RNIC (RDMA enabled NIC) of the server. After that, the server scans the data pool from the tail, moves the latest version (B_2, C_1, A_2) to the new data pool and skips subsequent stale versions (B_1, A_1). After each move, the server updates metadata: if the migrated version is not directly indexed by the hash entry, the server modifies the pointer (PrePTR) in the next version (B_3, C_2) and sets the transfer flag; otherwise, if the object is the latest version (A_2), another offset ("new" offset) in the hash entry is updated to record the object's position in the new data pool. As a result, the server can handle clients' requests concurrently. Specifically, the server writes to the old data pool and updates the "old" offset as usual. To serve reads, the server checks whether the relatively new offset (based on the mark flag) exists to determine the object is in the new or old data pool. When traversing the version list, the transfer flag in the object metadata also allows locating objects correctly.

After finishing reverse scanning, log cleaning march on to the log merging stage (Figure 7(b)). In this stage, new writes during log compressing are merged into the new data pool. The server reserves

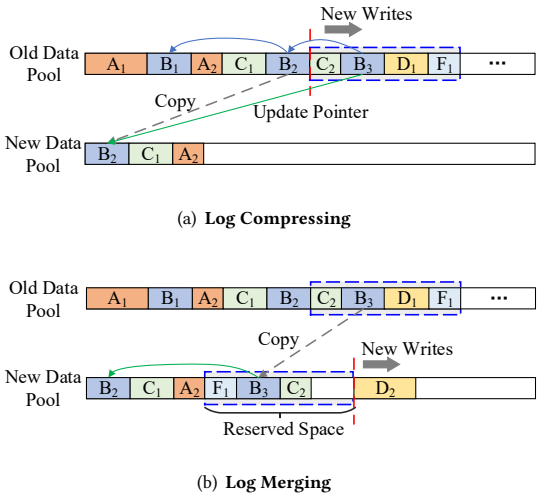


Figure 7: The process of log cleaning

space for the objects written after the start of log compressing, so the server can handle writes to the new data pool while conducting log cleaning. The log cleaning thread performs a reverse scan and replicates the objects into the new data pool. If the object to be replicated has already been updated (D_2) and the latest version already or can be made durable completely, then it (D_1) will be skipped. Read requests are processed as in the log compressing stage.

When all objects written during log compressing have been completely merged, the log cleaning process ends. At this time, for all objects replicated from the old data pool, the server clears the transfer flag in the object and the old offset in the hash entry. Meanwhile, the mark bit in the hash entry is flipped. Finally, the server informs the client that log cleaning has finished. After receiving the notification, the client returns to use the hybrid read scheme. During the process of log cleaning, the client can only perform reads by RPC+RDMA read scheme.

5 EXPERIMENTAL METHODOLOGY

5.1 Evaluation Platform

Our experiments run upon the servers, each of which contains two 10-core 2.4GHz Intel Xeon E5-2640 v4 processors and 128GB of DRAM. Each core has a private 32KB L1 cache and a private 256KB L2 cache, and all 10 cores on a single processor share a 25MB L3 cache. Each machine is equipped with a Mellanox ConnectX-5 InfiniBand NIC (MCX555A-ECAT), connected to a Mellanox 100Gbps Infiniband switch. All machines run Ubuntu 16.04 with the MLXN OFED LINUX-4.7 InfiniBand driver. Like the simulation method used by Forca, we reserve 6G of DRAM from the server machine and use PMDK to emulate persistent memory [4, 13].

