Design and Implementation of a File System and a Distributed KV store on Non Volatile Memory

Thesis submitted

in partial fulfilment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

by

Chandan Kalita

Roll No: 11610102

Under the guidance of

Prof. Gautam Barua

Department of Computer Science and Engineering

Indian Institute of Technology Guwahati

Guwahati - 781 039, INDIA

February 2022

Declaration

I certify that:

a. The work contained in this thesis is original and has been done by me under the guidance of my supervisors.
b. The work has not been submitted to any other Institute for any degree or diploma.
c. I have followed the guidelines provided by the Institute in preparing the thesis.
d. I have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.
e. Whenever I have used materials (data, theoretical analysis, figures, and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references.

Chandan Kalita

Roll No: 11610102

This is to certify that the thesis entitled "Design and Implementation of a File System and a Distributed KV store on Non Volatile Memory", submitted by Chandan Kalita, a research student in the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati, for the award of the Degree of Doctor of Philosophy, is a record of an original research work carried out by him under my supervision and guidance. The thesis has fulfilled all requirements as per the regulations of the Institute and in my opinion has reached the standard needed for submission. The results embodied in this thesis have not been submitted to any other University or Institute for the award of any degree or diploma.

Date: 28/02/2022

Guwahati

Gautam Barua,

Professor,
Department of Computer Science and Engineering,
Indian Institute of Information Technology Guwahati,
Guwahati- 781015, India

Acknowledgements

Firstly, I would like to express my sincere gratitude to my thesis supervisor Prof. Gautam Barua for the continuous support of my Ph.D. study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better supervisor and mentor than him for my Ph.D. study.

Besides my advisor, I would also like to thank the rest of my doctoral committee members: Prof. Hemangee K. Kapoor, Prof. Santosh Biswas, and Dr. Gaurav Trivedi, for their insightful comments and encouragement, and also for the hard question which incented me to widen my research from various perspectives. I also like to thank Dr. Tamarapalli Venkatesh sir for helping me in the submission process as an administrative supervisor of my thesis.

The Departments as well as the Institutes, technical and secretarial staff provided valuable administrative support. Here, I would like to specially mention Bhriguraj Borah, Nanu Alan Kachari, Hemanta Kumar Nath, Nava Kumar Boro, Raktajit Pathak, Pranjit Talukdar, Prabin Bharali, Monojit Bhattacharjee, Gauri Khuttiya Deori.

I also like to thank my family for the loving support to carry out this research without which this work would not have been possible.

Abstract

Non-volatile memory (NVRAM) is becoming available. With the availability of hybrid DRAM and NVRAM memory on the memory bus of CPUs, a number of experimental file systems on NVRAM have been designed and implemented. In this thesis we present the design and implementation of a file system on NVRAM called DurableFS, which provides atomicity and durability of file operations to applications. It provides ACID properties to transactions involving multiple files. Due to the byte level random accessibility of memory, it is possible to provide these guarantees without much overhead. We use standard techniques like copy on write for data, and a redo log for metadata changes to build an efficient file system which provides durability and atomicity guarantees to transactions. Benchmarks on the implementation shows that there is only a 7% degradation in performance due to providing these guarantees.

Since the storage devices are not only meant for files, we also have designed a Key-Value store for such hybrid systems. In this thesis we also present 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 hierarchical naming structures, relational database, NoSQL systems etc. Keeping RDBMS 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 to fetch a value from 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. DKVS was compared with a Redis cluster having six nodes and found that our implementation is comparable with Redis with durability and consistency guarantees.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Introduction

Storage is a necessary unit of electronic devices. Different types of storage devices have been in use for decades. Hard disk is the most popular storage medium that is in use since 1956. Because of fast access and low cost per storage unit, hard disks are still being used as the main data storage devices in computers. Due to the mechanically rotating platters, access latency and power consumption in hard disks are higher than Solid State Disks (SSD) which are integrated circuit assemblies and used as memory to store data persistently. Compared with electromechanical magnetic disks, SSDs are typically more resistant to physical shock, run silently, and have less latency. Although SSD latency is smaller than hard disk, the performance gap between memory and storage is still very wide. There are some new memory technologies invented that are categorized as "Storage Class Memory" which we refer to as NVRAM in this thesis, and which reduces the gap between memory and storage [1].

NVRAM combines the benefit of DRAM and SSD. Some recent NVRAM technologies are: Phase-Change Memory (PCM) [2], Spin- Transfer-Torque RAM (STT-RAM) [3], Memristor [4], and Intel 3D XPoint [5]. While each NVRAM technology has different performance measures, they all provide byte granularity access and the ability to store data persistently across reboots without battery backing. Since the access latency of NVRAM is comparable to DRAM [1], it will outperform traditional hard disk drives and SSDs in terms of performance. Besides performance, power efficiency is a major advantage of NVRAM. Out of the total power consumption, a major amount is consumed by DRAM and Hard Disks. Along with direct power consumption, the cooling system of disk drives also consumes a huge amount of power. Another downside of using a Hard disk is space. Large data centers use millions of hard disks which occupy several thousand square feet of area. NVRAM can remove these two major obstacles in designing exascale systems. Now the question is "How to use NVRAM on modern computers?" Some recent work investigates the use of NVRAM within file storage [6] [7] [8] [9] or as a low-power volatile DRAM replacement [2] [10] [11] [12] [13]. Performance-wise NVRAM is comparable to DRAM when attached to a memory bus. From the power consumption point of view, NVRAM outperforms traditional hard disk and DRAM.

Although NVRAMs are large in capacity, less power consuming, and non-volatile in nature, the existing operating systems are designed for a memory device that is volatile, fast, random access, and erased on reboot, and storage device for a persistent, slow block-based device. An operating system manages the volatile memory using Virtual Memory Manager or VMM and on the other hand storage devices are managed using a file system and a block driver. But NVRAM devices are both byte-addressable (like

volatile memory) and persistent (like storage). Traditional system software is not designed to work with such devices. System software can be designed to manage NVRAM in several ways which will allow programmers to create persistent data structures. Thus, trees, lists, and hashes can survive program and system failures. Furthermore, this also should allow direct access to any memory location by bypassing several software layers, including system calls and device drivers which increases latency to storage. To use NVRAM as a storage device we need to think about an efficient storage system that allows us to access such devices using LOAD/STORE instructions. On the one hand, the fast storage randomly accessible at the byte level provides opportunities for new file system designs. On the other hand, the presence of a cache hierarchy on the path to the NVRAM and a lack of feedback when data actually reaches NVRAM poses challenges in providing guarantees on the durability of operations. Introduction of special instructions by Intel (`clwb`, `sfence`, and `movnti`) provides support for providing these guarantees.

As a result, a number of file systems have been designed and implemented for NVRAM [6]- [9]. A study in [14] shows that existing file systems can be ported to NVRAM without much degradation in performance. Previous file systems for NVRAM are POSIX compliant, with durability semantics as is present in UNIX and Linux disk file systems. The main goal of durability in these file systems is to enable a consistent state of the file system to be available after a crash. This involves the logging of only metadata changes. The durability of individual file data operations is expensive in a disk-based system and so is only provided as an option at much-reduced efficiency. With NVRAM as the media of storage, providing the durability of data operations is no longer very expensive. Further, many data-intensive applications need to implement transactions that provide ACID properties to a sequence of operations. Since standard file systems do not provide transaction facilities, such applications either implement restricted versions of transactions [15] or incur significant overheads to implement full ACID transactions. Zhao et al. [12] describe a system that provides ACID properties in an NVRAM file system. It, however, requires non-volatile cache memory too. Although file systems are considered a primary storage system, storage devices are not only made to store files. For example, many vendors have been trying to avoid the overheads associated with a file system for database uses. Oracle ASM file storage is such an example used below DBMSes by the biggest DBMS vendor, Oracle. The availability of large amounts of data has resulted in data storage and manipulation systems such as NoSQL systems, systems for semi-structured data like photo albums, and Map-Reduce systems. Specialized storage systems in the form of Key-Value storage have been designed to provide efficient access to such data. Besides unstructured and semi-structured data, at the same time, many applications still require a traditional file system interface.

This thesis focuses on different prospective of NVRAM and presents two major contributions- the design and implementation of an NVRAM-based file system and a distributed KV store system that is at a lower level than a traditional file system.

1.2 Contributions of the thesis

The first contribution of this thesis is the design and implementation of a file system DurableFS, for NVRAM attached to the memory bus. DurableFS has been designed by keeping in mind that many applications will require support for atomic and durable operations, and so support for transactions should be provided at the file system level.

DurableFS contains the following novel features:

- It is designed on the premise that many applications will require support for atomic and durable operations, and so support for transactions should be provided at the file system level.
- We provide atomicity of file operations between an open and a close of a file, with only a successful close making all changes to the file permanent. We call a sequence of operations on a file starting with an open and ending with a close, a transaction. This feature is provided so that applications that do not use explicit locking can run without change. If an application is providing locking at its level, this feature can be turned off for an opened file.
- We use standard instructions, *clwb*, *sfence*, and *movnti* to implement the above features. Therefore, our system can be used in existing hardware.
- It uses different mechanisms for data and meta management, combining techniques in wide use. Data management is handled by directly accessing data in NVRAM, using the standard copy-on-write used commonly in operating systems, and used in the context of NVRAM in [6]. Metadata management is handled in a manner similar to what is done for all information in ARIES [16]. Active metadata is kept in DRAM to improve access, and a write-ahead redo log is used to note the changes to metadata. However, unlike ARIES, no tail is maintained in DRAM. The log is written directly to NVRAM. Such writes are relatively cheap as the amount of data to be written is small and a *movntq* instruction can be used for the writes. Use of DRAM with a redo log with a log tail also in DRAM (like in ARIES]) has been used for storing variables in NVRAM transactional memory in SoftWrAP [17].

We show through implementation and by comparison with other systems, that the inclusion of atomicity and durability incurs an acceptable loss in performance, which we assert will more than make up the overheads applications will otherwise have to incur if they need these features.

The second contribution of this thesis is the design and implementation of a distributed key-value store DKVS, on NVRAM attached to the memory bus. Key-Value storage is a system that is increasingly being adopted by different products which manage data in terms of Key-Value pairs. It was first proposed at Carnegie Mellon University's Parallel Data Lab as a research project in 1996 [18]. Each

Key-Value is composed of data, a variable amount of metadata, and a globally unique identifier. Such a system is basically used to store unstructured data.

DKVS contains the following novel features:

- Standard feature of providing transparent access in a distributed system to allow scaling of storage as per need.
- Minimal structure of data to allow different higher levels of structuring: RDBMS, NoSQL systems, File systems, all can be implemented efficiently on top.
- As local access to data is at near RAM speed, it is important that remote access does not incur too much overhead. Use of RDMA to read remote data without remote CPU intervention helps in this.
- Provision to replicate KVs for uninterrupted service in the face of node failures.
- Support for concurrent access: Providing concurrent access is expensive; many applications with large, unified data, need concurrent access; providing this facility at a low level gives an efficient implementation, and makes higher-level software easier to implement.
- The key design points are
    - KVs are statically stored and the key identifies the node (virtual node) of storage. This node owns the KV. Higher level applications decide the location of new KVs
    - Two phase locking is used to implement conflict serializable schedules.
    - After obtaining a read lock from the owner of the KV, a node caches a copy of the KV locally and continues to use it till a write occurs which invalidates the cached copies.
    - All writes occur at the owner nodes.
    - The node executing a transaction acts as the transaction coordinator to handle commit of transactions.
    - Read copies are obtained from the owner node through RDMA if this feature is available. Metadata to access the remote KV is obtained with the read lock grant.
    - Multiple copies are maintained at a level of node replication for failure tolerance but copies are not used for reading. This simplifies replication consistency management, and with caching of KVs, the need to access copies is obviated.

1.3 Organization of the thesis

The rest of the thesis is organized as follows:

Chapter 2: This chapter includes some Non-Volatile Memory Technologies and related issues.

Chapter 3: This chapter describes the Design and Implementation of a Durable File System (DurableFS)

Chapter 4: This chapter describes some related works in NVRAM File Systems.

Chapter 5: This chapter describes the Design of a distributed Key-Value store(DKVS).

Chapter 6: This chapter describes the Implementation of DKVS.

Chapter 7: This chapter describes the evaluation of DKVS.

Chapter 8: This chapter describes some related works in NVRAM Key-Value Store.

Chapter 9: This chapter concludes the thesis.

Chapter 2

Non-Volatile Memory

2.1 Introduction

Storage is a necessary unit of electronic devices. Flash memories (mainly NAND Flash) are electronic storage units which are non-volatile in nature and are commonly used. They are now replacing hard disks in units called solid state disks (SSD). The interface is the same as a hard disk, such as SATA or SAS, and so they can easily replace hard disks. Since NAND flash memories are inherently faster than the speeds of existing hard disk interfaces, such devices became available through the PCIe bus using an interface standard called NVMe (Non-Volatile Memory Express). NAND Flash based memory is not suitable for placing on the main memory bus of a computer system because in-place writes cannot take place. New device technologies have been proposed that can handle random reads and writes and memory based on these device technologies can be placed on the memory bus of a computer system. Such memory systems have been given various names, such as Storage Class Memory (SCM), Persistent Memory (PM), and Non-Volatile RAM (NVRAM). We are using the term NVRAM in this thesis.

Some systems refer to memory that is backed up by battery as NVRAM. Such systems can only be used for small memory sizes as the power requirement of DRAMs is large due to the need to refresh memory. Static RAM based battery backed systems will be expensive.

2.2 NVRAM Technologies

Some recent NVRAM technologies are Phase-Change Memory (PCM) [2], Spin- Transfer-Torque RAM (STT-RAM) [3] , Memristors [4], and Intel 3D XPoint. While each NVRAM technology has different performance measures, they all provide byte granularity access and the ability to store data persistently across reboots without battery backing. Neither of the flash technologies is considered as NVRAM because NOR suffers from long erase and write time and NAND does not provide random access to its contents. Since the access latency of NVRAM is comparable to DRAM, it will outperform traditional hard disk drives in terms of performance. Besides performance, power efficiency is a major advantage of NVRAMs. Out of the total power consumption by a computer, a major amount is consumed by Hard Disks. Along with direct power consumption cooling system of disk drives also consume a huge amount of power. Another downside of using Hard disks is space. Large data centers use millions of hard disks which occupy several thousand square feet of area. NVRAM can remove these two major obstacles in designing exascale systems. Now the question is “How to use NVRAM on modern computers?” Some recent work investigates the use of NVRAM within file systems [6] [7] [8] [9], or as a low-power volatile DRAM replacement [2] [10] [11] [12] [13]. Performance-wise

NVRAMs are comparable to DRAM when attached to the memory bus. From the power consumption point of view, NVRAM outperforms traditional hard disk and DRAM.

2.2.1 Phase Change Memory (PCM)

PCM is a random-access non-volatile memory. It uses chalcogenide glass as a phase change material. Switching the state of such material from amorphous to crystallization makes a noticeable difference in its resistive capacity. The resistive capacity of such phase change materials can be up to six orders of magnitude higher in the amorphous state than in the crystalline state. Storing zero and one in PCM cells is different than flash-like memory. Here using electrical pulse, the phase change material is crystallized to represent 1 (SET), and to represent 0 (RESET) large electric current is applied and suddenly stopped to melt-quench the phase change material. PCM is considered to be a very high-performance memory device (comparable to DRAM). For write-intensive applications PCM is a good alternative because unlike flash cells memory elements can be toggled between 0 and 1 without first erasing the entire block of cells [19]. Samsung announced a 64 MB PCM device prototype in 2006. Later in the same year, Intel and STMicroelectronics demonstrated a 16 MB PCM. The first multi-level cell was announced by Intel and STMicroelectronics in 2008. Here addition to the fully amorphous and fully crystalline, partial amorphous and partial crystalline phases are also used. This arrangement of phase change material allows storing two logical bits in a single physical cell. In 2011, IBM also announced high-performance multi-level PCM. However, progress has been slow since then, and the future of this technology is uncertain.

2.2.2 Spin Transfer Torque RAM (STT-RAM)

When an electron passes over a medium, it contains an angular momentum. The angular momentum of spin-polarized current can be transferred to a magnetic layer by directing the current through it. Where the spin-polarized current is nothing but the current passes through a thick magnetic layer. This is called spin-transfer torque. This can modify the orientation of the magnetic layer and flip the orientation of the magnet. This idea is used to flip the active elements in a magnetic random-access memory [3]. This is how STT-RAM (or STT-MRAM: STT-magnetic RAM) works. STT-RAM has been developed by different industries in different capacities. A 32-Mbit STT-RAM was developed by Hitachi and Tohoku University and was demonstrated in June 2009. In 2011, Qualcomm presented a 1 Mbit Embedded STT-MRAM. The first commercially available DDR3 DIMM made up of spin-transfer torque was released by Everspin Technologies in 2012 of size 64 Mb. In 2019, Everspin Technologies started pilot production of 1Gb STT-MRAM chips. Intel has demonstrated STT-MRAM for L4 caches.

