BTU Cottbus – Senftenberg
Faculty 1 – Institute for Computer Science
Distributed Systems and Operating Systems Group
Prof. Dr.-Ing. Jörg Nolte

Brandenburg
University of Technology
Cottbus - Senftenberg

– Master Thesis –

Exploiting NVRAM for a Key-Value Store with Serializable Transactions

Jacob Lorenz
Computer Science

3rd May 2018

Reviewer:	Prof. Dr.-Ing. Jörg Nolte
Reviewer:	Prof. Dr.-Ing. habil. Ingo Schmitt
Adviser:	M. Sc. Jana Traue

Eidesstattliche Erklärung

Der Verfasser erklärt, dass er die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt hat. Die aus fremden Quellen (einschließlich elektronischer Quellen) direkt oder indirekt übernommenen Gedanken sind ausnahmslos als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden.

Ort, Datum Unterschrift des Verfassers

Contents

1 Introduction **1**

2 Non-Volatile RAM **5**

2.1 Architectures and Applications 5

2.2 Technologies 8

2.3 Challenges 9

2.3.1 Unintended Durability 9

2.3.2 Memory Management 10

2.3.3 Preserving Consistency 12

2.4 Summary 16

3 Key-Value Stores **17**

3.1 Overview 17

3.2 Transactions 20

3.3 Concurrency Control Protocols 26

3.3.1 Strategies 26

3.3.2 Multiversion Concurrency Control 28

3.3.3 Snapshot Isolation 30

3.3.4 Serializable MVCC 33

3.4 Key-Value Stores for NVRAM 35

3.5 Summary 38

4 A Key-Value Store with Serializable Transactions for NVRAM **41**

4.1 Overview 42

4.2 System Architecture 44

4.3 Key-Value Store Design 47

4.3.1 Two-Level Store 47

4.3.2 Transactions 48

4.3.3 Structure 50

4.4 Crash Consistency . . . 53

5 Implementation 55

5.1 Usage Examples . . . 55

5.2 Architecture . . . 58

5.2.1 NVRAM Management . . . 59

5.2.2 Durable Data Structures . . . 61

5.2.3 Components . . . 64

5.3 Transaction Processing . . . 68

5.3.1 Visibility . . . 68

5.3.2 Reads . . . 70

5.3.3 Updates . . . 70

5.3.4 Deletions . . . 71

5.3.5 Commits . . . 72

6 Evaluation 75

6.1 Challenges . . . 75

6.2 System Configuration . . . 76

6.3 Transaction Throughput . . . 77

6.3.1 Setup . . . 77

6.3.2 Methodology . . . 79

6.3.3 Results and Discussion . . . 81

6.4 Summary . . . 86

7 Conclusions and Future Work 89

A Acronyms 91

List of Figures 93

List of Tables 95

Bibliography 97

1 Introduction

Databases are an integral part in many areas such as accounting, trading, and machine learning. While use cases may vary, databases are generally used to manage large amounts of data fast and reliably. An important method to access data in a database are *transactions*. A transaction is a sequence of operations on data that appears as a single atomic operation. Therefore, an important factor in delivering the mentioned technologies is *transaction throughput*.

There are several ways to increase transaction throughput. One approach is to hold large portions of data or even the entire database in fast volatile memory as in Main-Memory Databases (MMDBs) [GKP$^+$10, LKF$^+$13, DFI$^+$13]. In this case, recovery is crucial as all data in volatile memory is lost in a crash. Therefore, recovery usually involves access to non-volatile, yet slower, storage media which impairs throughput.

Another approach to improve transaction throughput is to utilize multi-core architectures by allowing transactions to execute concurrently. This way, an incoming transaction does not have to wait until a running transaction terminates. However, race conditions between concurrent transactions may result in inconsistent data. Therefore, concurrent transactions must be isolated by means of a dedicated *concurrency control*. There are different *isolation levels* but only *serializability* guarantees that no sequence of concurrent transactions can yield inconsistent data [BBG$^+$95, LBD$^+$11, NMK15]. Although desirable, serializability is often unsupported or discouraged due to lower transaction throughput.

An important class of databases that often feature both in-memory operation and concurrent transactions are *key-value stores* [DHJ$^+$07, CDG$^+$08, LM10]. Unlike relational databases, a key-value store consists of a single associative collection. Lacking query languages and data schemas, key-value stores avoid a lot of overhead that may not be required in certain circumstances.

Recent research suggests that byte-addressable Non-Volatile Memory (NVM) with parameters close to Dynamic RAM (DRAM) will be available in the near future [DKK$^+$14, OL17, ALR$^+$17]. Notable technologies currently in research are Phase-Change Memory (PCM), Spin Torque Transfer RAM (STT-RAM), and 3D XPoint. It is suggested that, due to their characteristics. these technologies are strong candidates for non-volatile main memories. Commonly accepted terms for such memory are Non-Volatile RAM (NVRAM) and Storage Class Memory (SCM).

The integration of NVRAM can have many implications on both operating systems and applications [CNF+09, BCGL11, PWGB13, BC16]. With projected densities higher than DRAM, NVRAM is suggested to replace disk storage in some cases. This could significantly reduce recovery overhead in MMDBs. For one, databases would no longer have to move their data from non-volatile storage to main memory, thus enabling instant restarts. More importantly, persisting data would be faster by multiple orders of magnitude.

Recent research indicates that NVRAM-resident MMDB and key-value stores can achieve higher transaction throughput than conventional systems [BHC+13, ZSLH16]. However, these works still rely on non-serializable transactions and do not address the opportunity to enable stronger isolation. Serializability is a desirable property to guarantee data integrity and should not be traded in favor of performance. In the end, NVRAM could be an opportunity to satisfy both demands. Therefore, this thesis aims to provide affordable serializable transactions by leveraging the superior performance of NVRAM over disks.

The task is to design and implement an in-memory key-value store that exploits the benefits of upcoming non-volatile Random Access Memory (RAM) to enable fast serializable transactions. For this purpose, an NVRAM-aware multiversion concurrency control protocol is implemented. In order to validate the approach and determine the overhead of serializability, benchmarks are used to compare the key-value store against non-serializable solutions.

Overview After this introduction, the thesis proceeds with a domain analysis in the fields of NVRAM and key-value stores. Chapter 2 examines NVRAM and discusses applications and challenges, whereas Chapter 3 deals with key-value stores and modern concurrency control protocols. Based on these insights, a concept for an NVRAM-resident key-value store with serializable transactions is developed in Chapter 4. The subsequent chapter presents selected details on the prototypical implementation of the concept. Based on the implemented prototype, the concept is evaluated by means of synthetic benchmarks in Chapter 6. The last chapter provides a summary of this thesis and gives an outlook onto future works in the field.

Terminology As of this writing there is no distinct consensus as to how byte-addressable non-volatile memory should be referred to. This thesis exclusively uses the term NVRAM. There are two reasons for this decision. For one, alternative terms such as Byte-Addressable Persistent RAM (BPRAM) or Persistent Memory (PM) suggest that non-volatile memory is also persistent which has ambiguous definitions and is not consistently used [NHH$^+$17]. Another reason is that some terms such as SCM, NVM, and PM may not reflect the property of byte-wise addressing which is central to these technologies. NVRAM, on the other hand, explicitly denotes all of the desired properties.

2 Non-Volatile RAM

Conventional non-volatile memories such as Hard Disk Drives (HDDs) or Solid State Drives (SSDs) provide large capacities but are only block-oriented and incur substantial access latencies compared to RAM. Therefore, Input/Output (IO)-bound applications tend to keep as much data in RAM as possible but are eventually forced to access slower media for durable storage. This led to the development of non-volatile variants of RAM. Recent research suggests that both fast and high-capacity NVRAM will become widely available in the near future.

This chapter gives an overview on the state-of-the-art in NVRAM research. Included is a discussion of opportunities and challenges, such as the notorious consistency issues in the presence of failures.

2.1 Architectures and Applications

There are numerous examples for applications of NVRAM. While earlier works often considered NVRAM as a way to improve fault tolerance, recent research suggests a broader range of applications. This development is especially driven by recent advances in the manufacture of standalone NVRAM.

Fault-Tolerance As pointed out earlier, a well-known use case of NVRAM is to increase fault tolerance towards crashes. The goal is to retain main memory content even in case of a crash, for instance by an abrupt power loss [GmS92, Eic86]. This way, critical data such as logs of file systems or databases remain durable and can be used to recover and even complete unfinished operations such as making a transaction durable [LGG$^+$91, CNC$^+$96]. In the past, such solutions relied on battery-backed RAM. This was subject to criticism as batteries only have a limited charge to ensure durability. Also, batteries degrade over time and need to be maintained to prevent unexpected failure [GmS92]. Therefore, modern NVRAM solutions which no longer require peripherals are a welcome improvement in this area.

Disk Caches A significant amount of research on NVRAM is dedicated to mitigating the IO bottleneck imposed by traditional disk storage. One way to do so is to defer disk IO via durable disk caches [CNC$^+$96, WZ94]. When an object on disk is requested, it is moved to the disk cache. Once an object is cached, it may be read or modified without accessing the underlying disk. Write-back is only required when there is not enough space for an incoming cache item. In doing so, the number of data accesses involving disk IO can be greatly reduced. Many operating systems including Linux or FreeBSD mimic this concept through the use of page caches in volatile memory. The difference is that volatile caches need to be flushed at some point which requires careful resource management. NVRAM caches on the other hand, only need to evict items when there is no slot for an incoming item.

Disk Replacement Another approach is to treat NVRAM as an equivalent to traditional disk storage. Early works, which were strongly influenced by the lack of high-capacity NVRAM, proposed hybrid storage systems where disk storage was used in conjunction with NVRAM [WRPK02, MBL01]. These works have very similar assignment policies in that they only store small files such as metadata or libraries in NVRAM whereas larger files remain on disk. While this does not remove disk access as a common bottleneck, it certainly alleviates latency for some frequently accessed files. In that regard, NVRAM-complemented disk storage systems are similar to those with NVRAM disk caches.

Modern Use Cases For the time being, many applications will have to deal with a scarcity of NVRAM. But as improving technologies achieve higher capacities with better parameters, new system architectures become feasible. In some cases, traditional storage may be eliminated altogether, making NVRAM the primary storage medium. A prominent use case for this architecture are MMDBs. These databases reside entirely in main memory which drastically reduces latencies when accessing their data. However, they are also vulnerable to crashes, as main memory is still mostly volatile. In order to prevent data loss, MMDBs have to mirror the entire database to non-volatile memory. For that, they perform logging or checkpointing to synchronize individual or groups of changes to non-volatile memory. When the database is restarted, for example in response to a crash, checkpoints and logging information are used to recover the most-recent state of the database that was durable at the time of the crash. That is, MMDBs require non-volatile memory for the sole purpose of recovery. Recurring recovery measures such as logging have been a long-standing issue with MMDBs as they incur expensive disk IO, thus limiting transaction throughput [GmS92, WBR$^+$12, MWMS14]. In addition, restarts reduce the availability of a system as recovering large databases from slow storage can be time-consuming. With NVRAM on the other hand, any MMDB would be implicitly durable, hence making disk IO obsolete. Moreover, it has been shown that with NVRAM logging is no longer necessary which paves the way for instantaneous restarts [OLK+15]. The concept of NVRAM-based MMDB is especially promising as some upcoming variants of NVRAM are projected to feature larger capacities than conventional DRAM [LIMB09, ZWT13, DKK+14]. For that reason, recent research has investigated NVRAM-aware designs for MMDB ranging from Key-Value Stores (KVSs) [BHC+13, ZSLH16, WNZ+16] to full-fledged database systems [OLK+15, SBD+16, ALR+17]. Research results in these areas are central to this work and are reflected in chapter 2 through 4. Given their importance for this thesis, existing KVSs for NVRAM are looked into in more detail at the end of chapter 3.

Experimental Use Cases While most works aim to improve existing architectures, some explore different computation models. A recent example is a proposal to use NVRAM to enable on-chip machine learning. The idea is to move away from the well-known Von Neumann architecture and implement Artificial Neural Network (ANN) by means of NVRAM [FNS+16]. ANNs perform a weighted sum over their inputs before an activation function classifies the result. But before satisfiable results can be obtained, ANNs need to be trained by properly adjusting the scalar weights of their inputs. It has been suggested that with NVRAM weight adjustment could be performed on-chip where updated weights would be durable without the need for write-back. However, neither artificial intelligence nor alternative computation models are subject of this work.

Most proposals concerning NVRAM assume the presence of volatile RAM. [OL17]. The reason behind this assumption is that not all parts of memory may be intended for durability. Nonetheless, recent research suggests that systems exclusively based on NVRAM can be built [NH12, Cou16]. Clearly, such an architecture would have severe consequences for both operating systems and applications [BCGL11]. For example, operating system processes would remain in memory even if terminated. On the one hand, it could significantly accelerate the procedure of invoking a process. On the other hand, all data belonging to a process' address space would be durable even if they were corrupted by a crash. Other issues are concerned about memory management, device drivers, and vital information. An early prototype of such a system is currently in development [Cou16]. This topic however, is beyond the scope of this work and it is henceforth assumed that volatile RAM will co-exist with NVRAM.

As shown above, NVRAM provides an opportunity to improve existing architectures and even create new computation models. Given sufficient parameters such as capacity and endurance, NVRAM could resolve the IO bottleneck of non-volatile storage media. Especially systems such as MMDBs have shown considerable gains in transaction throughput and recovery times. Consequently, MMDBs and in particular KVSs are at the center of this work.

2.2 Technologies

The design and integration of fast NVRAM is not a new research area. In the past, there have been multiple attempts to produce non-volatile equivalents of main memory. While earlier approaches were mainly designed to make systems more tolerant to crashes [GmS92], recent research suggests NVRAM to hold entire MMDBs and speed up recovery [OLK$^+$15, SBD$^+$16, ALR$^+$17].

One way to achieve byte-addressable NVM is to attach DRAM or Static RAM (SRAM) to backup power supplies as in [LGG$^+$91, WRPK02]. In other cases, conventional non-volatile storage, such as flash memory, is directly attached to the DRAM module [SXH$^+$10, HJ14, ONO16]. However, these approaches rely on batteries, which must be maintained, or block-oriented memory which is still much slower than DRAM. A more promising approach is to develop alternative memory techniques that provide the features of NVRAM, natively.

Among a range of recent NVRAM designs, the most promising are PCM and STT-RAM [ZWT13, MV16, JRRR17]. These technologies vary according to the following parameters:

- density
- endurance
- latency
- power consumption

While PCM is projected to have the highest density of all, including DRAM, it also features a lower endurance. In terms of endurance, STT-RAM fares better than PCM but is still surpassed by DRAM. Also, STT-RAM has a very low density which prevents high capacity memory modules. NVRAM is known to incur higher access latency than DRAM. This is certainly true for PCM which can have up to 10% higher latency than DRAM. STT-RAM, on the other hand, is projected to be on a par with DRAM. Concerning power consumption, STT-RAM appears to have advantages over both DRAM and PCM. In this case, PCM registers an even higher consumption than DRAM [MV16].

In total, STT-RAM poses a very promising technology but provides suboptimal capacities and is also less enduring than DRAM. PCM, on the other hand, provides a higher capacity but at the cost of higher latencies. Still, recent research suggests PCM to be the first fast high-capacity NVRAM to be ready to manufacture [ZWT13, DKK+14, MV16]. Nevertheless, this work is independent of the underlying NVRAM technology.

2.3 Challenges

Despite the conceivable advantages of NVRAM, there are also challenges to be addressed. Although most issues are of practical nature there are also conceptual concerns.

2.3.1 Unintended Durability

The key feature of NVRAM is to retain its data across restarts. However, not all data is necessarily intended to be durable. Notable examples include transient, confidential, and corrupt data. The former comprises data which may not be valid after a system restart, as is the case with data related to machine or device state.

Information Security Other data such as passwords, encryption keys, or decrypted data may be confidential and should not be durable. It has been shown that even volatile RAM holds its charge long enough so that a module can be moved to an attacker's machine for read-out [HSH+08, YADA17]. Such attacks would be trivial on NVRAM [BCGL11], but that is beyond the scope of this work.

Stray Writes When an operating system or application behaves in an erratic fashion or crashes, it may produce corrupt data in memory. This is called a stray write. In this case, systems incorporating NVRAM could face durable memory corruptions [CNF+09, VTR+11, DKK+14]. In contrast to conventional non-volatile memory, NVRAM is particularly vulnerable as it is directly accessible through the Central Processing Unit (CPU). However, it has been shown that, compared to disk storage, stray writes do not occur significantly more often in NVRAM [CNC+96]. Therefore, stray writes are not an issue in this work.

2.3.2 Memory Management

NVRAM is a new type of memory that can also be used as durable mass storage. In order to benefit from this new technology, both platforms and operating systems need to find ways to efficiently manage it. There are several issues to be addressed in this area.

System Integration An important aspect in managing NVRAM is the memory interface. Recent research suggests that NVRAM will be attached to the system memory bus using the Dual In-line Memory Module (DIMM) format known from DRAM [NHH$^+$17, OL17, ALR$^+$17]. A decisive advantage of this approach is the much lower latency compared to the alternative IO bus. Also, previous efforts to produce NVRAM, such as DRAM-attached SSDs, have also been integrated as DIMMs (known as Non-Volatile DIMM (NVDIMM)) [DKK$^+$14, HJ14]. Consequently, system designers can build on an existing software stack. Still, there are drawbacks to be considered. Clearly, the number of available DIMM slots in a machine is limited, so NVRAM may not scale well for mass storage. That situation is especially relevant in hybrid systems containing both RAM and NVRAM. Also, in hybrid systems both kinds of memory are likely to be attached to the same memory interface thus sharing its bandwidth.

Programming Model With NVRAM devices integrated into the system, programmers still need a way to access it. Several approaches have been proposed to this end [NHH$^+$17]. While it is always possible to operate on NVRAM by mapping individual device regions into virtual memory, there are considerable weaknesses to this approach [CNF$^+$09, VTS11, DKK$^+$14, NHH$^+$17]. A major challenge of working with NVRAM is to provide consistency guarantees across possible system failures. Yet, systems are largely unaware of these circumstances. With raw device access which is already error-prone, the complex task of preserving consistency is handed to the programmer. Another challenge is that, due to Address Space Layout Randomization (ASLR), virtual memory mappings are volatile and are not valid across restarts. Therefore, it has been proposed to rely on dedicated high-level programming primitives as in Mnemosyne, NV-Heaps, and PMDK [VTS11, CCA$^+$11, Rud17]. These systems provide interfaces for memory allocation and consistent updates based on transactions. An important distinction to the previous low-level approach is that memory is accessed through an NVRAM-aware software interface. Unfortunately, as of this writing, there is no evident consensus regarding the programming model to use for NVRAM [BC16]. Still, the middleware PMDK (formerly known as NVML) appears to be gaining the upper hand [OL17, NHH$^+$17, Mal17, ALR$^+$17]. Figure 2.1 depicts how PMDK sits between the application and the file system. It operates by mapping files into a process' address space to enable load-store semantics.

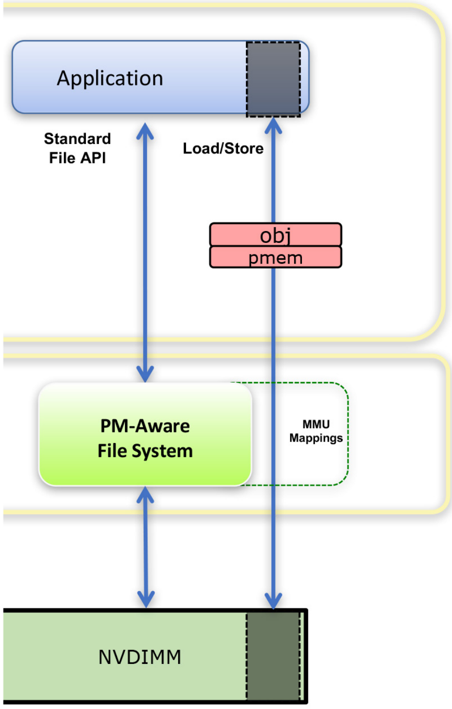


Figure 2.1: System architecture with PMDK to manage NVRAM [Rud17].