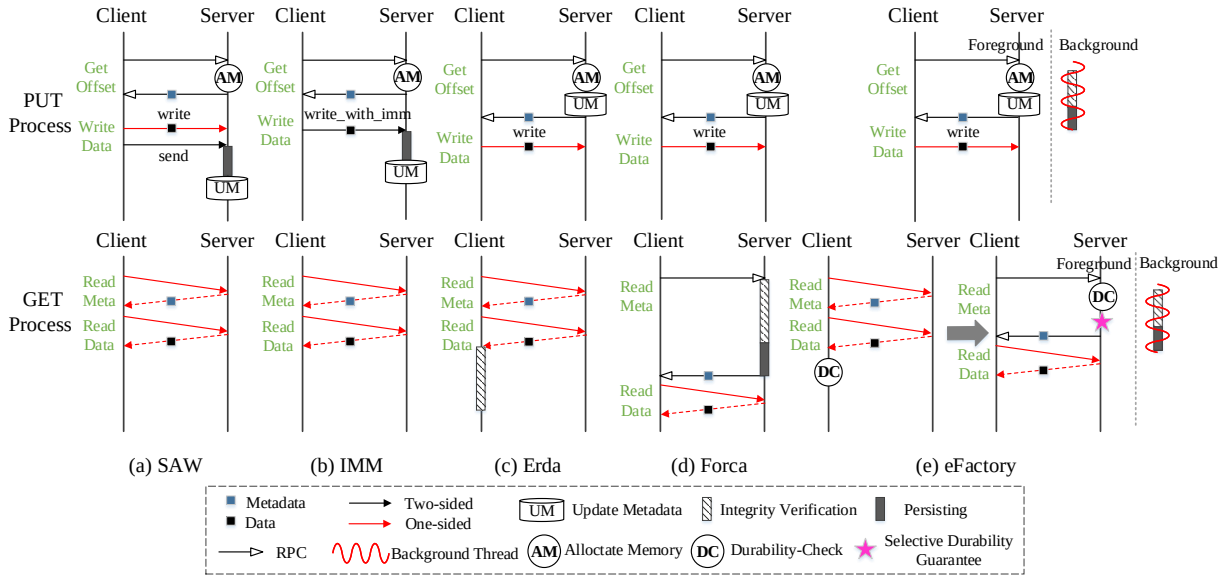


Figure 8: A comparison of eFactory with existing designs

5.2 Workloads

We use the YCSB [9] benchmark, which is widely used in related works [13, 16, 19, 21, 27, 28, 31], to generate four workloads following a long-tailed Zipfian distribution: (1) Read-only workload (YCSB-C) contains 100% GET. (2) Read-intensive workload (YCSB-B) contains 95% GET and 5% PUT. (3) Write-intensive workload (YCSB-A) contains 50% GET and 50% PUT. (4) Update-only workload contains 100% PUT. In the experimental results, each data value is the average of 5-run results.

5.3 Comparisons

We compare eFactory against send-after-write (SAW) [10], the system that uses `write_with_imm` for remote durability (IMM) [36], Erda [21], and Forca [13]. To provide an apple-to-apple comparison on different approaches, we implement SAW, IMM, Erda, and Forca on the same code base as eFactory. All implementations are strictly according to the primal designs [2, 10, 13, 21, 36]. To make it clear, the basic access paradigms are shown in Figure 8.

5.3.1 SAW (Send after Write). PUT: For a PUT, the client issues a write request by RDMA send. When the server receives the request, it allocates a data region in the data pool and returns the start offset of the data region with RDMA send (We term this procedure SEND-based RPC). Then, the client transfers data directly to the data region by RDMA write. Finally, the client leverages RDMA send to inform the server to persist data and update metadata.

GET: When there is no hash conflict, a GET in SAW consists of two RDMA reads, which fetch metadata and data respectively. Since metadata is updated after the data is completely durable, there is no need to verify data when reading.

5.3.2 IMM. PUT: IMM also uses SEND-based RPC to allow the remote server to allocate space for the data to be written. And then, the client uses `write_with_imm` to transfer data. Hence, the server can be aware of the write completion (discussed in Section 2.1). Finally, the server flushes the data into NVMM and updates the metadata.

GET: IMM uses two RDMA reads to fetch metadata and data respectively.