2.2.3 Resistive RAM (RRAM or ReRAM)

RRAM is a type of non-volatile (NV) random-access (RAM) computer memory that works by changing the resistance across a dielectric solid-state material. A dielectric material normally does not conduct

currents. But applying a high voltage to such dielectric material may cause a dielectric breakdown. Although dielectric breakdown is permanent damage of the material, for *memristors* the damage is temporary and reversible. That is, deliberately applying voltage produces a conductive filament and changes the resistive capacity of such material. In RRAM the two different states of the conductive filament are used to represent either zero or one. Different RRAM have been developed by different industries using different dielectric materials. At the IEDM conference in 2008, Industrial Technology Research Institute (ITRI), Taiwan demonstrated a high-performance RRAM technology of switching time less than 10 ns. In 2010 ITRI broke their own speed record by showing switching time less than 0.3 ns. In 2012 Panasonic launched an RRAM evaluation kit based on tantalum oxide. A dramatic change in the RRAM technology was made by Crossbar in 2013 by developing a 1 Tb RRAM 3D multi-layer cross-point chip of very small physical size. Fujitsu announced a 4Mb chip for mass production in 2016. Weebit, in 2021 tested and characterized a 1Gb RRAM array using 28nm technology.

2.2.4 Intel 3D-Xpoint

3D XPoint is a non-volatile memory (NVM) technology developed jointly by Intel and Micron Technology. This product was available on the open market under the brand name **Optane** (Intel) since April 2017 [5]. Bit storage is based on a change of bulk resistance, in conjunction with a stackable cross-gridded data access array. In terms of cost 3D XPoint products are cheaper than dynamic random-access memory (DRAM) but costlier than flash memory.

The system is available in the form of an SSD on PCIe (NVMe) (Intel Optane SSD) or in the form of a DIMM memory modules (Intel Optane Persistent Memory (Pmem)). The latter requires special support from the CPU, and the latest Intel chips have Optane support built-in. Using 512 Gb modules, the Pmem200 series offers up to 4TB of NVRAM per socket (and 2 TB of RAM for a total of 6 TB per socket of main memory) on Intel third generation Xeon processor based 2 socket systems. It provides a bandwidth of 4.64 GB/sec on 256B operations with 67% reads and 33% writes. The system runs at 3200 Mhz. This is about 80% of the performance of RAM that will co-exist with the NVRAM.

It can thus be seen that NVRAM is now available commercially with large memory capacities.

2.3 CPU Support for NVRAM

2.3.1 The Memory Subsystem

The memory subsystem is one of the most important parts of CPU architecture. In this section, we will explain how a processor reads (writes) data from (to) memory. When the processor needs to read from or write to a location in main memory, it first checks whether a copy of that data is in the cache. If so,

the processor immediately reads from or writes to the cache, which is much faster than reading from or writing to main memory. Most modern desktop and server CPUs have at least three independent caches: an instruction cache to speed up executable instruction fetch, a data cache to speed up data fetch and store, and a translation look-aside buffer (TLB) used to speed up virtual-to-physical address translation for both executable instructions and data. Data is transferred between memory and cache in blocks of fixed size, called cache lines. When a data read occurs from a specific memory location a cache line is copied from memory into the cache along with the address from which the data is read. Data and the address together are called a cache entry. When the processor needs to read or write a location in the main memory, it first checks for a corresponding entry in the cache. The cache checks for the contents of the requested memory location in any cache entry that might contain that address. If the processor finds that the memory location is in the cache, a cache hit has occurred. However, if the processor does not find the memory location in the cache, a cache miss has occurred. In the case of a cache hit, the processor immediately reads or writes the data in the cache line and a cache miss allocates a new entry and copies in data from the main memory. If some data is written to the cache, the processor needs to update the corresponding memory location at some point. The timing of this write is known as write policy. There are two write policies- write-through cache and write-back cache. In a write-through cache, every write to the cache causes a write to the main memory immediately. In a write-back cache, writes are not immediately written to the main memory, instead, track which locations have been modified. Marking those locations as dirty and the data in these locations is written back to the main memory only when that data is evicted from the cache. It depends upon cache eviction policies. In the existing model processor cache and main memory, both are volatile. Therefore, in case of system failures, all data is lost. Therefore, consistency between cache and main memory is not violated. But in presence of non-volatile memory, the situation is different. Now inconsistency between volatile cache and non-volatile RAM will arise.

In a modern system, there is more than one level of cache memory. The first level (L1) will typically be specific to a CPU core (instruction and data caches are usually separate). L2 cache will also be on chip, but is likely to be shared by all the CPU cores in the chip. There may be a L3 cache which is outside the CPU chips and which may be shared across multiple CPU chips, if they exist. Further, there are a number of intermediate units between the last cache and main memory. Figure 2.1 shows a typical memory subsystem. The Store buffer, the cache hierarchy, the WC buffer, are all volatile memory structures. The only non-volatile component in the figure is the NVRAM. The major challenge to using NVRAM is to ensure that data written to it has actually been stored in NVRAM. Only if data is written to NVRAM, is any change durable, that is, the change survives a subsequent failure. Further, it is required in many applications (such as transaction processing based applications) to inform an external unit of the completion of certain operations. The reason for the uncertainty is 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 (the area inside the shaded rectangle) on the completion of writes. But first of all, data is only written into the cache hierarchy and it may stay there for a long time. If we wish to be sure that data is actually written to NVRAM, we must be able to either write to NVRAM bypassing the cache hierarchy, or by being able to explicit flush data from the caches through instructions. Bypassing the cache for all accesses will become very inefficient. Intel has introduced a set of instructions to handle these issues. These instructions are briefly described now.

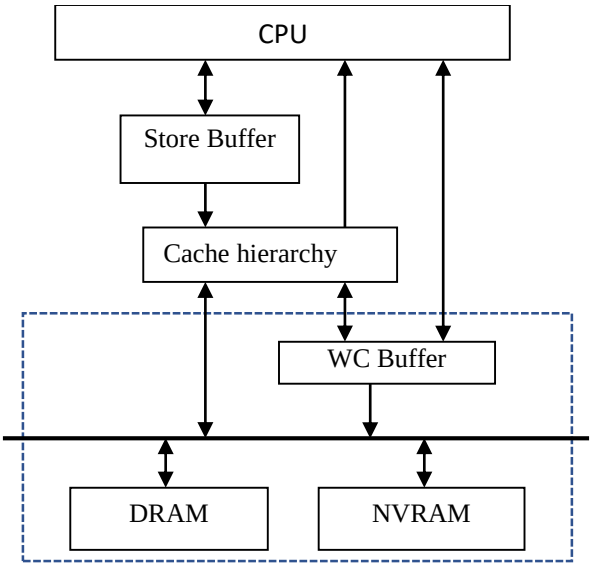


Figure 2.1: Memory Sub System

2.3.2 Intel Instructions for NVRAM

2.3.2.1 Cache Flush and Atomic Store Instructions

These instructions can be used to explicitly write the contents of a cache line (flush) to main memory. Since a line of cache is identified by a memory address, the instructions will automatically flush data through the cache hierarchy (thus if the cache line is in a L1 cache, it will be flushed to the L2 cache, then to the L3 cache, and then to main memory).

`CLFLUSH m8, CLFLUSHOPT m8, CLWB m8: CLFLUSH` invalidates the cache line that contains the linear address (m8) specified with the source operand from all levels of the processor cache hierarchy (data and instruction). The invalidation is broadcast throughout the cache coherence domain. If, at any level of the cache hierarchy, the line is inconsistent with memory (dirty) it is written to memory before invalidation. The source operand is a byte memory location. `CLFLUSHOPT` is an optimized version and is preferred over CLFLUSH. CLWB writes back to memory the cache line (if modified) that contains the linear address specified with the memory operand from any level of the cache hierarchy in the cache coherence domain. The line may be retained in the cache hierarchy in non-modified state. Retaining the

line in the cache hierarchy is a performance optimization (treated as a hint by hardware) to reduce the possibility of cache miss on a subsequent access. Hardware may choose to retain the line at any of the levels in the cache hierarchy, and in some cases, may invalidate the line from the cache hierarchy.

The above flush instructions are implemented asynchronously. That is, the CPU does not wait for the completion of the instruction before continuing with the next instruction. If there are two flushes to the same cache line issued one after the other, then they are ordered, but if there are two cache flush instructions for different cache lines, then there is no guarantee of order of writes of these two to main memory. It is therefore possible that a second flush completes before a flush issued earlier and so may survive a crash, while the earlier flush does not. In order to impose ordering, Fence instructions have to be used in conjunction with cache flushes.

`sfence`: Performs a serializing operation on all store-to-memory instructions that were issued prior the SFENCE instruction. This serializing operation guarantees that every store instruction that precedes in program order the SFENCE instruction is globally visible before any store instruction that follows the SFENCE instruction is globally visible. The SFENCE instruction is ordered with respect store instructions, other SFENCE instructions, CLFLUSH, CLFLUSHOPT and CLWB instructions, and MOVNTx instructions. Issuing an sfence instruction after a CLWB ensures that subsequent writes will be done after the CLWB flush completes. So, if we wish to order two writes to NVRAM, we need to use the following order of instructions: write-to-memory (m8) instruction, CLWB m8, sfence, write-to-memory (m'8) instruction.

If data is to be made durable in NVRAM, the sequence of instructions will be mov m8 -> CLWB m8 -> sfence. Note that this may need for a 64 byte cache line to be first read into cache, 8 bytes written into cache, and then the entire 64 byte cache line to be written to memory. Now if we only wanted to ensure that the 64-bit quantity whose address is m8 is made durable in NVRAM, we are incurring a lot of overhead by reading and flushing an entire cache line (typically 64 bytes). Intel provides a set of "non temporal" store instructions which can be used for this purpose. We mention only one of them here: MOVNTQ.

`MOVNTQ m64, mm`: Moves the quadword in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to minimize cache pollution during the write to memory. The source operand is an MMX technology register, which is assumed to contain packed integer data (packed bytes, words, or doublewords). The destination operand is a 64-bit memory location. The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. Although this instruction is present to reduce overhead for writes whose values will not be read again soon (hence the term "non-temporal"), a MOVNTQ followed by an SFENCE instruction

becomes an atomic write to NVRAM. It is atomic, in that either the entire 8 bytes is written or nothing is written. A cache flush on the other hand, may result in only some of the 64 bytes reaching memory at the time of a crash.

2.3.2.2 Support for ADR and e-ADR

Use of the above instructions only ensures that data reached the WC buffer in the memory subsystem (Figure 2.1). If a system crash occurs, then the data was not written to NVRAM. To prevent this from happening, Intel has introduced an "Asynchronous DRAM Refresh" (ADR) scheme in its new server CPUs. This scheme ensures that once data reaches a WC buffer, it is guaranteed to be written into the memory DIMMs even on a power failure (the area inside the dotted rectangle is the ADR region). While this has increased the likelihood of stores reaching NVRAM, guarantees are still not possible as the clwb instruction (and it variants) is not synchronous, completing the instruction before the flush has actually completed. The only way we can be sure that data has reached NVRAM is to write to NVRAM, clear data from all the stores in the path from the CPU to NVRAM, and then to read the data again from NVRAM to check if what is obtained is what was written. If a sequence of writes is followed by an sfence instruction and then another write and clwb is done, only the last write needs to be read again to ensure that all the writes have reached NVRAM. Intel has introduced yet another scheme called e-ADR in its systems supporting NVRAM pmem200 Optane memory. With e-ADR, all cache contents will be flushed to the memory units on a power failure. If this becomes available widely, then there will be no need to use clwb and sfence instructions to ensure durability of data in NVRAM. As e-ADR is a newly introduced feature, we have used clwb, sfence, and reading the last write to implement durable writes to NVRAM. Further, e-ADR implementation will most likely require a battery of some sort as with the large cache sizes on systems today, a lot of data transfer to memory will be required. So it may not see widespread use.

Chapter 3
DurableFS

3.1 Introduction

File systems are the most widely used storage technique to store data on a disk. Hard disks have been considered as the main secondary storage device in a computer for quite some time. Solid State Disks (SSDs) are becoming popular for their low access latency and compact physical size. Both hard disks and SSDs are attached to the IO bus of a computer and accessed as a block device. The invention of NVRAM devices which can be attached to the memory bus of a computer gives a new direction of research where researchers began to think about how to store data in NVRAM (using load and store instructions). With the availability of hybrid DRAM-NVRAM memory on the memory bus of CPUs as shown in Figure 3.1, a number of file systems on NVRAM have been designed and implemented [6] [7] [8] [9]. The implementation of a file system for an NVRAM device is not straightforward. This is because, although NVRAM has near RAM latency and a randomly accessible interface at the byte level that provides new file system designs opportunities, the presence of a cache hierarchy on the path from CPU to the NVRAM and a lack of feedback system when data actually reaches NVRAM, pose challenges in providing guarantees on the durability of operations. Introduction of special instructions by Intel (`clwb`, `sfence`, and `movnti`) provides support for providing these guarantees. As a result, a number of file systems have been designed and implemented for NVRAM. A study in [14] shows that existing file systems can be ported to NVRAM without much degradation in performance, but without utilizing the full potential of NVRAM. Previous file systems for NVRAM are POSIX compliant, with durability semantics as is present in UNIX and Linux disk-based file systems. The main goal of durability in these file systems is to enable a consistent state of the file system to be available after a crash. This involves the logging of only metadata changes. Due to the comparatively low bandwidth of disks, the durability of individual file data operations is expensive in a disk-based system. Therefore, data logging is only provided as an option at much-reduced efficiency. With NVRAM as the media of storage, providing the durability of data operations is no longer very expensive. Further, many data-intensive applications need to implement transactions that provide ACID properties to a sequence of operations. Since standard file systems do not provide transaction facilities, such applications either implement restricted versions of transactions [15] or incur significant overheads to implement full ACID transactions. Keeping in mind the importance of durability and atomicity we have designed a file system, DurableFS, which, provides transactions with ACID properties. In the current implementation, we have provided a transaction facility between the open and the close of a file. This will allow efficient implementation of RDBMS, NoSQL, and other data-intensive applications on NVRAM. However, this feature is optional. Since our implementation is only a prototype, a separate ioctl call to disable

transactions has not been implemented. Our design is based on a standard Ext3 file system design, including a meta-data log, except that we use copy-on-write for data blocks. Since Ext3 is primarily designed for hard disk drives, there is no need of flushing data from any volatile store to achieve durability. But as mentioned earlier, for a memory based storage we need to flush all volatile store and therefore, we include provisions for flushing data to NVRAM from the cache hierarchy. In our system metadata is also cached in DRAM in the normal way it is done in a Linux file system so that it can be accessed rapidly (DRAM is still faster than NVRAM). Our work demonstrates that a file system can be implemented efficiently on NVRAM using standard techniques, with the added bonus of providing support for ACID properties to transactions.

Our system contains the following novel features:

- It is designed on the premise that many applications will require support for atomic and durable operations, and so support for transactions should be provided at the file system level.
- We provide atomicity of file operations between an open and a close of a file, with only a successful close making all changes to the file permanent. We call a sequence of operations on a file starting with an open and ending with a close, a transaction.
- We provide durability of changes to files at the close of a file. These features are over and above the consistency guarantees existing file systems on NVRAM provide.
- We use standard instructions, `clwb`, `sfence`, and `movnti` to implement the above features [20]. Further, since we have implemented on DRAM only, we do a read after write of the last change to ensure completion of changes.
- It uses different mechanisms for data and meta management, combining techniques in wide use. Data management is handled by directly accessing data in NVRAM, using the standard copy-on-write used commonly in operating systems, and used in the context of NVRAM in [6]. Metadata management is handled in a manner similar to what is done for all information in ARIES [16]. Active metadata is kept in DRAM to improve access, and a write-ahead redo log is used to note the changes to metadata. However, unlike ARIES, no tail is maintained in DRAM. The log is written directly to NVRAM. Such writes are relatively cheap as the amount of data to be written is small and a movntq instruction can be used for the writes. Use of DRAM with a redo log with a log tail also in DRAM (like in ARIES]) has been used for storing variables in NVRAM transactional memory in SoftWrAP [17].
- We show through an implementation and by comparison with another system NOVA [21], that the inclusion of atomicity and durability incurs acceptable loss in performance, which we assert will more than make up the overheads applications will otherwise have to incur if they need these features.

