Write-Optimized and Consistent RDMA-based Non-Volatile Main Memory Systems

Xinxin Liu, Yu Hua*, Xuan Li, Qifan Liu
WNLO, Huazhong University of Science and Technology, Wuhan, Hubei, China
*Corresponding author: Yu Hua
E-mail: {xinxin, csyhua, xuanli, qifanliu}@hust.edu.cn

***Abstract*—To deliver high performance in cloud computing, many efforts leverage RDMA (Remote Direct Memory Access) in networking and NVMM (Non-Volatile Main Memory) in end systems. Due to no CPU involvement, one-sided RDMA becomes efficient to access the remote memory, and NVMM technologies have the strengths of non-volatility, byte-addressability and DRAM-like latency. However, due to the need to guarantee Remote Data Atomicity (RDA), the synergized scheme has to consume extra network round-trips, remote CPU participation and double NVMM writes. In order to address these problems, we propose a write-optimized log-structured NVMM design for Efficient Remote Data Atomicity, called Erda. In Erda, clients directly transfer data to the destination memory addresses in the logs on servers via one-sided RDMA writes without redundant copies and remote CPU consumption. To detect the atomicity of the fetched data, we verify a checksum without client-server coordination. We further ensure metadata consistency by leveraging an 8-byte atomic update in a hash table, which also contains the addresses of previous versions of data in the log for consistency. When a failure occurs, the server properly and efficiently restores to become consistent. Experimental results show that compared with state-of-the-art schemes, Erda reduces NVMM writes approximately by 50%, significantly improves throughput and decreases latency.**

I. Introduction

Cloud computing requires high performance in both network transmission and local I/O throughput. RDMA (Remote Direct Memory Access) allows to directly access remote memory via bypassing kernel and zero memory copy. Due to no CPU involvement, one-sided RDMA operations (read, write and atomic) provide higher bandwidth and lower latency than two-sided one (send and recv). Moreover, NVMM (non-volatile main memory) technologies have the strengths of non-volatility, byte-addressability, high density and DRAM-class latency in end systems. Many schemes thus synergize RDMA and NVMM to deliver end-to-end high performance [1]–[5]. However, RDMA NICs fail to guarantee persistence with NVMM [1], [3], and thus using one-sided RDMA to access remote NVMM becomes inefficient due to the challenges of guaranteeing **Remote Data Atomicity (RDA)**: Non-atomic writes from failures are durable in NVMM, which results in the inconsistency of data. The server is unaware of the non-atomic and invalid data in NVMM due to no CPU involvement in the context of the one-sided RDMA. The client is also unaware of the possible data loss in the server, because the returned ACK of RDMA write from the server merely means that the data have reached the volatile cache of the server NIC, and possibly fail to be flushed into NVMM.

Currently, in order to guarantee RDA, some schemes leverage an extra RDMA read operation after RDMA write to force data to be persistent and integrated [3], [6]. Undo/redo logging and copy-on-write (COW) are popular consistency mechanisms in persistent memory systems [5], [7]. There also exist some RDMA-based NVMM systems that ensure RDA by CPU involvement [1], [5]. However, these solutions fail to be efficient due to the following three problems. (1) **High Network Overheads:** existing schemes that leverage an extra RDMA read operation after RDMA write cause extra network round-trips for each RDMA write. (2) **High CPU Consumption:** logging and COW require the remote CPU to control the sequence among operations. However, CPU involvement decreases the benefits of using one-sided RDMA operations. (3) **Double NVMM Writes:** some CPU involvement solutions [8] need to check the written data in log regions or buffers, and then apply them into the destination memory addresses. These operations essentially require double NVMM writes, consuming the limited NVMM endurance.

In order to address these problems, we propose Erda (Efficient Remote Data Atomicity) that is an efficient write-optimized log-structured NVMM design. In Erda, an object with a CRC checksum inside is the basic unit of access operations. Clients read and write objects by interacting with remote servers. For the update operation from clients to servers, the metadata in a hash table are modified with an 8-byte atomic write, and then the object is directly transferred from clients to the destination memory address on servers without redundant buffer and server CPUs. The non-atomicity of the written object will be detected by subsequent read requests via verifying checksums. Once the verification results show that the fetched object is non-atomic, clients will re-read the previous version of the object, whose address information is also contained in the hash table, to ensure the consistency and atomicity of the fetched object. At the same time, servers are notified about the inconsistency and properly restore to a consistent version. Evaluation results demonstrate that compared with Read After Write and Redo Logging schemes, Erda significantly improves the throughput and decreases the latency, as well as reduces the NVMM writes approximately by 50%.