5.3.3 Erda. PUT: The client uses SEND-based RPC to obtain a memory space allocated by the server, and then it conveys the data directly to remote memory with RDMA write. In Erda, data are not forced into NVMM explicitly. And the hash indexing is updated immediately after allocating memory for the data.

GET: Erda uses Hopscotch hashing to index objects. Besides, an 8-byte atomic region, which includes offsets of the latest two versions and a tag, is contained in each bucket to ensure the atomicity of metadata modify. For a GET, the client exploits RDMA reads to gain metadata and data respectively. After that, the client verifies data integrity with CRC inside the object. If the fetched object is identified as incomplete, another RDMA read will be sent to get the previous one.

5.3.4 Forca. PUT: Write in Forca is performed like that in Erda.

GET: The client sends read requests to the server by RDMA send. After receiving the request, the server searches the address offset of expected data in the hash table. Then the server identifies data integrity by CRC and conducts data persisting before returning the offset. Finally, the client obtains data by RDMA read.

eFactory appoints a single background thread to conduct integrity verification and data persisting. Accordingly, the client can first optimistically use the pure RDMA read scheme and identify data integrity by checking the durability flag. It's reasonable because data is persisted only after it is recognized as complete by either background thread or request handler. And with durability

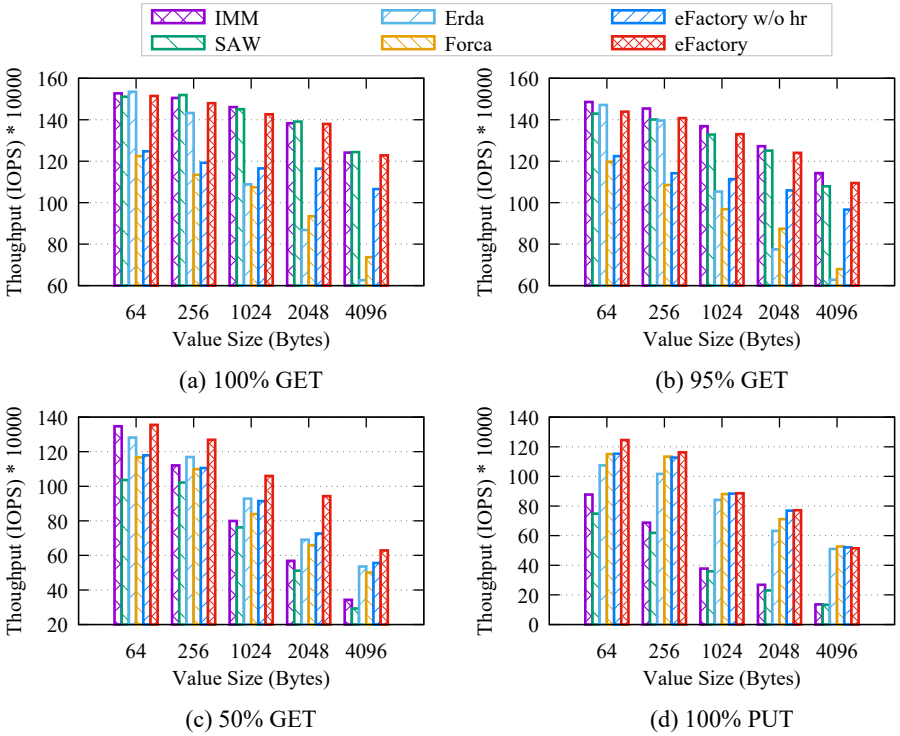


Figure 9: End-to-end throughput comparison with different value sizes

guarantee before reading, eFactory refrains from non-monotonic reads across crashes. If the fetched object is not completely durable, the client switch to the RPC+RDMA read scheme. Compared with Forca, the server performs a durability check first to determine whether the background thread has persisted the requested object. If the object has already been durable completely in NVM, a response is sent back immediately. Otherwise, the server carries out CRC calculating and data persisting (selective durability guarantee). With a selective durability guarantee, eFactory gains better performance without impairing consistency.