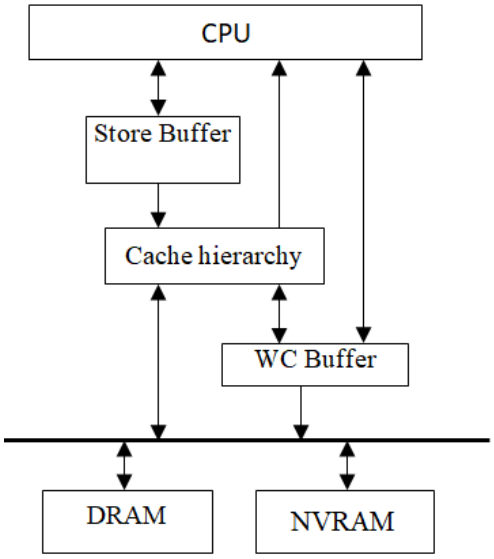


Figure 3.1: Intel Cache-Memory Architecture.

3.2 Durability and Consistency in a Cache Based System

As mentioned in chapter 2, the major challenge to using NVRAM is to ensure that data written to it has actually been stored in NVRAM but not in a volatile store. Only if data is written to NVRAM, is any change durable, that is, the change survives a subsequent failure. Further, it is required in many applications (such as transaction processing-based applications) to inform an external unit of the completion of certain operations. The reason for the uncertainty is 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 or acknowledgement from the memory units on the completion of writes. Although there is a durability issue while writing to disks in a traditional system, interrupts from the disk subsystem are signals to the CPU of the completion of operations. Figure 3.1 shows a simplified version of an Intel x86-64 architecture [22]. As the figure shows, even in the presence of NVRAM, volatility remains an important part of the memory hierarchy. If a program crashes because of a hardware or power failure, only the state present in the NVRAM at the time of the crash will survive a restart. Any state that was present in volatile structures at the time of the crash will be lost. Since the cache memory is under hardware control, data may remain in cache for long periods of time. In order to ensure that critical data survives a crash, programs must be able to flush data in a cache to memory. The Intel architecture has different instructions to achieve this: `clflush`, `clwb` and `movnti` are some such instructions. Details have been discussed in the previous chapter. Although the `clflush` and `clwb` instructions allow us to flush cache lines to memory, there is another instruction `movntq`, which can write 8 bytes of data from a register to memory atomically. It writes data using a non-temporal hint to minimize cache pollution during writes to memory. These instructions flush a cache line from the cache hierarchy to memory or writes data atomically to memory. However, the operations are asynchronous and so a sequence of flush

instructions may not write data into memory in the same order as the instructions. In order to ensure ordering among operations, the `sfence` instruction ensures that all writes before the instruction takes place before any succeeding writes [20]. Even after data is flushed from the cache system, it may be in the WC buffer and so may not survive a subsequent crash. To prevent this from happening, the Asynchronous DRAM Refresh (ADR) scheme of Intel can be used. This scheme ensures that once data reaches a WC buffer, is guaranteed to be written into the memory DIMMs even on a power failure. While this has increased the likelihood of stores reaching NVRAM, guarantees are still not possible as the `clflush` instruction (and its variants) is not synchronous, completing the instruction before the flush has actually completed. The only way we can be sure that data has reached NVRAM is to write to NVRAM, clear data from all the stores in the path from the CPU to NVRAM, and then to read the data again from NVRAM to check if what is obtained is what was written. If a sequence of writes is followed by an `sfence` instruction and then another write and `clflush` is done, only the last write needs to be read again to ensure that all the writes have reached NVRAM. Another problem is that of consistency. If the NVRAM is in a consistent state and a change takes it to another consistent state, then if the change is not atomic, then the NVRAM may become inconsistent. For example, if a change involves a line of cache, after a `clflush` only part of the cache line may have been written to the WC buffer before a power failure. Intel provides the `movntq` instruction which writes 8 bytes from a register to memory atomically. The instruction bypasses the cache hierarchy, writing all the 8 bytes into a WC buffer in one operation. This instruction can be used to make atomic changes in NVRAM and is used in our design to implement a write-ahead log.

3.3 Design and Implementation

Like a traditional system, in DurableFS, each file is identified by an inode. For the sake of simplicity, we kept the inode structure very simple and have not implemented ownership, access control etc. As shown in Figure 3.2, the inode structure is composed of only four fields: i_blocks, which represents the number of blocks being used by this object, a pointer i_block which points to a tree of pointers to data blocks (the tree grows as the file grows), i_size which represents the actual size of the file and type representing the type of the file: directory, file, symbolic link. Directories are files as in Ext3, and their structure is also the same as that in Ext3.

```
struct inode

{

    int   i_blocks; //# of blocks in the file

    int * i_block; //root pointer of a tree of pointers to blocks

    int   i_size; //size of the file in bytes

    int   i_type; // file type: file, directory, symbolic link

}
```

Figure 3.2 Structure of an inode

The file system design is simple because all the metadata and code ensuring contiguity of related data in a file system, are no longer required since NVRAM supports random access at no additional cost. The file system layout is given in Figure 3.3 and the fields are described in Table 3.1. Here the first 4 bytes (TS) represents the total size of the NVRAM in KB. The 5th (BS), 6th (BB) and 7th (IB) bytes represent the block size in KB, number of blocks for the free block bitmap and number of blocks for the inode bitmap respectively. The free block bitmap (FB Map) is of size BS*1024*BB bytes which starts from the 8th byte. The next BS*1024*IB bytes is the free inode bitmap (FI_Map) and next to it, BS*1024*IB*8*sizeof (inode) bytes represents the inode table (I_Table). The next K (configurable) blocks contain the log, and the remaining portion of the NVRAM is used as data blocks.

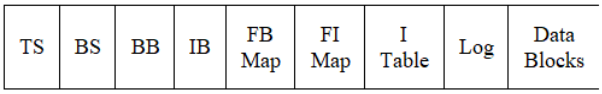


Figure 3.3: File System Layout.

3.3.1 Metadata Redo Log

Due to durability and consistency issues, metadata and data cannot be written in place. This is due to the limitation of size of an atomic write. A system failure or a crash of the write call during an ongoing write may lead to inconsistencies between the data and metadata of a file system if we allow in place writes. Copy on Write is a standard alternative of in place writes. However, after creating a new copy of an existing block, all metadata has to be updated. File system metadata contains several crucial information like the superblock, i-nodes, indirect data pointers, etc. corruption of which can crash the entire file system. Therefore, special care has to be taken to update metadata of a file system so that if

there is a failure at any stage, a recovery routine can recover the file system. To update metadata in such a manner that recovery will be possible in case of a failure, logging of metadata is a standard and useful technique. File system logs enable rapid and clean recovery of file systems if a system goes down. There are two types of logging, Redo and Undo. In redo logging, the new data that has to be written is written to the log and made durable before writing it to the actual location. The data is written to the actual location only after the transaction commits successfully. On the other hand, in undo logging the old existing data is first copied from the actual location to the log and made durable. Once the old data is durable, the new data is written to the actual location in place during the transaction. In the event the transaction fails, in case of undo logging all modifications to the actual location are rolled back using the old data in the undo log and in the case of redo log if the transaction was committed all entries in the redo log has to be re executed. If the transaction was not committed, all the log entries of that transaction is ignored. In undo logging we have to ensure the durability of each log entry. On the other hand, in redo logging we have to ensure that the *commit* record is durable after a fence call. In case of redo logging, all reads as part of the transaction have to first search the redo journal for the latest copy [9]. For this reason, in normal circumstances redo logging incurs an additional overhead for all the read operations in a transaction. However, in our design we have two versions of metadata- RAM and NVRAM. During a transaction the RAM version of metadata is updated in place and therefore any subsequent read of the transaction gets the latest copy of data. Therefore, in our design we have used write-ahead redo logging which is stored directly in NVRAM and which uses small writes to reduce the number of instructions required to make log entries durable. In our design, a copy of each metadata is made to act as a buffer and changes are made to the buffer first. The buffer is implemented in DRAM to take care of cases where NVRAM access may be slower than that of DRAM. 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 which allows us to only log the metadata changes but not an entire metadata block. Space is pre-allocated for the log in the file system, and besides space for the log entries, there are two pointers, *start_log* and *end_log* which are stored along with the log in NVRAM. The log space is used 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*. On a failure, the recovery process will clear the log. But if there is no failure or a system shut down for a long time, the log may grow very long and space allotted to it may get exhausted, so, when the length of the log exceeds a pre-defined threshold value, the log gets cleared except for the currently executing transactions. Each log entry is 16 bytes long and is written into with two `movntq` instructions. *end_log* is updated using a `movntq` instruction, only after 16 bytes are written, and then an `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 different entries of a log are shown in Table 3.2. Depending on the Entry Type, entries Data1, Data2, and Data3

may be present or the fields may be blank. Every entry is identified with a Transaction Number. An entry contains an offset (previous pointer) to the previous entry made for this transaction. Entry types are: *set/reset inode bit map; set/reset free block bitmap; update inode logical block address; update inode i size; update inode i blocks; begin transaction; commit transaction; end transaction*. The only type with three parameters is update inode logical block address: *Data1=inode number, Data2=logical block number, Data3=data block address*. When a transaction begins, a variable in the kernel keeps the index of the last entry for that transaction in the log. This is used when appending a new entry and the variable is updated with the index of the current entry.

Field	**Size (Bytes)**	**Description**
TS	4	Total size of the disk in KB.
BS	1	Block size in KB.
BB=(TS/BS)/BS*1024*8	1	No. of blocks for Free block bitmap.
IB	1	No. of blocks for Inode bitmap.
FB_MAP	BS*1024*BB	The free block bitmap.
FI_MAP	BS*1024*IB	The free inode bitmap.
I_TABLE	BS*1024*IB*8*size of(inode)	The inode table.
Start_log	8	First block of the log.
End_log	8	Last block of the log.
LOG	BS*2	The Log; First 8 bytes is the log end pointer.
Data Blocks		

Table 3.1: Description of the proposed file system layout.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	Target Bit Position of inode bitmap.				Set/Reset				Unused				Transaction No.		Previous Pointer
2	Target Bit Position of free block bitmap.				Set/Reset				Unused				Transaction No.		Previous Pointer
3	Inode number				Index of *i_block[]*				Data Block number				Transaction No.		Previous Pointer
4	Inode number				Value of *i_size*				Unused				Transaction No.		Previous Pointer
5	Inode number				Value of *i_blocks*				Unused				Transaction No.		Previous Pointer
100	Unused (Begin transaction)												Transaction No.		Previous Pointer
101	Unused (Commit Transaction)												Transaction No.		Previous Pointer
102	Unused (End Transaction)												Transaction No.		Previous Pointer

1: set/reset inode bit map.

2: set/reset free block bitmap.

3: update inode logical block address.

4: update inode i_size.

5: update inode i_blocks.

100: begin transaction.

101: commit transaction.

102: end transaction.

Table 3.2: Log Entries.

3.3.2 Data Blocks

We perform a copy-on-write whenever there is a write request to a data block. If a part of a block is to be written into, the block is copied to a free block and updates take place there. If a full block is to be written, then the write takes place on the new, free block. An update of the inode makes the changes permanent. The scheme is similar to that implemented in [6]. In Figure 3.4 the sequence of operations for a low level write function, WRITE (i_number, buffer, size, offset), is given. Buffer is a pointer to user memory containing the data to be written to the file with inode *i_number*, size is the number of bytes to write, and offset is the offset in the file to write to. For ease of exposition, it is assumed that the write is to a single whole block or to part of one block only. The case for larger writes follows a similar pattern. The design assumes that the difference in access times of DRAM and NVRAM will be small, and so the overheads of keeping copies in DRAM of NVRAM data, is avoided, even though the

overhead of flushing cache lines on a write has to be incurred. Since every write to separate areas require separate flushing, metadata is handled, as mentioned above, with copies in DRAM.

```
1. Copy the inode i_number from NVRAM to RAM if it is not already
   in RAM.
2. Determine the logical block number i from the offset and obtain
   the starting block S to which write has to take place, by
   traversing the tree with root at i_block and finding entry i.
3. Find a new free block F from the RAM copy of the free block
   bitmap (fbb)
4. If partial modification of a block is to be done, Copy the block
   of data from S to F
5. Write size bytes from buffer to F
6. Flush cache lines containing data written into, using clwb
7. Execute the sfence instruction;
8. Set Fth bit of RAM fbb;
9. Write entry to log: set Fth of fbb
10. If partial modification Write entry to log: reset Sth bit of fbb
11. Update the RAM copy of the inode of i_number: (entry i's pointer,
    i_size if required, i_blocks if required)
12. Write entry to log: block i = F of i_number
13. Write to log: i_size of i_number
14. Write to log: i_blocks of i_number
// log writes don't need clwb as movntq instructions are being used
// write entry to log includes a sfence instruction at the end.
```

Figure 3.4: Write function.

3.3.3 Atomicity and Durability

In order to provide atomicity and durability, we need to define a point of durability for file operations. Under Linux semantics, there are no guarantees of durability of file writes since writes may be only to the buffer / page cache. A "flush" operation is required if durability is required. Even with this,

durability support is weak, since there is no acknowledgement of storage completion in NVRAM. As we assume that the file system will be used to provide durable writes, we need to define a point of durability, which we define to be the closing of a file. On a successful close, the changes to the file is also made permanent, thus providing atomicity of operations between an open and a close of a file. Consistency and Isolation, if required, have to be provided by applications. Operations other than reads and writes are individually treated as atomic and durable. This implementation can be extended to support full-blown transactions involving multiple files, with the open of a file corresponding to locking the file, and the closing of a file to unlocking the file. This may require a commit system call which should identify a set of open files and all these files should be closed together, and the operations on these files should be treated as a transaction and the transaction should be committed. The current implementation of having only one file's open and close forming the beginning and the end of a transaction was to allow existing programs to run without modification (that is, without having to add commit calls). This allows us to compare the performance of our system with other implementations.

3.3.4 Implementing Durability

As can be seen in Figure 3.4, after data is copied to block F, the cache lines are flushed. This is done to ensure that before making log entries for updating the inode of the file concerned (the log is in NVRAM), data written to block F is flushed. The `sfence` following the flush is to ensure that the log entry writes using `movntq` instructions take place after the data is in NVRAM. The `clwb` instruction is of the form `clwb m8`. The cache line containing the memory address m8 is flushed to main memory. The cache line size is 64 bytes. So, to flush a 4KB data block whose starting address is A, we have to issue 64 `clwb` instructions of the form `clwb A, clwb A+26, clwb A +26 + 26`, and so on (64 X 64 = 4KB). When a file is closed, the situation is as follows: all data writes have been made to NVRAM and cache flushes issued for these writes. All changes to metadata as a consequence of the file operations have been made in the RAM copy of the concerned metadata (except for release of resources). All metadata change operations are written into the log. Now, a *commit transaction* record is written to the log and then the changes recorded in the log for this transaction are made in the actual metadata locations in NVRAM. When all such changes have been made, an *end transaction* record is written into the log. If a metadata change involves a reset of either the free block bit map or a free inode bitmap, then these changes are not made in the RAM version of these bitmaps. Suppose transaction A deletes a block x from the file it is operating on. The x^{th} bit of the free block bit map has to be reset. But this should be done only when the transaction commits as otherwise there can be problems. If a subsequent transaction B requires a free block, it may find x and use it. Later, transaction B commits, but before A can commit, a failure occurs. So, no changes are made to the inode of transaction A and one of its entries is still pointing to x. But x has been reused by B. On the other hand, if transaction A requires a free block and acquires block y, it must set the y^{th} bit of the free block bit map as otherwise another transaction may also use block y, finding it free. During recovery, if it is found that transaction

A has not committed, while nothing needs to be done in the metadata in NVRAM, the setting of bit y in the free block bit map in the RAM copy must be undone. The sequence of operations on a commit is shown in Figure 3.5.

1. Write *commit transaction* Record in log;
2. Execute sfence
3. Read the last record of the commit record and compare with the value written (stored in a CPU register); If the value read matches, commit record (and all previous records) is durable in NVRAM.
4. Follow the back pointers in the transaction records, and for each record, make necessary changes in the metadata in the file system in NVRAM.
5. If a change is a reset to a bitmap, make that change to the RAM version too.
6. Flush all cache lines involving the metadata changes
7. Execute sfence
8. Write end transaction record in the log.

Figure 3.5 The Commit Process on the close of a file

We must ensure that the commit record is durable before we can make changes to the metadata in NVRAM. As already discussed, the only way to ensure this is to reread the data already written into NVRAM. Since we are using `movntq` to write into the log, and since `movntq` bypasses the cache hierarchy, there is no need to flush any cache lines. The `sfence` is required though, to ensure that metadata changes are made to NVRAM only after the commit record actually reaches NVRAM.

