DKVS: A Distributed Key-Value Store for Non-Volatile Random-Access Memory

Chandan Kalita and Gautam Barua

Computer Science and Engineering, Indian Institute of Technology, Guwahati, Guwahati, 781039, Assam, India.

Abstract

Non-volatile memory (NVRAM) is becoming available, and several experimental file systems have been built for such systems. This paper presents the design and implementation of a distributed key-value storage system on a cluster (DKVS) having NVRAM in each node. We have chosen a key-value system to enable different storage architectures to be implemented on top, such as file systems with a hierarchical naming structures, relational databases, NoSQL systems etc. Keeping RDMS primarily in mind, DKVS provides support for transactions with ACID properties for sequences of operations on multiple key-value items. We provide resilience by allowing replicated copies of each key-value pair in another node. We have used techniques used in clustered file systems to provide caching of locations of key-values, and RDMA as an optional feature to fetch a value from a remote node. The implementation is on a six-node system as a loadable Linux module. RDMA is implemented in software using Soft-RoCE. A portion of RAM was simulated as NVRAM using the "memmap" feature of Linux. We evaluated our implementation using our custom workload as well as with the YCSB benchmark and compared it with a six node Redis cluster on the same experimental setup and found that the two have comparable performance.

Keywords: Concurrency, Key-Value, NVRAM, RDMA, Soft-RoCE, Transaction

1 Introduction

The promise of the widespread availability of Non-Volatile Ram (NVRAM) on a system's memory bus, has resulted in several designs and implementations of storage software for NVRAM. While the first round of designs was for file systems[1, 2], the next round saw some Key-Value (KV) storage systems, in line with interest in KVs as the underlying storage system for systems such as NoSQL systems. Even though the only commercially available product, Intel Optane, was discontinued in July 2023, work on producing NVRAM products based on the Computer Express Link (CXL) protocol, has revived interest in NVRAM based solutions. This paper describes a distributed key-value store on NVRAM (DKVS). The store is for a traditional clustered system on a LAN, augmented with terabytes of NVRAM at each node. It is assumed that all data can then be stored in these NVRAMs. To handle node failures, there are provisions to replicate KVs across multiple nodes of the cluster. As NVRAM provides RAM like performance (although slower), in such a system, device latency is no longer a major determiner of performance. Instead, in such systems network performance plays an important role. Therefore, the design assumes RDMA [4, 5] access to remote memory from a particular node. However, RDMA is not central to our design and it has been included to increase the throughput of access of remote key-value items. Our system can be used without RDMA. This differentiates it from other systems in which RDMA is an integral part of the system design [6].

Our system design targets traditional RDBMS systems operating in clusters on a LAN with a shared storage area network. Such systems will operate in our system by using the NVRAM available in large configurations at each node, as shared storage. Control data and data to be written travels on the LAN, while data to be read from a remote node is done through RDMA. A common security domain is

assumed to allow remote nodes to handle authentication and access control of KVs to be read. To read a KV, its control information (inode information) has to be first accessed through the LAN. An application requesting read access to the KV to a node will be checked for access control and then, on behalf of the application, the node will read the value of the KV through RDMA. The application may be running on the node itself, or on a separate application server.

Our research demonstrates the efficiency of employing standard features in a Non-Volatile Random Access Memory (NVRAM) system. The innovation lies in the comprehensive design rather than the introduction of new features. A key distinctive feature is the centralized handling of writes at the owner's site, where static owners are assigned to each key-value pair. This approach simplifies the processing of writes, with other nodes obtaining copies for reading from the owner node. The use of RDMA for data retrieval, or LAN in its absence, further enhances system performance. The decision to embed location information in the key, rather than using consistent hashing, allows for greater flexibility in item placement, empowering higher levels of software to dictate data distribution. The system's replication strategy prioritizes fault tolerance, leveraging a fast interconnection network for efficient data caching in remote nodes. The inclusion of standard write-ahead redo logs and a transaction owner for each transaction allows the implementation of distributed transactions. Two-PL locking with deadlock detection is used for implementing concurrency control. Use of AVL tree for various index structures is another contribution of our system. Since NVRAM devices are byte addressable, AVL tree or other balanced binary tree structures are more appropriate for search and insert operations than other structures, such as B+ trees, LSM trees, which are normally effective for block-oriented storage devices. Overall, our work presents a well-designed NVRAM Key-Value system that optimally balances performance, fault tolerance, and offers transactional integrity.

2 The KV System at each node

```
struct inode
{
   8 bytes key;
   8 bytes starting NVM address;
   8 bytes v_size;
   32 bytes ACL;
   2 bytes replica_count;
   2 bytes replica[replica_count];
   2 bytes version;
   8 bytes pointer to nodes reading
}
```

Fig. 1The inode structure of a KV pair.

2.1 Use of AVL Trees

The most common structure for handling KV Stores is an LSM (log structured merge) tree [12]. However, this assumes that the bulk of the data is stored in block based secondary memory. There have been proposals to modify the LSM structure when NVRAM is also present in the system [X,Y,Z]. But all these proposals assume that NVRAM is not sufficient to hold all the data of a KV store. So the proposals use the NVRAM as some form of cache for secondary storage. But in our system, the KV store is distributed across multiple nodes and all data is stored in NVRAM in the nodes of the distributed system and there is no block based secondary store. So there is no need of introducing the

complexities of LSMs. We only need an indexing structure into the data stored in NVRAM. To handle reads, writes, deletes and range queries, a tree structure is the accepted indexing structure. Since the index is in byte-addressable memory, the most common structure, the AVL tree, is being used. Some in-memory designs [13] have used B+ trees arguing that if nodes fit a cache line, accessing a node will not require memory access. Our view is that with large cache memory available nowadays, with many cache lines, AVL tree nodes will get cached and the advantage of searching within a B+ tree node will not be there. Therefore, we have used three AVL trees - K_AVL for the inode table, F_AVL for free space management and CACHE_AVL for cache entries. Details are given below.