6 EVALUATION

6.1 Throughput

We now evaluate the throughput of eFactory against other systems. Figure 9 illustrates the throughput of these systems with different value sizes for four types of workloads. These experiments involve 8 concurrent clients issuing operations as fast as possible.

For a write-only workload, intuitively, eFactory should have comparable performance with Erda and Forca. The reason is that all of them use RDMA write without an immediate persistence guarantee. Figure 9(d) shows the result. Surprisingly, the PUT throughput of eFactory is slightly better than Forca when the value size is small. This is because, in eFactory's implementation, the object is co-located with object metadata and directly pointed by the hash entry. However, Forca has an extra intermediate layer of object metadata, which causes the performance disparity. Besides, eFactory gains 5%-22% higher throughput than Erda. This is due to eFactory utilizes multiple receiving regions to optimize the simultaneous processing of a batch of packets. Moreover, as we expected, eFactory outperforms IMM and SAW by 0.42x-2.79x and 0.66x-2.85x respectively. The performance gain comes from fewer network round trips and CPU's involvement on the critical path.

Figure 9(a) demonstrates the result for a read-only workload. eFactory shows nearly the same performance as IMM and SAW. The gap is merely 2%. With a hybrid read scheme, eFactory can leverage the high performance that RDMA reads bring as much as possible. Erda has comparable performance when the value size is small, however, as value size becomes larger, its performance becomes worse due to CRC calculating overhead on the read critical path. Forca also shows a bigger gap when the value size increases. Differently, even when the value size is small, Forca achieves poor throughput. The reason is that Forca always needs the server's participation to ensure strong consistency during reading. Since the latency of RPC, which contains two RDMA primitives, is higher than RDMA read, Forca fails to gain high performance as other systems. Specifically, when the value size is 4KB, the throughput of eFactory is 1.96x and 1.67x of Erda and Forca respectively.

We now compare the performance across workloads that have a mix of reads and writes. As shown in Figure 9(b), for the read-intensive workload, eFactory's throughput is nearly the same as SAW, which accounts for 95% of IMM's throughput. Compared with read-only workloads, the gap between eFactory and IMM increases a little. Because of the read-write race, more read operations should be handled with RPC+RDMA read scheme. Moreover, eFactory

still outperforms Erda and Forca by 0.74x and 0.61x respectively. Figure 9(c) illustrates the result for the write-intensive workload. From the figure, we identified that eFactory achieves the highest throughput for all the value sizes. This confirms the benefits of asynchronous durability and the hybrid read scheme.

Factor Analysis. To investigate the contribution of background verification and durability as well as hybrid read scheme, we also evaluate the performance of eFactory w/o hr (eFactory without hybrid read). Take the 4KB value as an example, for the read-intensive workload (Figure 9(b)), eFactory w/o hr achieves 42% higher throughput than Forca. By leveraging the hybrid read scheme, the throughput can increase 13% further; for the write-intensive workload (Figure 9(c)), eFactory w/o hr outperforms Forca by 10.9%. With the hybrid read scheme, eFactory improves throughput by 13%.

6.2 Scalability

We now compare the number-of-clients scalability of eFactory against other systems. The experiment uses 32-byte keys and 2048-byte values. Figure 10 shows the results. From the figure, we observe that the throughput of eFactory grows approximately linearly with the increasing client numbers for all the cases. However, when write dominates, IMM and SAW fail to scale well. The reason is that eFactory uses the background thread to perform asynchronous durability, which reduces the network round trips and CPU involvement on the critical path as much as possible.

eFactory without hybrid read improves Forca's throughput by 16%-45% for read-only workload and 19%-48% for the read-intensive workload. With the hybrid read scheme, eFactory enhances the throughput further by 15%-23% for read-only workload and 11%-24% for the read-intensive workload. These confirm the benefits of using background verification and durability scheme as well as the hybrid read scheme.