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 61772212.

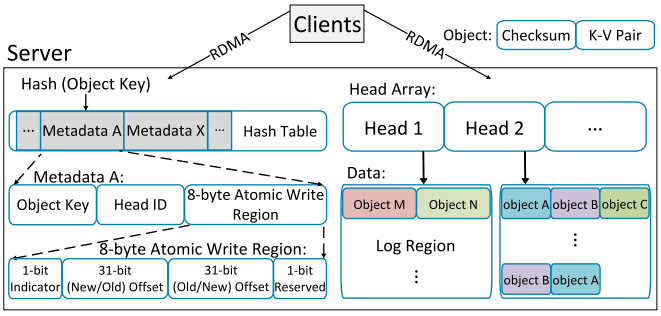

Fig. 1: The overall architecture of Erda.

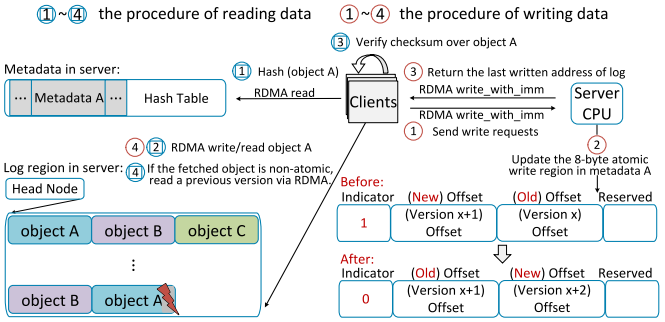

Fig. 2: The procedures of reading and writing objects.

II. System Design

Data and metadata are persistent in the server's NVMM. **The Structure of an Object:** As shown in Figure 1, an object is the basic unit of one access, which consists of a 32-bit CRC checksum and a KV pair. Specifically, the 32-bit CRC checksum computed over the KV pair is used to check the atomicity of the object. **The Structure of a Log Region:** We store and manage objects using a log-structured manner. We use a head array of the fixed addresses to link logs, and the Head ID is used to distinguish different head nodes. Each head links a continuous memory region (such as 1GB) as a *log region*. For scalability, when a larger memory region is needed, we allocate and register another continuous memory region and link it to the first 1GB memory region following the same head. To enable one-sided RDMA reads from clients to a log region, we analyze workloads and predefine a size for a unit that stores an object in the log region. When the size of the object is no larger than the preset size, the object is stored in the log region. Otherwise, the log region stores the pointer that points to the full object outside the log region. **Metadata in a Hash Table:** We adopt the RDMA-friendly hopscotch hashing to index objects. The hash entries store the metadata of objects. We design a structure for the metadata, which consists of the object key, the head ID and an 8-byte atomic write region, including 1-bit *Indicator* which indicates whether the following 31-bit offset is "new" (the latest address of the object) or "old" (the previous address of the object), 31-bit new/old offset, 31-bit old/new offset and 1-bit reserved position to support variable-size objects. All the information in this region can be updated in an 8-byte atomic write.

Figure 2 shows the procedures of reading and writing data (objects) using RDMA. Once the connection is established, the server will send the head array containing the corresponding relationships between head IDs and pointers to the client. To read an object, the client hashes the requested object key, and uses one-sided RDMA read to directly read the corresponding hash entry in the server. Then, after verifying the received object key, the client queries the local cached head array for the pointer corresponding to the received head ID. Finally, with the aid of the 8-byte atomic write region and the pointer, the client directly fetches the requested object using one-sided RDMA read. When the client verifies the checksum of the object correctly, this RDMA read operation finishes. To write an object, the client sends a write request to query the server about the last written address of the log using RDMA write_with_imm, which notifies the receiver of the immediate value. The server further updates the hash entry of desired key's metadata and returns the last written address of the log. With the returned information, the client posts one-sided RDMA writes to directly write data in the log region of the server without the participation of the server's CPU, and thus the server obtains higher processing capacity and removes redundant copies.