3.3.5 Recovery

If a crash occurs, we redo the entire log for all transactions that have a commit transaction but not an end transaction entry in the log. Changes are written to the actual locations of the metadata in NVRAM. The RAM version of the metadata has in any case got wiped out due to the crash. Because of the `sfence` instruction between data block writes and the log write, if we find the log write in NVRAM after a crash, it means that the data writes have also reached the NVRAM (`sfence` ensures that previous writes to memory take place before subsequent writes). If there are log entries for a transaction whose commit transaction entry is not there in the log, then that transaction is assumed to have been

aborted. Since changes to metadata due to this transaction (which are in the log) are not redone, the effects of the transaction are not persisted. The data blocks that were written into may have been written into NVRAM, but they remain free in the free block bitmap and so the contents are not used. Once the changes made by a transaction, as noted in the log have been made permanent in NVRAM metadata, an end transaction entry has to be written. In the next round, when end transaction is found, we are sure that the changes are permanent and so the entries of this transaction can be removed. The basic scheme is similar to the ARIES system [16]. However, since the log is only redo and there is no undo, there is no need to write "compensatory log entries" as described in ARIES. A failure taking placed during a recovery has no impact, in that in the second recovery, the entire process will be repeated. Since all changes to metadata are idempotent, repeated changes to the same metadata can be made. Recovery starts from the end of the log as shown in Figure 3.6 (which is the "first" entry in the description below):

```
Loop:
endptr ← end_log
Loop1:
Entry ← log[endptr]
Unused_flag ← True
if Entry == "end transaction" then
   Follow the back pointers and make every entry traversed
      "unused". The end transaction entry is made "unused" last,
      to handle a crash during a recovery.
   Unused_flag = false
end if
if Entry == "commit transaction" then
   Make changes to metadata as per the entries which are followed
      through the back pointers. Flush all relevant cache entries,
      and execute sfence. Write end transaction in the log.
   Unused_flag = false
end if
if Entry == "some other type but not unused" then
   Follow the back pointers and make every entry traversed
      "unused". The first entry is made "unused" last, to handle
      a crash during a recovery.
   Unused flag = false
end if
if Entry == "unused" then
   Skip the entry.
end if
if endptr > start then
   go to the previous entry in the log.
   goto Loop1
else
   if Unused_flag == True then
      remove all log entries.
      Exit
   else
      goto Loop
   end if
end if
```

Figure 3.6 The Recovery Process

3.4 Implementation

We have designed and implemented an Ext3-like file system below the Linux VFS interface, for hosting in NVRAM. As hardware with NVRAM was not available, we designated a portion of RAM as NVRAM and carried out our implementation. We used the Linux kernel version 4.13.0 and implemented the file system as a Linux kernel module. When the module is added, the DurableFS is registered using register_filesystem() call sothat the kernel is aware of for mount and other system calls. During mount, DurableFS allocate and maps the simulated NVRAM using the ioremap interface. The ioremap() returns a kernel virtual address corresponding to start of the requested physical address range. The memory region returned by the ioremap is partitioned into 4KB blocks and initialized different mandatory data structures like inode table, bitmaps etc. To mount the file system, we have used the generic mount implementations- *mount_nodev(),* which mount a file system that is not backed by a device. Call to the mount_nodev() also initialize the superblock. Once the FS is mounted, we can use different filesystem calls. We have implemented four DurableFS specific system calls- open(), read(), write(), unlink(). Along with these we also have implemented few inode operations- lookup(), rmdir(), and rename().

3.5 Experiments

Experiments were carried out in a 6 core Xeon system with 32 GB of memory. 4 GB was assumed to be NVRAM and set up accordingly in the kernel. We have created files of different sizes in two versions of our system- enabling and disabling cache flushes. In both cases, we noted the Timestamp Counter (TSC) [23] values and analyzed the results. Besides creating files of different sizes, to run more realistic workloads we compared our implementation with the NOVA file system by running FIO and Filebench benchmarks. To compare our system with NOVA, we installed NOVA in the same system.

3.5.1 Creating Files of Different Sizes

We first carried out experiments by writing to files of different sizes. For each size, the code was executed 100 times, and the average execution time of these 100 runs was used for further analysis. Time was measured by reading the Timestamp Counter (TSC), available in Intel processors, at the beginning and the end of each run. The difference gives us the number of clock cycles between the two readings. This implementation was carried out under kernel mode as a kernel module. In this experiment, we ran the implementation with flushing of caches as described above, without flushing the caches and bypassing the cache memory. This experiment was carried out to compare the performance among flushing, without flushing and bypassing the cache memory. The difference in time of execution gives us the overhead due to cache flushing and bypassing the cache. To bypass the cache, we need to reserve some memory from the kernel by using the *memmap* kernel parameter and to reallocate the reserved memory as un-cacheable. Bypassing cache memory removes the problem of consistency, as data is written directly into main memory.

Result:

The Table 3.3 shows the time of execution with different write sizes in different modes.

Size	No Flush	Flush	No-Cache
64	881.34	901.41	5,162.67
128	1,019.00	1,048.85	12,600.67
256	2,029.42	2,111.69	27,907.33
512	4,038.77	4,200.70	71,893.33
1024	8,044.50	8,437.72	1,21,309.33
2048	16,211.66	16,929.30	2,48,624.67
4096	30,739.10	31,989.05	4,90,624.67
8192	60,001.07	63,101.12	9,82,078.67
16384	1,22,927.29	1,29,100.52	19,65,632.67
32768	2,46,170.33	2,58,299.78	42,62,037.33
65536	4,58,889.91	4,82,001.32	77,22,460.67
131072	10,97,789.84	11,52,798.21	1,57,94,688.67
262144	21,56,887.00	22,88,980.91	2,94,79,959.33
524288	39,68,917.87	42,88,967.91	5,45,32,034.00
1048576	74,66,848.58	79,91,280.84	10,67,48,924.00

Table 3.3: TSC values for different write sizes, in # of clock cycles.

As the results show (Figure 3.7), bypassing the cache results in significant degradation of performance. This is probably because the system has to wait for completion of a write to memory which is in units of only 64 or 128 bits at a time. When a cache is there, data is written into cache, and flushing a cache takes place in parallel with subsequent writes to cache. Use of flush and fence instructions resulted in only about a 7% degradation of performance.

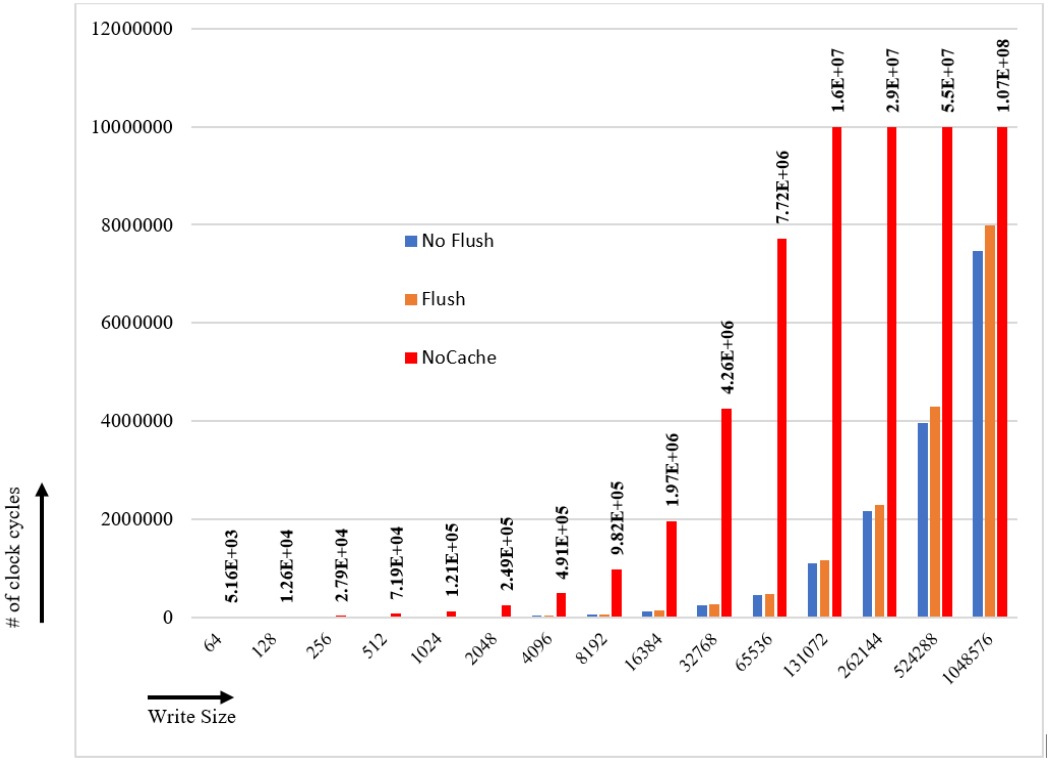


Figure 3.7: Write time vs size of file

3.5.2 Running the FIO and Filebench benchmarks

To compare our system with NOVA, we used Linux kernel version 4.13.0. We ran two benchmarks Fio [24] and Filebench [25] on NOVA and on our system. The benchmark workload characteristics are given in Table 3.4. For DurableFS, there were two versions: 1) the atomic and durable version, and 2) the second in which data writes were not flushed from cache. We ran the benchmarks FIO and Filebench on our system and compared the results with a version of our system without using any flush instructions for data writes. Thus, we measured the impact of having to cater to the NVRAM. Results show that in this case too, there is only a 7% degradation in overall bandwidth achieved when flush instructions are used. This shows that the scheme to ensure consistency of persistent data almost fully utilizes the high bandwidth that a memory bus persistent storage gives. Of course, the copy-on-write for updating a block partially and the use of a metadata log adds overheads in comparison to a RAM file system. But the durability that is achieved requires this overhead. We also ran the same benchmark in NOVA and compared the results with our system. Figure 3.8 gives the results of our experiment. Each experiment was run 10 times and the average result was taken. The results show that when we compare our durable implementation with NOVA, it is slower by 11.14%, 9.25%, and 2.26% in the Fio, Filebench (fileserver), and Filebench (webserver) benchmarks respectively. On the other hand, if we compare our implementation with data write durability removed (version 2), it is slower by 1.74%, 2.32% and -1.08% in the three benchmarks. What the results show is that our implementation is as efficient as

NOVA when we do not have data durability (NOVA too does not have data durability). When we add data durability, our implementation is slower by 11 to 9 percent in write intensive loads (See the R/W ratio in Table 3.4). We feel that this is an acceptable penalty to pay for introducing data durability in the form of transactions. In a read intensive load (Filebench (webserver)), the difference in performance is not significant (2.26%). So, our work shows that introducing transaction capabilities to an NVRAM file system is a practical proposition.

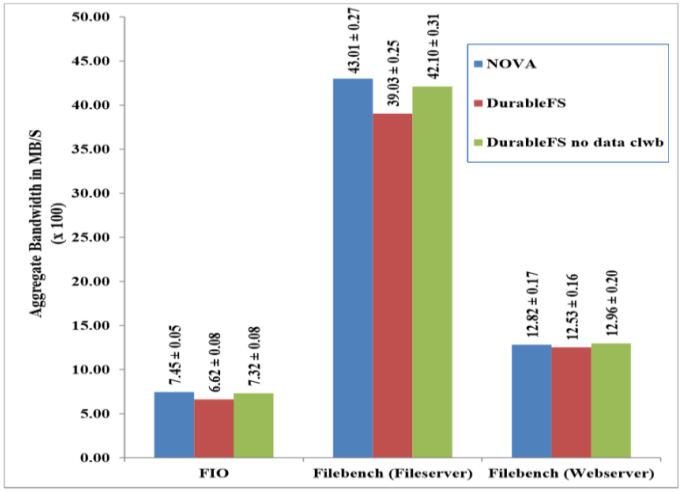


Figure 3.8: Benchmark Results with confidence interval.

Workload	File Size	I/O Size	Threads	R/W Ratio	No of files
Fio	256M	4K	10	1:1	10
Filebench (Fileserver)	128K	4K	10	1:2	1K
Filebench (Webserver)	64K	4K	10	10:1	1K

Table 3.4: Benchmark workload characteristics.

Chapter 4
Related Work in NVRAM File Systems

With the availability of NVRAM which provides fast storage and byte level randomly accessible interface, a number of experimental file systems on have been designed and implemented. In this chapter we have mentioned some such file systems.

4.1 BPFS

An early design of a file system for NVRAM was the BPFS system [6] developed at Microsoft in 2008. The design assumes that NVRAM will be on the system memory bus, but will be 2-5 times slower than DRAM. At the time when the system was being designed, there was no hardware support in the form of instructions such as *clflush*, *movnti*, *sfence*. So, they assumed that there would be an *movnti* like operation through the use of capacitances on DIMMs to complete writes on a power failure. They further assumed that there would be an instruction which they called an *epoch* barrier which combines the functionality of *clflush* and sfence. BPFS provides durability and consistency guarantees by using the above instructions. The system also allows small updates in place. Taking advantage of the fact that writes do not need to be at the block level (even though space is allocated on a per block basis), they have combined all pointer information in the file system into one tree. The top portion of this tree allows the accessing of a node, given an inode number. The nodes containing inode information (including the file size) are the root of the file represented by that inode. It could be a data file or a directory file. So, the inode table, the directory structure, and the data file indices of an ext3-like file system are combined into one tree. All operations then are on this tree. In place updates of up to 64 bits are allowed into a block. Appends into a file are allowed up to a block boundary and the change of the file size in the inode node ensures atomicity of the append. General changes in files are handled by using Copy-on-write with shadow pages. A shadow sub-tree is created in NVRAM based on the operation and the top-level pointer in the original tree is atomically written into (using *movnti*) to include the shadow subtree in the main tree. Some data structures are stored in volatile memory due to performance issues and to handle the atomicity of a copy-on-write. They are the global free block list, the list of freed and allocated blocks from an in-flight copy-on-write operation, and a cache of directory entries (this is for performance). Each directory entry is stored in a list and in a hash table to provide quick, ordered directory lookups. This design uses no logs to ensure consistency of data. Most metadata operations involve changing one node (most of the time the node corresponding to an inode of a file) and this is handled by copy-on-write, as described above. If multiple nodes have to be changed (moving a file from one directory to another), then a large portion of the tree may need to be copied to form a

shadow sub-tree and this is the price they pay for not having a log. They argue that such operations are rare and so the overall performance will be good.

4.2 PMFS

PMFS [9] is another early file system for NVRAM developed at Intel on 2013. It uses the byte addressability property to reduce the overhead of block-oriented devices. In PMFS, as shown in Figure 4.1, NVRAM can be accessed directly from user space using memory mapping. At the time of their design, Intel CPUs had implementations of *clflush* and *sfence* instructions and also atomic 64-bit writes using the Intel *movnti* instruction. But there was no support for the "ADR" facility of flushing memory buffers on a failure. So, they assumed in their design that a *pm_barrier()* instruction is present which essentially performs the "ADR" functionality. Like BPFS they use a single tree structure for all metadata and data. A B-tree structure is used. For metadata updates, they use an undo log where changes are first written and then written into the B-tree. It may be noted that we have used a redo log but with a RAM copy of the metadata used to access changes to metadata that have not yet been committed (a problem with redo is that every access has to check the log to see if there has been a change which is not in NVRAM as yet). Undo logs require cache flushes after every log entry. But PMFS does keep a copy of some of the metadata in RAM, namely the free block list.