Overall, eFactory performs best among six key-value stores. Specifically, for write-intensive workload, eFactory beats IMM' and SAW's throughput by up to 2.14x and 2.18x when concurrency reaches 16. In addition, eFactory gains about 24% and 50% higher throughput than Erda and Forca respectively.

6.3 Log Cleaning

As the server in eFactory handles requests from clients while performing log cleaning, we evaluate the performance impact on the client's requests caused by the log cleaning. Figure 11 shows the average latency of eFactory with or without performing log cleaning. The experiment uses 32-byte keys and 2048-byte values. Overall, log cleaning incurs 1%-21% performance overhead. For 100% PUT, the latency when conducting log cleaning is very close to that when not performing log cleaning, but it still increases a little. This is because the server needs to switch back and forth between the new data pool and the old data pool in order to copy the data and modify the object metadata in the previous version. This access mode is not conducive to exploiting cache locality. For read-only workload (100% GET), the latency during log cleaning is 21% higher than that under the normal case. The reason is that the server needs to be involved in the reading process to determine the correct locations

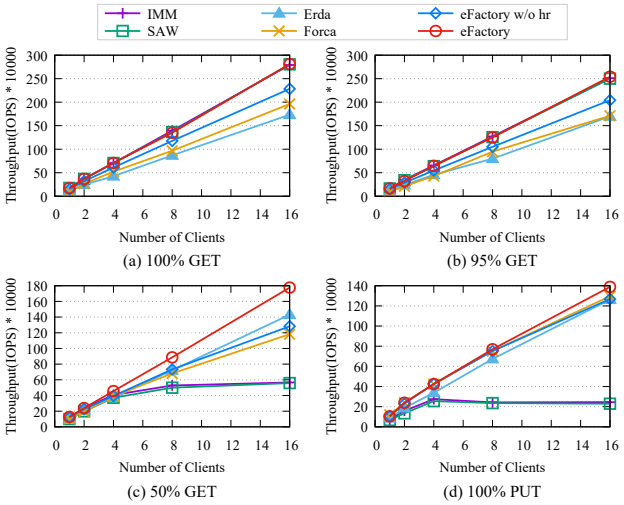


Figure 10: Thoughput with variable number of client processes

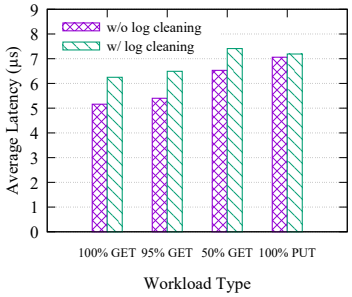


Figure 11: The performance impact caused by log cleaning

of accessed objects, thus eFactory can't take benefit from the hybrid read scheme during log cleaning and can only achieve worse performance.

7 RELATED WORK

7.1 Remote Persistence

Currently, typical solutions [10, 13, 21, 27, 36] for remote persistence include RPCs and one-sided RDMA followed by another extra network round-trip. Besides, researchers implement new primitives to extend the existing RDMA protocol. For example, remote commit (rcommit) [29], a new RDMA verb being proposed in the draft for RDMA standard, allows the client to push data with RDMA write and subsequently make it durable, thus a full round trip, as well as remote interaction, can be eliminated from the operation. Persistence Parallelism Optimization [12] puts forward a new primitive named `rdma_pwrite` and leverages the advanced network controller to return persist acknowledge. Arash et al. [30] find rcommit is expensive because it ensures both ordering and durability of synchronous mirroring updates. Hence, they propose new RDMA primitives including rofence (remote ordering fence) and rdfence

(remote durability fence). However, these designs [12, 29, 30] require either new PCIe command or specific hardware. Contrarily, our work is based on current RDMA primitives and requires no special hardware.