Erda is able to provide Remote Data Atomicity (RDA) guarantee to ensure crash consistency under RDMA and NVMM scenarios: **Out-of-Place Updates.** We adopt a log-structured NVMM design to prevent in-place updates, and always maintain an "old" version of the updated object (similar to an undo log). **CRC Checksum.** We add a 32-bit CRC checksum over each object, so clients can detect the atomicity of the fetched object by verifying the checksum. Once the fetched object becomes non-atomic, the client can issue another RDMA read to fetch the previous version of the object. **8-Byte Atomic Write.** We design an 8-byte atomic write region in the hash entry to ensure metadata consistency. The update operation only needs to modify the atomic write region in the metadata. Therefore, the inconsistency will only occur when the metadata have been atomically updated but a failure occurs before the object data have been fully written into a log. Fortunately, the 8-byte atomic write region also contains the address information for the "old" object version. When a failure occurs, the client can fetch the previous version, and the server can properly restore to a consistent version. However, for create and delete operations, the entire hash entry needs to be modified, which is larger than 8 bytes. In order to guarantee the consistency for the create operation, the atomic write region in a hash entry is the last modified part. Hence, if the metadata is partially written, the data offsets in the atomic write region will be equal to 0. During recovery, the server compares the key in that hash entry with the key of the first object in the log region. If the two are not the same, the server will delete this entry. For the delete operation, the 8-byte atomic write region is cleared before the rest of the metadata entry, in order to ensure consistency across failures.

III. Implementation Details

Read-Write Competition. In Erda, there are three read-write scenarios. 1) After receiving an update request, a server atomically modifies a hash entry, but the client has not

completed the object write. Thus, synchronous read operations from other clients find that the object is invalid by checking the checksum. In this case, the clients issuing the read requests will read a previous version of the requested object by using the old offset from the obtained hash entry. Then the client will notify the server to correct the inconsistency. 2) Clients remotely read a hash entry that is being created/deleted by the server and hasn't been completed. The clients will know if the metadata is partially created/deleted by comparing the key in the hash entry with the key of the object whose data offset is 0 within the log region. In addition, the scenario, where clients remotely read the object whose hash entry has been created but the object write has not been completed, is similar to the first scenario. 3) When a client has read the entry but not the corresponding object, another client concurrently writes the updated object in the log after requesting the server to modify the same entry. This read-write competition does not lead to errors, because the update in our log-structured mechanism is an out-of-place update, which does not affect the previous version of the object to be read by the first client.

Lock-Free Log Cleaning. Log cleaning reclaims free space of the append-only log by removing deleted objects and stale versions of objects for space saving. When the occupied space following a head reaches a pre-defined threshold, the cleaner in a server will allocate another continuous memory region, and starts log cleaning after notifying the connected clients and waiting maximum round trip time. After receiving the notification, clients can still read and write objects, but in different ways: clients issue read/write requests using RDMA write_with_imm. Furthermore, in the 8-byte atomic write region of metadata, the server does not flip the Indicator. The previous "new offset" region in metadata now stores the address of the original log region (Region 1), and the previous "old offset" region in metadata now stores the address of the newly allocated region (Region 2). Log cleaning consists of two phases: log merging and replication. In the log merging phase, the cleaner traverses Region 1, writes the latest version of objects to Region 2 and updates the "old offset" region as metadata. For read/write requests from clients, the server accesses the "new offset" region where the address information of Region 1 is stored. In the replication phase, the cleaner replicates objects that were written by clients after the start of the log cleaning into Region 2, and the server handles new read/write requests concurrently. Specifically, for the write requests, the server updates the "old offset" region, and appends the new object into Region 2. When the log cleaning is completed, the server changes the pointer of the corresponding head from pointing to Region 1 to Region 2. Then, the server flips the Indicators in the hash table, returns the new pointer to the connected clients and notifies these clients that the log cleaning finishes.

IV. Performance Evaluation and Analysis

Our experiments run upon 2 – 4 servers for different performance metrics, each of which contains two 2.4 GHz Intel Xeon $E5620$ CPUs (4 cores) and 12GB of DDR3 RAM. Each server is also equipped with a 40Gbps Mellanox ConnectX InfiniBand network adapter and runs on CentOS 7.3. we add 150ns as extra write latencies of DRAM to simulate NVMM [9]. We use YCSB to generate three workloads that follow Zipfian distribution: the update-heavy workload (YCSB-A), the read-mostly workload (YCSB-B) and the read-only workload (YCSB-C).