2.2 RAM and NVRAM Versions of AVL Trees

We assume that RAM access will be faster than that of NVRAM, and so copies of K_AVL and F_AVL are maintained in RAM. Modifications are first done in the RAM version of a tree. Then a log record is created in a Write-Ahead Log (WAL) for metadata which is stored in NVRAM. Writing into WAL uses the by now standard method of using clwb, sfence, and movnti instructions of the Intel instruction set to write atomically to WAL [14]. After WAL writing is complete, changes to the NVRAM version is made. Once this is complete, an "end" statement is written into the WAL. Standard, ARIES-like redo is done during recovery [ARIES].

2.3 The Inode Structure

As this is a key-value store, there is no naming directory structure. Each KV pair is accessed based on the key. The key may be of random size up to a maximum, which can be assigned during system setup. A key contains the owner node id (see section 3) and the actual key. For each key, there is an inode having some metadata. As shown in Figure 1 the inode structure is composed of eight fields: key, which represents the unique identifier, a pointer starting address which points to the beginning of a value, an integer value size which is the size of the value, an access control list, number of replicated copies, location of replicas, a 2-byte version number and a pointer to a list of nodes maintaining a cached copy. Each value is therefore stored contiguously in memory.

```
struct inode
{
   8 bytes key;
   8 bytes starting NVM address;
   8 bytes v_size;
   32 bytes ACL;
   2 bytes replica_count;
   2 bytes replica[replica_count];
   2 bytes version;
   8 bytes pointer to nodes reading
}
```

Fig. 21 The inode structure of a KV pair.

2.4 Inode Table and K_AVL

Each node maintains an independent table of inodes. The inode table is implemented as an AVL tree (K_AVL) to enable efficient searching. The tree is formed based on the value of the key. A fixed-size NVRAM block is reserved to store the tree nodes. Insertion into the AVL tree allocates space from the fixed size block only. This fixed size block is partitioned into sub-blocks of the size of one node of the

tree. Memory management of this fixed size block is done using a bitmap. The arrangement of the bitmap and blocks for tree nodes is shown in Figure 2.

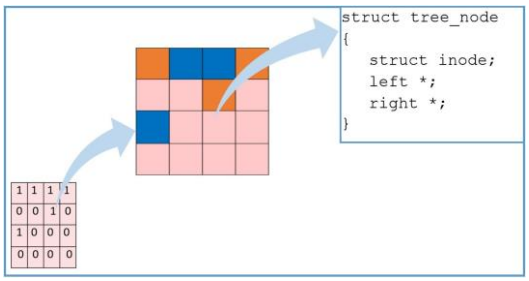

Fig. 32 Bitmap and Tree node blocks.

2.5 Free Memory Management and F_AVL

Apart from the K_AVL tree, in each node, there is another AVL tree F_AVL, to manage NVM free space. Units of free space are ordered in F_AVL by the size of the free space nodes. Initially, the entire free space is represented as a free entry as the root of the F_AVL tree with a value equal to the amount of free space available. The free NVRAM keeps splitting based on the required size when a new entry occurs. To add a new entry, the first fit (in in-order traversal) algorithm is used to find a node 'N' in the F_AVL tree. If the size of 'N' is the same as required, that node is removed from the F_AVL and added to the K_AVL tree at the appropriate place. If the size of 'N' is larger than required, 'N' is partitioned and the required segment of NVRAM is taken out from the F AVL tree to form a new node that is inserted into the K_AVL tree. The remaining part of 'N' remains in the F_AVL tree. Accordingly, on a delete operation, the respective node from the K_AVL tree moves to the F_AVL tree. Due to delete operations number of small unused NVRAM areas can arise. Two or more contiguous unused blocks (hole) are merged to make a large unused block by running a GC (Garbage Collection) routine. Figure 3 illustrates the changes that take place as KVs are added to, and deleted from the system. This is essentially a modified form of the Buddy algorithm for space management, removing the "power of 2" allocation.

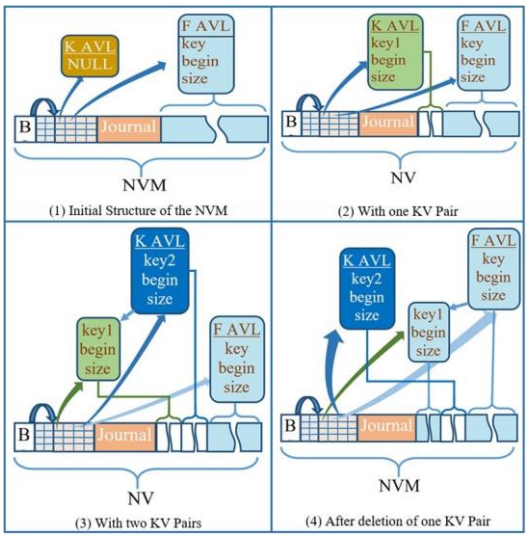

Fig. 43 Example of NVRAM Data Structures.

3 The Distributed System

3.1 Locating Key-Values

One instance of the KV store will run at each node of the distributed (clustered) system. It is a Key-Value storage, to be implemented in a distributed cluster (nodes) of servers interconnected by a LAN and also through a shared RDMA link. For access to single KVs, the operations are PUT and GET. For accessing multiple KVs in transaction mode, READ and WRITE are used, as described in more detail below. Each Key-Value (KV) has an owner node, and the current version of the KV is stored in the owner node. The owner node of a KV may change, but it is going to be infrequent – due to failure of the owner node or due to redistribution of data based on usage information. Another node may obtain a copy of the KV by transferring it from the owner node to itself by RDMA. But this copy will not be updated if an update on the KV takes place. All KV updates take place on the owner node and in replicas, if they exist. The Key of a KV is composed of two parts: the first part is the owner node id, and the second part is a number uniquely identifying the KV in the owner node, the length of the Key part in KV's can be set at system configuration time. Associated with each KV is an inode which, as already explained, contains all required information about the KV: length of the value, starting address of the value which is stored contiguously in NVRAM, access control information, etc.