Another discussed approach to manage NVRAM is through designated file systems [OL17, ALR+17]. File systems provide a well-known and suitable abstraction for non-volatile storage. In order to enable regular memory access in a load-store manner, individual files can be mapped into virtual memory. However, traditional file systems are not directly well-suited for use with NVRAM. One reason is that most operating systems provide a page cache which is used by file systems to defer expensive disk IO. In the case of NVRAM, page caches may be no longer needed, as updates to NVRAM incur far less latency compared to other non-volatile memories. In this regard, page caches even add overhead instead of mitigating it. Apart from that, they add a level of indirection which makes writes to NVRAM more likely to be torn by failures. Also, traditional file systems are usually designed for block-oriented devices which may no longer be the best option. Therefore, several NVRAM-aware file systems have been proposed [CNF+09, WR11, DKK+14, XS16]. The key feature of these file systems is a zero-copy mechanism by circumventing page caches. This enables true store-load semantics for memory-mapped files. Other aspects include attempts to leverage the byte-addressable nature of NVRAM and crash-related consistency issues. Yet, there appears to be a growing consensus that instead of designing new file systems for NVRAM, existing file systems should be outfitted to support NVRAM. A prominent result is the adoption of Direct Access (DAX) in the Linux kernel [OL17, ALR+17, Rud17]. DAX is a mechanism to bypass the operating system's page cache and is used by PMDK [Rud17, Cor].

2.3.3 Preserving Consistency

As pointed out earlier, the consistency of data stored in NVRAM is vulnerable to crashes or power failures [CNF+09, DKK+14, OL17]. Since NVRAM is directly attached to the processor memory interface, there is no need to use techniques such as Direct Memory Access (DMA) to transfer a modified page to external storage. This also means that a memory operation solely relies on the CPU which usually gives no confirmation when that operation completes. In this context, there are two major issues that threaten the consistency of data written to NVRAM, namely out-of-order execution and deferred write-back.

Out-Of-Order Execution

When executing a program, processors fetch instructions in a consecutive manner. Some instructions may inflict minimal latency, while others such as load operations may delay further execution for hundreds of cycles. In an attempt to optimize instruction throughput, individual instructions may be reordered. While compilers may statically define promising orders, processors are able to reorder instructions at runtime. This enables processors to optimize resource utilization and hide latencies of time-consuming instructions. However, only reorderings that do not violate data dependencies between instructions are possible. While processors do prevent such conflicts, there are dependencies that cannot be observed. For example, in order to mark a chunk of data as durable in NVRAM, one might store a designated flag immediately after the operation completed. With out-of-order execution it is possible that the flag is written before the payload. This can lead to severe inconsistencies especially when a crash prevents the chunk from being written.

A common method to counter this issue is to enforce memory order with memory barriers (also fences) [DKK+14, SBD+16, OL17]. A memory barrier prevents the CPU from proceeding until all prior memory operations have completed. Although a barrier does not directly order its preceding instructions, it can be used to impose an order on separate sequences of instructions. An example for a memory barrier is `sfence` on x86 architectures. While this approach solves the initial problem, it has a notable drawback. Memory barriers defeat the purpose of out-of-order execution. As a result, CPU pipelines are likely to stall, hence reducing resource utilization. Furthermore, store buffers are flushed leading to higher latencies when accessing data of deferred store operations. Therefore, barriers can have a significant impact on runtime performance, unless used judiciously. With *epoch barriers* a similar approach has been proposed to address both order and durability issues [CNF$^+$09].

Deferred Write-Back

In many modern processor architectures store operations may not immediately lead to an update in main memory. This behavior can be caused by intermediary buffers such as memory order buffers, caches, and memory controller buffers. While their individual purpose may vary, they all defer memory write-back operations. This is a known vulnerability for consistency in NVRAM as the mentioned buffers are volatile and deferred stores may be lost when power fails [CNF$^+$09, OL17]. In order to preserve consistency, it is necessary to force write-back in all of these cases. Figure 2.2 shows a typical memory hierarchy with several layers by which stores can be deferred.

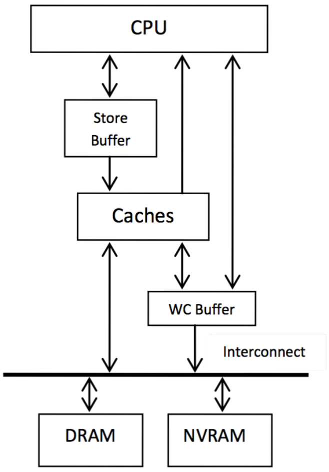


Figure 2.2: Architecture of memory subsystem [BCB12].

Memory Order Buffers (MOBs) In conjunction with instruction scheduling and cache coherency protocols a memory order buffer may be present. It holds all loads and stores, with the exception of non-temporal operations. In order to prevent a deferred write-back through MOBs, their store buffers must be flushed. On x86 architectures this can be achieved with a store fence operation such as `sfence`. As mentioned above, memory barriers carry a considerable overhead. However, if memory barriers are already used for enforcing program order, then flushing store buffers is a desirable side effect and incurs no overhead.

Caches Processor caches help avoid access latencies and reduce memory bus traffic for frequently used data. A possible exception are non-temporal stores and data chunks marked as uncacheable. Similar to MOBs, caches are volatile so an abrupt power failure may lead to lost updates. The issue with this is not that updates are lost but that it is unclear which updates are lost, if any. The reason for this circumstance is the cache eviction policy trying to compensate for typically narrow cache volume. Depending on policy, cache content, and system load, a modified chunk may or may not be flushed to main memory. An application scenario where such behavior is unacceptable is transactions. In the example in Figure 2.3, two stores are cached. One store becomes durable because its cache line is evicted, while the other remains in cache and is lost in a crash.

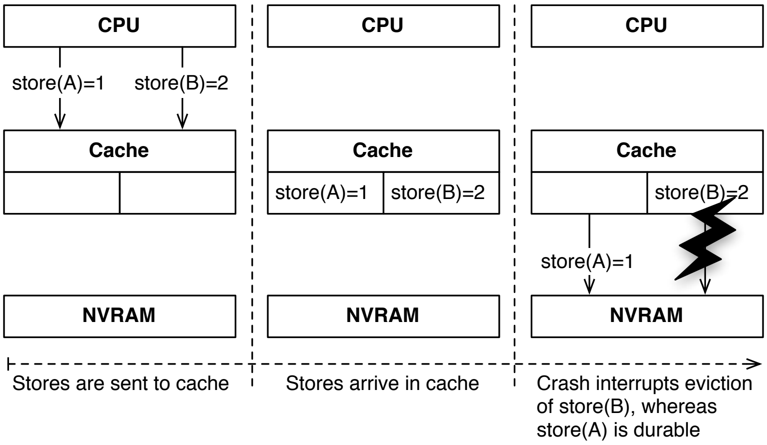


Figure 2.3: A sitation where only one of two cached stores reaches durability.

An approach to prevent such inconsistencies is to disable caching for selected memory regions but that could introduce considerable overhead for frequently used data. A more popular approach is to evict cache lines programmatically whenever necessary [CNF+09, DKK+14, OL17]. On x86 architectures this can be done with the `clflush` or `clflushopt` instructions. However, the problem with a cache line flush is that wanting to make a cache line durable does not always mean that is should be evicted. Therefore, there are proposals for instructions, such as `clwb`, that send a cache line to a memory controller without evicting it [KPS+16, OL17].

Write-Pending Queues (WPQs) Once a cache line is flushed, it is propagated to a memory controller where it is buffered in a WPQ. Again, the problem is that such a buffer is usually volatile. This means that a power failure could lead to lost updates to NVRAM. Even though residual power in DRAM has been shown to be substantial, there is no reliable way to ensure a full WPQ flush [HSH+08]. This circumstance has given rise to many discussions in the past [CNF+09, DKK+14, KPS+16]. Some authors proposed a designated instruction for flushing WPQs. An example is the meanwhile deprecated `pcommit` instruction (formerly known as `pm_wbarrier`) [DKK+14, OLK+15, NHH+17]. Others have developed more general mechanisms for preserving consistency in NVRAM that also address this issue [CNF+09, PCW14]. The dotted rectangle in Figure 2.4 indicates the domain in the memory subsystem that must be protected from power failures.

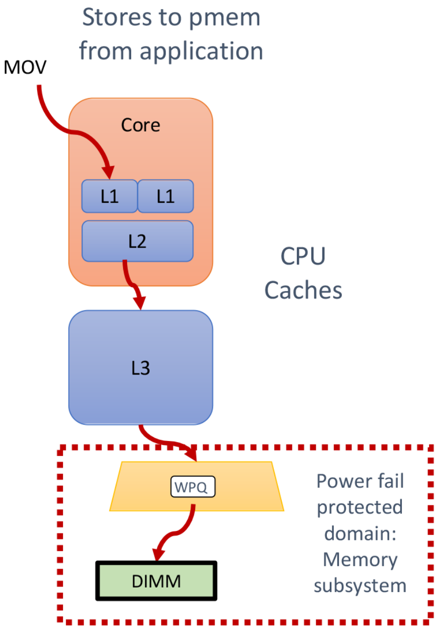

Stores to pmem from application
MOV
Core
L1
L1
L2
CPU Caches
L3
WPQ
DIMM
Power fail protected domain: Memory subsystem

Figure 2.4: The memory domain to be protected against power failures [Rud17].

The current state of affairs is that platforms must provide a feature called Asynchronous DRAM Self-Refresh (ADR) [NHH+17, Rud17]. It works by exploiting the fact that, even in case of a power failure, there is sufficient time and power to flush WPQs in all memory controllers. When the system's power supply unit detects a power failure, it signals all memory controllers to flush their WPQs. As a result, the programmer does not need to worry about WPQs and no overhead is incurred.

2.4 Summary

NVRAM is a promising technology, especially in storage-bound environments. Until recently, both latencies and capacities had fallen short of expectations. However, recent advances in the manufacture of NVRAM have shown promising results. Technologies such as PCM provide larger capacities than DRAM at slower but comparable latencies.

Apart from other more intricate use cases, recent research shows that MMDB can benefit greatly from NVRAM. While access latencies are comparable to conventional MMDB, NVRAM largely eliminates the need to ensure recoverability on slower mass storage. This enables higher transaction throughput and near-instantaneous restarts.

Still, there are also challenges to be addressed. First, current NVRAM technologies tend to have either high latency, low capacity, or low endurance. Even though PCM is a promising candidate, it is still slower than DRAM, especially when writing. Hence, manufacture still needs significant improvement. Furthermore, there is, as of this writing, no generally accepted programming model for NVRAM. In general, there is no way for software to determine whether a store has arrived in NVRAM or not. Since CPUs are unaware of any transactional semantics between stores, programmers need to manually ensure consistency for data in NVRAM. Otherwise, a torn write from a crash may lead to irreversible data corruption. The counter measures presented in this chapter must be applied judiciously as they may incur significant runtime penalties.

3 Key-Value Stores

A prominent use case for NVRAM are main-memory databases. Even though main memory and processor caches have become more affordable, MMDBs still suffer from recovery on slower disk drives [OLK$^+$15, SBD$^+$16]. NVRAM on the other hand, provides an opportunity to eliminate recovery altogether. An important class of databases often implemented as MMDB is KVS. Due to their simplicity and low overhead, KVSs have been adopted both in big-data computing and database research [DHJ$^+$07, LM10, WZT$^+$15]. In recent works, KVSs were used to explore database design for NVRAM [BHC$^+$13, ZSLH16, WNZ$^+$16].

This chapter provides a domain analysis of KVSs. First, a brief overview of KVSs is given. The aim of this work is to exploit NVRAM for a KVS with fast conflict-free concurrent transactions. Therefore, a substantial part of the remaining chapter is dedicated to transactions and concurrency control. The chapter is concluded with an examination of existing KVSs for NVRAM.

3.1 Overview

KVS form an integral part in modern database technology [FFP16]. This section gives an overview of their properties, classes, and applications. Compared to other types of databases, KVSs are very simple databases that are sometimes better described by what they are not or do not provide:

- non-relational data model
- no data schemas
- no query languages

In general, a KVS consists of a single associative container, where each key is mapped to exactly one value. A key is an arbitrary string with possible restrictions on its length. In terms of relational databases, KVSs comprise a single table of two columns. As a result, much of the structural complexity adherent to relational Database Management System (DBMS) is omitted, thus making way for profound optimization and better response times. Common data structures for associativity in KVS are hash tables and search trees, in particular B-trees.

Unlike traditional databases, KVSs do not impose data schemas. Consequently, arbitrary chunks of data can be stored as values which is especially useful in scenarios with no fixed data format or when enforcing one is not a priority. Furthermore, KVSs do not provide query languages such as SQL to store and retrieve data. Instead, KVSs are accessed programmatically through a concise set of operations which is why KVSs are also referred to as *embedded* databases. Although their Application Programming Interface (API) is not standardized, it can be essentially broken down to the following operations:

- insertion
- retrieval
- removal

Applications Traditional DBMSs are often based on complex architectures featuring query front ends and sophisticated storage mechanisms. While this works well in many cases, it severely limits the performance in situations where a simpler storage paradigm (e.g. key-value pairs) is sufficient. As a consequence, high access latencies and convoluted, error-prone concurrency schemes inhibit the scalability of storage systems. KVSs on the other hand are designed to compensate for these shortcomings. A driving force in this regard, are large internet platforms, e-commerce for instance, and cloud computing services.

A longstanding example of a KVS is BerkeleyDB which acts as a database in a variety of software solutions. Apart from open-source software such as OpenLDAP or Apache Web server, BerkeleyDB is also used in a number of proprietary software such as messaging servers, switches, and routers [Käs07, OBS99].

A more recent use case are distributed in-memory caches often found in big-data environments. Web caches have received great attention as service providers struggle to scale with rising traffic where many requests target only a small amount of data [XFJP14]. With caching, a dedicated eviction policy ensures that *relevant* items reside in memory. As a result, caching can improve response times significantly. For this purpose, KVSs provide an appropriate abstraction. Important representatives of this class are Redis and memcached [Lab, Dor].

Not only have these KVSs been deployed at companies such as Facebook or Twitter, but they have also formed the basis for considerable amounts of research in this area [XFJP14]. Examples include Field-Programmable Gate Array (FPGA) acceleration [LAC14], memory partitioning for better cache hit rates [CM14], and NVRAM integration [WNZ+16, Mal17, VTR+11]. Still, large companies tend to maintain in-house solutions to suit their needs [CDG+08, DHJ+07, LM10, WZT+15].

Beyond databases and caches, KVSs have also been proposed as a basis for file systems. In the past, there have been several attempts to integrate database concepts into file systems, some of which are logging [RO92, Twe98] and transactions [SS90, WSSZ07, SGC+09]. Some studies even suggest that traditional hierarchical file systems may often be suboptimal [Ste05, SM09]. While databases in general are still considered too heavy-weight for use in file systems [SM09], KVSs may be a viable alternative. Examples include the network file system DBFS which is based on BerkeleyDB [MTV02] and FlatFS, a simple file system for NVRAM [VNP+14]. In addition, KVSs are also used to complement file systems, for example, to store metadata as in PVFS [CLR+09]. Still, the predominant use case of KVS is found in light-weight databases and caches on top of existing file systems.

Transactions An essential feature of most databases are transactions. Transactions enable a sequence of database operations to appear as a single atomic operation. If a single operation involved in a transaction fails, the entire transaction fails and its side effects are rolled back. Transactions are a powerful mechanism that enables aggregated operations without worrying about inconsistencies even in case of failure. Given the prevalence of transactions, most KVSs support them. A notable exception is memcached [WNZ+16]. Due to their importance for this work, transactions are covered in more detail in the next section.

In-Memory Operation The performance of a database is often denoted in terms of transaction throughput. One way to increase throughput is to mitigate data access latencies. Apart from faster storage, this can be done by placing the entire database in main memory which enables speedups by multiple orders of magnitude. This approach, which dates back to the mid 1980s, has been adopted in many high-performance databases such as the more recent HANA database [GmS92, FCP+12]. Likewise, most KVSs are explicitly designed for in-memory operation. Notable exceptions are the popular BerkeleyDB or Apache's Cassandra where in-memory operation is only an option [Oraa, LM10].

Concurrency Another approach to increase transaction throughput is to utilize multi-core processors by executing transactions concurrently. In order to achieve maximum performance, it is common for main-memory databases to also support concurrency [GKP$^+$10, FCP$^+$12, DFI$^+$13]. Further, it has been shown that KVS can gain substantial performance benefits through concurrency [FAK13, LLL$^+$15, XFJ14]. In fact, most KVSs natively support concurrency with the exception of Redis [Lab]. Unfortunately, concurrency also introduces new issues such as inconsistencies through race conditions on shared data. Mitigating this issue can degrade performance which is why many designs trade full consistency against faster relaxations [DHJ$^+$07]. This issue is dealt with in the next section about transactions.

Distributed Databases As mentioned earlier, KVSs play a crucial role in big-data environments. Since availability is often a requirement in this area, KVSs are often implemented as distributed services [DHJ$^+$07, LM10, WZT$^+$15]. Distributed databases and their mechanisms such as distributed transactions are beyond the scope of this work.

3.2 Transactions

Transactions are a powerful concept that has been adopted in various branches of computer science. Examples include databases, transactional memory, and operating systems. With transactions, multiple operations, such as reading or updating a record, can be grouped into a single unit that succeeds if and only if neither of its operations fails. Especially in high-performance computing environments, the utilization of computing resources through concurrent transactions plays an essential role. This section describes the concept of transactions and its properties with regard to concurrency, in particular.

Definition A transaction is a sequence of operations that is treated as a single atomic operation, i.e. it either succeeds if all its operations succeed or it fails. In general, an incomplete or failed transaction must not have any observable side effects. A transaction *commits* when all its subordinate operations have completed. Once this process is complete, the transaction is *committed* and all its side effects, if any, become visible.

In general, the concept of a transaction does not impose any restrictions on the kind of operation enclosed inside a transaction. That is, apart from primitive operations such as read or update, transactions may also consist of inner transactions as well. This concept is known as *nested* transactions [G+81]. In contrast, *flat* transactions only permit primitive operations.

Despite their general nature, nested transactions are not a subject of discussion in this thesis, for a couple of reasons. First, nesting has been found useful primarily for distributed transaction systems and transactional programming models neither of which are within the scope of this work [Mos81, Mos06]. In addition to implementation issues, there are several semantic models for nesting which further complicates its discussion [HR93, WS92]. In the end, nesting has not found wide adoption with the prominent exception of transactional memory [MH06, MBM+06, JSG12] and a few databases [OBS99]. Hence, unless stated otherwise, the term transaction always refers to flat transactions in this thesis.

Especially when discussing concurrent transactions, a meaningful notation is required to describe their interactions. For this purpose, the concept of a *schedule* is used. A schedule is a list of operations enclosed in one or more transactions. Within a schedule, all operations appear in the same order they are executed. Although there seems to be no standard notation, the operations of a schedule typically comprise abstract operations for reading and writing a record, as well as the basic transactional primitives of commit and abort. A transaction is implicitly started by its first operation, so there is no need for an explicit primitive. In order to make schedules more readable, operations are denoted by shorthands as shown in Table 3.1.