We compare Erda with two consistency schemes: Redo Logging (a CPU involvement scheme) [7] and Read After Write (a network-dominant scheme) [6]. **Redo Logging** scheme adopts two-sided RDMA (RPC) to access remote NVMM. To write objects, all a client needs to do is to send a request to the redo log region of a server using RDMA send. The server asynchronously verifies the atomicity of the message in the redo log and applies the write request to the destination memory address. To read objects, after a client issues a request via RDMA send, the server looks for the object in the redo log and the destination memory address. For **Read After Write** scheme, to write objects, a client first sends a request to a server like Erda and obtains the address to be written in the ring buffers. Moreover, the client uses one-sided RDMA write to push the object into the ring buffers, and issues RDMA read following the RDMA write to force the object to be persistent and integrated into the ring buffers. The server CPU polls for these operations asynchronously and applies them to the destination memory address. The procedure of read operations follows the operations of redo logging scheme.

Latency. In Figures 3 – 5, compared with Redo Logging and Read After Write, Erda reduces the average latencies by approximately 34.40% and 34.43% respectively. Erda performs especially better for YCSB-B and YCSB-C where read operations dominate, because the clients of Erda use two one-sided RDMA reads to perform read operations (one for the metadata, and the other for directly fetching the requested object) without the CPU involvements of servers.

Throughput. Figures 6 – 8 show the throughputs with different workloads and numbers of client threads. The value size is 16Bytes. We observe that the average throughput of Erda is 1.53x and 1.51x those of Redo Logging and Read After Write respectively. We further evaluate Erda's throughput using one server and three clients (each client with one thread). As shown in Figure 9, the system scalability has been significantly improved since Erda's remote reads do not occupy the server CPUs, and the remote writes require the participation of the server CPUs only when requesting the remote addresses to be written.

The Number of Written Bytes. Table I shows the benefits of Erda in terms of NVMM endurance. N is the size of one KV pair. $Size(key)$ is the key size. For Erda, a create operation first writes metadata in the hash table, i.e., an object key, a head ID (1Byte), an Indicator and an offset (4Bytes). Then a client directly writes an object (4Bytes+N) in a log region. For an update operation in Erda, the server rewrites an Indicator and an offset (4Bytes) in metadata, and then writes the updated object (4Bytes+N) in a log region. A delete operation in Erda is to directly delete/reset the corresponding

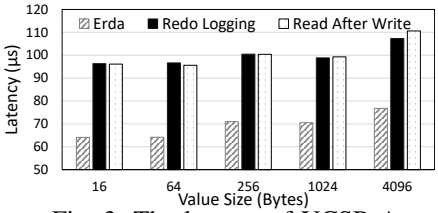

Fig. 3: The latency of YCSB-A.

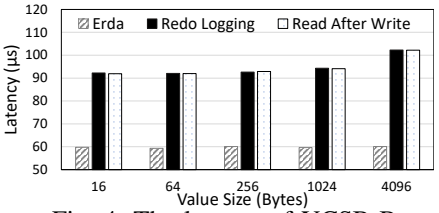

Fig. 4: The latency of YCSB-B.

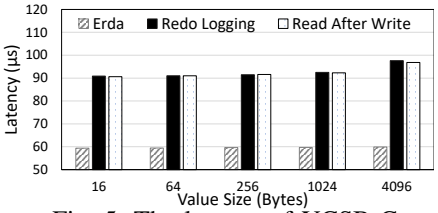

Fig. 5: The latency of YCSB-C.

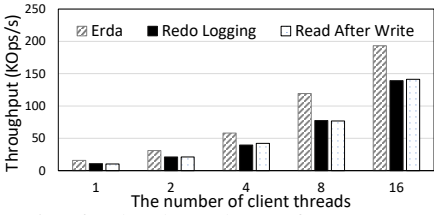

Fig. 6: The throughput of YCSB-A.

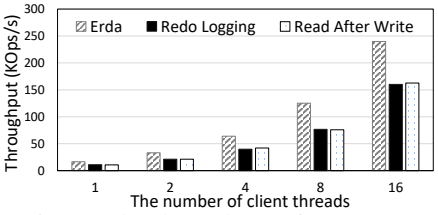

Fig. 7: The throughput of YCSB-B.

Fig. 8: The throughput of YCSB-C.

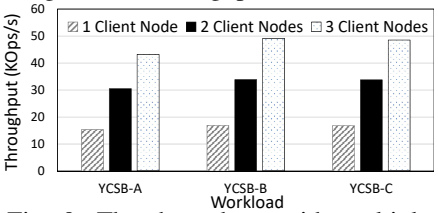

Fig. 9: The throughput with multiple client nodes. The value size is 16Bytes.