3.2 Reading and Writing of KVs

Given a request for a particular key at a node, its first part may identify the owner node to be other than the current node. To access the remote KV, the current node has to send a request to the owner node seeking the inode of the KV. The owner node accesses the inode and checks access permission for the current read request and, if permission can be granted, the inode of the KV is returned to the requesting node. The value of the KV is obtained directly by the requesting node, bypassing the owner node CPU, through RDMA, as the inode contains the starting physical address of the value of the KV. The owner node keeps track of nodes that have a copy of the inode

as it sends the inode to other nodes. This information is stored in the inode itself (this information is not shared with remote nodes). When there is a write to the KV, the owner node uses this information to invalidate all the copies of the KV.

Remote nodes cache the values of KV's it has read. The inode of the KV is also available. Due to space limitations, the value may be removed, or both the value and inode of the KV may be removed. To manage this, nodes manage a cache of inodes and values obtained from other nodes. Each entry is accessed by a unique key. Pointers to cached inodes and values are kept in each entry. Only the inode or both the inode and value of the Key can be cached. If none of the two are cached, the entry is removed from the cache. Given a key, a node, therefore, checks the cache to find out what is available locally, and if nothing is available, it fetches the inode through the LAN and the value through RDMA and creates a cache entry in the process. If only the inode is cached, the value is obtained through RDMA. Cached entries are stored in RAM, using an AVL tree (CACHE_AVL).

To write to a KV, a remote node sends the new value to the owner node through the LAN. The owner node invalidates all remotely stored inodes of the KV and then updates the KV locally. To implement fault-tolerance, all the KVs of a node may be replicated in another node. The number of copies can vary from one to as many nodes that are there. It is expected that most KVs will have two copies. So, the discussion here is restricted to the case of two copies. The first copy is in the owner node, and the second copy is in another node which we call the slave node for this owner node. The identity of the slave node is recorded in the inode of the KV. The slave copy does not participate in read operations and other nodes do not even know the slave node of a KV. On a write request, the owner node, after updating the local copy of the KV, sends the new copy to the slave node, and after receiving acknowledgment of the update by the slave node, the home node declares the write is complete to the requesting node. Suitable protocols have been implemented for a slave node to take over KVs of a failed node. The node id in Keys is not changed during such a take-over, and so a mapping of node id (logical) to the actual node id (physical) must be maintained by every node in the system.

3.3 Concurrency Control

A two-phase locking facility with a minor restriction is implemented to provide transaction locking facilities that span several KVs. In a strict two phase locking scheme, read or write locks on items can be obtained in any order as a transaction progresses. All locks are released at the time a transaction commits. Our limitation is that locks are to be acquired in two parts: first, all required read locks are acquired, the items read, and write locks on required items are then obtained before writes take place. This is not a major restriction as the need for a subsequent read lock after the initial set of read locks can always be implemented by acquiring write locks on those items in the next step. Our restriction results in a more efficient solution than the general case and our assessment is that in applications such as databases, our restriction will not matter. For a single KV, the protocols described above take care of locking. A read of a KV is accompanied by a read lock. Multiple nodes may read lock a KV. On a write to the KV, the inode invalidation messages serve as requests to unlock the read lock being held by remote nodes. For a multiple KV access through a transaction, the node where the access is initiated is designated as the transaction owner (TO). In the read phase, all the reads are partitioned on a per-node basis, and each node is sent a list of KVs to read lock and whose inodes are to be returned to the TO. The KVs are then read by the TO through RDMA. Following this reading phase, the writes are similarly partitioned and sent, but with the values of the KVs (either new or existing) sent to the nodes. Nodes reply when their writes are complete, and when all replies are received, a commit message

is sent to all participating nodes. More details are given below in the section on implementation. Read and write requests may be delayed due to a lock on the KV. A deadlock detection process runs in the background, which periodically polls all nodes for the current state of locks and detects deadlocks using standard algorithms. Abort messages are sent to appropriate TO nodes. Note that since the slave nodes are passive, they do not participate in the lock and unlock processes. They only need to be sent updated versions of KVs, whether locking is present or not.

4 Implementation

The system has been designed and implemented as a loadable Linux kernel module. User level implementation was not chosen due to the decision to have a single security zone where each node is trusted. The need to handle RDMA accesses also taken into account. As hardware with NVRAM was not available, we designated a portion of RAM as NVRAM using the Linux memmap kernel parameter. Each kernel module in every node has two kernel processes. One is responsible for managing the NVRAM and the local KV store, while the second process is an interface process that accepts requests from clients and interacts with the first kernel process to provide the requested service. If a request is from a local user process, then communication with the interface process is done using Netlink sockets [15]. The use of Netlink keeps us away from polluting the kernel and damaging the stability of the system. Moreover, since we have developed our code as a loadable kernel module, it is not appropriate to include system call code in a loadable module. In case the user request is from a remote node the interface process is contacted on a known TCP port. The interface process sets up the RDMA network with remote nodes when required. To use RDMA we have used Soft-RoCE [16]. Soft-RoCE is a software implementation of RoCE (RDMA over Converged Ethernet) that allows RoCE to run on any Ethernet network adapter whether it offers hardware acceleration or not.