Sanidhya et al. [18] present a comprehensive study on correct remote persistence of RDMA updates when system configurations vary. This includes persistent area scope, whether DDIO [3, 14] is enabled, and receive queue's work request buffer is located in DRAM or NVM. Unlike it, our work focuses on providing high performance and data consistency for the current general configuration.

7.2 Consistency for RDMA-based NVM

Octopus [22], an RDMA-enabled distributed persistent memory file system, directly accesses a shared persistent memory pool and actively fetches and pushes data all in clients to reduce server load. However, the design that updating remote data in-place with RDMA writes fails to ensure crash consistency. Erda [21] adopts a log-structured mechanism for out-of-place updates of data, as well as an 8-byte atomic structure design to ensure atomic update of metadata. Unfortunately, the 8-byte atomic region only contains the location of the latest two versions, which is not enough to restore to a consistent state if multiple threads concurrently update the same object. In comparison, our proposed eFactory maintains multiple versions for each object in the form of a linked list, which is more robust. Besides, Erda doesn't persist data explicitly, thus dirty updates become durable through natural eviction. This leads to non-monotonic reads: after the crash, clients aren't ensured to gain the value read before the crash. By contrast, eFactory leverage background thread for asynchronous durability, and adopt hybrid read scheme to prevent non-monotonic reads. Forca [13] uses a log-structured approach to eliminate in-place updates. Moreover, Forca allows the server to actively detect one-sided write finish by self-verifying. However, Forca performs self-verifying on the read path, which makes the read performance decreasing. In doing so, the read access policy must also be based on RPC, so reads can not completely be offloaded to the client. Unlike Forca, eFactory leverages a background thread to conduct self-verifying and data persisting. Besides, eFactory adopts the hybrid read scheme to take advantage of one-sided RDMA reads to the maximum. Orion [36], a distributed file system built upon NVM and RDMA, provides weak consistency and strong consistency according to when the metadata is updated. But it adopts the IMM strategy. In contrast, our proposed eFactory delivers higher throughput while providing strong consistency.

8 CONCLUSION

It's challenging to guarantee data consistency of the RDMA-based NVM system without sacrificing performance. This work presents a multi-version log-structuring design, named eFactory, which delivers high performance with a consistency guarantee. eFactory enables remote atomic updates with a log-structuring mechanism. To cope with the crash inconsistency that many concurrent threads update the same object, eFactory maintains a version list for each object. In eFactory, data are delivered directly with RDMA write. Besides, a single background thread is appointed to perform integrity verification and data persisting. Hence, durability and CRC calculating overhead on the critical path can be reduced to the greatest extent. Moreover, eFactory adopts a hybrid read scheme to achieve high performance for reads without losing consistency guarantee. Experimental results demonstrate the benefits of these novel designs, as well as the fact that eFactory overall outperforms all previous solutions.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful and helpful comments, which improve the paper. This work is supported in part by National Key R&D Program of China No.2018YFB1003305, NSFC No.61832020, No.61821003.