Notation	**Operation**
r(A)	Read a record A
w(A)	Write a record A
c	Commit
a	Abort

Table 3.1: Shorthands for transactional operations in schedule notations.

There are several ways to print a schedule. A common method is to render a linear list containing shorthands of indexed operations where the indices denote the associated transaction, respectively (see Figure 3.1). Schedules are read from left to right.

$$r_1(A)\ r_2(A)\ w_2(A)\ c_2\ w_1(A)\ c_1$$

Figure 3.1: Interleaved notation of a write-write conflict.

For the better readability, this work chooses to list operations for each transaction individually, while retaining their global chronological order (see Figure 3.2).

$$\begin{array}{lllll} t_1 : & r(A) & & & w(A)\ c \\ t_2 : & & r(A) & w(A)\ c & \end{array}$$

Figure 3.2: Projected version of Figure 3.1.

Applications Transactions are useful when a series of operations must either execute in conjunction or not at all. A simple example is the transfer between bank accounts. The action of withdrawing a value from one account and depositing it on another comprises two separate actions that must both be successful in order to take effect.

The predominant domain of transactions is databases where they are used to achieve consistent and reliable data access. However, there were also attempts to establish transactional semantics as an operating system feature [PHR$^+$09, Spi09, Bla91]. This way, subsequent system calls could be executed as a unit and be undone if one of them fails. Another application that sees increasing attention is transactional memory which aims to provide a synchronization alternative to locking [Kni86, HM93, ST97]. With regard to this thesis, recent use cases include transactions in a MMDB [LKN14] and durable updates to NVRAM [VTS11]. Despite being an intriguing concept, transactional memory is beyond the scope of this work. Instead this work sets its focus on plain software-based transactions in the field of databases and KVSs, in particular.

Transactional Semantics The previous definition of transactions was of rather intuitive nature. However, in order to be useful, the semantics of a transaction need to be described more precisely. The predominant characterization of transactional semantics is ACID [G$^+$81, HR83]. It comprises a set of necessary properties:

- atomicity
- consistency
- isolation
- durability

Atomicity captures the all-or-nothing notion of a transaction, i.e. either all operations in its context succeed or none. As a consequence, any already completed operation of a transaction must be undone should the latter fail. Reverting the affected data to their previous state is often referred to as *rollback*.

The property of *consistency* asserts that if the underlying data is in a consistent state, then any transaction must preserve consistency. For example, an ACID-compliant database cannot be transitioned into an illegal state by means of a transaction. If a transaction is bound to break the consistency of the database, then it has to be aborted and rolled back.

In case multiple transactions are executed concurrently, each transaction could observe intermediate side effects of other concurrent transactions. In order to prevent this scenario and ensure the consistency property, *isolation* precludes transactions from seeing any concurrent activity. The property of isolation is later dealt with in more detail.

The last of the four ACID properties is *durability*. It ensures that all side effects incurred by a committed transaction must be durable across any subsequent system failure. Durability can be very hard to enforce, especially in the face of catastrophic failures with failing backup media. Therefore its notion is often relaxed to a reasonable extent.

The ACID criteria have become the prevalent reference for characterizing transactional systems. However, not all systems enforce the complete set of properties. Notable exceptions are transactional memory for conventional RAM and some cache-like databases. In these cases durability is not required as they are volatile by design. A prominent example for relaxation that is also fundamental to this thesis is the isolation of concurrent transactions.

Serial Transactions In a serial transaction-processing system all transactions are executed in a sequential order. That is, only one transaction, if any, is being executed at a time and overlapping is not possible. If a transaction t_1 attempts to start while another transaction t_0 is still active then t_1 has to wait until t_0 terminates.

This reveals two important drawbacks. First, transaction throughput does not scale as the number of overlapping transaction requests increases. Second, with only one computing resource active at a time, resource utilization is low on multi-core architectures. The same is true on single-core systems as execution latencies cannot be hidden by switching to other transactions. To mitigate these issues, transactions can be allowed to run concurrently.

Concurrent Transactions Concurrent transaction execution can largely remove the shortcomings outlined above. Now, an incoming transaction does not need to wait for an in-flight transaction to complete. In addition to increasing transaction throughput, it also enables better resource utilization. This works as long as data is read but not written. Allowing concurrent updates however, bears potential conflicts that threaten the consistency of data and must therefore be addressed. Possible conflicts are:

- write-write
- write-read
- read-write

When a transaction t_1 attempts to update a record A that was previously written but not committed by another transaction t_2 then t_1's update could overwrite t_2's update to A before it has become visible. Unaware of the condition t_2 will successfully commit even though its update is lost (see Figure 3.3). This situation is called a *write-write* conflict.

$$\begin{array}{lllll} t_1: & & w(A) & c & \\ t_2: & w(A) & & & c \end{array}$$

Figure 3.3: A write-write conflict resulting in a lost update for t_2.

A *write-read* conflict occurs when a transaction reads data that has not been committed yet. Imagine a transaction t_1 that reads a record A that was previously updated but not committed by another transaction t_2. If t_2 updates the same item again or fails, then t_1 has processed a value that was never committed. This situation is also called *dirty read.* For an example see Figure 3.4.

$$\begin{array}{lllll} t_1: & & r(A) & w(B) & c \\ t_2: & w(A) & & a & \end{array}$$

Figure 3.4: A write-read conflict resulting in an erroneous update for t_1.

The last conflict is called *read-write* conflict and denotes a situation when a transaction updates a record that was previously read by another transaction that is still running. Consider two transactions t_1, t_2 where t_1 reads a record A which is later updated by t_2 before either transaction commits. If t_1 reads A again, then the result may be inconsistent with the earlier read as shown in Figure 3.5. The situation is also referred to as *inconsistent read* or *non-repeatable read.*

$$\begin{array}{lllll} t_1: & r(A) & & & r(A) \\ t_2: & & w(A) & c & \end{array}$$

Figure 3.5: A read-write conflict resulting in an inconsistent read for t_1.

Note that, since isolation must be guaranteed for the entire lifetime of a transaction, conflicts are not precluded by protecting individual read or update operations. Instead, a dedicated synchronization mechanism for transactions is needed. An important formalism in this regard is serializability.

Serializability A core concept to preserve consistency in the presence of concurrent transactions is *serializability.* It is based on the observation that in an ACID-compliant serial transaction processing system, every sequence of transactions always yields consistent data. Likewise, a schedule of concurrent transactions should yield consistent data if it behaves in a way that is equivalent to a serial sequence of the same transactions.

More precisely, a concurrent schedule is called *serializable* if and only if there exists a serial schedule of the same transactions that produces the same output. A transaction processing system provides serializability if and only if it guarantees that all its concurrent schedules are serializable.

In order to enforce serializability, a decidable classification for serial transaction schedules is required. An early definition of serializability appeared in ANSI SQL [BBG$^+$95, MS93]. The idea is to identify and detect *anomalies* of non-serializable schedules at runtime. If an anomaly is detected then any affected transaction must fail. Based on whether these anomalies were permitted, several *isolation levels* were defined. In terms of ANSI SQL, a transaction is serializable if none of the following anomalies was present:

- dirty read
- non-repeatable read
- phantom

A *phantom* is similar to a non-repeatable read but differs in that the item in question is not modified but added or removed. Imagine a transaction t_1 making a conditional selection of items. If another transaction t_2 adds an item and t_1 repeats its selection then the result may contain the item which is inconsistent with the first result.

Note that this formalization is built around observable artifacts of non-serializable schedules, rather than their cause such as read-write conflicts. While it is pragmatic to address only observable anomalies it is also unreliable as more complicated consistencies may remain undetected. In fact, it was later found that the above characterization is insufficient as further anomalies were discovered [BBG$^+$95, FOO04]. In the wake of these findings, additional restrictions were imposed on the notion of serializability.

Nevertheless, all of the discovered anomalies can be attributed to the access conflicts shown above. For example, the most recently discovered anomalies, *write skew* and *non-serializable read-only* are essentially results of read-write conflicts. In this sense, a transaction is serializable if and only if all possible conflicts are precluded. This characterization has several advantages. Most importantly, it is more plausible to discuss non-serializable schedules in terms of causes instead of effects. As opposed to anomalies, the number of conflict scenarios is smaller and also fixed. Therefore, this thesis primarily defines serializability in terms of conflicts.

3.3 Concurrency Control Protocols

Concurrency is a major building block for scalable transaction processing. It enables higher transaction throughput and resource utilization compared to serial processing. On the downside, concurrent schedules are subject to potential conflicts that may result in data corruption. However, it is not sufficient to provide mutual exclusion for individual operations within a transaction. In other words, the scope in which isolation is required spans beyond critical sections. Therefore, a dedicated concurrency control is required to ensure isolation. Unfortunately, concurrency controls do not come without overhead which is why, in most cases, a compromise between isolation and performance must be found. This section deals with concurrency control strategies and outlines the state of the art with a focus on optimistic approaches, in particular multiversion concurrency control.

3.3.1 Strategies

There are two fundamental approaches to the design of concurrency controls: *pessimistic* and *optimistic* protocols [KR81, LBD$^+$11, SCB$^+$14]. The distinction is based on whether conflicts are assumed to be frequent or infrequent. Still, both strategies share their intent to prevent conflicts from manifesting into inconsistencies.

Pessimistic Concurrency Control A pessimistic concurrency control assumes that conflicts are frequent and strives to prevent conflicts before they can even emerge. To this end, pessimistic control mechanisms employ some form of exclusive ownership. That means, a transaction must acquire the ownership of all data items it wishes to access. If the resource acquisition succeeds, then the transaction can freely operate on the temporarily owned data. Only when that transaction terminates, will this ownership be released. If a transaction fails to claim the exclusive ownership on its data then it has to wait until the required ownership is granted or abort. Pessimistic concurrency control is usually implemented with locks. Locking provides a solid mechanism to ensure serializability and most database systems implement it [KR81, BBG+95, LBD+11].

Despite their prevalence, pessimistic concurrency controls have notable drawbacks. Locking-based concurrency controls are prone to *deadlocks* [BG81, KR81]. In order to prevent deadlocks, they must be detected and resolved which introduces runtime overhead. Another problem is *lock contention* which occurs when a large portion of concurrent threads in a system compete for a single shared resource [BBG+95, SCB+14]. Since only one transaction can manage to acquire ownership, all remaining transactions are left waiting for it to complete and contend again. As a result, only one transaction is executed at a time, thus defeating the purpose of concurrency.

Optimistic Concurrency Control Optimistic concurrency controls form the opposite of pessimistic control schemes. Instead of preventing conflicts altogether, optimistic control schemes do not enforce consistency until a transaction commits. Only when a transaction commits, the concurrency control starts to check for violations, a step called *validation*. If no conflict is detected, then the transaction may commit, otherwise, it must abort. Validation is crucial for consistency and its implementation substantially determines the achievable isolation level [LBD+11].

Optimistic concurrency control protocols rely on Copy-On-Write (COW) and timestamp ordering to synchronize data races of competing transactions [BG81, KR81]. While readers can access data without further means of synchronization, writers apply their modifications only on copies of the original data. Apart from short-duration locks for critical sections, this approach works without locking data for the duration of an accessing transaction. As a result, readers do not block other readers or writers. Depending on the type of concurrency control, however, concurrent writers may need to operate under mutual exclusion. This issue is addressed later on. An important class of optimistic concurrency controls is multiversion concurrency control or multiversioning [Ree78, BG83].

There are, however, disadvantages to this type of concurrency control. While validation is necessary to ensure that a transaction is not involved in data conflicts, it can also introduce a significant overhead. First, even when there are no conflicts, validation is conducted nevertheless. Second, validation usually requires certain metadata about the operations within a transaction. As a result, validation complexity scales in size of metadata required to determine conflict freedom. Another drawback is that aborting a conflicting transaction means that its entire progress is discarded. In this case, valuable computing resources are wasted.

Despite some drawbacks, optimistic concurrency control has found wide adoption especially in domains where reads are dominant and conflicts are known to be infrequent or negligible [CM86, LBD+11, WAL+17]. The scenario of read-dominated workloads has been shown to apply more often than not [ALR+17, WJFP17]. Given its promising properties, optimistic concurrency control is discussed in more depth in the subsequent sections.

3.3.2 Multiversion Concurrency Control

Multiversion Concurrency Control (MVCC) or multiversioning, is an optimistic concurrency control method. Initially, MVCC was designed as a solution for concurrency control in distributed systems [Ree78]. However, it was also studied in non-distributed settings and was soon considered a promising alternative to locking [KR81, BG83, Car83, HP86, CM86]. Subsequently, MVCC has been adopted in both commercial and non-commercial transaction processing systems, ranging from general-purpose database systems to high-performance in-memory databases [LBD+11, LMM+13, DFI+13, SFW+15]. More recent examples include prototypes of MMDB and KVS for NVRAM [BHC+13, ZSLH16, OBL+14, SBD+16].

Principle

A *version* is a snapshot of a particular data item within a database. In terms of KVS, that would be a value. In traditional concurrency schemes, there is exactly one version of each item. These are also referred to as *single-version concurrency controls.* If a transaction issues an update to a version, then it is performed in-place. In order to ensure isolation, a transaction has to be protected against concurrent reading or writing.

In a *multiversion concurrency control*, there can be multiple versions of data items. This fundamentally changes the nature of both read and write operations. Instead of updating versions in-place, which would have to be isolated, write operations create copies of existing versions and only modify those copies. As a result, read operations are implicitly decoupled from concurrent updates. That means, in particular, that a read operation may access an item even when newer versions have been committed. In order to keep track of when versions were created or modified, versions are usually equipped with timestamps, respectively.

Through its COW approach, multiversioning can effectively isolate read operations from concurrent operations without the need for locking. An important implication of this circumstance is that read operations never wait for write operations and vice versa. This is a significant advantage over single-version schemes especially in applications where reading is much more frequent than writing. In fact, it has been shown that many workloads are dominated by queries [KKG+11, ALR+17]. This is also reflected in the MMDB benchmark TATP which assumes 80 % of all transactions to be read-only [LBD+11]. Another useful side effect of multiversioning is that it forms an implicit logging infrastructure which can be used for recovery purposes [CNF+09, VTR+11]. MVCC is similar to Read-Copy-Update (RCU) [MS98] but differs in that it is more general and can manage more than two versions at a time.

While read operations benefit from COW, updates can be expensive. Updating a version incorporates additional overhead for allocating a new version and copying the original version before modifying it. In the case of KVS, however, copies may not be necessary if entire values are updated.

Visibility

A central aspect of multiversioning is the concept of *visibility*. Since there may be multiple versions of a single data item, an operation must first figure out to which version it should apply. For this purpose, a visibility property is defined. It determines which versions can be accessed by the operations of a transaction. In other words, a version is *visible* to a transaction and its operations if and only if it satisfies the visibility property. In general, visibility is defined as a predicate over timestamps of transactions and versions. The concrete definition of the predicate is subject to the respective MVCC protocol.

When a transaction attempts to access an item, it would typically traverse the versions of that item and determine for each version whether it is visible to the transaction. Only if a version is visible to the issuing transaction, it can be selected for reading or writing. The implementation of visibility is of paramount importance for MVCC protocols and the desired isolation level [LBD+11].

Challenges

The most promising feature of multiversioning is the non-blocking nature of read operations due to the absence of in-place updates. However, there are also important problems that need to be addressed in order to leverage the merits of multiversion concurrency control.

First, the maintenance of more than one version per item implies a significant overhead in storage. Note that a version may not only contain payload but additional data such as pointers to adjacent versions. A version may also be required to hold certain metadata such as timestamps, further increasing the overall memory footprint. This is relevant especially in areas where memory is comparatively scarce as is in main-memory databases. However, not all versions need to be retained. Instead, only the versions that are visible to at least one transaction are needed. All other versions are considered *stale* and can be disposed of. This task is usually achieved by a designated garbage collection mechanism. Although garbage collection may improve the overall memory footprint, it is also known to have adverse effects on performance.

Second, whenever an item is accessed, the system first has to find a visible version of the item. Accessing an item may therefore incur a significant runtime overhead. The overhead mainly depends on the size of the history which, in theory, is only bounded by the amount of available memory. Employing a garbage collection mechanism can help by reducing the size of histories. Another optimization would be to have a transaction keep track of all the versions it references. This way, visibility would only have to be computed once for each item.

3.3.3 Snapshot Isolation

The most widely used MVCC protocol to date is Snapshot Isolation (SI) [LBD+11, NMK15]. Originally, SI was developed as a response to the insufficient definition of serializability in the SQL standard [BBG+95]. Since then, it has been deployed in numerous databases and KVSs [CRF09, WAL+17]. In the context of isolation levels, SI defines a new isolation level that goes by the same name. Although SI is weaker than serializability, it provides a good trade-off between performance and consistency. This section describes the concept of SI and its properties.

The core principle of SI is that a transaction t only sees a private snapshot of the database as of when t started. In this sense, the notion of a snapshot comprises the set of the latest versions that have been committed before t was invoked. The key to this behavior is the definition of the visibility property.

Visibility

Each version v stores two timestamps $begin_v$ and end_v, denoting when v was created and when it was invalidated by an update or deletion, respectively. The interval $[begin_v, end_v]$ is called the *lifetime* of v. Also, when a transaction t starts, it is given a timestamp $begin_t$ to capture when t started. In order to determine which version is visible to the operation of a transaction, the concurrency control needs to test for each version whether t started during the lifetime of v. The latest valid version satisfying this property is selected to be visible by the requesting transaction. More precisely, the version seen by a transaction t is

$$\max_{i \in \mathbb{N}} \{\, v_i \mid begin_{v_i} < begin_t < end_{v_i} \}.$$

Conflict Handling

Concurrent updates by a transaction t_2 that happen after a transaction t_1 started, are not included in the snapshot of t_1 and are therefore invisible to t_1. If however, t_1 decides to also update the same data item, then a write-write conflict emerges. In this case, the *first-committer-wins* principle is applied and t_1 must abort as t_2 also modified the same item and committed earlier [BBG+95]. A popular variant of this property is the equivalent *first-updater-wins* principle [FOO04, LBD+11]. According to this property, a writer fails immediately if he is not the first to attempt an update on a given version, thus making the earlier transaction the first committer. Note, that based on this strategy, SI can be implemented without validation on commit. It can also reduce the size of individual rollbacks as write-write conflicts are detected immediately.

Shortcomings

Under SI reads can always be satisfied, provided the requested item exists. Note that even in the face of concurrent updates, a transaction under SI always sees the same items. This precludes inconsistent reads, read skew, and phantoms. In addition, SI does not allow dirty reads, since snapshots only contain committed data. However, SI does not prevent all possible anomalies and is therefore not serializable [BBG$^+$95, FOO04]. In particular, these conflicts are write skew and non-serializable read-only transactions.

Write Skew The earliest known anomaly of SI is write skew. The reason for its occurrence is that under SI a transaction does not see modifications to versions that have been read during the transaction.

Imagine two transactions t_1, t_2 reading two data items x, y constrained by a predicate C. Next, t_1 updates x and finds that $C(x^*, y)$ still holds true. At this point t_2 is unaware that x has been modified and updates y. Since t_2 does not see the modifications of t_1, it also evaluates $C(x, y^*)$ to be true. Finally, both transactions may commit even if C is now violated because none observed the others changes (see Figure 3.6). Note that no write-write conflict occurs as both updated items are distinct. In fact, write skew is said to occur if read sets overlap while write sets are distinct.

$$\begin{array}{llllll}
t_1: & r(x,y) & & w(x) & & c \\
t_2: & & r(x,y) & & w(y) & c
\end{array}$$

Figure 3.6: Write skew due to transactions t_1, t_2 not seeing each others changes.

In the field, write skew has been countered by inducing artificial write conflicts between transactions that are expected to exhibit write skew [FLO$^+$05].