The layout of DKVS is given in Figure 4. Here the BS field is of 4-bytes and is used to store the size of the tree node bitmap (as shown in Figure 2 above). The size of this field restricts the maximum number of tree nodes and hence restricts the number of KVs that the node allows. The value of the BS field is assigned during system setup and based on that value, the size of the BitMap and Tree Nodes are dynamically initialized. The next K (configurable) bytes contain the journal (the write-ahead metadata redo log), and the remaining portion of the NVRAM is used as data storage. The size of the journal area does not affect any other part of the system. However, if the journal area is small and there is no failure or a system shut down for a long time, the log may grow long and the space allotted to it may get exhausted. Therefore, when the length of the log exceeds a pre-defined threshold value, the log gets cleared except for the currently executing transactions.

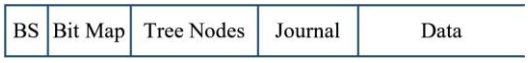

Fig. 4 The layout of the KV Store.

4.1 Metadata Redo Log

Due to durability and consistency issues, metadata and data cannot be written in place in NVRAM. To improve performance, we have kept DRAM versions of metadata, which are updated in place. Changes made to metadata are also recorded in a write-ahead, redo log which is stored in NVRAM. For logging the metadata changes, we have implemented a very simple log structure whose position in NVRAM is shown in Figure 4 (journal). The different entries of a log are shown in Table 1.

Besides log entries, there are two pointers, start log, and end log which are stored along with the log in NVRAM. The log is implemented as a circular array with start log and end log denoting the start and end of the active log. New entries are always appended using end log. Each log entry is 16 bytes long and is written into with two movnti instructions. “end log” is updated using a movnti instruction, only after 16 bytes are written, and then a sfence instruction is issued. This ensures that on a crash, partial log entries cannot exist. This thus ensures an “all-or-nothing” update of the log. The first byte of a log entry defines the nature of the entry. The major challenge to using NVRAM on a memory bus is to ensure that data written to it has actually been stored in NVRAM. Problems arise due to the presence of cache memory in the path from a CPU to memory on the memory bus and the lack of any signal from the memory units on the completion of writes [17]. Here also we have used the same method as DurableFS [1] of re-reading written data after cache flush to ensure that the data has actually reached a durable point.

Table 1 Different Types of Log Entries

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	Parent Key							Left Child Key							P. Ptr
2	Parent Key							Right Child Key							P. Ptr
3	Node Number							Left Child Node Number							P. Ptr
4	Node Number							Right Child Node Number							P. Ptr
5	Key							Value Size							P. Ptr
6	Key							Version		Unused					P. Ptr
7	Key							Read Ref Index		Read Ref Index		Unused			P. Ptr
100	Unused														P. Ptr
101	Unused														P. Ptr
102	Unused														P. Ptr

1: Update left child of K_AVL, 2: Update right child of K_AVL, 3: Update left child of F_AVL, 4: Update right child of F_AVL, 5: Update value size, 6: Update version, 7: Update Read Ref, 100: Begin Transaction, 101: Commit Transaction, 102: End Transaction

Commented [IEU1]: This table needs to be edited. Why are the length of columns different. Why are there node numbers instead of parent key in items 3 and 4? What is P.Ptr?
The types need to be explained in more detail in the form of another table with two columns: column 1 is the type field and column two contains a complete explanation of the entry.

Commented [IEU2]: Needs formatting too

4.2 The PUT Operation

To perform a PUT operation a client has to provide the Key and the Value. This operation can be used to store a single KV pair. Algorithm 1 shows the algorithm of the PUT operation. For storing multiple KV pairs in one node or in multiple nodes we use a different version of the PUT operation with transaction support. As mentioned earlier, in a PUT operation to ensure that the value has actually reached the NVRAM we use different cache flush/ fence instructions with re-reading the last written cache line.

Algorithm 1 PUT for one node
Require: key,value
1: Write Log ”Beg metadata key”
2: Obtain free space F from RAM version of F_AVL of sizeof(value) and update RAM F_AVL
3: Search for key in K_AVL and get node KA; if not present, obtain free node KA for key. Set pointer in KA to F and insert KA into RAM K_AVL for new KV.
4. insert old F space to F_AVL if this is an update.
5: Copy value to F (write is to NVRAM); sfence; clflushopt;
6: Write Log “remove F from F_AVL” (insert old F to F_AVL).
7: Write Log “Insert KA into K_AVL”.
8: Write Log “Commit metadata key”.
9: Reread end log to confirm durability.
10: Update K_AVL and F_AVL in NVRAM; sfence

Commented [IEU3]: How does this match with Table 1 contents? How are we identifying a transaction?

11: Write Log “End metadata key”.

4.3 Transaction over multiple nodes

If a transaction spans multiple nodes, we propose a different algorithm. A transaction over multiple nodes is handled by maintaining a log of committable transactions from participating nodes by the transaction owner. Here each participating node is the owner of one or more of the target KVs and the transaction owner (TO) is the serving node of any request. In this case, the serving node will form multiple independent sub-requests for each participating node. In the first phase, read requests will be sent. Then write requests will be sent. For writes, Algorithm 2 shows the algorithm at the TO, while Algorithm 3 shows the algorithm at each participating node. For writes (Algorithm 3), first, a begin transaction is written (line 1). Then the required NVRAM area (from F_AVL tree) is reserved for each KV. Then each KV is write locked, and all copies in other nodes are invalidated. If the KV is already locked, then there will be a wait and this may lead to a deadlock as mentioned above. Unlike the Put algorithm, instead of writing commit metadata, a lock ok message will be sent to the transaction owner first (line 10). This is required because if we write commit metadata, this write will be recovered after a failure. But it should not be recoverable since failure may happen at some other node and that failure may not be recoverable at that node. That is, a participating node cannot commit the metadata until it is allowed by the transaction owner. The transaction owner will send an allow write message to the participating node once it receives lock_ok messages from all participating nodes. Once a node receives an allow write message from the transaction owner it writes a commit metadata record in the log and executes the log entries and then writes an end commit metadata record (lines 12 to 20). Then the node sends a ready to commit message to the TO. The TO, on receiving such messages from all nodes, sends commit ok to all nodes. A node then writes end transaction and the process is complete (line 22-23). Finally, all locks are released. If an abort comes after the end metadata log record has been written, then the log entries from the beg transaction record have to be removed, and the updates in line 18 have to be undone (undo records have not been shown here for the sake of clarity; the abort process has also not been shown). During recovery, if end transaction is not found in the log record (line 22), then the log actions from beg transaction have to be undone as the transaction was not committed, even though the changes in this node may have been completed. Deadlocks have to be detected and transactions have to be aborted. This has not been shown in the figures for the sake of clarity in reading. If a KV is being accessed in non-transaction mode, then no locks will be acquired. We are thus implementing “advisory locking” and not “mandatory locking”. That means user programs will take care that particular KVs are not being accessed both in transaction mode and non-transaction mode. So, in non-transaction mode, nodes can pick up copies of a KV without the owner knowing about it (through RDMA, once the location of the KV is known). On a write, the owner node issues a broadcast that the KV has been updated. Transactions may also contain read operations. For reads, the transaction owner will obtain a read lock from the KV owner and get a copy through RDMA. A separate algorithm for Reads has not been shown.