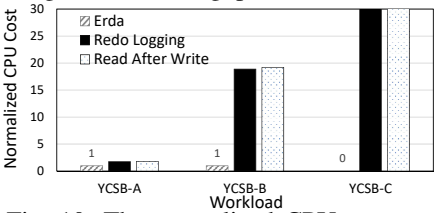

Fig. 10: The normalized CPU costs at server side (64Bytes value size).

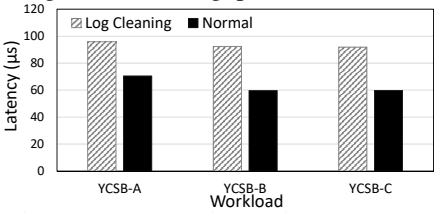

Fig. 11: The latencies during log cleaning ($1,024$Bytes value size).

hash entry, which contains an object key, a 1-byte head ID and an 8-byte atomic write region. For Redo Logging and Read After Write, the metadata consists of a key and an address (8Bytes). We omit the detailed analysis due to space limit.

TABLE I: The number of written bytes in different operations.

	Create	Update	Delete
Erda	Size(key)+9+N	8+N	Size(key)+9
Redo Logging	Size(key)+12+2N	4+2N	Size(key)+8
Read After Write	Size(key)+12+2N	4+2N	Size(key)+8

Server CPU Utilization. Figure 10 evaluates the utilization of the server CPU that possibly becomes a bottleneck when handling a large number of client requests on a fast network. For YCSB-C, the CPU cost of Erda is 0 since the read procedure of Erda does not involve server CPUs. Hence the normalized CPU costs of the other two schemes are positive infinity. The normalized CPU costs of Redo Logging and Read After Write for YCSB-B are on average 20.90x and 21.75x higher than the cost of Erda, respectively. For YCSB-A, the normalized CPU costs of Redo Logging and Read After Write are on average 1.92x and 2x respectively.

Log Cleaning. We evaluate the impact of log cleaning on the concurrent read/write requests. In Figure 11, "Log Cleaning" represents the average latencies of read/write requests during the log cleaning. "Normal" represents the average latencies of read/write requests under the normal cases of Erda. The average latencies during the log cleaning are worse than those under the normal cases of Erda. The main reason is that the read procedure of Erda does not involve server CPUs with one-sided RDMA read, while the read procedure during the log cleaning uses RDMA write_with_imm involving server CPUs (similar to Redo Logging and Read After Write).

V. Conclusion

In order to address the problems of high network overheads, high CPU consumption and double NVMM writes when ensuring RDA with RDMA and NVMM, we propose an efficient write-optimized log-structured NVMM design, called Erda. Erda transfers data directly to the destination memory address without buffer and copy, and guarantees consistency and atomicity by leveraging out-of-place updates, the CRC checksum and the 8-byte atomic write. Evaluation results demonstrate that Erda reduces NVMM writes approximately by 50%, significantly reduces CPU costs, decreases the latencies and improves the throughputs of existing schemes.

References

[1] J. Yang, J. Izraelevitz, and S. Swanson, "Orion: A distributed file system for non-volatile main memory and rdma-capable networks," in *FAST*, 2019.

[2] X. Hu, M. Ogleari, J. Zhao, S. Li, A. Basak, and Y. Xie, "Persistence parallelism optimization: A holistic approach from memory bus to rdma network," in *MICRO*, 2018.

[3] D. Kim, A. Memaripour, A. Badam, Y. Zhu, H. H. Liu, J. Padhye, S. Raindel, S. Swanson, V. Sekar, and S. Seshan, "Hyperloop: group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems," in *SIGCOMM*, 2018.

[4] Y. Lu, J. Shu, Y. Chen, and T. Li, "Octopus: an rdma-enabled distributed persistent memory file system," in *USENIX ATC*, 2017.

[5] Y. Shan, S.-Y. Tsai, and Y. Zhang, "Distributed shared persistent memory," in *SOCC*, 2017.

[6] C. Douglas, "RDMA with byte-addressable PM, RDMA Write Semantics to Remote Persistent Memory, An Intel Perspective when utilizing Intel HW." https://downloads.openfabrics.org/ofiwg/dsda_rqmts/RDMA_with_PM.pptx, 2014.

[7] M. A. Ogleari, E. L. Miller, and J. Zhao, "Steal but no force: Efficient hardware undo+ redo logging for persistent memory systems," in *HPCA*, 2018.

[8] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "Farm: Fast remote memory," in *NSDI*, 2014.

[9] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *ASPLOS*, 2011.