Non-Serializable Read-Only Anomaly Another anomaly was discovered almost 10 years after the introduction of SI. It proved, contrary to common understanding, that even read-only transactions may not always be serializable [FOO04]. The proof consists of a schedule of three transactions with one being read-only. The schedule is constructed in a way that only two but never all three transactions can execute without a conflict.

Suppose a pair of data items $x = 0$ and $y = 0$ and transactions t_1, t_2, and a read-only transaction t_{RO}. Further, let t_1 compute $y = y - 10$ and also subtract one if $x + y < 0$, while

t_2 sets $x = x + 20$. The schedule given in Figure 3.7 shows that while t_1 is the first transaction to start execution, both t_2 and t_{RO} start and commit sequentially before t_1 issues its update on y. This means that t_{RO} will see the update of x by t_2 while t_1 does not. According to the output of t_{RO} ($x = 20$, $y = 0$), a serializable schedule would require t_1 to have been executed after both t_2 and t_{RO}. This however, is not possible since t_1 would have seen the update of t_2 and no penalty would have been applied as $20 - 10 \geq 0$. Likewise, in order for t_1 to yield $y = -11$, it would have had to be executed before t_2 (and t_{RO}) which, however, is not consistent with the output of t_{RO}. In fact, the output of t_{RO} corresponds the exact opposite serial ordering as do those of t_1 and t_2.

$$\begin{array}{rllll}
t_1 : & r(x,y) & & & w(y)\,c \\
t_2 : & & r(x)\,w(x)\,c & & \\
t_{RO} : & & & r(x,y)\,c &
\end{array}$$

Figure 3.7: Transaction t_{RO} is read-only but not serializable.

Both symptoms can be attributed to the fact, that SI fails to observe read-write conflicts. When a transaction requests an item, it reads the latest version that has been committed before the transaction started. This way, a transaction always reads the same version even if a newer version has been committed concurrently. This relieves the system from locking a version when accessing it. The downside is that every transaction is effectively isolated from any concurrent modifications. As a consequence, transactions may successfully commit even if one or more versions they have read has been updated in the meantime. All anomalies known to be emitted by SI can be reduced to read-write conflicts.

3.3.4 Serializable MVCC

Snapshot Isolation provides affordable isolation but fails to preclude all non-serializable schedules such as write skew. Nevertheless, SI is the most widely adopted MVCC implementation [LBD$^+$11, BHC$^+$13, NMK15]. In some cases, serializing alternatives are available but disabled by default. This policy is often motivated by significant performance benefits compared to strictly serializing concurrency control [CRF09]. Others argue that SI anomalies may be negligible as even renowned ACID-compliant benchmarks such as TPC-C do not expose them [FLO$^+$05]. However, the need for data integrity should not depend on benchmarks failing to prove it. Therefore, there are efforts to overcome the weaknesses of SI and provide strong consistency without falling back to pessimistic approaches. This section presents an overview of approaches to make serializing MVCC affordable.

The main reason for non-serializable schedules in SI is that it cannot detect read-write conflicts. While SI keeps track of each transactions' updates for installment on commit, reads are not tracked. Therefore, a naive approach to achieve serializable schedules with SI is to track the read operations of each transaction. In doing so, read-write conflicts can be detected during validation by looking at the timestamps of all versions read. If at least one of these versions has been invalidated after the transaction started, then a read-write conflict has emerged. Unless the conflicting updater is still running, the reading transaction must abort. This method is both simple and effective but introduces significant overhead. Note that traditional SI does not perform any validation if first-updater-wins is used for precluding write-write conflicts. The additional overhead especially affects read-mostly transactions which is undesirable in read-dominated environments. Since the latter is where SI has been especially successful, tracking reads followed by validation is often stated as prohibitively expensive [CRF09]. Lacking viable alternatives, research interests primarily focus on mitigating the footprint of read tracking and validation.

Exploiting Query Languages

Several authors have proposed to detect conflicts from query language statements [FLO$^+$05, FA15, NMK15]. For instance, a common strategy to prevent write skew in SI is to inject detectable conflicts whenever the required access patterns are detected [FLO$^+$05]. Others have found efficient ways to determine whether an item is included in a range query, thus improving validation time [NMK15]. Although intriguing, these approaches cannot be applied to KVSs since they operate through ad hoc queries instead of query languages.

Reducing Contention

A major challenge for high-performance implementations of MVCC and SI, in particular, is that important aspects such as timestamping or validation are often centralized which can cause considerable contention. Therefore, a substantial amount of research is dedicated to providing protocols in the spirit of SI but lower contention. Note that, in contrast to locking, contention for MVCC denotes much smaller intervals of a transaction's lifetime.

A major bottleneck in MVCC implementations is validation [TZK$^+$13, BHC$^+$13, DKDG15, FA15, WJFP17, ZZYT17]. The reason is that, validation typically requires mutual exclusion since concurrent updates to items from the read set could falsify the validation result.

As a result, validation does not scale, thus inhibiting transaction throughput. Therefore, many authors propose protocols featuring concurrent validation [BHC$^+$13, DKDG15, FA15, WJFP17]. Parallel validation has already been proposed in the early days of MVCC [KR81], but received renewed attention lately.

Another point of contention is the assignment of timestamps. A typical SI implementation requires multiple timestamps for both versions and transactions. However, most implementations rely on a global assignment policy which introduces a significant contention on multi-core systems [TZK$^+$13, ZZYT17]. First, concurrency is reduced as mutual exclusion is required to guarantee strictly monotone timestamps. Second, the CPU must be informed that changes to the cached timestamp counter must be globally visible. This is usually done with fences which can further reduce performance. In response, some authors have proposed protocols featuring decentralized timestamp assignment [TZK$^+$13, ZZYT17]. Further sources of contention addressed in these works are transaction id assignment and shared memory access, in general.

3.4 Key-Value Stores for NVRAM

The previous sections describe the fundamentals on transactions, serializability, and modern concurrency control protocols. Since the aim of this thesis is to design and implement a concurrent KVS for NVRAM, this chapter concludes with an overview of existing KVSs for NVRAM.

NVRAM is not yet commercially available, but there is a number of studies involving KVS and NVRAM. In essence, there are three research branches:

- evaluation of programming facilities for NVRAM
- evaluation of implications of NVRAM for existing KVSs
- design of NVRAM-aware KVS

While, ultimately, all branches aim to understand the implications of NVRAM, individual scenarios and approaches differ substantially. As for the aim of this work, focus is given to KVS specifically designed for NVRAM.

At the moment, there are only few designs for NVRAM-aware KVS: Echo [BHC$^+$13], NVHT [ZSLH16], and MetraDB [MGA16]. Below, the architecture of these KVSs is outlined with an emphasis on transactions and concurrency control.

Echo

Echo is one of the earliest KVSs designed to leverage the benefits of NVRAM [BHC$^+$13]. It aims to achieve scalable high-performance transactions through optimistic concurrency control and light-weight persistence management. For this purpose, Echo uses a two-level store architecture featuring both volatile and non-volatile RAM. Only committed data is written to NVRAM, while uncommitted data is kept in volatile RAM. Moreover, there are two groups of threads: workers and masters. The former execute the operations of transactions and buffer their updates, whereas masters are in charge of committing them. By keeping updates from NVRAM until commit, persistence guarantees need to be enforced less often and degradation effects such as wearing are reduced. Also, updates are buffered locally in each thread which avoids contention on shared data. In order to ensure isolation between concurrent transactions, Echo employs classic SI. As a result, some non-serializable schedules of transactions are permitted.

Contrary to many other works, Echo resolves write-write conflicts using the original first-committer-wins strategy. This eliminates the need to acquire exclusive ownership when updating an item but leads to late conflict detection with larger rollbacks. The core data structure beneath Echo is a hash table which maps keys to version histories. Concerning NVRAM consistency, Echo settles for existing instruction sets with cache line flushes and store fences. The authors further anticipate a hardware capability, such as the now obligatory ADR, which ensures that cache flushes always become durable. During the evaluation of Echo, its design is shown to provide strong durability and consistency while providing performance characteristics of volatile main-memory KVS. However, the evaluation is carried out on volatile RAM and does account for latencies of NVRAM and cache flushes. Also, the separation of worker and master threads is effectively removed, as workers *become* masters and commit their own transactions.

Being one of the most meticulously designed and documented KVSs for NVRAM, Echo is a guiding example for this thesis. It achieves high performance through its two-level architecture and optimistic concurrency control. Drawbacks include non-serializing SI and the first-committer-wins strategy. In the end, the authors' evaluation is not entirely conclusive as some design aspects were considerably altered.

NVHT

The goal of NVHT is to leverage NVRAM to achieve fast updates without sacrificing durability [ZSLH16]. The architecture of NVHT differs significantly from that of Echo. Most importantly, NVHT only supports transactional updates instead of full-grown transactions. That means, that each operation is implicitly committed, which is sometimes called auto-commit. Nevertheless its architecture may give valuable insights into the design of a KVS for NVRAM. Similar to Echo, NVHT relies on a hybrid memory architecture consisting of both volatile RAM and NVRAM. However, Echo is a two-level store where each update is buffered in volatile RAM until it is committed to NVRAM. In NVHT, on the other hand, all updates are directly written to NVRAM. Similar to Echo, the key data structure of NVHT is an NVRAM-aware hash table. However, NVHT does not use multiversioning to control concurrent operations. Instead, it applies half-coarse synchronization by locking individual partitions of the hash table upon access.

Despite NVHT having shown good performance against prominent KVSs, there are some problems with its design. First, accessing an item in the hash map locks an entire partition. The authors point out that this design decision can increase lock contention, thus reducing concurrency. More importantly, NVHT does not address the issues of ordering and deferred write-back on NVRAM. Instead, the authors merely devise a kind of commit record whose presence or absence denotes whether the preceding item should be taken into account. However, without enforcing an ordering on store operations, the commit record could be durable before the actual item. Also, in order to ensure durability, cache lines need to be flushed. Omitting these precautions could lead to inconsistent data.

MetraDB

In [MGA16] KVSs are proposed as a middleware for NVRAM. As an example, the authors present MetraDB, a solution for distributed storage based on NVRAM. Like NVHT, MetraDB is a single-level store which means that changes are written to NVRAM immediately. Since MetraDB is required to support overlapping namespaces, the KVS consists of multiple hash tables referred to as containers. While this seems to complicate memory allocation schemes for NVRAM, the authors assert that the size of hash tables is fixed. There are two kinds of transaction is MetraDB: transactions on containers and transactions on metadata. Since containers are designed for single-threaded access, transactions on containers need not worry about isolation. In contrast, transactions on metadata can be multi-threaded. These transactions are responsible for adding and removing containers. The authors point out that these operations are infrequent and, therefore, need not be very efficient. For this reason, transactions on metadata are protected by a global lock on the collection of containers. Recovery is based on redo logging, since undo logging would require additional writes to NVRAM during a transaction. With redo logging, log entries only need to be flushed on commit which can be optimized, for example with non-temporal stores. In an evaluation, MetraDB has shown superior performance when compared to several lookup data structures contained in NVML (currently known as PMDK). Given that NVML provides general-purpose NVRAM facilities, as opposed to MetraDB's use case optimizations, the comparison is not always fair. Still, the evaluation exposes scalability issues related to the operating system rather than the KVS itself. Upon finding scalability to falter when increasing the number of CPU cores, the authors link the issue with kernel locking on memory-mapped files.

When compared to Echo or NVHT, MetraDB is a very application-specific KVS. Especially its use of containers, which can only be accessed by one thread at a time, conveys little guidance for the design of a KVS with concurrent transactions. Apart from a few isolated cases, single-threading and very coarse locking may not be the most promising approaches to achieve high transaction throughput.

3.5 Summary

KVSs are a vital database technology in modern information processing. While KVSs are especially relevant in distributed systems, this work concentrates on single-node KVS for convenience. As in MMDB, KVS can achieve maximum performance when combining in-memory operation and concurrency. In order to preserve consistency in the presence of concurrent transactions, a dedicated concurrency control is required. A modern control scheme which is also used in KVS is MVCC. However, many MVCC protocols, such as the popular SI, are not serializing and are therefore prone to inconsistencies. On the other hand, alternatives which guarantee serializability tend to be error-prone and expensive. Despite having shown promising results, all existing KVSs for NVRAM still only implement SI or locking. This circumstance gives rise to the question whether NVRAM can provide enough leverage to make serializable transactions affordable without resorting to locking.

4 A Key-Value Store with Serializable Transactions for NVRAM

The aim of this thesis is to design a KVS with affordable serializable transactions by exploiting the benefits of NVRAM. In order to provide the necessary groundwork, the previous chapters give an overview on recent research in NVRAM, KVS, and concurrency control in databases. It is further pointed out that, as of this writing, there appears to be no previous work on leveraging NVRAM for concurrency control in KVSs or MMDBs, respectively. While recent works primarily see NVRAM as a means to reduce recovery overhead, this thesis explores a different approach.

As mentioned in Chapter 3, many transaction processing systems do not support or encourage serializable transactions due to severe performance degradation. Therefore, the idea is to use the benefits of NVRAM to make serializable transactions affordable. While this may not provide the highest possible transaction throughput, the aim is to achieve performance on a par with non-serializing solutions for traditional storage. In other words, instead of increasing maximum performance, this work attempts to increase performance with maximum consistency.

NVRAM is especially significant for memory-resident databases as data no longer needs to be copied to a much slower storage device for recoverability. Also, restarts can be near-instantaneous as all data is already in memory and does not need to be fetched. Therefore, the proposed concept is exclusively targeted at MMDB. Given the vast complexity of fully-featured in-memory DBMS, it seems appropriate, for an initial study, to resort to much more manageable KVS. Based on whether the approach turns out to work well for KVS, it may still be applied on MMDB in future work.

This chapter presents the concept for an NVRAM-aware KVS with serializable transactions. After a brief overview in the next section, follow-up sections outline the architecture, concurrency control, and consistency measures.

4.1 Overview

This section provides an overview of the concept. For this purpose, goals, assumptions, and design constraints are outlined. The section concludes with an outline of the desired API and practical examples.

Goal

The intent of this thesis is to determine whether MMDB could exploit NVRAM to make transactions with strong consistency affordable. Given the overwhelming complexity of full-scale DBMS, this work resorts to in-memory KVS. NVRAM significantly reduces the required recovery overhead. While others have used this condition to increase transaction throughput alone, this work chooses to leverage the headroom to compensate for the cost of serializability. The goal is to design a serializing in-memory KVS for NVRAM which performs on par with non-serializing KVS based on volatile RAM.

Assumptions

The concept is based on several assumptions concerning both technical aspects and workloads characteristics.

Hardware In order to take advantage of concurrency, the concept is designed for multi-core architectures. While this increases the number of threads that can be run in parallel, it also introduces synchronization issues for access to shared memory which must be handled with care. However, to keep complexity manageable, the concept refrains from distributed computing and targets single-node databases. In accordance with recent research, it is assumed that volatile RAM will continue to be present and share the same memory interface with NVRAM. This reduces individual bandwidths but enables uniform access methods. It is further assumed that target systems provide sufficient hardware and software facilities to manage NVRAM. Details concerning crash consistency are provided in Chapter 4.4

Workloads When designing systems and transaction processors, in particular, it is helpful to know in advance which kind of workloads are expected or should be given priority. Given that many MMDBs are dominated by read operations [ALR$^+$17], this work is intended for read-mostly workloads. While read-only transactions are supported, there seems to be no hard evidence on the importance or quantity of such transactions. Likewise, long-running transactions are not handled separately, as their share could not be determined.

Design Constraints

In-Memory Operation Since the intent of this work is to evaluate opportunities of NVRAM for MMDB, the target KVS must hold all its data in volatile or non-volatile main memory with no disk storage involved. This way, access latencies are limited to main memory rather than slower disk storage or SSD.

Transactions Contrary to full-grown MMDB, a number of KVSs do not support transactions that span multiple primitive operations. However, in order to allow conceptual conclusions to MMDB, it is important to maintain sufficient generality. Therefore, the concept requires full-featured transactions as in MMDB. As a result, multiple operations, such as reading or writing an item, may be enclosed within a transaction. Likewise, full ACID support is required to guarantee sound transactional semantics. In order to guarantee strong consistency and isolation in the presence of concurrent transactions, serializability is a central requirement. Note that, in contrast to some KVSs, which are designed as caches, the target KVS in this work supports durability. Concerning the nature of key-value pairs, this work imposes no restrictions on their data types. Still, implementations are free to limit the length of keys, for instance, if the underlying data structure requires it.

API

Given the simplistic nature of KVS, this concept anticipates a narrow API that features the very basic building blocks of transactional semantics. The API can be described as a tuple of three instruction sets. First, there are routines to create or manage instances of the KVS. The second set consists of routines to start and end transactions. Transactions are managed through handles which are retrieved when creating them. Such transaction handles are required for the third set of instructions, namely inserting or deleting pairs and retrieving values. Table 4.1 gives an outline of the intended API.

Function	Description
kvstore() : void	Create a key-value store instance
begin() : tx	Start a transaction
commit(tx) : bool	Commit a transaction
abort(tx) : void	Abort a transaction
get(key, tx) : value	Retrieve value for a given key
put(key, value, tx) : bool	Insert a key-value pair
remove(key, tx) : bool	Remove a key-value pair

Table 4.1: API of the intended key-value store.

This API is sufficient to power a basic database with transactional semantics. Listing 4.1 shows an example where a transaction checks whether the balance is negative and, if so, applies a penalty.

```
kvstore kvs;

/* ... lots of transactions ... */

auto tx = kvs.begin();
auto deb = kvs.get("debit", tx);
auto sav = kvs.get("saving", tx);
if (getBalance(sav, deb) < 0.0) {
    kvs.put("debit", applyInterest(deb), tx);
}
tx->commit();
```

Listing 4.1: An example program showcasing the intended API of the KVS.

4.2 System Architecture

The KVS is designed for multi-core architectures and relies on both volatile and non-volatile memory attached to the system memory interface. In order to take advantage of both types of memory, the KVS is designed as a two-level store which only updates NVRAM when a transaction commits. Consistency across crashes is ensured with existing hardware primitives and upcoming platform features. Concurrent transactions are controlled by a serializable variant of SI. Background on these design decisions is given below.

Designing a runtime-critical software such as databases not only involves knowledge about expected workloads but also about the underlying computing device. While workloads have been discussed earlier, this section describes the system architecture of the intended KVS.

Concurrency

The KVS is designed for a single-node architecture. Even though distributed databases are fairly common, there seems to be no apparent reason for them to reveal any more insight on leveraging NVRAM for concurrency. Also, distributed systems involve much more complex mechanisms such as consensus among distributed transactions, all of which are beyond the scope of this work. However, future work should investigate whether the conclusions of this work also hold for distributed databases.

In order to achieve scalable transaction throughput through concurrency, the target system is a multi-core architecture. That means, the system features one or more processors with multiple cores, where each core may support multiple hardware threads. On such a system, each transaction is executed in the context of a thread which is scheduled and assigned to a processor core by the operating system. Processors usually coordinate their work by communicating via some form of chip interconnect. In order to preserve generality, this work makes no assumptions concerning the nature of such interconnect networks.

Memory Architecture

Recent research shows that on traditional hardware it is advisable to continue integrating volatile RAM together with NVRAM. The reason is that not all data is meant to be durable which is especially true for NVRAM where crash consistency is linked with considerable overhead. Manufacturing NVRAM is still challenging, especially in terms of access latency and endurance, but it is expected that these issues will be resolved in the near future. Therefore, in an effort to combine the benefits of both technologies, the memory subsystem is required to feature both volatile RAM and NVRAM. In accordance with recent research it is assumed that both kinds of memory can be accessed through the same memory interface (see Figure 4.1). This work assumes a shared memory architecture. That is, processors may have one or more private cache levels but main memory is accessible to all processors. Conceptually, cache coherence is not required but has the advantage that less effort is spent on coordinating concurrent access to shared data.