Algorithm 2 Write in the transaction Owner
Require: K[M],V [M]
1: Partition the M KVs into N sets, one set for each of N nodes. Let the set for the ith node be denoted by Si of size n, where each element is a two tuple <Kij,Vij>, j=1 ..n;
2: for (k=1; k++; k<=N) do
    3: Send write(Sk) to node k
4: end for
5: while (true) do

```
        6:      i=receive(lock ok)
        7:      Commit node[i]=true
        8:      if (count(commit node)==N) then
        9:      send(allow write) to all N nodes
        10:     break
        11:     end if
12: end while
13: while (true) do
        14:     i=receive(ready to commit);
        15:     ready node[i]=true
        16:     if (count(ready node))==N) then
        17:     send(commit ok) to all N nodes
        18:     break
        19:     end if
20: end while
```

Algorithm 3 Write multiple KVs to a node
Require: S = {<Kj,Vj>|j=1,n}; cardinality (S) = n
1: Write Log ”Beg Transaction S”
2: for (j= 1; j++; j<= n) do
3: Obtain free space Fk from RAM version of F_AVL of sizeof(Vj)
4: Update RAM F_AVL
5: Obtain free node KAj and set Kj and pointer to Fj into KAj
6: Insert KAj into RAM K_AVL
7: Copy Vk to Fk (write is to NVRAM)
8: if Kj is write locked, wait for the lock to be released
9. Else if there are read locks on Kj, send “invalidate” messages to all nodes holding read locks and Write Lock Kj
10: end for
11: Send lock ok to transaction owner (TO) and Wait for allow write from TO
12: for (k= 1; k++; k<= cardinality (S)) do
13: sfence; Write Log “remove Fk from F_AVL”.
14: Write Log “Insert KAk into K_AVL”;
15: end for
16: Write log “Commit metadata S”
17: Reread end log to confirm durability.
18: Update K_AVL and F_AVL in NVRAM
19: sfence
20: Write Log “End metadata S”
21: Send ready tocommit to TO and Wait for commit ok from TO
22: Write log “End Transaction S”
23: Reread end log to confirm durability
24: Unlock all Kj

5 Evaluation

As we could not find any NVRAM based distributed KV store, it was not possible to compare our system with any other similar system. The closest system available for comparison was the distributed version of Redis [23]. This is a memory-only system, with no write-ahead-logs, and no NVRAM support. We compared our system with Redis using the YCSB benchmark. In order to understand how our system performs with different workloads, we desinged our own custom workloads. Our system was

implemented in a cluster of 6 nodes. This was implemented in one Linux server (with 32 GB RAM) by loading six copies our kernel module into a single kernel. Each of the six nodes was allotted 2 GB for use as NVRAM, and RAM data structures were allotted space from the rest of the common RAM area at runtime at module initialisation (insmod). A Redis Cluster with six nodes was also implemented in a single server using six user processes and this was used to compare the two systems using the YCSB benchmark. Since Redis provides only limited transaction support, it was not possible to compare the performance of our system using transactions, with Redis.

5.1 Custom Workloads

We designed our own set of nine workloads and compared their results. These set of experiments were carried out to test our system, and to evaluate the impact of writes, locks, and transactions, on performance. In each workload, 50% of the KVs accessed are local and the remaining 50% are remote. For each experiment, the code was executed 5 times, and the average execution time of these 5 runs was used. All operations are on KVs with 2 MB of data. Real-time is recorded. The result of this workload is shown in Table 2.

In the first four workloads, there is no contention. Every thread operates on a distinct KV. As can be seen, the performance reduces as the number of reads reduces.

It can be also seen that writes are more efficient than updates (comparing 3 and 4). These runs verify the expected behavior. It may be noted that the performance reduction is not very much since reading and writing to RAM takes almost the same time In workloads 5 and 6 we run single operation transactions with each of 30 threads updating one randomly chosen KV out of a set of 30. In 5 we do not do locking, and so the result of 6 as compared to 5 shows the overheads of locking on a single KV. There is a 23% reduction in the rate achieved. In workload 7 we increase the level of contention by making the 30 threads competing to update 10 KVs (instead of 30 KVs in runs 5 and 6). There is a 45% decrease in rate in workload 7 in comparison to workload 6, but with contention among 10 instead of 30 KVs, the result is better than expected. In the last two workloads 8 and 9, we run transactions with multiple operations in each thread. There are 30 KVs and 10 threads. In workload 8 each of the 10 threads performs 5 operations in transaction mode (each operation being randomly chosen as read or write) with each operation operating on a randomly chosen KV. In workload 9, the number of operations per thread is increased to 10, increasing the probability of contention, and increasing the chances of deadlocks. There were deadlocks in both runs. The performance, as expected, goes down drastically in comparison with earlier workload runs.