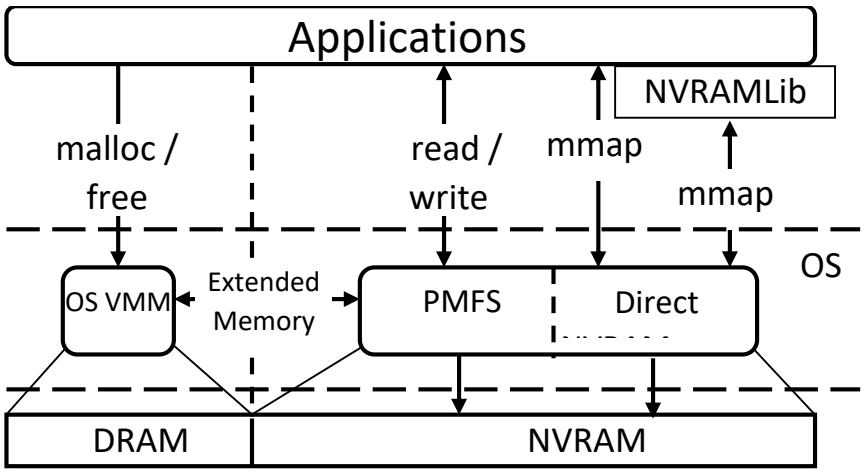


Figure 4.1: PMFS System Architecture

Applications using POSIX file system calls like read and write go through the PMFS kernel module to copy data from NVRAM to user space on a read and vice versa for a write. Using the mmap feature, a file or a portion of a file can be directly accessed from a user application, as can be done in a standard Linux system. The difference here is that data is being written directly to non-volatile store rather than to kernel memory in a disk-based system. Such memory mapped applications which work on disk storage cannot however be used without changes as special instructions have to be used to ensure durability. They therefore suggest the use of library functions (depicted by NVRAMLib in the figure)

to access the memory mapped area. These library functions will provide functions such as read and write and will implement these operations using special instructions for durability

4.3 NOVA

NOVA [21] (NOVAFORTIS [26] is the system with reliability features included) is a log-structured file system implemented in NVRAM. Log-structured file systems accumulate multiple random write requests into a comparatively larger and sequential write so that the underlying storage device (hard disks, SSDs, etc.) can process the request efficiently. To store data contiguously, a conventional log-structured file system needs to maintain contiguous free space using expensive garbage collection operations. Sine their system is on NVRAM with byte accessibility, contiguous space is not required. Their design aim is to allow existing applications using POSIX system calls to run on their file system without change. They therefore assumed Unix semantics for file system access, adding the journaling feature of ext4 to prevent inconsistent metadata on a crash. They do not handle the durability of writes to data as Unix semantics does not either. This makes their system faster than other designs which provide durability for data writes too. Each file in their system is stored as log structured file. The inode of a file has a head and a tail pointer to a log which contains the file. To enable quick access to the contents of such files, for each file and each directory, a DRAM-based radix tree structure is maintained. This tree is updated whenever any change is made to the corresponding file. There is also a free block list kept in DRAM. These DRAM structures are not written to NVRAM except when the file system is unmounted. Their design, therefore, reduces the number of writes of meta-data to NVRAM during run-time (keeping only the information in logs) at the cost of more complex recovery in case of a failure (essentially an fsck has to be done as free list information is not there in non-volatile memory) and also more time to find attributes of files since they are stored in a log structure. They use copy-on-write to change data blocks, but they do not flush caches after writing to data blocks. Not doing this does not guarantee the durability of data. Meta-data consistency is maintained by using *clwb* and *movnti* instructions. To cater to multiple CPUs, they have separate structures on a per-CPU basis.

4.4 SoupFS

SoupFS [27] is another recently developed file system on NVRAM. They seek to reduce the critical path time by doing cache flushes asynchronously. They introduce the notion of a consistent view and the latest view. The latest view is used in operations and it is made the consistent view by doing cache flushes in the background. Their design sacrifices the durability of operations and is suitable for implementing current file system semantics. They also combine the file name and inode store to improve efficiency. Both of these features go against our design goals of a distributed system for data-centric applications with guaranteed durable stores.

4.5 DAX

DAX [28] support for File Systems in Linux is to enable implementation of file systems on NVRAM. It is a simple and easy way of using NVRAM as permanent store for existing applications using POSIX system calls. DAX removes the Page Cache from such file systems, preventing a copy of data from the file system to DRAM, and allowing direct access of the data from NVRAM. The support enables existing file systems such as EXT4 to be implemented on NVRAM. However, DAX does not provide any support to ensure consistency and durability of writes to NVRAM. If UNIX like semantics is required, then this solution is the easiest and probably the fastest way to use NVRAM as the secondary store.

4.6 ORION

ORION [29] is a distributed file system for NVRAM-based storage. It exposes a normal POSIX interface. An Orion cluster consists of a metadata server, several data stores organized in replication groups, and clients all connected via an RDMA network. To encourage local access, Orion migrates durable data to the client whenever possible. Orion caches file system data structures on clients. A client can apply file operations locally and only send the changes to the metadata server over the network. Data for a file can reside at a single DS or span multiple DSs. A client can access a remote DS using one-sided RDMA and its local NVRAMMs using load and store instructions.

4.7 Aerie

Aerie [7] is a file-system architecture that allows programs to access NVRAM from user mode without kernel interaction. It does not use POSIX calls, having a library, FSLib which provides access to the file system. Considering their base implementation, metadata is handled by the kernel in a module called a “Trusted FS Service” (TFS). Data is accessed after a file is mapped to the user address space. To map a file, a call to TFS has to be made where permissions will be checked and then the file will be mapped. Operations on files such as changing permissions, deleting, renaming, etc. will required TFS intervention. In the bare system, naming is at a single level with a unique key for each file. Metadata is made crash resistant using the usual methods mentioned above. Intel instructions *clflush, sfence, and movnti* are used to provide durability. A redo log is used to record changes to metadata before the actual changes are made. There is no provision for durability of writes to data as data is directly written into using load / store instructions. A central locking service is included to allow different levels of concurrency control that applications may require. The block diagram of Aerie architecture is shown in Figure 4.2.

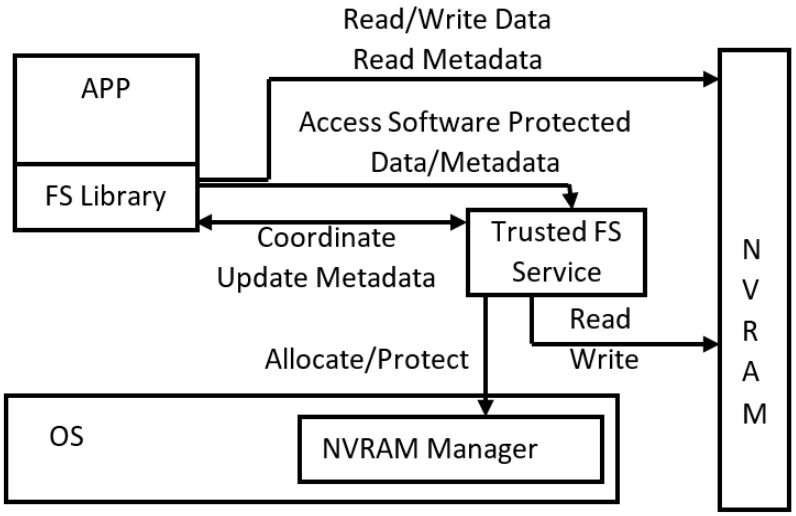


Figure 4.2: Aerie architecture

4.8 SoftWrAP

SoftWrAP [17] is a system to provide transaction memory support in an NVM based system. It does not implement any file system, allowing programmers to make changes to variables atomically by specifying the start and end of transactions. It uses caching of variables in RAM with a redo log whose tail is also stored in RAM. The technique is similar to that used in RDMS systems based on the ARIES recovery system [16].

4.9 Comparison

The Table 4.1 below lists the main features of the systems discussed in this chapter. Durability is implemented using Intel instructions. As Intel introduces more instructions or features, the implementation becomes more efficient in the designs. The latest Intel proposal of including an "e-ADR" feature (see chapter 2) may require no special instructions to be executed to ensure durability of writes into NVRAM. Most of the designs use a log of some kind to provide consistency of metadata. Durability of writes to data is provided only in Nova (and Orion) and our system. Mapping of files to use space is supported by BPFS and Aerie. Of course, memory mapping of files to user space without worrying about durability and about sharing and about relocalability, can be provided by all designs.

	Transaction support	**Implementation of Durability**	**Metadata treatment**	**Data Handling**	**File memory map to user space**
BPFS	No	Spl instruction: epoch barrier, movnti	No logs. Everything is Copy-on-write, and limited update in place and append.	Copy-on-write, no durability	No, but proposed
PMFS	No	cmpxchg16b instruction (with LOCK prefix), movnti, clflush, Spl instruction: pm wbarrier	Undo log; in place small updates; global free list in RAM	Mapped to user space; ext4 "ordered data" mode only (no durability guarantee); POSIX version also there.	Yes
NOVA	No	Clwb, movnti,	Per inode log structured system ; global redo log for directory ops; RAM version for fast access; slower attribute access	Durable Copy-on-write	No; special atomic-mmap supported
SoupFS	No	Intel RTM to enforce cache line atomicity; clflush / sfence, movnti	Consistent & latest views in NVRAM and RAM resp. background daemon to map latest to consistent.	In-place updates	No
Aerie	Locking service	clflush/sfence, movnti	Redo logs used for metadata	Direct write from user space; no durability	
SoftWrAP	Yes	Cflush /sfence / Pcommit	Redo logs used	Cached in RAM	Not applicable
DurableFS	Yes – opening and closing of a file.	clflush/sfence, movnti	Write-ahead redo log in NVRAM, active metadata in RAM	Durable Copy-on-write on NVRAM directly	No

Table 4.1: A brief comparison of different file systems

Chapter 5
KV Store Design

5.1 Overview of the system

DKVS is a distributed Key-Value store designed by keeping in mind the memory characteristics of NVM and using RDMA to access remote memory to improve performance.

The following is an overview of the 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 a home node, and the current version of the KV is stored in the home node. The home node of a KV may change, but it is going to be infrequent – due to failure of the home 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 home 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 home node.
- The Key of a KV is composed of two parts: the first part is the home node id, and the second part is a number uniquely identifying the KV in the home 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.
- Given a key at a particular node, its first part may identify the home node to be other than the current node. To access the remote KV, the node has to send a request to the home node seeking the inode of the KV. With the inode, the node checks permissions for the current read access and if permission can be granted, the value of the KV is obtained directly, bypassing the home node CPU, through RDMA.
- The home node keeps track of nodes that have a copy of the inode through information stored in the inode itself (this information is not shared with remote nodes). If there is a write to the KV, then the home node sends messages to all nodes with a copy of the inode of the KV, invalidating the inode (and value contents too by implication).

- 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 of remote 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 home node through the LAN. The home node invalidates all remotely stored inodes of the KV and then updates the KV locally.
- Fault-tolerance: Each KV may be replicated as per requirement. 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 home node, and the second copy is in a slave 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 home 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.
- Concurrency Control: A limited two-phase locking facility is implemented to provide transaction locking facilities that span several KVs. 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. A transaction starts at a node called a transaction owner (TO), with a read phase, where all the required reads are done. 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 pools all nodes for the current state of locks and detects deadlocks using standard algorithms. Abort messages are sent to appropriate 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.

5.2 Design Decisions

Its key design decisions were made with the characteristics of NVRAM in mind. Based on our observation, we make six major design decisions in our system as the following.

a) Use of RDMA network

Remote Direct Memory Access (RDMA) is a technology that allows computers in a network to exchange data in main memory without involving the processor or operating system of the computer, which is not possible in a disk-based system. In DKVS we are using NVRAM on the memory bus for storing data. Use of RDMA to transfer large data dramatically improves the overall performance of our system. In our system we have used RDMA for transferring the value of a key. However, all metadata checking and metadata update operation are not performed through RDMA since we need to perform locking and all in such operation which need the operating system. Thus our use of 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.

RDMA has been used in many KV-store proposals, Chapter 8 gives summaries of some of the systems relevant to our design. The main issue in using RDMA to access remote data is ensuring that up-to-date data is retrieved. This is difficult since the remote request bypasses the host CPU. Further, locating the remote data will require fetching index metadata too, resulting in read amplification. FaRM [30] uses one-sided RDMA (like we do) to provide remote access to a distributed key-value store. It has to locate the key-value, and so some metadata reads are required. Locks are stored with the value to allow lock-free read access. By using optimistic concurrency control with strict serializability, overhead of locking and synchronization is postponed to transaction commit time. FaSST [31] implements RPCs over RDMA to handle these problems. But this requires a RDMA process to be running in the remote system's CPU and NIC. This increases the latency.

b) Why a home node for each KV and why not let the KV owner be dynamic.

The main assumption is that reads will dominate writes. Having a home node makes lock management easy as it is centralised. Writes are not done through RDMA. A write request along with the key and value are sent to the home node over the LAN. However, since a transaction may involve multiple KVs with multiple locks, a deadlock detection process is required which periodically polls all nodes for current locks. The KV owner is changed only if there is a failure of the master node. In this case, if a cached entry refers to a failed node of the system, RDMA access will fail and the node will broadcast a request on the LAN to find the new owner.

c) Why slaves do not service read requests

Since it is assumed that the cluster of nodes is interconnected by broadcast networks, both for RDMA and for regular messages, there will be no advantage in terms of network cost to allow accesses from slaves. With RDMA access for reads, the home node of an item is unlikely to become a bottleneck. So, by not allowing reading from slave nodes the update process becomes simple and there is no need for three phase commit or such other techniques while updating slave nodes.

d) Use of 2PL for Concurrency Control

With the decision to have home nodes for each KV, 2PL is the automatic choice for providing concurrency control. The alternatives of optimistic, timestamp or multi-version concurrency control will incur more overhead in a clustered system compared to 2PL.

e) Why values are stored contiguously

The Value of a KV has to be kept contiguously to allow standard RDMA to access the value. Doing so also allows local access of a value through memory mapping in a user process.

f) What if we run out of space? Are we considering archiving on disks?

We are not considering a scenario where the available NVRAM is not sufficient to hold all the active KVs. Clearly, a production system will have provisions for archiving unused KVs to disk store, as well as taking full backups to disk.

5.3 Design Goals and Contributions

a) Design Goals

DKVS is a distributed Key-Value store built for the performance characteristics of NVRAM and RDMA. NVRAMs are byte addressable and operates on very low latency in comparison to SSDs. On the other hand, enabling RDMA to access storage can dramatically improve the overall performance of a system. These properties of NVRAM and RDMA fundamentally alter the relationship among memory, storage, and network. It motivates us to design a storage system where the primary storage is

on the memory bus so that we achieve a very high throughput and remote access to storage does not create any bottleneck in the system. DKVS achieves the following design goals:

I. Transparent access to data across a cluster: Standard feature of providing transparent access in a distributed system; allows scaling of storage, computing power, as per need; improves reliability of the system
   - Unique Key for every item (The key is composed of the node id and a locally unique identifier for each node)
   - Mapping of node ids to physical node ids is cached to remove lookup time. A node id may be different from the physical node id due to failure of nodes.
   - Standard, well established, distributed system algorithms are used wherever feasible.

II. Minimal structure of data to allow different higher levels of structuring: RDBMS, NOSQL systems, File systems, all can be implemented efficiently on top.
   - Key Value Store implemented, and not a file system.

III. Fast access of remote data: As local access of data is at near RAM speed; it is important that remote access does not incur too much overhead.
   - Use of RDMA to read remote data without remote CPU intervention.

IV. Uninterrupted service in the face of single node failures: standard requirement of reliability
   - There is provision to replicate KVs at a node in another node.

V. Support for concurrent access: Providing concurrent access is expensive; many applications with large, unified data, need concurrent access; providing this facility at a low level gives an efficient implementation, and makes higher levels of software easier to implement.
   - Transactions involving access to multiple items supported.

a) Contributions of this Design

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 features. The system uses RDMA as a performance enhancement feature and the system can be implemented efficiently without RDMA. No complex structures on top of RDMA is used as has been done by many designs. 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. For example, all rows in a RDMS table may be kept in one node by a RDMS system, and so the placement of a new KV item (representing a table row) will be based on where the table is located. 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. The transaction facility is optional.

5.4 Design

In our design each key-value pair is identified by a unique key. Each node is equipped with its local key-value store. The consistency of each local key-value system is maintained locally by each node. On each node there will be a kernel process which will be responsible to manage the NVM and one interface process which will accept requests from client processes and interact with the kernel process to provide the requested service. A node will be called a *serving node* if a client requests a key to that node. A node will be called *owner* with reference to a key-value, if the pair is physically present in that node.

In our approach there is no directory structure. Each KV can be accessed in a single level. The key may be of random size up to a maximum, which can be assigned during system setup. A key contains the master node id and the actual key. For each key there is an inode having some metadata. As shown in Figure 5.1, 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 *v_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 cache copy.

```
struct inode
{
      32 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;
}
```

Figure 5.1: The inode structure of a KV pair.

5.4.1 Use of AVL Tree

To speed up the search of a KV pair, some existing systems [32] [33] [34] [35] employs a B+ tree index and some [21] [29] use a radix tree. The most common structure is an LSM (log structured merge) tree [36]. B+ tree is good for storing data for efficient retrieval in a block-oriented storage device like HDD because B+ trees have very high fanout which reduces the number of I/O operations required to find an element in the tree. B+ trees group a larger number of keys into each node to minimize the number of seeks required by a read or write operation but increases per-node search time. Since disk read is expensive it is better to reduce the number of disk reads, and that is accomplished by increasing the number of positive hits on the cache (reads a greater number of keys in per disk read operation).

Similarly, there are disadvantages of radix tree like – it is not height balanced; hence it may require more lookup time. Moreover, in order to access a key, a radix tree is traversed depth-first, following the links between nodes, which represent each character in the key. Breaking the key into some smaller units and using them in the lookup purpose is not beneficial for us. Similarly, LSM trees are suitable when the KV store, including the index is stored in secondary storage. The top level of an LSM tree resides in RAM to improve access. We do not need this extra complexity when everything is in main memory.