The KVS is designed to exclusively reside in main memory. All data that is not required across restarts is stored in volatile RAM, whereas all other data is stored in NVRAM. Multiple recent

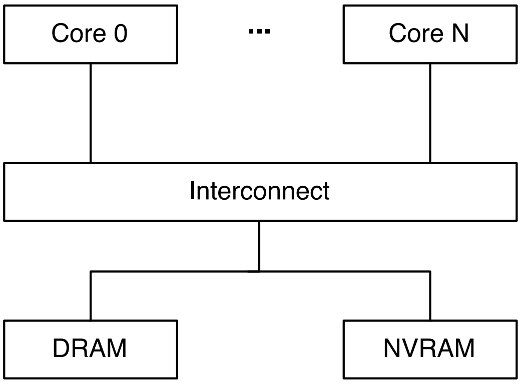


Figure 4.1: Simplified example of expected system architecture.

works have demonstrated that NVRAM can be used to build MMDB without conventional non-volatile storage such as hard drives. As a result, ensuring recovery, which has always been an inherent bottleneck of MMDB, can be eliminated. In addition, near-instantaneous restarts become feasible. As a consequence, conventional storage is not part of the concept for this KVS. While such components may very well be present in a system, they are never used to store any data of the KVS other than its binaries. This way, data access incurs no I/O and restarts do not have to fetch data from slower storage devices. In return, candidate systems must provide sufficient NVRAM capacity to hold the entire database.

A disadvantage of this approach is that the size of the database is bounded by the amount of available NVRAM. In contrast, MMDBs usually allow for larger data sets by keeping frequently used data in memory, while others are moved to slower mass storage media. However, main memory capacities have been steadily growing and NVRAM capacities are projected to have at least twice the capacity of DRAM. Another drawback is recoverability in case of device failures. Mass storage not only scales better in terms of capacity, but it also supports redundancy through Redundant Array of Independent Disks (RAID), for instance. With NVRAM, both capacity and scalability are lower, so employing information redundancy may be prohibitively expensive. Without such measures of fault tolerance, however, a single failed NVRAM module may lead to permanent data loss. This issue is not within the scope of this thesis.

4.3 Key-Value Store Design

This section describes the software architecture of the KVS. That includes the operation principle in terms of transactions, storage, and concurrency as well as the general structure.

4.3.1 Two-Level Store

As mentioned above, the KVS resides entirely in main memory. This enables fast access to all data within the database and makes swapping obsolete. In return, the size of the database is bounded by the total NVRAM capacity. Apart from capacity, operating NVRAM currently exhibits greater access latencies compared to DRAM. As pointed out in Chapter 2, these latencies mainly affect writes and are attributed to both technology parameters and crash consistency measures. This work assumes, that even as technology improves, crash consistency will continue to come a cost.

In an effort to mitigate these issues, the KVS is designed to use volatile RAM in addition to NVRAM. In order to achieve maximum performance, it attempts to exploit the benefits of either technology while limiting the impact of their drawbacks. For that purpose, the KVS employs a two-level store architecture as in [BHC$^+$13].

In a two-level store architecture, in-flight data from memory accesses may be buffered in an intermediary storage medium. In this case, write operations are buffered in volatile memory until their associated transaction commits. Only when a transaction commits, all its updates are propagated to NVRAM. Otherwise, no user data is written to NVRAM. Read operations are not directly affected by the two-level store paradigm. In some cases, however, an implementation may choose to buffer read operations, for instance to determine serialization order. A draft of the architecture is depicted in Figure 4.2.

The aim of the two-level store architecture is to reduce the impact of NVRAM-related latencies. Buffering updates to NVRAM in volatile memory has several advantages. First, updates are only posted to NVRAM when they need to be which can save both latency and memory bandwidth, especially when aborts are frequent. Second, limiting updates to NVRAM to commit phases allows for bulk writes. This way, expensive consistency procedures do not have to be performed repeatedly for a single transaction. Third, updated items in volatile memory can be accessed with lower latencies, which is true for both read and write operations. In the end, buffering updates also aids recovery, as uncommitted data is guaranteed to be lost after a restart.

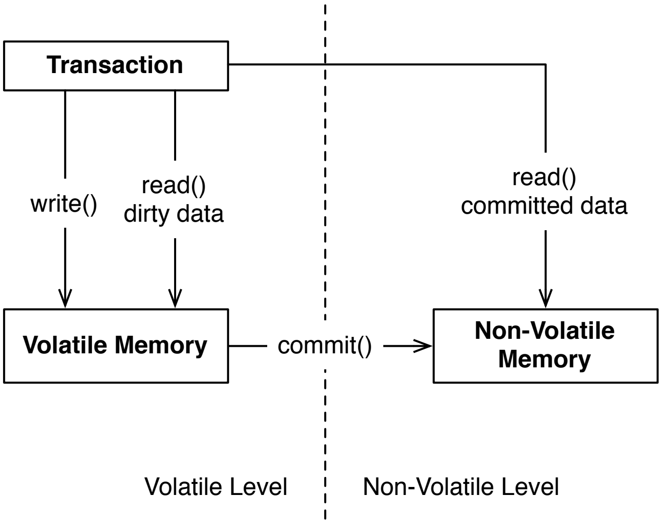


Figure 4.2: Transactional data flow in the two-level store.

4.3.2 Transactions

The KVS supports full-featured and ACID-compliant transactions. Unlike other works, this KVS allows multiple operations to be enclosed in a single transaction. The primary motivation behind this decision is to preserve generality with regard to more complex DBMS. Likewise, all transactions must conform to the ACID properties to ensure data consistency. Nesting transactions is not supported as use cases are too few to justify the additional complexity. Providing ACID support requires a conjunction of preventing erroneous behavior and restoring a previously sane state if an error occurs.

Isolation The cornerstone of this work is to ensure isolation between concurrent transactions, that is concurrent transactions cannot observe each others uncommitted actions. This is achieved with the two-level store architecture and a serializing MVCC protocol.

Each transaction is run in a separate thread. All modifications within a transaction are buffered in thread-local volatile memory as opposed to updating in-place. As a result, the database in NVRAM does not reflect uncommitted changes and can therefore not be used to observe such activity. Technically, threads could spy on each others change buffers but such behavior is neither required nor intended.

Protecting transactions from observing concurrent updates, however, does not imply transactionally consistent data. For this purpose, the KVS employs a serializing MVCC protocol. When compared to locking-based approaches, MVCC protocols have shown better performance in read-intensive environments and are used in many databases. In this case, the concrete protocol is a serializing variant of SI and serves two purposes: recovery and ensuring all transactions behave as if run serially. On the one hand, timestamped versions are used to keep track of modifications. On the other hand, a copy-on-write mechanism is used to enable recoverable version histories without logging. In order to achieve serializability, all read operations are tracked in addition to the usual change sets. On commit, a validation phase ensures that no read versions have been modified by other transactions. If validation fails, so does the corresponding transaction. For write-write conflicts, the first-updater-wins strategy is applied. This helps aborting conflicting transactions earlier than at commit which saves computing resources.

Atomicity Atomicity means that a transaction either succeeds as a whole or it fails entirely. This means that any traces of a failed transaction must be either neutral or reverted. As with isolation, atomicity is achieved by the two-level store architecture and the concurrency control protocol. The former ensures that only modifications of committing transactions are written to durable memory. That way, an incomplete or failed transaction cannot be globally observed. In addition, the SI protocol allows the system to always retrieve the latest committed version of an item. Even if an update to NVRAM fails, the KVS can always go back to the latest committed version without performing an actual rollback. However, in the event an update propagation to NVRAM is interrupted, partial write-backs may become durable. A typical scenario would be a system crash or a power failure. In this case, the KVS must ensure that torn writes do not harm the consistency of data. Possible solutions for this problem are recovery routines that locate and invalidate corrupted data or designated bit fields that are guaranteed to be set only after the entire payload has been written. The concrete recovery method to be used is implementation-defined. For details, see Chapter 5.

Durability & Consistency Ensuring durability and consistency with NVRAM requires special attention and is covered in Chapter 4.4.

4.3.3 Structure

The KVS resides entirely in main memory, that is, both volatile and non-volatile memory. According to the two-level store architecture, the KVS is partitioned into two sections: a volatile and a non-volatile section. An overview of the store's components is given at the end of this section in Figure 4.3.

Volatile Section

The volatile section only contains strictly volatile data, that is, losing these data in a crash can never affect the durable part of the database. Most importantly, that includes transaction control blocks and a transaction table.

Transaction Control Blocks From a software point of view, transactions can be modeled as a tuple of attributes that describe the current state of a transactional context. Typical attributes could be:

- begin and end timestamps
- execution phase
- change sets

Throughout its lifetime, a transaction usually transitions through several execution phases. Beginning with an `active` state once a transaction has started, it may transition to `try_commit` upon commit and finally `committed` when it succeeds. Upon failure, a transaction could indicate a `failed` state. The concrete set of phases is left to the implementation (see Chapter 5).

Change sets are required to buffer all modifications that a transaction carries out. When a transaction commits all modifications are propagated to durable memory. There may be several different change sets depending on the type of operation, such as deleting or updating.

Transaction control blocks are volatile because, otherwise, incomplete transactions would still have to be rolled back to satisfy atomicity. Furthermore, preventing or removing partially committed change sets is handled by recovery and NVRAM management.

Transaction Table In order to manage currently running transactions, each new transaction is placed inside a container. Especially with MVCC, transactions may need a way to inspect other transactions to detect conflicts. Since transactions run on different threads or cores, respectively, the container is a globally shared lookup table. In order to protect critical sections when accessing the table locking or lock-free append-only approaches could be used. Completed transactions may not be automatically removed from the table but collected by a garbage collector. The transaction table is volatile by implication as it only contains transaction control blocks which are explicitly volatile.

Non-Volatile Section

The non-volatile section stores all data that is durable across restarts. It holds a control block, the index structure, and all data items mapped by the index.

Control Block The control block is used to store a small set of essential metadata and for locating the index after a restart. For that purpose, the control block is placed in a fixed position of the KVS' non-volatile region. Possible locations are the front or rear end of the memory region. The index can be found by storing an offset or pointer. Either way, the underlying system must provide a way to reuse or recover a previous memory mapping, so both location methods are sufficient.

Index The index implements the actual KVS paradigm by mapping keys to individual data items. Note that, due to MVCC, instead of mapping concrete data items, each key maps to a history of versions of its associated item. It is the core data structure of a KVS and accessed by all transactions concurrently. As such, the index is a strong contention point that is also very critical for the performance of the KVS. For that reason, selecting a suitable data structure is crucial. The domain analysis in Chapter 3.4 shows that many, if not most, KVSs rely on hash tables. Reasons are amortized constant access, well-known array-like allocation schemes, and comparably low complexity. B-trees or radix trees, on the other hand, are slower and optimized for disk storage which was shown to be inappropriate for NVRAM. Therefore, this work opts for an index based on a hash table.

Operations on the index include adding, retrieving, and removing a key-value pair. To prevent inconsistencies through race conditions, access must be mutually exclusive. While locking can quickly become a bottleneck in read-dominated scenarios, non-blocking data structures are generally slower and more complex. The actual hashing method is an implementation detail (see Chapter 5).

Histories The index maps each key to a chain of versions of a data item. Before an operation on an item can begin, the MVCC algorithm has to determine which version of the item is visible to the requesting transaction. As a result, histories may be iterated frequently. Also, multiple transactions may access the same history at a time, so there may be overlapping accesses. However, transactions never remove and only append new versions to the history. Therefore, contention may be high but mostly read-only. In order to account for these characteristics, an array-like data structure may be the most appropriate choice as opposed to a list. Consecutive access in arrays is less likely to cause page faults and can benefit from hardware prefetching. Also, arrays have simpler allocation patterns compared to lists. Problems arise with garbage collection which could perform random-access modifications to the history. Since garbage collection would break the scope of this work, it is assumed to be absent or as non-invasive as possible. The concrete data structure used for histories is left to the implementation (see Chapter 5). For an overview of all the components of this section, see Figure 4.3 below.

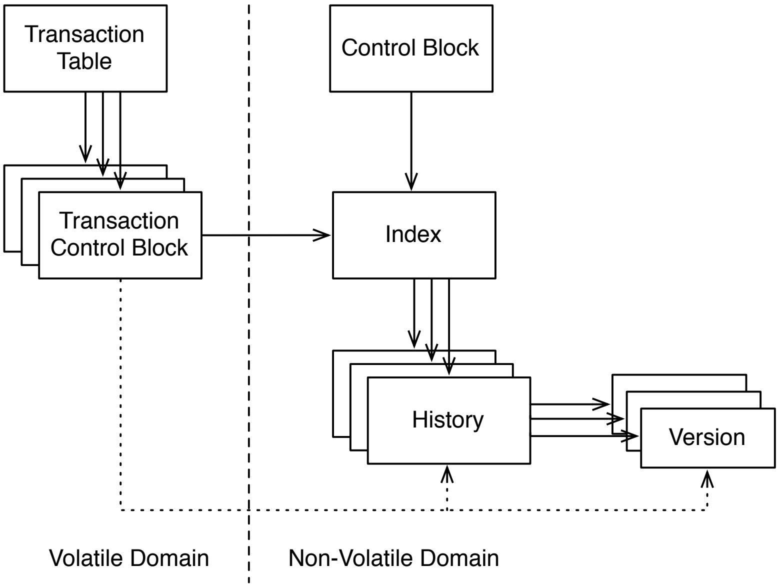


Figure 4.3: Components of the store and their interconnections.

4.4 Crash Consistency

Posting updates to NVRAM is different from traditional non-volatile storage media. With NVRAM, write-back is not directly observable by software and the underlying memory subsystem is unaware of transactional semantics. As a result, updates may be reordered or get stuck in store buffers and caches, thus threatening consistency across crashes.

As pointed out in Chapter 2.3.3, programmers need to manually enforce write-back, in order to preserve consistency. At first, it is important to enforce strict program order for transactionally related memory operations. This can be done with memory barriers or fences. While such measures usually include flushing store buffers, in-flight updates may still get stuck in caches. To prevent this, cache line flushes or non-temporal stores could be used. Even when reaching a memory controller, stores are once again buffered to speed up subsequent reads to that item. While earlier works anticipated designated flush instructions for controller buffers, both researchers and hardware vendors have agreed on the platform feature ADR. When power fails, it receives a signal and utilizes the remaining electrical energy to flush all memory controller buffers.

With the exception of ADR, all of these methods can significantly increase execution latencies as they work against many aspects of modern microprocessors. Deferred write-backs for example are useful to decrease access times for recently written data. Flushing store buffers, however, also affects data that is not involved in transactions. In addition, enforcing strict program order usually results in pipeline stalls. Since there are no other options at the moment, programmers have to meticulously manage and optimize updates to NVRAM.

Summing up, preserving consistency across crashes is necessary but also introduces adverse effects on performance. In accordance with recent research, this work requires the following features:

- means to enforce program order for memory accesses
- means to flush cashes or individual cache lines
- Asynchronous DRAM Self-Refresh

Both memory order enforcement and fine-grained cache flushes are provided by many instruction sets including `x86_64`, SPARC, and IBM POWER and zSeries [McK07]. ADR, on the other hand, is a new platform feature that is already supported on some systems. Therefore, no changes to existing hardware are needed.

5 Implementation

In this chapter a prototype implementation of the concept introduced in Chapter 4 is presented. The chapter is divided in three sections. By showing a few usage examples, the first part gives an impression on how the implemented KVS can be used. The second part deals with the architecture of Midas and outlines selected topics such as NVRAM handling and the components involved in transaction management. The last section delves into the details of transaction processing and the underlying algorithms. In order to make addressing the KVS more convenient, it is henceforth referred to by its working title *Midas* [1].

5.1 Usage Examples

In order to get an understanding of how Midas can be used, this section presents a selected set of usage examples. The first example deals with how an instance of Midas is created or recovered, respectively. The remaining examples showcase the transaction API of Midas.

Before Midas can be instantiated, an underlying persistent object pool must be created or recovered from the file system. Details on object pools are given in the next section. Once the object pool is initialized, Midas can be instantiated and transactions can be performed. The example is given in Listing 5.1.

```cpp
#include "midas.hpp"

/* declarations for this example */

int main() {
    // Create persistent object pool
    midas::pop_type pop;

```

[1] The name is taken from Greek mythology. According to the legend, King Midas was given the ability to turn everything he touches into gold. This resembles the idea of making information non-volatile as in NVRAM.

```
    // Initial size of the object pool (here: 1GB)
    const size_t pop_size = 1024ULL * 1024 * 1024;

    // Initialize persistent object pool from file.
    if (!midas::init(pop, "midas.db", pop_size))
        return EXIT_FAILURE;

    // Create Midas key-value store based on object pool
    midas::Store store(pop);

    // Run example transactions
    run_single_transaction(store);
    run_concurrent_transactions(store);

    return EXIT_SUCCESS;
}
```

Listing 5.1: Midas Bootstrapping.

Midas uses an ordinary transaction API. In order to execute a transaction, it must first be started with a call to `Store::begin()` which yields a handle to the new transaction. Using the transaction handle, the store can be modified in the context of the associated transaction. The available modifiers are `Store::put()`, `Store::get()`, and `Store::drop()`. As a compromise between usability and flexibility, the store supports strings as keys and values, exclusively. Unless a transaction is aborted prematurely, it must be concluded with a call to `Store::commit()`.

```
void run_single_transaction(midas::Store& store) {
    midas::Transaction::ptr tx = store.begin();

    std::string deb = store.get("debit", tx);
    std::string sav = store.get("saving", tx);
    if (get_balance(sav, deb) < 0.0)
        store.put("debit", apply_interest(deb), tx);

    if (!store->commit(tx))
        // Error: transaction failed
}
```

Listing 5.2: A single transaction on account data.

The resulting return code denotes whether the transaction committed successfully or failed. The following example shows a transaction checking the balance of an account and applying an interest fee if the balance is negative (see Listing 5.2).

Running transactions in concurrent threads requires no additional setup other than creating the respective threads to begin with. Each thread only needs a reference to the Midas store. While the store is a global resource, each transaction buffers its updates in its private memory region. The last example starts 10 threads which all execute the accounting transaction from Listing 5.2. Because this transaction may modify a value that is read by all concurrent transactions, many transactions are bound to fail due to read-write conflicts. The example is shown in Listing 5.3.

```
void run_concurrent_transactions(midas::Store& store) {
    std::vector<std::thread> threads;
    for (unsigned i=0; i<10; ++i)
        threads.push_back(
            std::thread([&](){ run_single_transaction(store); }));

    for (auto& t : threads)
        t.join();
}
```

Listing 5.3: Several concurrent accounting transactions on the same store.

5.2 Architecture

This section presents the software architecture of Midas. After a brief overview, selected aspects of the implementation, such as durable memory management, are elaborated in more detail.

Midas is implemented as a C++ library on top of PMDK. PMDK is a set of libraries for NVRAM management that acts as a middleware between programs using NVRAM and the underlying system [Rud17, Cor]. As depicted in Figure 5.1, Midas consists of two components: the key-value store and a set of durable container structures based on PMDK. The latter are used to provide the same level of abstraction of conventional containers for NVRAM. The components of Midas are largely congruent to Figure 4.3 from Section 4.3 and is described in more detail later in this chapter.