Table 2 Results of Custom Workloads

Workload No.	No. of Threads (R=read, W=write, U=update)	Total Data (MB)	No of KVs	Access Type	Time (sec)	Rate (MB/Sec)
1	R=20, W=0, U=0	40	20	Each thread accesses distinct KVs	0.518	77.22
2	R=10, W=10	40	20	Each thread accesses distinct KVs	0.554	72.20
3	R=10, W=10, U=10	60	30	Each thread accesses distinct KVs	0.871	68.89
4	R=W=0, U=30	60	30	Each thread accesses distinct KVs	0.899	66.74
5	R=W=0, U=30	60	30	Random access of KVs, no locking	0.887	67.64

6	R=W=0, U=30	60	30	Random access of KVs, locking	1.166	51.46
7	R=W=0, U=30	60	10	Random access of KVs, locking	2.063	29.08
8.	10	100	30	Each thread runs a transaction with 5 operations, each randomly R or W	6.184	16.17
9.	10	200	30	Each thread runs a transaction with 10 operations, each randomly R or W	14.88	13.44

5.2 Contention Based Workload

In this experiment setup, in each node, there are 1000 KVs preloaded and out of them 10 are marked as Shared KVs. The remaining KVs are marked as private and will not be shared among multiple threads. We ran 30 concurrent threads with a read probability R and contention probability C. R tells us whether the thread is read intensive or write intensive. The higher the value of R higher the probability of read. Based on C the access may be to a Shared KV or a Private KV. The higher the value of C higher the probity of contention. In each workload, there are 30 threads that perform 12 operations either read or write of 3 MB data per operation. Hence each run deals with 1080 MB of data. Real-time is recorded for these operations and transfer rates are calculated. The results of these workloads are shown in Table 3. As we can see in the table when the contention is 100% and all thread performs write operation only (SL 3 in Table 3), the system performs the worst. On the other hand, when the contention is 0% and all threads perform read operations the system performs the best. In these experiments, we also observed that in our system reading and writing takes a comparable amount of time if the contention level is 0 (SL 1 and 7 in Table 3).

Table 3 Result of Contention Based Workload

SL	Read	Contention	Time	MB/S
1	0	0	13.29	81.26
2	0	.5	23.93	45.13
3	0	1	119.66	09.02
4	.5	0	13.00	83.07
5	.5	.5	17.09	63.19
6	.5	1	99.72	10.83
7	1	0	11.96	90.30
8	1	.5	12.21	88.45
9	1	1	12.33	87.59

5.3 YCSB Benchmark

Commented [IEU4]: Can we provide details of the YCSB parameters we used in our experiments?

To evaluate performance with more realistic workloads, we ran the YCSB [18] benchmark in the six node setup. We compared the throughput of our system with a Redis cluster by running three YCSB workloads. Workload A is composed of 50% reads and 50% updates; Workload-B has 95% reads and 5% updates; Workload-C includes 100% reads. To run the benchmark, we created six instances of Redis on different ports (using the existing create-cluster script of the Redis distribution) and formed a cluster. We compared the Redis Cluster with two versions of our system- enabling and disabling cache flushes. For each experiment, the code was executed 5 times, and the average result of these 5 runs was used as shown in Table 4. When we disabled cache flush, we observed that our implementation performed slightly better than Redis in read-intensive workloads (Workload B and C). On the other hand, in Workload A (where numbers of Reads and Writes are equal) Redis is better (about 2%) than

ours.. When we enabled cache flush, which ensures data durability in an NVRAM, we observed that Redis is up to 14% better than our system in workloads A and B. But in workload C the performance of Redis is comparable to our implementation. It is to be noted our system incurs overhead due to the implementation of a write-ahead metadata log, which is absent in Redis as it is a memory-only system. From the results, we conclude that the DKVS system's performance is comparable to the performance of Redis.

Table 4 Result of the YCSB Experiment. Throughput(Ops/Sec)

	A (50-50 R-U)		B (95-5 R-U)		C(100-0 R-U)	
	ops/sec	Vs Redis (%)	ops/sec	Vs Redis (%)	ops/sec	Vs Redis (%)
Redis	5076	0	5160	0	5182	0
DKVS(NF)	4950	-2.48	5211	0.98	5219	0.71
DKVS	4365	-14	4644	-10	5078	-2

6 Related Work

Key-Value Stores were first proposed at Carnegie Mellon University's Parallel Data Lab as a research project in 1996 [19]. Each Key-Value is composed of data, a variable amount of metadata, and a globally unique identifier. Such systems are used to store unstructured data. Various such key-value stores [20–22] have been designed and implemented on traditional hard disks. A number of in-memory KV systems have also been proposed as have been hybrid systems with the bulk of data in slower secondary store, and recent data in RAM. With the advent of high-performance, byte-addressable NVRAMs, there have been a number of designs for NVRAM-based systems also. None of the systems reported in the literature have provided multi-KV transaction support as most designs are for NoSQL systems rather than RDMS systems.

Distributed File Systems on secondary storage have been available for quite some time. Most of the systems handle meta data separately as the naming space in a file system requires traversing to find a file [Lustre, NFS4]. So the techniques used are not suitable for a distributed key value store. Clustered File systems [OCFS2] handle a single file system accessed by the nodes of the cluster. Caching of location of file blocks, locking mechanisms, replication use algorithms similar to our design. But the context is different, and emphasis is more on meta data management and distributed space allocation.