Since our system is for byte addressable NVM devices and in such devices a memory read is not so expensive, we have used AVL trees for indexing which are intended for in-memory use, where random access is relatively cheap. By using AVL trees we can make a compact and faster index structure than a B+ tree or a radix tree or an LSM tree. AVL tree gives us O(log(n)) complexity for search and insert. Therefore, we have used three AVL trees- K_AVL for the inode table, CACHE AVL for cache entries and F_AVL for free space management.

Some in-memory DBMS designs [37] 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 disadvantage of searching within a B+ tree node will not be incurred.

5.4.2 Inode Table

Each node maintains an independent table of such 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 first byte of the key. A fixed size NVM 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 size 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 node is shown in Figure 5.2.

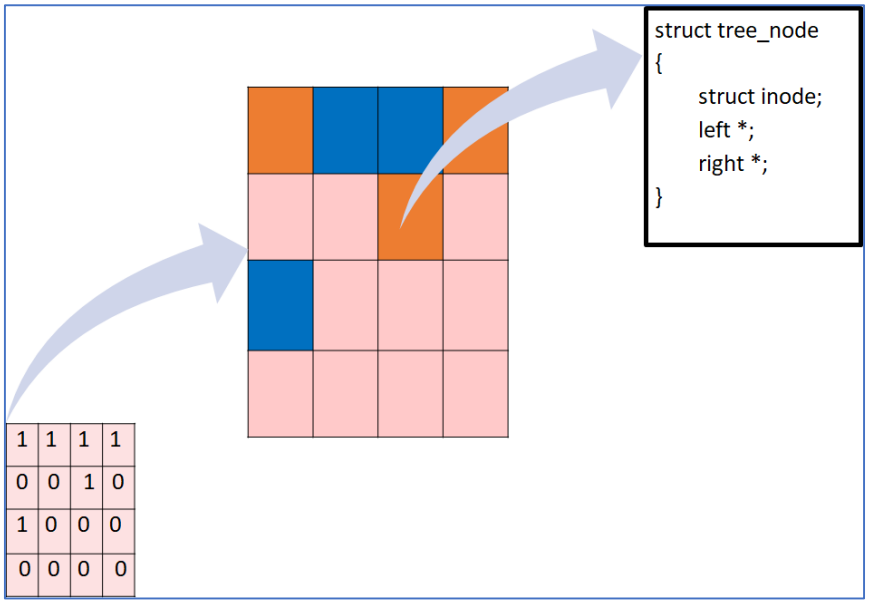


Figure 5.2: Bitmap and Tree node blocks

5.4.3 Caching and Replication for High Availability

For fast retrieval of data in our system Keys are cached by a node once it reads the values. Cache entries are stored in RAM, using an AVL tree (CACHE AVL). Each node of CACHE AVL contains the same metadata as K_AVL node. This will reduce the number of lookup operations for a key. Each node of this tree will remain valid until there is a modification of the KV pair. Once an inode of a KV pair is cached, any further read of that KV pair is served through RDMA using the cached metadata. But, if there is any modification to the KV pair, the cached copy will be invalidated by broadcast, and anyway a lookup has to be called. In this Key Value system, each KV pair can be replicated for failure recovery. The number of replicas for each pair is stored in the 2-byte replica count field of the inode. Along with replica count an array of node id is also maintained in inode to keep addresses of replicas. In the replica list, the first entry is the master and other are slaves. Read and write to a KV is allowed only in the master copy and slave copies are used for recovery purpose only. This structure allows random number of replicas for each KV pair. If we need to add multiple copy of a KV pair, first one copy will be created, and that node will send write request to other nodes according to the number of replicas requested. If there is a failure of master node, a required key can be found from its replica. Replica node is discovered by broadcasting the key or from a cached inode if cached earlier. In this situation if a cache entry refers to a failed node of the system; after RDMA times out, the serving node will request the key from the next replica and update the cache entry accordingly. Once the master is active, it synchronizes itself with the latest version of the KV pair from the replication list. We have used eventual consistency model in our system among replicas. If there is a write request (and the key already exists, then) the master node will invalidate cached copies by sending a broadcast message and acquire write locks for the KV pair in all replicas. Once the write is over the current replica will be unlocked and multicast a sync

request for that key to all nodes who have slave copies of the key. Once such a message is received by a node, the node will synchronize itself by reading data from an unlocked pair. Because of this approach, we do not need distributed lock managers like OCFS [38].

5.4.4 Free memory management

Apart from the K_AVL tree, in each node, there is an AVL tree F_AVL, to manage free space. Units of free space are also considered as a pair of key-value, where the key is not relevant. A node of F_AVL, which refers to a block of free NVM area is not discoverable in K_AVL. Such nodes are inserted into the F_AVL tree. Initially the entire free space is represented as a free key-value pair and its inode is the root of the F_AVL tree. The arrangement of nodes in K_AVL is based on the first byte of the key of a KV pair and arrangement of nodes in F_AVL is based on the size that it refers. The free NVM keeps splitting based on the required size when there is a new entry. To add a new pair, 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 exactly 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 NVM is taken out from the F_AVL tree to form a new node which is inserted into the K_AVL tree. The remaining part of 'N' is remains in the F_AVL tree. Accordingly, on a delete the respective node from the K_AVL tree moves to the F_AVL tree. Due to delete operations number of small unused NVM area can arise. Two or more contiguous unused blocks (hole) are merged to make a large unused block by running a GC (Garbage Collection) routine.

In our design, value is stored in contiguous memory location to enable RDMA. We do not allow in place update of any metadata in our proposed design. In-place update may result in inconsistency of the system due to failure during the update process. Therefore, we provide Write Ahead Logging (WAL) of metadata. Such log records are written in a fixed size journal block of the NVM. In Figure 5.3 different possible states of the NVM are shown.

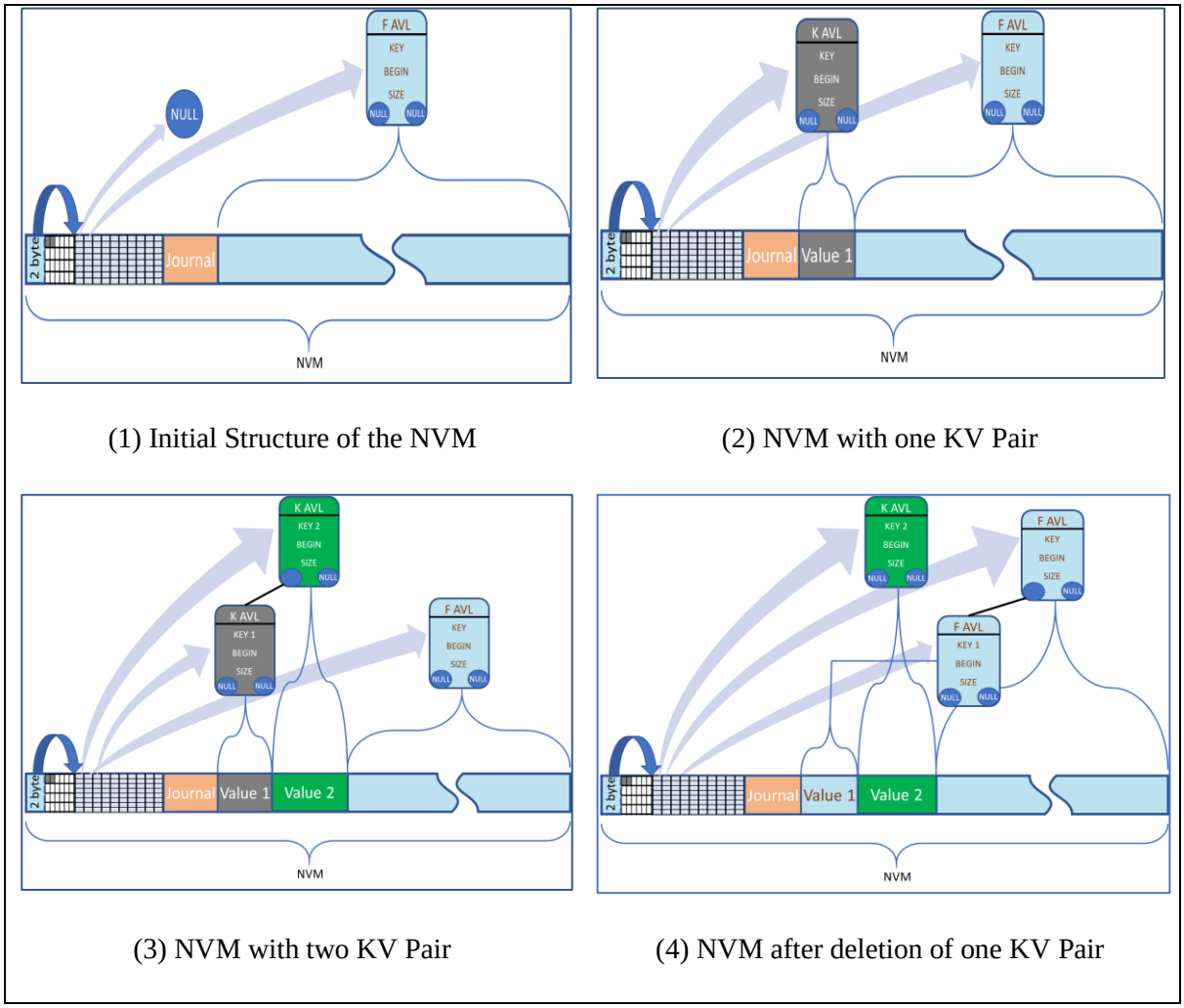
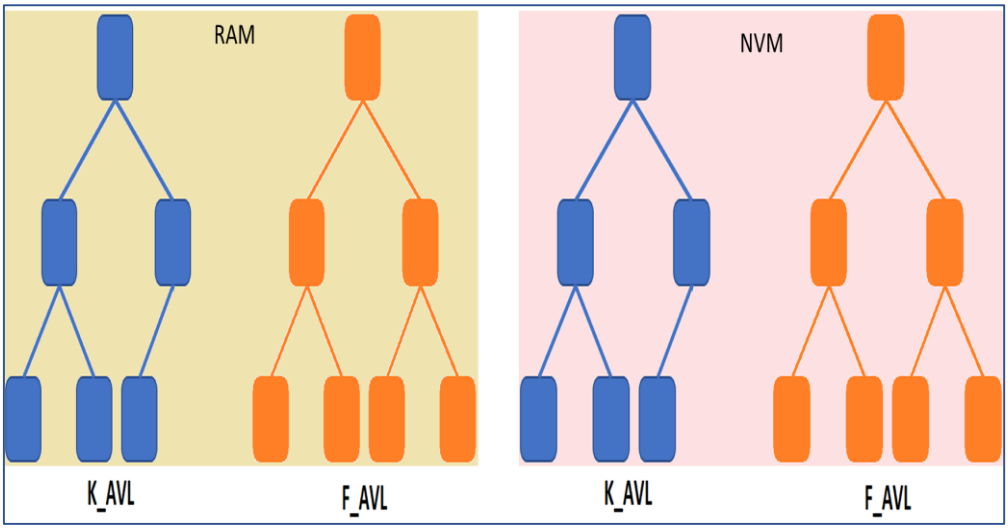


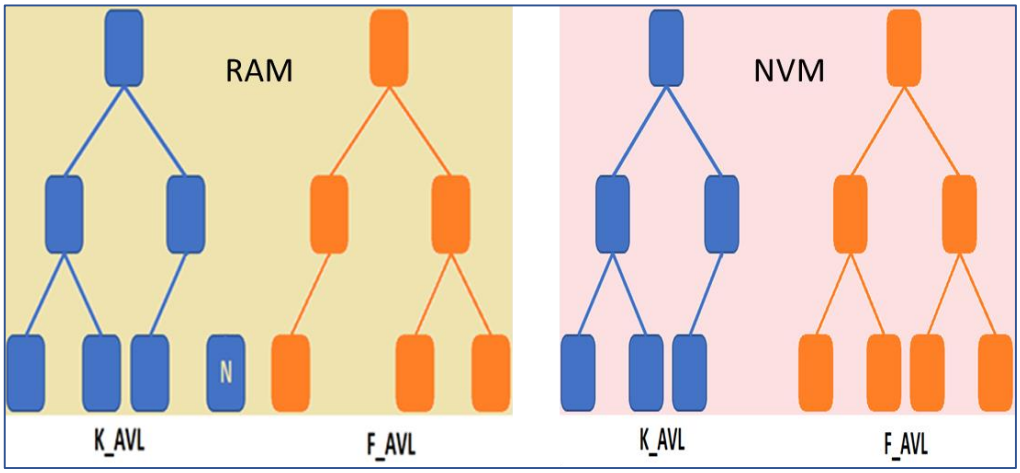
Figure 5.3: Structure of the NVM

5.4.5 Performance

Although NVRAM is attached to the memory bus and is byte addressable, it is assumed to be slower than the currently available DRAM. For metadata information, accessing NVRAM every time will decrease the overall performance of the system. Therefore, in our system there are RAM versions of both K_AVL and F_AVL trees and other metadata. The RAM version of metadata will remain up to date at any time. Any modification to such metadata will take place in the RAM version first. The NVM version of metadata will be updated through Write Ahead Logging (WAL). In Figure 5.4 two versions of F_AVL and K_AVL are shown. In Figure 5.4 (2) it is shown that the new entry ‘N’ is visible only in the RAM version. The NVM version will be updated using WAL.



(1) RAM and NVM versions of K_AVL and F_AVL



(2) Insertion of new node. NVM version yet to update

Figure 5.4: Two versions of F_AVL and K_AVL

5.4.6 Supported Operations

Our Key value store primarily support GET, PUT and DEL operations. Besides these it also supports UPDATE in a conditional way. As the name suggest GET operation is to read a value by providing the key, PUT operation is to write a new KV pair into the system and the DEL operation is to delete an existing KV pair. Our design also supports an update operation which deletes the value and inserts a new value, keeping the same key.

5.4.7 Node Selection, GET and PUT operations.

To retrieve a value, a client has to provide the key with a buffer address to store the value. Keys are composed of the key and the node id. The prototype of the GET operation is GET(Key, &buf). In GET operation if the node id of a Key is the same as the serving node id, a lookup call retrieves the inode and read value. But, if the node id is not same as the serving node id, the key will be searched in the CACHE AVL. The CACHE AVL is a cache copy of remote KVs stored in the RAM of a node for future use. If the key does not exist in the CACHE AVL, the key will be requested by the serving node from the owner. The owner node will verify permission etc. and send back the corresponding inode of the KV pair. After receiving the inode of the requested key, the serving node can read the value through RDMA. If the key is invalid or there is no KV with that key, the serving node will receive a negative acknowledgement from the requested node.

To perform the PUT operation, the uniqueness of the requested key must be maintained. Therefore, a lookup call is essential in a PUT operation also. The prototype of the PUT operation is PUT(Key, value). If the key does not exist, a new inode will be created by considering the key value size and other parameters of the KV pair and the inode of the KV will be returned. In case of the PUT operation, transaction support is required to maintain the consistency between K_AVL, F_AVL. A contiguous memory segment of the right size also needs to be available for the value.

The UPDATE operation is conditional. That is, if the new size of the value fits in the previous location, then optional in place update will be done. Any remaining memory area will be handed over to the F_AVL tree. In this case any measures for a consistent update will be handled by the upper layer application if necessary. On the other hand, if the new size is greater than the existing value size, a new KV pair with the same key will be inserted after deleting the existing KV pair using copy-on-write (CoW).

5.4.8 Metadata redo log

Due to durability and consistency issues, metadata and data cannot be written in place. To improve performance, we have kept DRAM versions of metadata, which is 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. The different entries of a log are shown in Table 5.1.

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

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

Table 5.1: Log entries

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. Together these operations are referred to as *write_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 [22]. Here also we have used the same method like DurableFS of re-reading a written data after cache flush to ensure that the data has actually reached a durable point.

Chapter 6

KV Store Implementation

6.1 Introduction

We have designed and implemented a key-value storage for NVM. Here also we have used the simulated NVRAM similar to DurableFS. The system has been designed and implemented as a loadable Linux kernel module. Each kernel module in every node has two kernel processes. One is responsible to manage the NVRAM and the local KV store, while the second process is an interface process which 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 [39]. 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 (either another node in the cluster, or from an application server) 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 [40]. 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 6.1. Here the BS field is of 4-bytes and is used to store the size of the tree node bitmap. The size and value of this field restrict the maximum number of tree nodes and hence restrict 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, 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.