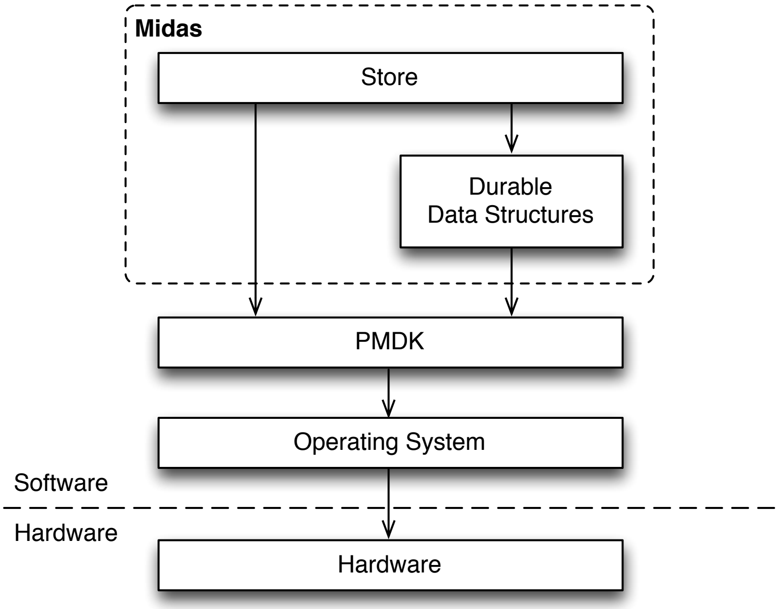


Figure 5.1: System architecture layers of Midas.

Overview Section 5.2.1 describes PMDK and shows how it can be used to access and manage NVRAM. Based on the programming model provided by PMDK, Section 5.2.2 presents the design of two containers classes that are used to handle data in NVRAM more efficiently. Finally, Section 5.2.3 provides an outline of all the components involved in the transaction management of Midas.

5.2.1 NVRAM Management

As explained in Chapter 2 there are several challenges to the integration of NVRAM into current systems. Among these are the following:

- Accessing NVRAM
- Programming against NVRAM
  - Separation of volatile from non-volatile data
  - Recovery of non-volatile objects after restart
  - Preserving consistency across restarts

In the past, several approaches have been discussed to address these challenges. Recent research, however, shows rising interest in PMDK (formerly known as NVML). PMDK is a set of both C and C++ libraries and appears to be the first practical holistic approach to managing NVRAM. Therefore, this work uses PMDK to implement NVRAM management.

NVRAM Access

In PMDK, NVRAM is accessed via ordinary file systems such as `ext4` or `tmpfs`. That way, NVRAM can be mapped into a process' address space via plain files. Using the kernel feature DAX file, systems can bypass the operating system's page cache [OL17, ALR+17, Rud17]. This enables true load and store semantics. In addition, swapping is disabled for the respective memory region.

Programming Model

Apart from its core, the essential part of PMDK used in this work is called `pmemobj`. With it, a persistent object pool holds all data that is meant to be durable. Memory inside the object pool is managed by a designated memory allocator. For each object which is meant to be durable, it allocates the required amount of memory and registers the resulting object in the object pool. This way, all objects can be recovered on restart. From a programmer perspective, objects are managed in a graph rooted in the object pool. Using this graph, which is recovered after restart, the programmer can access all previously durable objects.

Durable Data In order to separate volatile from non-volatile objects, there are templated wrapper classes to manage integral values and dynamically allocated objects:

- `pmdk::p<T>` for durable integral values
- `pmdk::persistent_ptr<T>` for pointers to durable objects

Durable Objects and Pointers A `persistent_ptr` consists of a non-volatile unique object id and a virtual memory address within the object pool. The virtual address is volatile because it is not valid across restarts. Therefore, the object id is mapped to a relative memory address within the object pool. Using the object id, the virtual memory address of an object can be computed from its relative address to the pool base on every restart. This is necessary, because in most operating systems virtual address space layout is different for each session. When creating an object, the function `pmdk::make_persistent<T>()` is used. The lifetime of a durable object spans across scopes and restarts. In order to deallocate an object, the function `pmdk::delete_persistent<T>()` must be used.

Preserving Consistency In order to ensure consistency across crashes in NVRAM, stores must be flushed immediately. In PMDK, this can be done explicitly for selected memory regions or implicitly by using a so-called transaction. A transaction receives the current object pool and a functor. Each persistent property or pointer that is modified by the functor, registers itself with the transaction. When the transaction commits, all registered objects are made durable in NVRAM. A transaction commits when its functor completes without errors. In the example below a durable object of type `T` is created inside a pool which has a graph root of type `Root` (see Listing 5.4). Assume the global variable `objCount` is supposed to reflect the correct number of durable objects. For that purpose, both creating of objects and incrementing the counter are consolidated in a transaction to form a single memory update.

```
#include "libpmemobj++/persistent_ptr.hpp"
#include "libpmemobj++/transaction.hpp"
#include "libpmemobj++/pool.hpp"
namespace pmdk = pmem::obj;

pmdk::p<size_t> objCount = 0;

```

```
template <class T, class Root>
pmdk::persistent_ptr<T> create(pmdk::pool<Root> pop)
{
    pmdk::persistent_ptr<T> obj = nullptr;

    // Consolidate updates in transaction
    pmdk::transaction::exec_tx(pop, [&, this](){
         obj = pmdk::make_persistent<T>();
         objCount = objCount + 1;
    });

    // At this point, all changes to NVM are either completely
        durable or absent
    return obj;
}
```

Listing 5.4: Consistent updates to NVM using PMDK transactions.

Even though object creation is always transactional in PMDK, initialization is not. Instead, the memory of an object is only nulled. As a result, each class intended for durability must be *trivially default constructible*, which means that a compiler-generated default constructor must be sufficient to initialize an instance of the class. Otherwise, the programmer has to use custom initialization procedures.

Internally, PMDK uses largely the same mechanism for consistency as described in Chapter 2.3.3. That means, modified data is flushed from memory order buffers and caches. In some situations, non-temporal writes are used which make use of write combine buffers. When it comes to flushing write-pending queues inside the memory controller, PMDK relies on ADR.

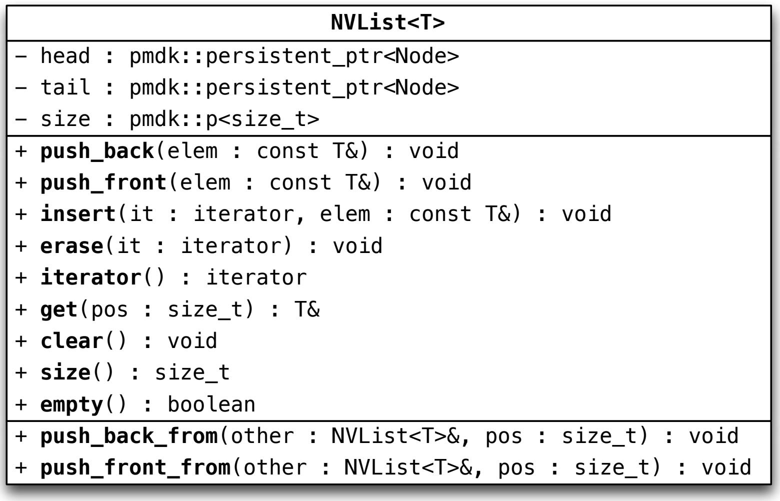
5.2.2 Durable Data Structures

The implementation of Midas relies on several fundamental data structures, such as arrays, lists, and hash maps. Since Midas is divided in two separate volatility domains, each one requires its own dedicated data structures. The volatile part of Midas is composed of containers from the C++ standard library. The non-volatile part, on the other hand, is based on custom container implementations using PMDK. This section outlines the design of durable data structures in Midas.

Durable data structures in Midas are the index and the version histories contained therein. The index is used to lookup version histories, thus it is implemented as a hash table. Histories, on the other hand, are implemented as linked lists because they are accessed in linear order. Lacking general-purpose container libraries based on PMDK, both non-volatile data structures are developed from scratch.

Doubly-Linked List

The doubly-linked list is implemented in the template class `NVList`. Its attributes consist of a pointer to first element, a pointer to the last element, and an element counter. All these attributes must be durable in order to recover the list after a restart. Therefore, they are declared as persistent pointers or properties in terms of PMDK. The API of `NVList` is closely related to the C++ standard library containers. It also provides iterator facilities for traversal (see class diagram in Figure 5.2).



```
NVList<T>
- head : pmdk::persistent_ptr<Node>
- tail : pmdk::persistent_ptr<Node>
- size : pmdk::p<size_t>
+ push_back(elem : const T&) : void
+ push_front(elem : const T&) : void
+ insert(it : iterator, elem : const T&) : void
+ erase(it : iterator) : void
+ iterator() : iterator
+ get(pos : size_t) : T&
+ clear() : void
+ size() : size_t
+ empty() : boolean
+ push_back_from(other : NVList<T>&, pos : size_t) : void
+ push_front_from(other : NVList<T>&, pos : size_t) : void
```

Figure 5.2: Class diagram for the class `NVList`.

An important aspect of implementing durable data structures is to preserve consistency. This is done by consolidating related allocations and operations on durable objects in a transaction as shown in Chapter 5.2.1. Listing 5.5 shows the implementation of the function `push_back`, which is used to append an item to the back of the list.

```
void push_back(const elem_type& elem)
{
    pmdk::transaction::exec_tx(pool, [&,this](){
        auto new_node = pmdk::make_persistent<node>();
        new_node->mValue = elem;
        append(new_node);
        mSize = mSize + 1;
    });
}
```

Listing 5.5: Implementation of `push_back()` with transactional updates to NVRAM.

An additional feature of `NVList` is a zero-copy mechanism for migrating individual list nodes between different lists. This is useful when nodes from a single list must be scattered to several other lists as is done when rehashing a hash table. The feature is provided by the functions `push_back_from()` and `push_front_from()`.

Hash Table

The hash table is implemented as a template class called `NVHashmap`. Essentially, the table consists of a single persistent array that is reallocated as needed. Each cell in the index is called slot or bucket. As with `NVList`, the API of `NVHashmap` is closely aligned to the C++ standard library. Also, the class employs a policy-based template design which allows changing its behavior through template parameters (see Figure 5.3).

There are three important aspects to the design of hash tables: the hash function itself, collision handling, and growth policy. The hash function used in Midas is a variant of the hash function used in the string library of Java [Orab], which is a polynomial of prime powers and character bytes as coefficients. The final access index is computed modulo of the current table size.

When disjoint keys result in equal hashes a collision occurs. In the case of Midas, collisions are handled by *chaining*. With chaining, each slot in the table is a linked list and keys are always stored in pairs with their mapped value. When hashing a key, the corresponding bucket is searched for a pair with a matching key. If no such pair is found, then the given pair can be inserted. As a result, each bucket only contains pairs with conflicting key hashes. Both the collision lists and the pairs are durable.

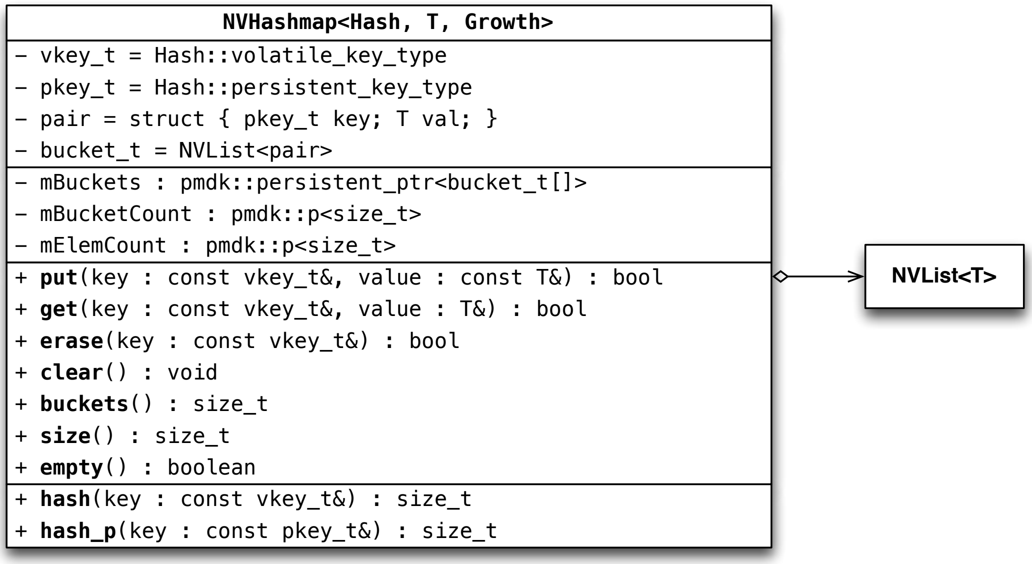

NVHashmap<Hash, T, Growth>
- vkey_t = Hash::volatile_key_type
- pkey_t = Hash::persistent_key_type
- pair = struct { pkey_t key; T val; }
- bucket_t = NVList<pair>
- mBuckets : pmdk::persistent_ptr<bucket_t[]>
- mBucketCount : pmdk::p<size_t>
- mElemCount : pmdk::p<size_t>
+ put(key : const vkey_t&, value : const T&) : bool
+ get(key : const vkey_t&, value : T&) : bool
+ erase(key : const vkey_t&) : bool
+ clear() : void
+ buckets() : size_t
+ size() : size_t
+ empty() : boolean
+ hash(key : const vkey_t&) : size_t
+ hash_p(key : const pkey_t&) : size_t
NVList<T>

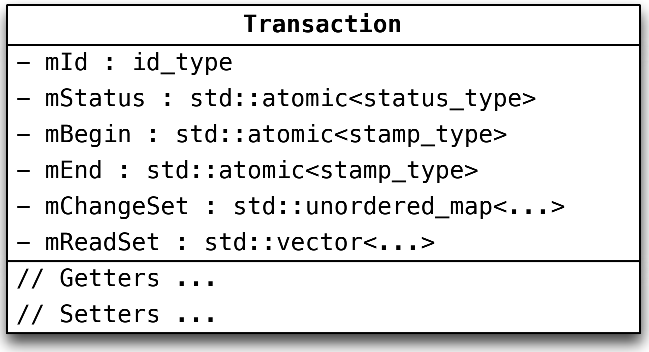
Figure 5.3: Class diagram for the class `NVHashmap`.

Because searching long buckets increases access time, the table is expanded regularly. More precisely, expansion occurs when the maximum load factor is exceeded. The load factor is defined as the ratio of the number of pairs and the number of buckets. The maximum load factor is set to 0.75. In order to amortize the cost of reallocations, the table size is doubled on each expansion. After an expansion, all pairs are rehashed into the new table. This is where the zero-copy list node migration of `NVList` is used. Parameters such as the maximum load factor or the growth factor can be adjusted using the policy-based template design.

5.2.3 Components

This section presents an outline of all components involved in the transaction management of Midas. A condensed overview is given at the end of the section.

Transaction Control Blocks Transactions are represented by transaction control blocks (TCBs). A TCB holds a unique transaction id (TID), a status code, two timestamps, as well as read and write sets. For historical reasons, a TCB is internally denoted as a `Transaction` (see Figure 5.4).



```
Transaction
- mId : id_type
- mStatus : std::atomic<status_type>
- mBegin : std::atomic<stamp_type>
- mEnd : std::atomic<stamp_type>
- mChangeSet : std::unordered_map<...>
- mReadSet : std::vector<...>
// Getters ...
// Setters ...
```

Figure 5.4: Class diagram of a transaction control block.

The TID is required to lookup the associated transaction. The status code indicates the current execution phase of a transaction and can be one of the following:

- Active
- Failed
- Committed

The execution phase of a transaction t is used by other transactions to determine t's state if they encounter dirty versions of t. In addition, there are two timestamps that denote when t started and when it ended. This information is vital to determine if the lifetimes of two transactions overlap. Since both transaction state and timestamps must be accessible and consistent to transactions from other threads, they are implemented as atomic variables.

When a transaction modifies the store with calls to `Store::put()` or `Store::drop()`, a designated change entry is stored in the associated TCB. Depending on the operation performed, a change entry contains the following items:

- an operation descriptor
- a pointer to the original version
- the raw updated value
- a pointer to the updated version

Using the operation descriptor, Midas knows how to perform a change on commit. For instance, if the change resulted from a `Store::drop()`, then it only contains a pointer to the original version which has to be invalidated. Change sets are implemented as hash tables. This is because a transaction might update an item multiple times and looking up a previously written item is faster than repeatedly scanning an item's history. In order to detect read-write conflicts, and thus enable serializability, a TCB also has to record all items that were read. Because read sets can grow large in read-mostly environments, they are implemented as dynamic arrays to amortize allocation cost. Both read and change sets are thread local storage and are therefore not synchronized.

When a transaction is started with `Store::begin()`, a TCB is created. During initialization the TCB is assigned a TID and a start timestamp. Once the TCB is initialized, its TID is used as a key to insert it into the transaction table. The lifetime of a TCB, including its transaction, ends when it is removed from the transaction table. This happens when the associated transaction encounters a conflict or is aborted with `Store::abort()`.

TIDs and Timestamps TIDs and timestamps are represented by 64-bit unsigned integers. They are acquired by incrementing a designated global counter, respectively. In order to ensure that the counters are consistent across all cores, they are incremented using atomic fetch-and-add instructions. Because there are situations where TIDs and timestamps must be distinguished from another, their value ranges are interleaved. TIDs are always odd numbers, whereas timestamps are always even. Each counter is incremented by two, accordingly. The reason for this design is explained in the next section.

Transaction Table As explained above, there are situations when transactions may need to determine the state of other concurrent transactions. Therefore, TCBs are stored in a global transaction table. In order to provide fast lookup, the transaction table is implemented as a hash table. Each TCB is hashed using its TID. However, the table is a globally shared resource that must be synchronized. In order to prevent a bottleneck, the concurrent hash table implementation *libcuckoo* is used [FAK13, LAKF14, GFL$^+$].

Versions With MVCC, concurrent access to data is managed by having multiple versions of an item. In this case, the value of each key-value pair is versioned. In Midas, a version is durable and consists of the actual data value and two timestamps indicating the lifetime of the version. The data value is represented by a durable dynamic array of characters, similar to std::string from the C++ standard library. Because all transactions must have a consistent view on the lifetime of a version, its timestamps are only accessed by means of atomic instructions. The timestamp fields can be used in several ways, as shown in the next section.

Histories For each value of a key-value pair, there are several versions. The versions of a single item are managed within a history. When a transaction wishes to access an item, the transaction management has to search the item's history for version that is *visible* to the transaction. In Midas, a history is a durable linked list by means of NVList from Section 5.2.2. At startup, existing histories are truncated to contain only their most recent version. After that, histories are in append-only mode.

Index Another core data structure in Midas is the index. It accomplishes the mapping between the key of an item and the item's version history. Whenever a transactional operation wishes to access an item, it has to query the index with the corresponding key. The index must provide fast lookup and is therefore implemented as a hash table. For that purpose, the durable hash table NVHashmap from Section 5.2.2 is used. The index is accessed by all concurrent transactions and must therefore be synchronized. Due to the complexity of concurrent data structure design, a lock is used to protect the index. This is likely to become a serious bottleneck and should be revised in future work. For a list of all components in Midas and their underlying data structures, see Table 5.1.

Component	**Data Structures**	**Memory Domain**
Transaction Control Block	std::vector,	volatile
	std::unordered_map	volatile
Transaction Table	libcuckoo_map	volatile
Index	NVHashmap, NVList	non-volatile
History	NVList	non-volatile
Version	NVString	non-volatile

Table 5.1: An overview of all components and their underlying data structures.

5.3 Transaction Processing

Unlike other KVSs, Midas can only be operated through transactions. Transactions that are either terminated or belong to another store instance are rejected. Once a transaction has been started with `Store::begin()`, it can be used to access the store with `Store::read()`, `Store::write()`, and `Store::drop()`. For the full list of procedures, see Table 5.2.