Redis [23], is an open-source in-memory key-value store, and is widely used in various enterprise solutions. It uses hashing to locate a value given a key. A distributed (clustered) version is available. The features of the clustered version are described below. Hash slots of a hash table are mapped to different nodes and each such node is the Master of all keys mapping to that slot. In our design, the hash slot equivalent is part of the key prefix. The prefix is used to identify the logical node storing the Master copy of the object. We chose this method as our design is for a relatively small cluster with a fast interconnect. Redis is targeted towards large clusters. Because of this, it allows an object to be replicated and it allows all copies to be accessed by applications for read-only operations (our system does not allow this; as reads are done through RDMA, a node is unlikely to become a bottleneck due to many reads on the same node). Consistency of replicas with the Master is done in Redis asynchronously (as is done in our design) but this requires special care to reduce inconsistent reads and loss of writes due to partitioning (we do not consider partitioning). Due to the possibly large number of nodes in a cluster, failure recovery protocols are much more complex in Redis than in our case. This includes handling of partitioning of the network cluster. In single-server Redis, limited

transaction support involving multiple keys is provided. But full ACID properties are not guaranteed. There is no redo or undo logs being maintained. In clustered Redis, this limited transaction facility is provided only if all the keys map to the same node in the cluster. A read of a replica is allowed but there is no guarantee that it is up-to-date. Redis has not been implemented for NVRAM and so there are no NVRAM durability concerns.

An early paper on the subject describes a system called FaRM [24]. They assume that nodes in a cluster have RAM backed up by batteries to provide NVRAM at every node. On a failure, the NVRAM portion of RAM is written out to SSDs to preserve its contents. Their system therefore does not have to deal with durability issues present in pure NVRAM systems. They use RDMA extensively for most operations. Their system is similar to ours in that they provide transaction support with strict serializability on operations on multiple KVs.

Commented [IEU5]: Could we have used this system for comparison? If not, why not?

HiKV [28] assumes a hybrid memory system of DRAM and NVRAM for a Key Value store. In HiKV, there is no disk, and data is persisted to NVRAM only. They consider only one system, and there is no distributed version. The main idea behind HiKV is the hybrid index: a persisted hash index placed in NVRAM, and a B+ Tree index placed in DRAM. Here, the persistent hash index in NVRAM is used to process read and write operations efficiently and the B+-tree index in DRAM is used to support range query operations. As is well known while a hash-based index has fast lookup time, a B+ tree index allows range queries. Our system ensures fast lookup (in a distributed scenario) by embedding the node location information in the key itself and uses KVL trees instead of B+ trees (both as an index to cached values at remote sites and as an index to the primary KV store at a node). They do not provide transaction support.

In [29], authors proposed Telepathy, a system implemented on a cluster with RDMA and NVRAM which handles replication of Key-value objects. Unlike our system, they allow applications to access any copy of an object. Their protocols implement strong consistency of the copies of an object. They do not consider multi-object transactions like we do. Our design reduces write overhead and handles possible read bottlenecks by caching objects. The first read of an object is more expensive in our case as a read lock has to be obtained.

Caribou [30] is an intelligent distributed system that provides access to DRAM/NVRAM through a key-value interface over the network. The main motive of Caribou is to allow near-data processing. For fault-tolerance Caribou uses replication. It is a hardware-based key-value store that uses field programmable gate arrays (FPGA). The FPGA has no caches in front of memory and therefore reads and writes are directly performed in the memory. The absence of a cache in front of the main memory reduces the complexity to achieve durability during writes.

DrTM+R [6] is a distributed transaction processing system designed to handle increasing data volume and concurrency demands in systems like web services, stock exchanges, and e-commerce. DrTM+R supports in-memory transactions, using HTM (hardware transaction memory) and RDMA. It employs an opportunistic concurrency control (OCC) which uses, during the validation phase, HTM for local resources, and RDMA for remote locking. It also introduces an optimistic replication scheme that utilizes Seqlock-like versioning to manage the race condition between the immediate visibility of records updated by HTM transactions and the delayed replication of those records. Atomic writes to NVM is achieved using HTM instructions, whereas our proposed system is a software solution without any special hardware facility. Our design and implementation seeks to demonstrate that a simple design using standard techniques can perform as well as schemes such as these, which are fairly complicated.

There is no report in the literature of any implementation of a distributed Key Value store supporting multiple object transactions on NVRAM, with durability guarantees.

7 Conclusion

This paper presents a distributed KV store in a cluster of NVRAM-based systems with a high-speed interconnect. No disk-based system is assumed to be present. It shows that the use of standard features in an NVRAM system can result in an efficient system. The novelty is in the overall design rather than the introduction of any new algorithms. The system uses RDMA as a performance enhancement feature and the system can be implemented efficiently without RDMA. Using the concept of static owners of each key-value, writes are handled centrally at the owner's site. Other nodes obtain a read lock from the owner node. (a prefix of every key identifies the owner node) and cache the inode information returned with the lock grant. Data obtained through RDMA (or through the LAN if RDMA is not available) is also cached. On a write, all cached copies are invalidated. The decision to keep location information in the key, instead of using consistent hashing or some such scheme to distribute data, has been mainly motivated by letting higher levels of software to decide on placement of items. This design decision also makes locating key-values easy. Replication for efficiency is not included as the system runs on a fast interconnection network, which enables caching of data in remote nodes. So, replications are there only for fault tolerance, and consistency of copies can be handled asynchronously with low cost. Standard use of write-ahead redo logs at each node along with a coordinator node for each transaction allows the implementations of transactions. Two-PL locking with deadlock detection is used for implementing concurrency control.

To the best of our knowledge, this is the first presentation of a distributed KV store on NVRAM that provides distributed multi-object transaction support. Comparison with other similar systems has therefore not been possible. However, we have compared our system with a Redis cluster using RAM only and no NVRAM. Comparison was done using custom workloads as well as using the YCSB benchmark. The comparisons show for a RAM only implementation, our system's performance is comparable to that of Redis even though our system implements a write-ahead log for metadata changes, which Redis does not. The system performs about 14% slower than Redis if we enable cache flushes required for achieving data durability in NVRAMs. This is the price to be paid for achieving fault tolerance.