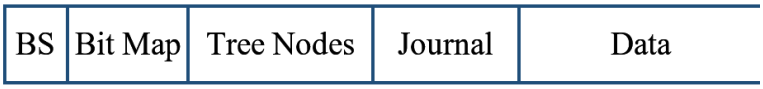


Figure 6.1: The layout of the KV Store.

6.2 The PUT Operation

To perform a PUT operation a client has to provide two parameters. The first parameter is the unique key and the second parameter is the value for the key. This operation can be used to store a single KV pair. For storing multiple KV pair in one node or in multiple node 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 NVM we use different cache flush/ fence instructions with re-reading the last written cache line. Figure 6.2 shows the algorithm of the PUT operation.

```
Put(key, value)

 1.  Write_Log "Beg_metadata key".
 2.  Obtain free space F from RAM version of F_AVL of
     sizeof(value)& 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 & insert
     KA into RAM K_AVL for new KV.
 4.  Copy value to F (write is to NVRAM); sfence; clflushopt;
 5.  Write_Log "remove F from F_AVL" (insert old F to F_AVL).
 6.  Write_Log "Insert KA into K_AVL";
 7.  Write_Log "Commit metadata key".
 8.  Reread end_log to confirm durability.
 9.  Update K_AVL in RAM; Delete old F if it is an update
10.  Update K_AVL and F_AVL in NVRAM.
11.  Sfence
12.  Write_Log "End metadata key".
```

Figure 6.2: PUT for one node

6.3 Transactions over multiple nodes

Our system supports two modes of access for every KV: non-transaction mode, and transaction mode. For transaction mode two phase locking protocol is used. Here every node will handle locks for the KV's it owns. In our system multiple transactions can be executed in parallel. In such an environment there is a possibility of deadlocks. Deadlock can be dealt with prevention mechanism, avoidance mechanism and detection mechanism. Prevention and avoidance mechanism are useful in a relational database where shared data are being used frequently and there are frequent updates. But our proposed system is a KV store where we assume that conflicting updates will be relatively few. Therefore, we have adopted a deadlock detection mechanism. There is a central lock server (which maintains a wait-for graph) which receives information in the form of commands on all locks granted and all locks released from the nodes. These commands will contain the node id and transaction number of the transaction which has locked / unlocked the KV. Periodically, the lock server checks for cycles in the wait-for graph. If a cycle is found, a victim transaction is chosen, and the server informs the node where the transaction is located. The concerned node aborts the transaction by releasing all locks the transaction holds, and it will restart the transaction.

In a transaction reads and writes are likely to span multiple nodes and so an algorithm different from the GET and PUT algorithms are required. A transaction will operate on a number of KVs which are owned by participating nodes. The node which initiates the transaction is the transaction owner and is also referred to as the serving node for that transaction. The serving node will maintain a log of actions on the KVs of the transaction. The serving node will form multiple independent sub-requests for each participating node. In our transaction model, read locks are first obtained on the read set of a transaction. Then the KVs are read and processed. Based on the processing, the write set of the transaction along with the new values of write set KVs, are determined. Request for write locks along with the new values are sent to all participating nodes. Each sub-request will contain a list of KVs owned by that node and whether a read or a write lock is needed on each KV. On a request for a read lock, the KV is locked if it is unlocked, and a lock grant along with the inode of the KV is returned to the transaction owner (TO). The TO will check that the transaction has permission to read the KV, and if so, the value will be read through RDMA. For write locks, each participating node acquires space for the new value to be placed in NVRAM (through the F_AVL tree). The new value is placed in NVRAM and a *lock_ok* is returned to the TO. The owner, on granting a write lock, informs all nodes holding a copy of the KV to invalidate it. As shown in Figure 6.3 in the first phase, read requests will be sent. Then write requests will be sent. This algorithm runs at the TO. Figure 6.5 shows the read algorithm at each participating node, while Figure 6.4 shows the write algorithm at each participating node. For writes, first a "*beg_transaction"* is written. 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. A two phase commit process is used to

ensure that all participating nodes complete their writes. On receiving *lock_ok* from all nodes, a commit record is written into the TO log. The transaction is now committed and during recovery operations may have to be redone to complete the transaction. "*allow_write*" is now sent to all nodes. Nodes respond with "*ready_to_commit*". Once all nodes respond, the TO send the final message "*commit_ok*" to all nodes. Then an "*end_transaction*" is written in the transaction log in the TO node. During recovery, if a TO log has "*end_transaction*" it means that the transaction was completed at all nodes and nothing needs to be redone.

Transaction (Ti)

```
{
    Determine the M KVs that need to be read by this transaction.
    Partition the M KVs into N sets, one set of Keys, for each of N
    nodes. Let the set for the ith node be denoted by RSi

    for(k=1; k<=N; k++)
    {
        send read(Ti, RSk) to node k;// RSk is the set of Keys to
        read
    }
    while(!timeout)
    {
        j=receive(lock_ok, inodes[]); // each of k nodes will send
        lock_ok and inodes of the locked KVs
        send(lock_ok, Ti, RSk ) to deadlock detector;
        Read_lock_i[j]=true;
        if(count(Read_lock_i))==N)
        {
            Break;
        }
    }
    if(timeout)  //some lock_oks not received
    {
        for (k=1; k++; k<=N)
        send abort(Ti) to node k;
        trigger_deadlock_detect();
```

```
}
for (j=1; j++; j<=N)
   foreach (K in RSj)
   {
      Determine read permissions for K from received inode;
      VK = Rdma_read(K); // read the values
   }
```

Process the values read and Determine the W KVs that need to be written by this transaction. Partition the M KVs into N sets, one set of Keys, for each of N nodes. Let the set for the ith node be denoted by WSi. WSi elements are two tuples, a key and a value to be written against that key.

```
insertlog(beg_trans, i); //a transaction log is maintained in
each node.
for(k=1; k<=N; k++)
{
   send write(Ti,WSk) to node k;// WSk is the set of <Key,Value>
   to write
}
while(!timeout)
{
   j=receive(lock_ok);
   send(lock_ok, Ti, WSk) to deadlock detector;
   commit_node_i[j]=true;
   if(count(commit_node_i))==N)
   {
```

```
        send(allow_write) to all N nodes;
        insertlog(commit, i);
        break;
    }
}
if(timeout)
{
    for (k=1; k++; k<=N)
        send abort(Ti) to node k;
    trigger_deadlock_detect();
}
While(!timeout)
{
    j=receive(ready_to_commit);//commit written
    Commit_node_i[j]=true;
    if(count(commit_node_i))==N)
    {
        Send commit_ok to all N nodes;
        insertlog(end_trans, i);
        Break;
    }
}
If(timeout)
{
    for (k=1; k++; k<=N)
        send abort(Ti) to node k;
```

```
        }
    }
```

Figure 6.3: Transaction Execution in the Transaction Owner

The Write algorithm executed at a participating node is shown in Figure 6.4. Unlike the Put algorithm, instead of writing *commit_metadata*, a lock_ok message will be sent to the transaction owner first (line 12). 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 for each key in the log. 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_commit_metadata* S_k record for each key and finally writes an *end_transaction* record and the process completes (line 31-39). Finally, all locks are released. 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 have to 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.

```
Write (WS_k)
 1.  Write_Log "Beg_Transaction WS_k".
 2.  for (k= 1; k<= cardinality(WS_k); k++)
     {
 3.      Obtain free space in NVRAM F_k from RAM version of F_AVL of
         sizeof(V_k)& update RAM F_AVL;
 4.      KA_k=lookup(K_k);
 5.      if(KA_k)
         {
 6.          WriteLock(K_k) & invalidate other copies of K_k (may wait
             if KV is already locked);
 7.          set pointer in KA_k to F_k in RAM K_AVL;
         }
 8.      Else
         {
 9.          Obtain free node KA_k & set pointer in KA_k to F_k & insert
             KA_k into RAM K_AVL;
10.          WriteLock(K_k);
         }
11.      Copy V_k to F_k (write is to NVRAM);
     }// if there is failure before sending lock_ok, TO will
     timeout
12.  Send lock_ok to Transaction Owner (TO);
13.  While(!timeout)
     {
14.      If(receive(allow_write)==true)
         {
15.          for (k= 1; k<= cardinality (WS_k); k++)
             {
16.              sfence;
17.              Write_log "Begin_metadata WS_k, k";
18.              Write_Log "remove F_k from F_AVL";
19.              Write_Log "Insert KA_k into K_AVL";
20.              Sfence;
21.              Write_log "Commit_metadata WS_k, k";
23.              Inform_replica(KA_k,Tx, TO);
             }
```

```
25.         Send ready_to_commit to;
26.         Break;
        }
    }
27. If(timeout)
    {
28.     send abort(Tx);// Either TO failed or lock_ok was not
        received from some other nodes by TO.
    }
29. While(!timeout)
    {
30.     If(receive(commit_ok)==true)
        {
31.         for (k= 1; k<= cardinality (WSk); k++)
            {
32.             Update K_AVL and F_AVL in NVRAM;
33.             Sfence;
34.             Write_Log "end_commit_metadata WSk";
37.             Unlock Kk;
38.             Inform_replica_to_update();
            }
39.         Write_Log "End_Transaction";
            break;
40.     }
41. }
42. If(timeout)
    {
43.     Release acquired space by updating F_AVL;
44.     Update K_AVL node;
45.     send abort(Tx);
    }
```

Figure 6.4: Write multiple KVs.

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. The algorithm for read is shown in Figure 6.5.

```
read(Sk)
{
    for (k= 1; k<= cardinality(S); k++)
    {
        inode[k]=ReadLock(Kk);
    }
    Send lock_ok and inode[] to transaction owner (TO)
}
```

Figure 6.5: Read multiple KVs.

6.4 Recovery from a failure

To recover from a failure, the algorithm shown in the Figure 6.6 will executed upon restart of the node. It will first see the last log entry of each transaction in the log. If for a particular transaction Tx, an end entry was written, there is nothing to recover. Hence such transactions can be ignored. But, if the last entry of the transaction is a commit, it means the global status of the transaction may be still committed, ended, or aborted. Based on the global status of the transaction we do different operations. If the global status is committed, we can assume that there is no further update of the transaction processed in any node including TO. Again, such transactions can be ignored and any new updates will be accepted. But if the last entry of a transaction in a node is commit and the global status is end after reboot, we execute the log entries of the transaction and end the transaction in the failed node also by writing an *end_transaction* record. The last combination of a transaction may be that the last entry of a transaction in a node is commit and the global status is aborted. This may be due to failure of another node involved in the transaction. In this case, since the transaction is already aborted by some other node, the recovery node also has to abort the transaction. While recovering we also ignore all such transactions whose current status in the local log is not yet committed and send an abort message to the TO to inform all other nodes to abort that transaction.

```
recovery ()
{
    for each transaction Tx in log
    {
        if (Tx.status == "end")
        {
            continue;
        }
        If(Tx.status == "commit")
        {
            status=lookuptx(Tx);
            If(status == "commit")
            {
                Continue;
            }
            If(status == "end")
            {
                Update K_AVL and F_AVL in NVRAM;
                Sfence;
                Write_Log "End_Transaction";
                Unlock all Key of Tx;
                Inform_replica_to_update();
            }
            If(status == "abort")
            {
                abort(Tx);
            }
        }
        If(Tx.status < "Commit")
        {
            send abort(Tx);
        }
    }
}
```

Figure 6.6: Recovery.

6.5 Takeover by a replica

As mentioned earlier, to mask failures, we can create a slave copy of each KV. Once a node A detects the failure of a node that contains the master copy of a KV, node A tries to mask the failure. But masking the failure is not always possible. It depends upon the current status of an ongoing write. Each node maintains a set of keys for which it is a slave, called the *slave_set*. There is one *slave_set* for every node for which this node has slave keys. Further, every key which is taking part in a transaction, is in a set called *Active_set*. Each element of this set is a three tuple, <k,t,o> identifying the key k, the transaction id t and o the identity of the node which is the transaction owner. When the failure of a node is detected (there could be many ways this could happen, including broadcast messages from other nodes), the function failure() shown in Figure 6.7 is called. This function first creates a set called Master_set which

is populated from elements that are in *slave_set* but are not in *Active_set*. This set represents all the keys for which this node can immediately become the Master. So, the node broadcasts a message declaring itself the Master of all these keys. Now the keys in the *Active_set* have to be checked. The function takeover() calls for each key in *Active_set*, the corresponding transaction owner to find out the status of the transaction. As shown in Figure 6.8, if a "commit" has been written to the transaction log at the TO then the copy of the KV in this node may be stale (the failed node may have committed the write and so has the up-to-date copy). So, it cannot become the master for this key. If on the other hand, the transaction has not committed, this node informs the TO abort the transaction and it takes over as the owner. For a committed transaction, the KV will not be available till the Master node recovers.

```
failure (node) //function called when the failure of a node is
detected
{
   // Slave_set is a set of keys for which this node is a slave and
   the failed node is the Master. There is one slave_set for each
   node whose keys are slaves in this node.
   // Master_set is a set of keys for which this node can become
   the Master immediately
   // Active_Set is a set of three tuples <key#, transaction#,
   owner_node>   Each item denoting that key# is active in
   transaction# and that the transaction owner is owner_node
   Master_set = NIL;
   FOREACH I IN Slave_set AND NOT IN Active_set
      Insert I in Master_set;
   Broadcast (Master, Master_set); //broadcast Master status of
   keys
   Master_set = NIL;
   Foreach <k,t,o> IN Active_Set
      Takeover(k,t,o);
   Broadcast (Master, Master_set)
}
```

Figure 6.7: The failure() routine.

```
takeover(k,t,o)
{
    status = lookuptx (o,t); // ask o the status of t; status is
    "commit" or "not_commit"
    If(status != "commit")
    {
        Insert k IN Master_set;
        send abort(o,t);
    }
}
```

Figure 6.8: The takeover() routine.

Chapter 7
Evaluation of DKVS

7.1 Introduction

We verified our implementation by running our custom workload 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 the 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). Since a Redis Cluster with six nodes is also implemented in a single server using six user processes, comparing our setup with a Redis setup became possible. We compared the two systems using the YCSB benchmark.

7.2 Custom Workloads

We designed our own set of nine workloads and compared their results. This 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 7.1.

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 write and update increases. 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.

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

Table 7.1: Results of Custom Workloads

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 the runs. The performance, as expected, goes down drastically in comparison with earlier workload runs

7.3 YCSB Benchmark

To evaluate performance with more realistic workloads, we ran the YCSB [41] benchmark in the six node set ups. 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 have created six instances of Redis on different ports (used existing create-cluster script of the Redis distribution) and formed a cluster. We have compared the Redis Cluster with two versions of our system- enabling and disabling cache flushes. When we disabled cache flush, we observed that our implementation performed slightly better than Redis in read-intensive workload (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. It may be noted here that Intel has recently announced NVRAM system with their Optane technology, where they are providing an "e-ADR' option. If this option is supported, then, there is no need to explicit flush caches as the system will automatically flush all caches to NVRAM on detecting a power failure. So, if this feature becomes common, then our system will be achieving performance the same as clustered Redis. 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. From these observations, we can conclude that the DKVS system's performance is comparable to similar systems. For each experiment, the code was executed 5 times, and the average result of these 5 runs was used as shown in Table 7.2.

	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	-14.00	4644	-10.00	5078	-2.00

Table 7.2: Result of The YCSB Experiment. Throughput(Ops/Sec)

Chapter 8

Related Work in NVRAM KV Stores

Key-Value Stores were first proposed at Carnegie Mellon University's Parallel Data Lab as a research project in 1996 [18]. 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 [42] [43] [44] 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. Most of the systems reported in the literature have provided multi-KV transaction support as most designs are for NoSQL systems rather than RDMS systems.

8.1 Redis

Redis [45], 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.

8.2 FaRM

An early paper on the subject describes a system called FaRM [30]. 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. Further, their NVRAMs in each node is small, 2GB in size. 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. They use optimistic concurrency control. A central coordinator is used to implement this. We also use a central coordinator (the node initiating the transaction), but use two phase locking. A ring, located at each receiver of every possible sender-receiver pair serves as a one-way RDMA communication channel between a pair of nodes. Logs and data to be read / written are placed on the ring for the receiver to use it as appropriate. To read data by node n1, a request is placed on the ring of the node with the data, say n2. n2 node reads the data locally, and places it on n1's ring for n1 to read the data. A similar method is used for writes. Due to the use of battery backup, these rings survive node failures. This simplifies the issues of durability and atomicity. During commit of data in the optimistic concurrency control validation, backups of objects are also committed. Since every node manages the reading and writing of data locally, the scheme is independent of the underlying storage scheme: it could be a DBMS, a Key-Value store, flat files, etc. The system decides where items and their backups will be stored, while our scheme allows applications to decide the placement of items, and replication is done at a node level, and not at an object level. That is, all the objects in one node are all replicated in another node. This allows easy switchover to a backup version when the primary fails.