Operation	Procedure		
Begin	`begin()`	:	`tx_ptr`
Abort	`abort(tx_ptr)`	:	`int`
Commit	`commit(tx_ptr)`	:	`int`
Read	`read(tx_ptr, const key_type&, value_type&)`	:	`int`
Update	`write(tx_ptr, const key_type&, value_type&)`	:	`int`
Delete	`drop(tx_ptr, const key_type&)`	:	`int`

Table 5.2: An overview of all procedures provided by Midas.

Each operation has its own procedure. The control flow of most procedures is structured in the same way. At first, inputs are checked for errors. Next, the history of the item associated with the specified key is looked up. While `Store::read()` and `Store::drop()` will fail if no history is found, `Store::write()` will perform an insertion by adding a new pair to the transaction's change set. If a history is found, then it is scanned for a version that is visible to the specified transaction. If a valid version is found, the operation can perform its read, write, or deletion, respectively. Otherwise, the operation cannot continue and aborts the associated transaction with an error code. Finally, a transaction must be committed with `store::commit()` to make its changes durable. Until then, it can always be aborted with `Store::abort()`. The subsequent sections give details about the implementations of visibility, operations, and commits.

5.3.1 Visibility

A transaction can only access an item if that item's history contains a version that is visible to the transaction. Therefore, transactions have to scan the histories of all items that they wish to access. During the linear search on a history, for each version, a visibility test is performed. There is at most one version visible to a transaction, if any.

Since Midas is based on SI, it features the same visibility conditions. Simply put, a version v_0 is visible to a transaction t_1, when v_0 was committed before t_1 started. There are several facets to this property. For instance, during t_1's runtime, another transaction t_2 may have updated v_0 but has not yet committed. In this case, t_1 can still see v_0 because, t_2 may still fail or get aborted. For this to work, t_1 must be able to see that v_0 was touched by t_2. Therefore, when a transaction t updates or deletes a version v, t replaces v's end timestamp with the TID of t. This way, t_1 can see that v_0 was updated, lookup t_2 in the transaction table, and inspect its current status. Likewise, if t_2 encounters a version v_1 that was created by t_1, then t_2 would find t_1's TID in the begin timestamp field of v_1. When a transaction commits, it changes its status to `Committed` and propagates its own end timestamp to all versions it has created or modified. But other transactions do not have to wait until the propagation is finished, because, using the deposited TID, they can always lookup the respective transaction and see that its status has changed to `Committed`.

When testing the visibility of a version, the test first inspects the version's begin timestamp and then its end timestamp. As explained earlier, the value domains of TIDs and timestamps are disjoint in that the former are always odd while the latter are even. Therefore, it is sufficient for a transaction to test the least significant bit of the timestamp to distinguish TIDs from timestamps. A pseudo code example of the visibility test in Midas is outlined in Algorithm 1. The visibility properties implemented in Midas are based on the works of [BBG+95, LBD+11].

Algorithm 1

```
procedure isVisible(tx, begin, end)
    if isTID($begin$) then
        $tx^* \leftarrow$ getTx($begin$)
        if getStatus($tx^*$) $\neq$ Committed or getEnd($tx^*$) $>$ getBegin($tx$) then
            return false
    else
        if $begin >=$ getBegin($tx$) then
            return false

    if isTID($end$) then
        $tx^* \leftarrow$ getTx($end$)
        if getStatus($tx^*$) $=$ Committed and getEnd($tx^*$) $<$ getBegin($tx$) then
            return false
    else
        if $end <$ getBegin($tx$) then
            return false
    return true
```

5.3.2 Reads

The read operation is the simplest operation to perform with Midas. Still, with regard to serializability, reads are very important, because this is where Midas extends SI. Reads are performed by calling `Store::read()` and are implemented as follows. A pseudo code example is given below (see Algorithm 2). After initial input sanitization, it is checked whether the specified item has already been updated within the same transaction. In this case, the item's updated value is retrieved from the change set and the operation returns. Otherwise, the history associated with the given key is looked up. Note that the index is locked for the duration of the lookup. If no history is found, the current transaction is aborted.

Algorithm 2

```
procedure READ(tx, key, value)
    if inChangeSet($tx, key$) then
        $value \leftarrow$ getChangeEntry($tx, key$)
        return OK
    $history \leftarrow$ getHistory($key$)
    if !$history$ then
        return abort($tx$)
    $snapshot \leftarrow$ getReadSnapshot($tx, history$)
    if !$snapshot$ then
        return abort($tx$)
    updateReadSet($tx, snapshot$)
    $value \leftarrow$ getData($snapshot$)
    return OK
```

From the history, a readable version (i.e. *snapshot*) of the target item is determined. The call to `getReadSnapshot()` is where the visibility test from Section 5.3.1 comes into action. If this fails, the transaction has to abort. Note that for the duration of `getReadSnapshot()`, the respective history is locked. When a valid snapshot is found, it is registered in the read set of the transaction. This is necessary to detect potential read-write conflicts at the end of the transaction and thus enable serializability. After the read set has been updated, the snapshots payload can be retrieved and written to the output parameter `value`.

5.3.3 Updates

The update operation in Midas is called `Store::write()`. It can be used to update existing items and insert new items. Because there can be no duplicates, Midas can always determine which action to perform. A pseudo code of the implementation is given in Algorithm 3. As before, it is first checked whether the specified item has already been modified in the context of the current transaction `tx`. If so, the change set is updated with the new value. In case the change entry indicates that the item was previously removed by `tx`, the operation descriptor is changed from `mode::remove` to `mode::update`. This is possible because, due to the two-level store architecture, all updates and deletions are buffered until commit. If the target item is not in the change set, Midas proceeds as usual and looks up the item's version history.

Algorithm 3

```
procedure WRITE(tx, key, value)
    if inChangeSet(tx, key) then
        updateChangeSet(tx, key, value)
        return OK
    history ← getHistory(key)
    if !history then
        return insert(tx, key, value)
    snapshot ← getWriteSnapshot(tx, history)
    if !snapshot then
        return abort(tx)
    setEndTS(snapshot, getId(tx))
    updateChangeSet(tx, mode::update, snapshot, value)
    return OK
```

Unlike with reads, a missing history is not a reason to abort but indicates that a new version must be inserted. If a history exists, then a writable snapshot is determined. Note that there is no insertion if `getWriteSnapshot()` fails, because that indicates a write-write conflict. In this case, the most suitable version must have been invalidated with the TID of a transaction other than the current `tx`. Therefore, `tx` must abort according to the *first-updater-wins* principle seen in Section 3.3.3. If a valid snapshot is found, its end timestamp is atomically replaced with the TID of `tx`. Finally, a new change set entry is created and the operation is complete. Note that, apart from replacing the snapshots timestamp, no NVRAM is updated at this point.

5.3.4 Deletions

An item can be deleted by invoking `Store::drop()`. Deletion is very similar to updating. The corresponding pseudo code is shown in Algorithm 4. As with updates, the change set is first checked for an existing change entry. If an entry exists, then there are two possibilities: the entry denotes an insertion and is is deleted immediately or it denotes an update in which case the operation descriptor is changed from `mode::update` to `mode::remove`. In contrast to updating, deletion must abort if no history is found for the specified key. The rest of the function works like `Store::write()` (see Algorithm 3).

Algorithm 4

```
procedure DROP(tx, key)
    if inChangeSet(tx, key) then
        updateChangeSet(tx, key, value)
        return OK
    history ← getHistory(key)
    if !history then
        return abort(tx)
    snapshot ← getWriteSnapshot(tx, history)
    if !snapshot then
        return abort(tx)
    setEndTS(snapshot, getId(tx))
    updateChangeSet(tx, mode::remove, snapshot)
    return OK
```

5.3.5 Commits

In order to manifest the changes of a transaction in the store, the transaction must be committed using `Store::commit()`. Its implementation is outlined as pseudo code in Algorithm 5. A commit works as follows. At first, the transaction receives an end timestamp. In the next step, the read sets of the transaction are validated against read-write conflicts. If at least one read version has been updated or deleted, then the transaction aborts. This implementation could be too pessimistic but it preserves serializability. Once the validation succeeds, all change sets of the transaction are translated into durable changes. As a result, durable versions are allocated or deallocated, and histories are modified accordingly. After this step, all changes are durable and the transaction status is changed from `Active` to `Committed`.

At this point, the TID of the committed transaction is still deposited on all the versions that it has modified or created. This is not a problem because other transactions can still inspect the committed transaction. In order to be able to remove the transaction at some point, its TID must be removed from all these versions. This is done by replacing the deposited TID

Algorithm 5

```
procedure COMMIT(tx)
    setEndTS($tx$, nextTS())
    if !validate($tx$) then
        return abort($tx$)
    persist($tx$)
    setStatus($tx$, Committed)
    propagateTimestamps($tx$)
    removeTransaction($tx$)
    return OK
```

with the transaction's end timestamp. The timestamp will be installed as: end timestamp for updated or deleted versions or begin timestamp for newly inserted versions. Finally, the transaction is complete and its control block can be removed from the transaction table.

6 Evaluation

Based on the concept introduced in Chapter 4, a prototype was implemented. Its goal is to provide fast serializable database transactions through the use of novel byte-addressable non-volatile memory. This chapter evaluates the prototype to determine effectiveness of the concept.

It has been shown that MMDB can benefit from NVRAM [BHC+13, SBD+16, OL17, ALR+17]. However, serializable transactions are still traded in favor of performance, by default. If the performance gain from NVRAM can be leveraged to compensate for serialization overhead, then serializability could be adopted as the default isolation level.

An important measure in this context is transaction throughput, as it is known to degrade with stronger isolation levels due to higher abort rates. Therefore, it is investigated whether transaction throughput with serializability can be as high as in non-serializable systems and scale accordingly. For this purpose, the implemented prototype, *Midas*, and the NVRAM-based KVS, *Echo* from Chapter 3.4, are compared in a dedicated transaction throughput benchmark.

At first, however, there are a couple of challenges to discuss. Next, the system configuration used for the evaluation is described. Subsequently, the transaction throughput benchmark is conducted and evaluated. The chapter concludes with a brief summary.

6.1 Challenges

Evaluating database systems or database concepts in general is a complicated topic and there are even additional issues in this particular instance. Essentially though, the evaluation of Midas is facing two major challenges:

1. NVRAM is not part of the system configuration
2. Established database benchmark software cannot be used

Promising NVRAM technologies such as PCM or 3DXPoint are not yet commercially available as DIMMs. Therefore, the evaluation has to be carried out on volatile DRAM which does not account for the greater access latencies of NVRAM. In order to still obtain meaningful results, it is possible to enforce custom latencies by modifying the system BIOS as in [DKK$^+$14, OLK$^+$15] or by reprogramming the DIMM microcode [SBD$^+$16]. Some authors, however, have decided not to emulate latencies [BHC$^+$13, ZSLH16]. Arguments include that any emulated latency would be inherently inaccurate as final NVRAM parameters are yet unknown. Further, they argue that it is safe to assume that the current latency gap of up to 10% can be eliminated as technologies advance. The issue of emulating NVRAM is addressed in Section 6.2.

In general, database systems are very complex software systems with a wide range of mutually differing feature sets, technologies, and optimization goals. As a result, it is very hard to identify use cases or workloads that are meaningful for all systems under evaluation. Ultimately, it is difficult to compare the performance of these systems against each other. Therefore, vendors usually rely on established benchmark software such as TPC [SBD$^+$16] or TATP [OLK$^+$15] as a reference. However, these systems require all their test candidates to provide a query language front end which is used to run their benchmark routines. Since neither KVS in this evaluation supports a query language, none of these benchmark suites can be used. As a result, custom benchmark routines must be developed as is done in similar works [BHC$^+$13, ZSLH16]. This issue is discussed in Section 6.3.

6.2 System Configuration

The benchmarks are conducted on a platform of four Intel Xeon E7-4830 processors with a total of 32 cores running at 2.13GHz. Main memory consists of 256GB DDR3 DRAM running at 1066Mhz. For durable storage, the system relies on mechanical hard drives. The operating system is CentOS 7.4 based on Linux 3.10.

Since NVRAM is not available, all benchmarks are carried out on volatile DRAM. In order to emulate persistence, each KVS stores its contents in a memory-mapped file. To satisfy the consistency requirements shown in Chapter 2.3.3, it is important to prevent potential buffering of data in kernel page caches. This is achieved by using a 127GB instance of *tmpfs* as RAM disk. However, even with this setup, memory pages may still be swapped at will by the operating system. In order to prevent this, swapping is disabled. As a result, the hard disk is never used and no write operation within the mapped region is deferred. In accordance with [BHC+13, ZSLH16], this work refrains from emulating NVRAM latencies for the same reasons given in Chapter 6.1.

6.3 Transaction Throughput

Transaction throughput is a key benchmark for the performance of transactional databases. Systems with stronger isolation levels tend to have lower transaction throughput resulting from the increased number of aborted transactions. Therefore, in this experiment, transaction throughput is used to determine the performance impact of serializability in NVRAM-based key-value stores.

6.3.1 Setup

In order to perform the experiment, there are two prerequisites to satisfy. At first, data is required to populate each KVS. Second, workloads of concrete transactions are needed to operate on the populated stores. Key-value pairs may not be too hard to come by but in this case synthetic inputs are fine as well. Workloads are more complicated because it is hard to acquire transaction traces and determine their relevance. For that reason, the experiment is synthetic and relies exclusively on randomly generated data and workloads. All pseudo-random numbers were generated using uniform distributions. However, each random choice is reproducible as the generated data is stored on disk prior to the experiment. Hence, the only non-deterministic behavior is induced by multi-threading, the operating system, and the underlying hardware.

Key-Value Data

The data used in the experiment comprises two distinct sets of randomly generated key-value pairs. The first data set contains 1K pairs, whereas the second holds 100K entries. Lacking insights on meaningful layouts of key-value pairs, this evaluation relies on the same layout used in [BHC+13]. Accordingly, keys are 128-byte random strings and values are 1024-byte random strings. When the experiment starts, each individual KVS is populated with these key-value pairs.

Workloads

All transactions that are to be executed during the experiment are encoded as a workload. A workload specifies for each transaction its operations and the respective key-value pairs to operate on. When generating a workload, there are three important dimensions:

- number of transactions
- length of transactions
- type of operations

Workload Size In this experiment, each workload consists of 1000 transactions. If longer execution times are required, e.g. to obtain more stable measurements, workload size could be increased at will.

Transaction Length The length of a transaction is the number of operations enclosed in that transaction. Intuitively, the length of transactions should vary. Unfortunately, no reliable sources as to the absolute quantities of transaction lengths could be found. Nevertheless, the following ranges in Table 6.1 are deemed meaningful.

Name	min{#operations}	max{#operations}
Short	2	32
Long	64	256

Table 6.1: Types of transactions in terms of length.

Short transactions can be small updates like incrementing a numeric value, optionally based a small aggregation. Long transactions on the other hand, can be larger aggregations such as computing a sum over many items.

Transactional Operations When specifying the operations of a transaction, it must be decided whether an operation reads or updates an item. Insert and delete operations are omitted as they complicate the experiment when run concurrently. For example, a concurrent transaction might fail because an expected pair has not been inserted yet. Such an incident reduces transaction throughput without an actual conflict which could distort results. The remaining two operations are selected based on the empirical analysis in [ALR+17]. According to the source, read operations amount to 84% of all operations. The remaining 16% are subsumed as updates for this experiment. Each operation is programmed to act on a random key from the data set associated with the workload. Therefore, workloads are bound to the data set used during generation.

Scenarios and Expectations

Resulting from the dimensions shown above, four unique scenarios are derived. The following scenarios are supposed to simulate both low and high contention in different forms (see Table 6.2).

Name	Database Size	Transaction Length
S1	Small (1K)	Short (2 - 32)
S2	Small (1K)	Long (64 - 256)
S3	Large (100K)	Short (2 - 32)
S4	Large (100K)	Long (64 - 256)

Table 6.2: Scenarios are combinations of database size and transaction length.

In a database that is small compared to the number of concurrent transactions, it is very likely that multiple transactions operate on the same data which can cause conflicts. Likewise, longer transactions cause contention as they are more likely to access data of other transactions. That said, scenario S1 and especially S2 simulate high contention which is the worst-case for any concurrency control. The reason is that contention often causes conflicts which lead to aborts and reduced transaction throughput. With larger databases, short transactions are less likely to access the same data which reduces contention. However, as transactions become longer, they become more likely to collide and abort.

6.3.2 Methodology

The benchmark is performed separately for each KVS. The general procedure is to execute each workload with a different number of cores. Since the underlying machine has 32 physical cores, the experiment is performed with 1, 2, 4, 8, 16, and 32 cores. During each run, several statistics such as the total time taken and the number of aborts are captured. In order to reduce the influence of outliers, each run, i.e. a combination of workload and core count, is performed 10 times. As a result, for each store there is a total of 24 configurations which amounts to a total of 480 runs including repetitions.

A single run of the benchmark works as follows. At first, the respective KVS is initialized with dummy data and the workload decomposed so that it fits the number of cores of the current configuration. Then, for each core, a thread is created and pinned to that core. Each thread executes an equal share of the entire workload. The total time taken to execute the entire workload is measured by taking the difference of timestamps from before spawning the workers and after all workers have terminated. The general procedure of the experiment is depicted in Figure 6.1. The procedure performed by a single worker thread is shown in Figure 6.2. When a transaction fails, it is retried up to three times. If all retries fail, the transaction is canceled and the program proceeds to the next transaction in the workload.

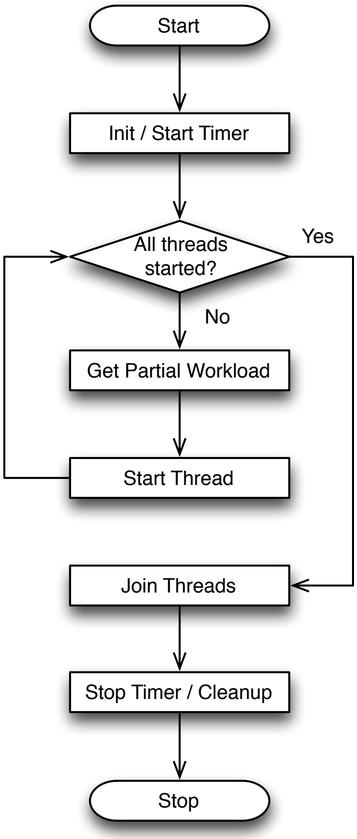


Figure 6.1: Logical flow of the benchmark application.

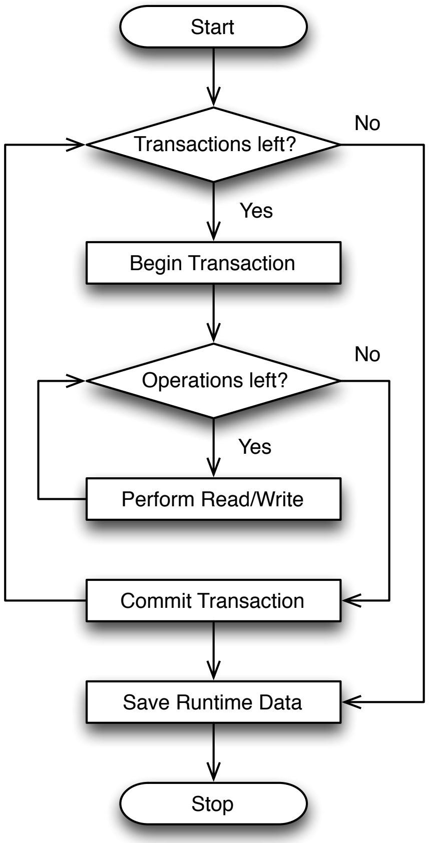


Figure 6.2: Logical flow of a single worker thread.