REFERENCES

[1] [n.d.]. Cyclic Redundancy Check. https://en.wikipedia.org/wiki/Cyclic_redundancy_check.
[2] [n.d.]. Forca. https://github.com/huanghaixin008/Forca.
[3] [n.d.]. Intel Data Direct I/O Technology (Intel DDIO). https://www.intel.com/content/dam/www/public/us/en/documents/technologybriefs/data-direct-i-o-technology-brief.pdf.
[4] 2016. How to emulate Persistent Memory. http://pmem.io/2016/02/22/pm-emulation.html.
[5] 2019. What Is Intel Optane DC Persistent Memory? https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html
[6] Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J Marathe, Athanasios Xygkis, and Igor Zablotchi. 2020. Microsecond consensus for microsecond applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 599–616.
[7] Thomas E Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N Schuh, and Emmett Witchel. 2019. Assise: Performance and Availability via NVM Colocation in a Distributed File System. *arXiv preprint arXiv:1910.05106* (2019).
[8] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, et al. 2013. Spin-transfer torque magnetic random access memory (STT-MRAM). *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 9, 2 (2013), 1–35.
[9] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.
[10] Chet Douglas. 2015. RDMA with PMEM: Software mechanisms for enabling access to remote persistent memory. In *Storage Developer Conference*.
[11] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th symposium on operating systems principles*. 54–70.
[12] Xing Hu, Matheus Ogleari, Jishen Zhao, Shuangchen Li, Abanti Basak, and Yuan Xie. 2018. Persistence parallelism optimization: A holistic approach from memory bus to rdma network. In *MICRO*.
[13] Haixin Huang, Kaixin Huang, Litong You, and Linpeng Huang. 2018. Forca: Fast and atomic remote direct access to persistent memory. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE, 246–249.
[14] Ram Huggahalli, Ravi Iyer, and Scott Tetrick. 2005. Direct cache access for high bandwidth network I/O. In *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 50–59.
[15] Nusrat Sharmin Islam, Md Wasi-ur Rahman, Xiaoyi Lu, and Dhabaleswar K Panda. 2016. High performance design for HDFS with byte-addressability of NVM and RDMA. In *Proceedings of the 2016 International Conference on Supercomputing*. 1–14.
[16] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2014. Using RDMA efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*. 295–306.
[17] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 185–201.
[18] Sanidhya Kashyap, Dai Qin, Steve Byan, Virendra J Marathe, and Sanketh Nalli. 2019. Correct, Fast Remote Persistence. *arXiv preprint arXiv:1909.02092* (2019).

[19] Antonios Katsarakis, Vasilis Gavrielatos, MR Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. 2020. Hermes: a fast, fault-tolerant and linearizable replication protocol. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 201–217.

[20] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th annual international symposium on Computer architecture*. 2–13.

[21] Xinxin Liu, Yu Hua, Xuan Li, and Qifan Liu. 2019. Write-optimized and consistent rdma-based nvm systems. *arXiv preprint arXiv:1906.08173* (2019).

[22] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: an rdma-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 773–785.

[23] Teng Ma, Tao Ma, Zhuo Song, Jingxuan Li, Huaixin Chang, Kang Chen, Hai Jiang, and Yongwei Wu. 2019. X-RDMA: Effective RDMA Middleware in Large-scale Production Environments. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 1–12.

[24] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, et al. 2019. Storm: a fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage*. 97–108.

[25] Diego Ongaro, Stephen M Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. 2011. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 29–41.

[26] Marius Poke and Torsten Hoefler. 2015. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. 107–118.

[27] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing*. 323–337.

[28] Yacine Taleb, Ryan Stutsman, Gabriel Antoniu, and Toni Cortes. 2018. Tailwind: fast and atomic rdma-based replication. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 851–863.

[29] Tom Talpey and Jim Pinkerton. [n.d.]. RDMA Durable Write Commit. https://tools.ietf.org/html/draft-talpey-rdma-commit-00

[30] Arash Tavakkol, Aasheesh Kolli, Stanko Novakovic, Kaveh Razavi, Juan Gómez-Luna, Hasan Hassan, Claude Barthels, Yaohua Wang, Mohammad Sadrosadati, Saugata Ghose, et al. 2018. Enabling efficient RDMA-based synchronous mirroring of persistent memory transactions. *arXiv preprint arXiv:1810.09360* (2018).

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

[32] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. 2017. Apus: Fast and scalable paxos on rdma. In *Proceedings of the 2017 Symposium on Cloud Computing*. 94–107.

[33] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. 2018. Deconstructing RDMA-enabled distributed transactions: Hybrid is better!. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 233–251.

[34] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 87–104.

[35] Jilong Xue, Youshan Miao, Cheng Chen, Ming Wu, Lintao Zhang, and Lidong Zhou. 2019. Fast distributed deep learning over rdma. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–14.

[36] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2019. Orion: A distributed file system for non-volatile main memory and RDMA-capable networks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 221–234.

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