8.3 uDepot

uDepot [46], is a key-value store designed to use fast NVRAM block-based devices (attached via a PCI Express bus). It provides variable-sized keys and values with key sizes up to 64 KB and value sizes 4 GB. uDepot maintains its index structure in DRAM using a two-level hash table. The NVRAM space is split into two types segments: index segments for flushing index tables from DRAM, and KV segments for storing KV records. Space management in uDepot is done using a log-structured approach by allocating spaces sequentially. Defragmentation is done using a garbage collection process. If uDepot is not cleanly shut down, the index structure has to be reconstructed from KV records found in KV segments. uDepot bypasses the page cache and accesses the storage directly. This system is not of direct relevance to our design as we use NVM for storage.

8.4 PapyrusKV

Another system using NVRAM block-based devices (NVMe systems) is PapyrusKV [47]. It assumes that the NVRAM system is a buffer for larger disk-based storage. So, it uses an LSM tree based indexing scheme and it is modelled on the BigTable / HBase implementations. The design parameters are very different from our system's parameters.

8.5 KVSSD

In KVSSD [48], another block device based system, the authors proposed compound commands which allow the host to specify multiple key-value pairs in a single NVRAM operation. KVSSD commands are defined to be processed independently of each other. Thus, even though a set of operations are consecutively requested to KVSSD, other operations can interleave. This independence among operations makes KVSSD unable to guarantee the atomicity and isolation properties of transactions.

8.6 HiKV

HiKV [49] 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. Consistency of data in NVRAM is ensured by using 16-byte atomic writes using the cmpxchg16b instruction (with LOCK prefix). For durability, HiKV use clflush and sfence instructions. The focus of their paper is on the index design. 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.

8.7 SLM-DB

SLM-DB [32] is similar to HiKV as it combines the Log-Structured Merge Trees design and a B+ tree to leverage NVRAM in an NVRAM and disk-based hybrid system. For fast lookup SLM-DB use a B+ tree index on NVRAM. SLM-DB has a single-level organization of KV pairs on disks and performs selective compaction for the KV pairs, collecting garbage and keeping the KV pairs sorted sufficiently for range query operations. Data durability in persistent memory is achieved through mfence and clflush instructions. In SLM-DB, it is assumed that for large KV stores, data will be stored on disks, and NVRAM is expected to coexist as a non-volatile memory buffer.

8.8 Telepathy

In [50], 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.

8.9 Caribou

Caribou [51] is an intelligent distributed system that provide 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 which uses field programmable gate arrays (FPGA). The FPGA has no caches in front of memory and therefore read and writes are directly performed in the memory. The absence of cache in front of the main memory reduces the complexity to achieve durability during writes.

8.10 FaSST

FaSST [31] is an RDMA-based system that provides distributed in-memory transactions with serializability and durability. The focus of FaSST is on challenging the common choice of one-sided RDMA operations in transaction processing systems. One-sided RDMA provides limited operations, and accessing data stores often requires traversing complex data structures, leading to lower throughput and higher latency. FaSST introduces the idea of using remote procedure calls (RPCs) over two-sided unreliable RDMA using datagram messages as a primitive. Unlike one-sided RDMA, FASST involves the remote CPU in message processing and offer greater flexibility for data access in a single round trip. It demonstrates that FaSST RPCs provide significantly higher throughput and CPU efficiency compared to existing systems, using one-sided RDMA. Our design does not consider RPC through RDMA as an alternative, as our view is that the extra complexity does not improve performance appreciably when compared to our scheme.

8.11 DrTM+R

DrTM+R [52] 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.

8.12 DrTM+H

DrTM+H [53] investigates the impact of different choices of RDMA (Remote Direct Memory Access) primitives and designs on the performance of distributed transactions. The work aims to provide guidance on optimizing distributed transactions with RDMA, comparing the one-sided and two-sided RDMA primitives. Authors systematically compared different RDMA primitives and designs for distributed transactions, providing their relative performance at the primitive level, phase-by-phase execution, and end-to-end system performance. The proposed hybrid design, DrTM+H, is shown to outperform existing systems in certain scenarios, emphasizing the importance of carefully choosing RDMA primitives for different phases of transaction processing. Our design uses RDMA as an add-on to improve performance. The system will work even without RDMA. Complex schemes to optimally use RDMA, we feel, does not add much to performance. Further, with faster LAN systems, and with servers with a large number of cores, the performance improvement in using RDMA is likely to reduce in the years ahead.

Chapter 9
Conclusion

NVRAM provides high bandwidth and byte level random access with non-volatility property. These devices are also large in capacity and less power consuming. The recent announcement of Intel of Xeon servers with TB of NVRAM storage on the memory bus has made it important to learn to use this new feature. This thesis is an attempt in this direction.

It first describes the design and implementation of a file system on NVRAM. This exercise revealed the overheads that has to be incurred to provide durability of writes to memory in the face of the use of cache memories - which are volatile - in the path to NVRAM. The design showed that availability of special instructions provide in the Intel architecture allows the efficient implementation of a file system on NVRAM. We have proposed and evaluated our scheme to ensure that data has reached NVRAM by flushing and re reading the data again from NVRAM to check if what is obtained is what was written. To implement the idea, we do not need to re-read all cache lines. If a sequence of writes is followed by a sfence instruction and then another write and then a clflush instruction is executed on the cache lines written into, only the last write needs to be read again to ensure that all the writes have reached NVRAM. Our system, named DurableFS, uses copy-on-write for writing into data blocks, and it uses a redo log to record changes to metadata while making the changes to versions of the metadata buffered in RAM. It Durability of data in NVRAM is ensured by flushing data in caches of the CPU by using cache flush and fence instructions. Consistent updates are made using movnti instructions. The point of durability is the closing of a file, and it is also the point of atomicity. We evaluated DurableFS running the Fio and Filebench workloads and compared the results with the NOVA file system, another file system for NVRAM and we got similar performance.

The real power of NVRAM systems can be realised only when they replace the current typical setup of a commercial business system. This is a cluster of computers with shared storage through a SAN, nowadays implemented in a so called hyperconverged architecture. Therefore, we designed and implemented a distributed Key-value store for a cluster of computers with only NVRAM as secondary storage and with RDMA facility through a high speed interconnection. We chose a Key-Value store rather than a file system as the current trend is to use a key-value store as the basic storage system. This allows a variety of high level structures to be efficiently implemented, be it an RDMS system or a NOSQL store. For remote reading, we have used RDMA to bypass the remote network stack. Use of RDMA improves the performance of read intensive workloads. We have compared our system with a Redis cluster using custom workloads as well as using the YCSB benchmark. The comparison shows for a RAM only implementation, our system's performance is comparable to that of Redis. The system performs about 14% slower than Redis if we enable cache flushes required for achieving data durability

in NVRAMs. DKVS is a KV store which is distributed in nature, provides distributed transactions to maintain ACID properties, and uses RDMA for fast data reading and provides replication of KVs to improve reliability.

We feel that there are two directions in which further work can be carried out. First of all, we need to build an application on top to show the feasibility of not only the new technology, but also of our design. One direction will be to port the MariaDB RDBMS system to use our clustered KV store and then to evaluate performance with real data. The second direction of research needs to consider how to organise NVRAM storage in a single, multi-socket computer. Each socket of modern servers has 36 to 40 cores with up to 8 memory channels. Each memory channel will be able to handle TBs of NVRAM. If we assume 2 TB of NVRAM per channel, then one socket with 40 cores will have 16 TB of NVRAM. A server with 8 sockets will have 320 cores and 128 TB of NVRAM (and an equal amount of RAM). The interconnection among the sockets is physically not a bus (each is not equally accessible from all others). Neither are the core to memory paths in a socket on a bus. So new algorithms will have to be devised. Most probably, we will need to consider distribution of computation besides distribution of data, resulting in a distributed computing architecture. These modern servers are targeted for use in Cloud data centers with virtual resources being allocated to user application. A cluster of virtual CPUS and storage may become an allocable unit. How such allocations are done is clearly another direction future work can take.

References

[1] January 2013. [Online]. Available: http://researcher.watson.ibm.com/researcher/files/us-gwburr/Almaden_SCM_overview_Jan2013.pdf. [Accessed 21 February 2022].

[2] B. Lee, E. Ipek, O. Mutlu and D. Burger, "Architecting phase change memory as a scalable DRAM alternative," in *36th International Symposium on Computer Architecture (ISCA 2009)*, Austin, TX, USA, 2009.

[3] H. Yiming, "Spin-transfer torque MRAM (STT-MRAM): Challenges and prospects," *AAPPS bulletin*, vol. 18, no. 6, pp. 33-40, 2008.

[4] D. B. Strukov, G. S. Snider, D. R. Stewart and R. S. Williams, "The missing memristor found," *nature*, vol. 453, no. 7191, pp. 80--83, 2008.

[5] "3D XPoint," [Online]. Available: https://en.wikipedia.org/wiki/3D_XPoint. [Accessed 2022].

[6] C. Jeremy, N. E. B, F. Christopher, I. Engin, L. Benjamin, B. Doug and C. Derrick, "Better I/O through byte-addressable, persistent memory," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[7] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena and M. M. Swift, "Aerie: Flexible file-system interfaces to storage-class memory," in *European Conference on Computer Systems*, 2014.

[8] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. Gupta and S. Swanson, "Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories," in *International Symposium on Microarchitecture, MICRO*, Atlanta, Georgia, USA, 2010.

[9] S. R. Dulloor, S. Kumar, A. S. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran and J. Jackson, "System software for persistent memory," in *European Conference on Computer Systems*, 2014.

[10] E. Ipek, J. Condit, E. B. Nightingale, D. Burger and T. Moscibroda, "Dynamically Replicated Memory: Building Reliable Systems from Nanoscale Resistive Memories," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, Pittsburgh, Pennsylvania, USA, 2010.

[11] C. Wang, S. S. Vazhkudai, X. Ma, F. Meng, K. Youngjae and C. Engelmann, "NVMalloc: Exposing an Aggregate SSD Store as a Memory Partition in Extreme-Scale Machines," *International Parallel and Distributed Processing Symposium*, pp. 957-968, 2012.

[12] J. Zhao, S. Li, D. H. Yoon, Y. Xie and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *Annual IEEE/ACM International Symposium on Microarchitecture*.

[13] J. Coburn, A. M. Caulfield, A. Akel, L. Grupp, R. Gupta, R. Jhala and S. Swanson, "NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, CA, USA, 2011.

[14] P. Sehgal, S. Basu, K. Srinivasan and K. Voruganti, "An empirical study of file systems on NVM," in *Symposium on Mass Storage Systems and Technologies (MSST)*, Santa Clara, CA, USA, 2015.

[15] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," *ACM Transactions on Computer Systems*, vol. 26, no. 2, pp. 1-26, 2008.

[16] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh and P. Schwarz, "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Transactions on Database Systems*, vol. 17, no. 1, pp. 94-162, 1992.

[17] E. R. Giles, K. Doshi and P. Varman, "SoftWrAP: A lightweight framework for transactional support of storage class memory," in *Symposium on Mass Storage Systems and Technologies (MSST)*, 2015.

[18] "wikipedia.org," wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Object_storage. [Accessed 21 February 2022].

[19] C. H. Lam, "Storage Class Memory," in *IEEE International Conference on Solid-State and Integrated Circuit Technology*, Shanghai, China, 2010.

[20] "Intel architecture instruction set extensions programming reference," [Online]. Available: https://software.intel.com/sites/default/files/managed/07/b7/319433-023.pdf. [Accessed July 2016].

[21] J. Xu and S. Swanson, "NOVA: a log-structured file system for hybrid volatile/non-volatile main memories," in *Usenix Conference on File and Storage Technologies*, 2016.

[22] B. Kumud, C. D. R and B. H. J, "Implications of CPU Caching on Byte-addressable Non- Volatile Memory Programming," HP Laboratories, 2012.

[23] "Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2," Intel.

[24] "FIO – FLEXIBLE I/O TESTER SYNTHETIC BENCHMARK," storagereview.com, [Online]. Available: https://www.storagereview.com/fio-flexible-i-o-tester-synthetic-benchmark. [Accessed 2022].

[25] "Filebench," [Online]. Available: https://github.com/filebench/filebench/wiki. [Accessed 2022].

[26] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. d. Silva, S. Swanson and A. Rudoff, "NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System," in *Proceedings of the 26th Symposium on Operating Systems Principles*, Shanghai, China, 2017.

[27] M. Dong and H. Chen, "Soft updates made simple and fast on non-volatile memory," in *Usenix Annual Technical Conference*, 2017 .

[28] "Direct Access for files," [Online]. Available: https://www.kernel.org/doc/Documentation/filesystems/dax.txt. [Accessed 2022].

[29] J. Yang, J. Izraelevitz and S. Swanson, "Orion: a distributed file system for non-volatile main memories and RDMA-capable networks," in *USENIX Conference on File and Storage Technologies*, 2019.

[30] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam and M. Castro, "No compromises: distributed transactions with consistency, availability, and performance," Symposium on Operating Systems Principles, 2015.

[31] K. Anuj, K. Michael and G. A. David, "FaSST: Fast, Scalable and Simple Distributed Transactions with Two-sided (RDMA) Datagram RPCs," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[32] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh and Y.-R. Choi, "SLM-DB: Single-Level Key-Value Store with Persistent Memory," in *USENIX Conference on File and Storage Technologies*, 2019.

[33] Zaitsev and Peter, "Innodb architecture and performance optimization," *O'Reilly MySQLConference and Expo*, 2009.

[34] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas and L. Vivier, "The new ext4 filesystem: Current status and future plans," *Proceedings of the Linux symposium*, vol. 2, pp. 21-33, 2007.

[35] R. Russon and Y. Fledel, "NTFS Documentation," 2013. [Online]. Available: https://dubeyko.com/development/FileSystems/NTFS/ntfsdoc.pdf. [Accessed February 2022].

[36] P. O'Neil, E. Cheng, D. Gawlick and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351-385, 1996.

[37] J. Rao and K. A. Ross, "Making B+- trees cache conscious in main memory," in *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, 2000.

[38] M. Fasheh, "The Oracle Clustered File System, Version 2," *Linux Symposium*, vol. 1, pp. 289-302, 2006.

[39] "netlink(7) — Linux manual page," [Online]. Available: https://man7.org/linux/man-pages/man7/netlink.7.html.

[40] "HowTo Configure Soft-RoCE," [Online]. Available: https://community.mellanox.com/s/article/howto-configure-soft-roce. [Accessed February 2022].

[41] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010.

[42] V. Srinivasan, B. Bulkowski, W.-L. Chu, S. Sayyaparaju, A. Gooding, R. Iyer, A. Shinde and T. Lopatic, "Aerospike: architecture of a real-time operational DBMS," *Proceedings of the VLDB Endowment*, vol. 9, no. 13, p. 1389–1400, 2016.

[43] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *ACM Symposium on Operating Systems Principles*, Stevenson, Washington, USA, 2007.

[44] Ghemawat, Sanjay, Dean and Jeff, "LevelDB, A fast and lightweight key/value database library by Google," 2014.

[45] "Redis Documentation," [Online]. Available: https://redis.io/documentation.

[46] K. Kourtis, N. Ioannou and I. Koltsidas, "Reaping the performance of fast NVM storage with uDepot," in *USENIX Conference on File and Storage Technologies*, Boston, MA, USA, 2019.

[47] J. Kim, S. Lee and J. S. Vetter, "PapyrusKV: a high-performance parallel key-value store for distributed NVM architectures," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017.

[48] S.-H. Kim, J. Kim, K. Jeong and J.-S. Kim, "Transaction Support using Compound Commands in Key-Value SSDs," in *USENIX Conference on Hot Topics in Storage and File Systems*, 2019.

[49] F. Xia, D. Jiang, J. Xiong and N. Sun, "HiKV: a hybrid index key-value store for DRAM-NVM memory systems," in *Usenix Annual Technical Conference*, 2017.

[50] Q. Liu and P. Varman, "Silent Data Access Protocol for NVRAM + RDMA Distributed Storage," in *IEEE Parallel and Distributed Processing Symposium Workshops*, New Orleans, LA, USA, 2020.

[51] Z. István, D. Sidler and G. Alonso, "Caribou: intelligent distributed storage," in *Proceedings of the VLDB Endowment*, 2017.

[52] C. Yanzhe, W. Xingda, S. Jiaxin, C. Rong and C. Haibo, "Fast and general distributed transactions using RDMA and HTM," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016.

[53] W. Xingda, D. Zhiyuan, C. Rong and C. Haibo, "Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better!," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[54] S. Scargall, Programming Persistent Memory, Berkeley, CA: Apress, 2020.