References

[1] Kalita, Chandan, Barua, Gautam, Sehgal, Priya: Durablefs: a File System for NVRAM, CSI Transactions on ICT **7**(4), 277–286 (2019)

[2] Xu, J., Swanson, S.: {NOVA}: A log-structured file system for hybrid {Volatile/Non-volatile} main memories. In: 14th USENIX Conference on File and Storage Technologies (FAST 16), pp. 323–338 (2016)

[3] Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Communications of the ACM **51**(1), 107–113 (2008)

[4] RDMA Aware Networks Programming User Manual. Accessed on February 02, 2024. https://indico.cern.ch/event/218156/attachments/351725/490089/ RDMA Aware _Programming user manual.pdf

[5] Kalia, A., Kaminsky, M., Andersen, D.G.: Using rdma efficiently for key-value services. In: Proceedings of the 2014 ACM Conference on SIGCOMM, pp. 295–306 (2014)

[6] Chen, Y., Wei, X., Shi, J., Chen, R., Chen, H.: Fast and general distributed transactions using rdma and htm. In: Proceedings of the Eleventh European Conference on Computer Systems, pp. 1–17 (2016)

[7] Kaiyrakhmet, O., Lee, S., Nam, B., Noh, S.H., Choi, Y.-r.: {SLM-DB}:{SingleLevel}{Key-Value} store with persistent memory. In: 17th USENIX Conference on File and Storage Technologies (FAST 19), pp. 191–205 (2019)

[8] Zaitsev, P.: Innodb architecture and performance optimization. O'Reilly MySQLConference and Expo, April 2009 (2009)

[9] Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A., Vivier, L.: The new ext4 filesystem: current status and future plans. In: Proceedings of the Linux Symposium, vol. 2, pp. 21–33 (2007). Citeseer

[10] Russon, R., Fledel, Y.: Ntfs documentation. Recuperado el **1**, 2 (2004)

[11] Yang, J., Izraelevitz, J., Swanson, S.: Orion: A distributed file system for {NonVolatile} main memory and {RDMA-Capable} networks. In: 17th USENIX Conference on File and Storage Technologies (FAST 19), pp. 221–234 (2019)

[12] O'Neil, P., Cheng, E., Gawlick, D., O'Neil, E.: The log-structured merge-tree (lsm-tree). Acta Informatica **33**, 351–385 (1996)

[13] Rao, J., Ross, K.A.: Making b+-trees cache conscious in main memory. In: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, pp. 475–486 (2000)

[14] Cooperation, I.: Intel architecture instruction set extensions programming reference. Intel Corp., Mountain View, CA, USA, Tech. Rep, 319433–030 (2016)

[15] netlink(7) — Linux manual page. Accessed on February 02, 2024. https://man7. org/linux/man-pages/man7/netlink.7.html

[16] HOWTO CONFIGURE SOFT-ROCE. Accessed on February 02, 2024. https: //enterprise-support.nvidia.com/s/article/howto-configure-soft-roce

[17] Bhandari, K., Chakrabarti, D.R., Boehm, H.-J.: Implications of cpu caching on byte-addressable non-volatile memory programming. Hewlett-Packard, Tech. Rep. HPL-2012-236 (2012)

[18] Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM Symposium on Cloud Computing, pp. 143–154 (2010)

[19] Object storage. Accessed on January 02, 2024. https://en.wikipedia.org/wiki/ Object storage

[20] Srinivasan, V., Bulkowski, B., Chu, W.-L., Sayyaparaju, S., Gooding, A., Iyer, R., Shinde, A., Lopatic, T.: Aerospike: Architecture of a real-time operational dbms. Proceedings of the VLDB Endowment **9**(13), 1389–1400 (2016)

[21] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. ACM SIGOPS operating systems review **41**(6), 205–220 (2007)

[22] Ghemawat, S., Dean, J.: LevelDB, A fast and lightweight key/value database library by Google (2014)

[23] Redis data structure store. Accessed on January 02, 2024. https://redis.io/docs/ get-started/data-store/

[24] Dragojevi´c, A., Narayanan, D., Nightingale, E.B., Renzelmann, M., Shamis, A., Badam, A., Castro, M.: No compromises: distributed transactions with consistency, availability, and performance. In: Proceedings of the 25th Symposium on Operating Systems Principles, pp. 54–70 (2015)

[25] Kourtis, K., Ioannou, N., Koltsidas, I.: Reaping the performance of fast {NVM} storage with {uDepot}. In: 17th USENIX Conference on File and Storage Technologies (FAST 19), pp. 1–15 (2019)

[26] Kim, J., Lee, S., Vetter, J.S.: Papyruskv: A high-performance parallel key-value store for distributed nvm architectures. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–14 (2017)

[27] Kim, S.-H., Kim, J., Jeong, K., Kim, J.-S.: Transaction support using compound commands in {Key-Value}{SSDs}. In: 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19) (2019)

[28] Xia, F., Jiang, D., Xiong, J., Sun, N.: {HiKV}: a hybrid index {Key-Value} store for {DRAM-NVM} memory systems. In: 2017 USENIX Annual Technical Conference (USENIX ATC 17), pp. 349–362 (2017)

[29] Liu, Q., Varman, P.: Silent data access protocol for nvram+ rdma distributed storage. In: 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 1–10 (2020). IEEE

[30] Istv´an, Z., Sidler, D., Alonso, G.: Caribou: Intelligent distributed storage. Proceedings of the VLDB Endowment **10**(11), 1202–1213 (2017)

[31] Liu, X., Hua, Y., Bai, R.: Consistent rdma-friendly hashing on remote persistent memory. In: 2021 IEEE 39th International Conference on Computer Design (ICCD), pp. 174–177 (2021). IEEE

[32] Kalia, A., Kaminsky, M., Andersen, D.G.: {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), pp.185–201 (2016)