6.3.3 Results and Discussion

In this subsection the results of the benchmark are shown and discussed. The benchmark captures two metrics: transaction throughput and the transaction abort rate. In order to express the scalability of both KVSs, throughput is expressed in terms of speedup factor. This also helps comparing the performance of both stores even if absolute throughput differs significantly. The abort rate is given by the ratio of the number of aborted transactions and the number of scheduled transactions. When interpreting throughput and speedup factors, it is important to take the abort rate into account because higher abort rates can lead to higher transaction throughput. The reason for this circumstance is that commits are very expensive and the resulting time saving of an abort has a stronger influence on throughput than reducing the number of committed transactions. This is also a direct consequence of the two-level store architecture and the additional measures to preserve consistency in NVRAM.

Legend The following analysis compares benchmark results of Echo (green), Midas (blue) and its optimized variant, denoted as Midas* (red). In the current implementation of Midas, the index is based on a naive hash table implementation for NVRAM. In order to protect critical sections in the hash table, a simple global locking mechanism is used. This approach forms a major bottleneck which can be addressed with a designated but often complicated concurrent design as in [FAK13]. However, this is an implementation detail, albeit crucial. In order to show the potential of Midas' design, a second variant of Midas without index locking was examined. This is possible because during the benchmark not the index but only histories are modified. This is a side effect of omitting insert and delete operations for simplicity.

Scenario S1 This scenario simulates high contention by means of a small database but short transactions. The expected outcome in this scenario is that Echo performs better than Midas because it detects fewer conflicts which are also less likely. At first, the plot of transaction throughput in Figure 6.3 confirms that expectation. However, it also confirms that the index lock in Midas is a massive bottleneck as Midas* achieves up to 50% higher throughput than Echo. Concerning speedup factors, all stores fail to improve for more than 8 cores (see Figure 6.4). This is better reflected in terms of parallel efficiency as shown in Figure 6.5. The reason is that there are very few key-value pairs which increases synchronization overhead. As expected, both variants of Midas exhibit much higher abort rates than Echo, due to stricter conflict detection (see Figure 6.6). While Midas* makes the best use of additional cores, it produces a slightly higher abort rate than Midas. This reveals a serializing nature of the global lock in the original implementation of Midas.

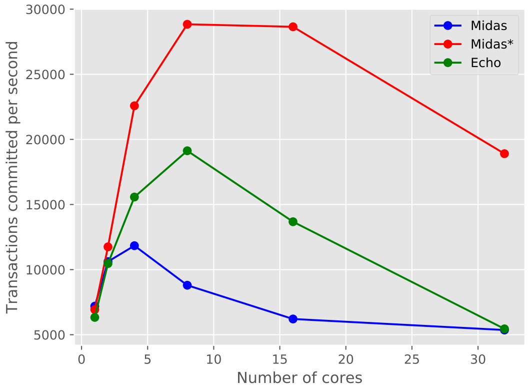


Figure 6.3: Transaction throughput for scenario S1.

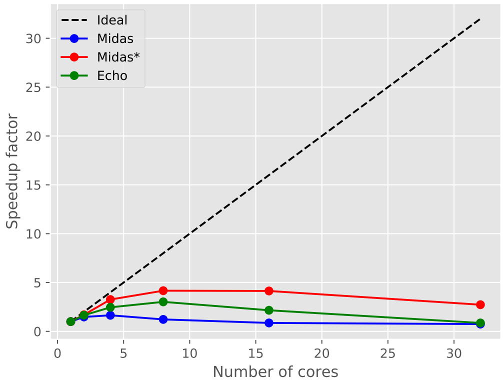


Figure 6.4: Transaction throughput speedup for scenario S1.

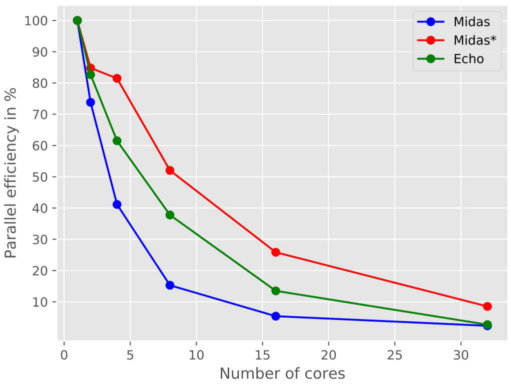


Figure 6.5: Parallel efficiency for scenario S1.

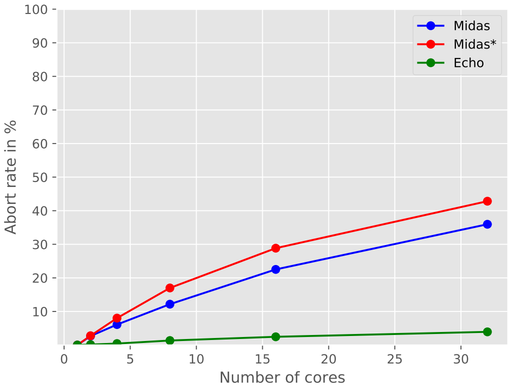


Figure 6.6: Abort rate for scenario S1.

Scenario S2 This scenario simulates very high contention by means of a small database and long transactions. It can be seen as the worst-case scenario for concurrency controls with strong isolation levels. Therefore, both versions of Midas are expected to be outperformed by Echo. The plot in Figure 6.7 confirms that, as transaction throughput for both Midas and Midas* plummets towards zero. This is also reflected in speedup factors and parallel efficiency in Figures 6.8 and 6.9. Unlike in the previous scenario, this result is dominated by extreme abort rates approaching 100% which can be seen in Figure 6.10. Note that Midas*, despite committing almost no transactions, still shows some throughput at 32 cores because it has less synchronization overhead than Midas and saved time by not committing. Echo, on the other hand, has much lower abort rates. This results in significantly better transaction throughput at first, but, starting at 8 cores, throughput starts to decline rapidly as well.

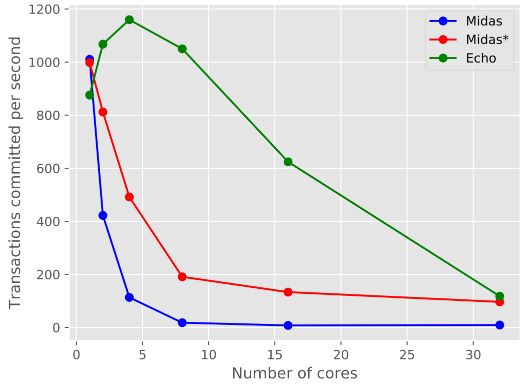


Figure 6.7: Transaction throughput for scenario S2.

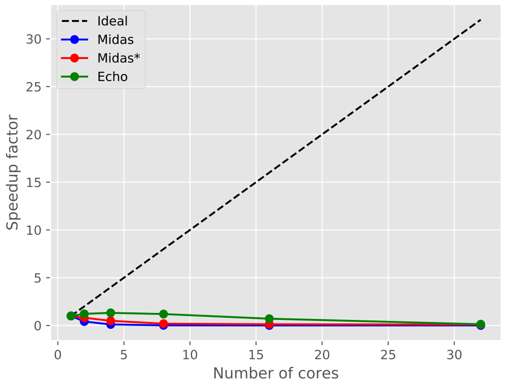


Figure 6.8: Transaction throughput speedup for scenario S2.

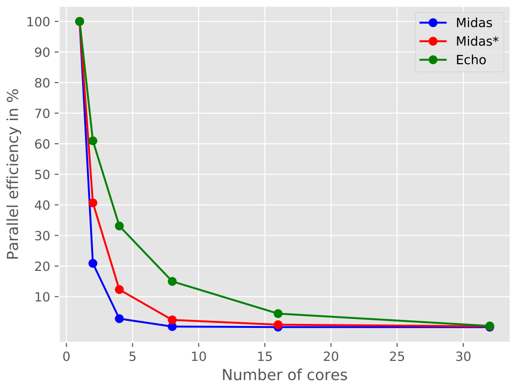


Figure 6.9: Parallel efficiency for scenario S2.

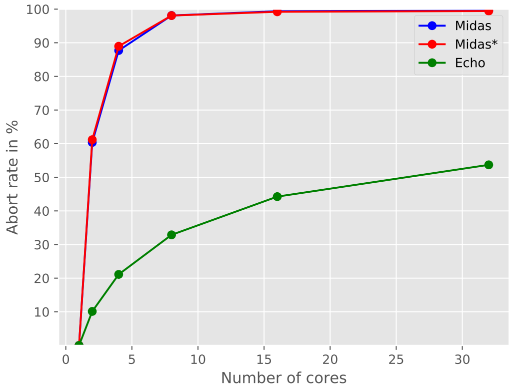


Figure 6.10: Abort rate for scenario S2.

Scenario S3 This scenario simulates low contention by means of a large database and short transactions. Based on the lower probability of conflicts, it is expected that Midas could be on a par with Echo. The plot in Figure 6.11 shows that transaction throughput in both Midas and Midas* is significantly higher than in Echo. While Midas is too limited by synchronization to benefit from additional cores, Echo and Midas* achieve close-to-ideal speedup until 4 cores (see Figure 6.12 and 6.13). Beyond that, Echo starts plunging at 8 cores whereas Midas* manages to improve until 16 cores, albeit slightly.

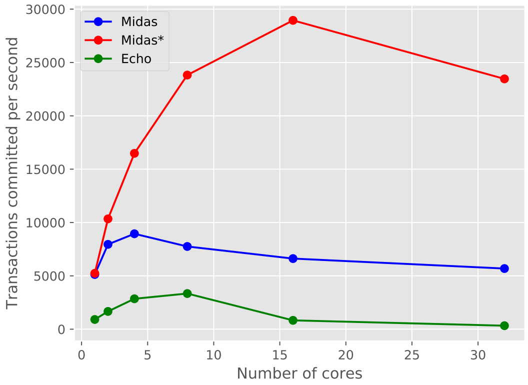


Figure 6.11: Transaction throughput for scenario S3.

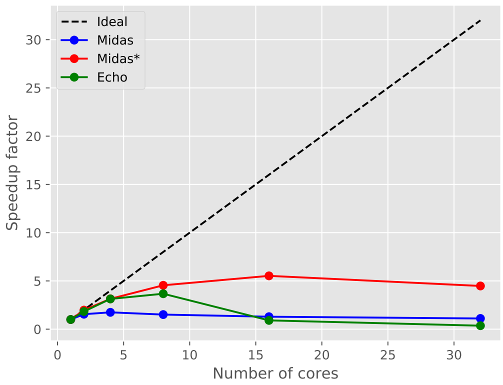


Figure 6.12: Transaction throughput speedup for scenario S3.

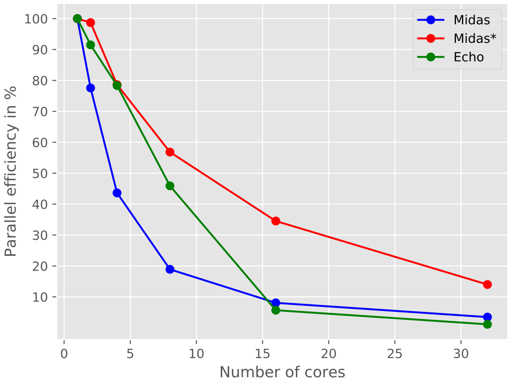


Figure 6.13: Parallel efficiency for scenario S3.

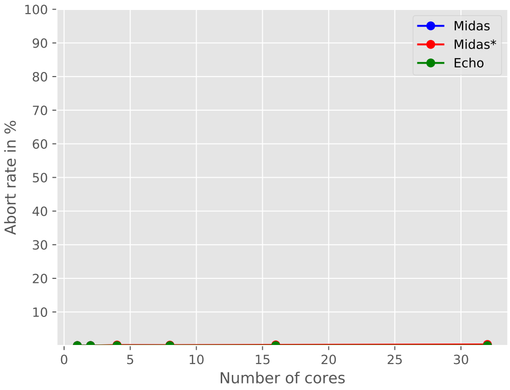


Figure 6.14: Abort rate for scenario S3.

This is a significant result, because it shows that serializable MVCC can still perform well for workloads where SI is known to excel at. After all, Midas* manages to surpass Echo by a large margin. However, Figure 6.14 shows that this scenario appears to be well-formed for strong isolation as abort rates are consistently below 1%. A side effect is that transaction throughput values cannot be distorted by time savings from aborted transactions. Also, the unsynchronized index in Midas* surely is too optimistic for an approximation of a concurrent index.

Scenario S4 The last scenario simulates medium contention by means of a large database accessed by long transactions. This scenario is similar to S3 but the longer transactions access the index more often and make aborts more likely, which is a disadvantage for Midas. This circumstance is confirmed in Figure 6.15 as Echo achieves higher transaction throughput than Midas. Also, Echo scales better than Midas which is depicted in Figures 6.16 and 6.17.

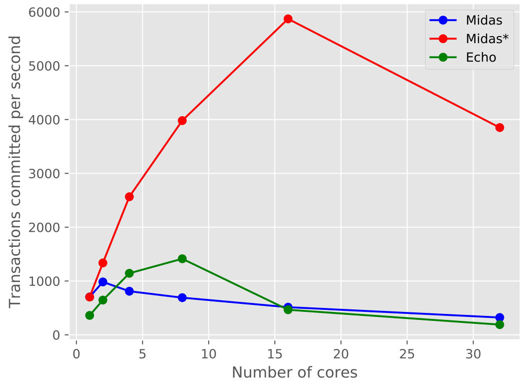


Figure 6.15: Transaction throughput for scenario S4.

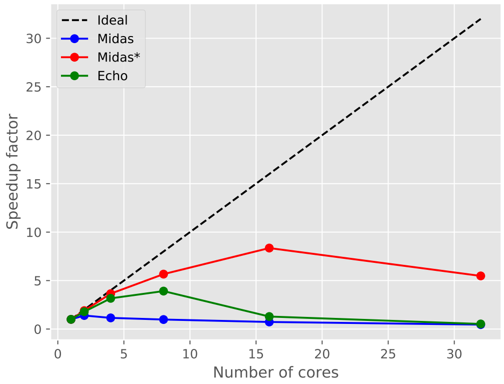


Figure 6.16: Transaction throughput speedup for scenario S4.

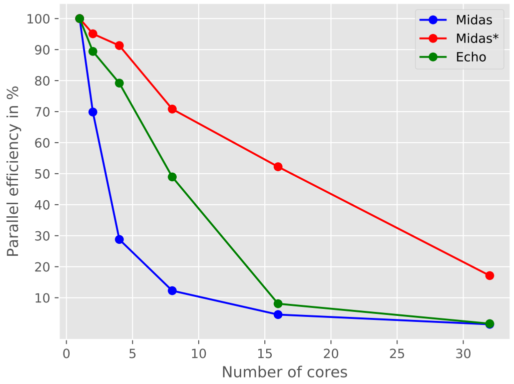


Figure 6.17: Parallel efficiency for scenario S4.

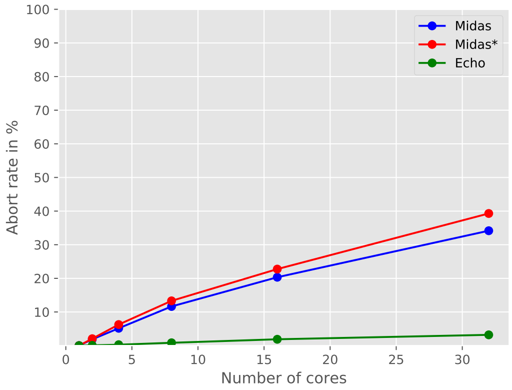


Figure 6.18: Abort rate for scenario S4.

However, this is largely attributed to synchronization overhead in Midas, as Midas* achieves up to 6 times higher throughput than Echo while it produces only slightly higher abort rates (see Figures 6.15 and 6.18). Note that Midas* shows even better speedups than in scenario S3, but that is most likely due to time savings from aborted transactions which are more numerous in S4. A troubling observation from Figure 6.18 is that, longer transactions alone can cause abort rates to scale in the number of cores for both variants of Midas. Independent of high abort rates, transaction throughput in Midas* suddenly declines rapidly at 32 cores. A possible explanation is that remaining synchronization points in Midas* incur too much overhead to be compensated by time savings from aborted transactions. Echo, on the other hand, has significantly fewer aborts but cannot exploit the advantage as it fails to leverage more than 8 cores efficiently.

6.4 Summary

In 3 out of 4 scenarios, the original implementation of Midas performs significantly worse than Echo. Especially in high-contention scenarios, it shows much higher abort rates, lower throughput, and insignificant speedup. However, at low contention, Midas consistently achieves higher throughput than Echo. This is a promising result as it shows that, even with serializability, MVCC can continue to excel in low-contention scenarios. However, despite negligible abort rates, which are on par with Echo, Midas shows poor scalability. This is due to the non-concurrent hash table implementation which requires a global lock for synchronization.

Since the concrete index implementation is not part of the synchronization concept, Midas* is used to approximate the performance of a lock-free concurrent hash table. Midas* has significantly higher throughput and scales well up until 16 cores, while maintaining comparable abort rates. However, this result may be too optimistic because concurrent data structures are often slower than their non-synchronized counterparts. Therefore, it is conceivable that Midas with a concurrent index may perform worse than Midas* but still better than Echo.

Echo has much lower abort rates in all scenarios, but only provides snapshot isolation. Moreover, it fails to leverage the advantage and hardly scales beyond 8 cores. Possible reasons for that include inappropriate synchronization and inefficient NVRAM management.

In total, the benchmark shows that, in certain scenarios, serializability may become affordable with NVRAM after all. Except in very high contention scenarios, Midas* can provide better throughput and scale better than Echo. Moreover, if contention is low, even Midas can perform better than its non-serializable counterpart Echo.

7 Conclusions and Future Work

Recent advances in non-volatile memory research indicate that fast high-density NVRAM is available in the near future. Later research found that novel NVRAM can significantly increase transaction throughput in main-memory databases. Still, for performance reasons, many databases resort to relaxed isolation of concurrent transactions. Therefore, this thesis investigates if NVRAM can provide sufficient leverage to make the isolation level of serializability affordable.

Due to the complexity of full-featured MMDB this work is focused entirely on key-value stores. Based on recent research, a concept for an NVRAM-resident key-value store with serializable transactions was developed. Subsequently, Midas, a prototype of the concept was implemented. Midas relies on MVCC and implements a serializable variant of Snapshot Isolation. Access to potential NVRAM and durable data structures is managed through PMDK, a state of the art library for programming against NVRAM.

In order to evaluate the concept, a synthetic benchmark over transaction throughput was developed. In the experiment, Midas is compared to Echo, another NVRAM-resident KVS but with non-serializable transactions. According to the results, Midas mostly performs much worse than Echo. Except at low contention, Midas has lower throughput, higher abort rates, and fails to utilize additional processors. However, it was found that the results can be largely attributed to an implementation flaw in Midas. A modified variant of Midas achieved higher throughput and better scalability than Echo at comparable abort rates. These are promising results and indicate that the concept of Midas may be worth pursuing.

Future Work There are some aspects of this work that should see further attention. First and foremost, additional experiments are needed to better evaluate the concept of powering serializability with NVRAM. For example, it should be investigated how Midas compares to other solutions, such as BerkeleyDB [OBS99], that support serializability but use disk storage for recovery. Ultimately, the concept of Midas could be evaluated for full-featured MMDB. Since many modern KVSs and MMDB, such as DynamoDB [DHJ$^+$07] or HANA

[LKF+13], are distributed systems, it would be interesting to see if the concept of Midas can be beneficial to distributed transactions.

On the implementation side, there is also room for improvement. First, the durable data structures in Midas are not concurrent and require locking which introduces undesirable bottlenecks. Therefore, future work should consider the design or adaptation of highly-concurrent data structures for NVRAM. A possibility would be to port existing implementations to NVRAM, such as the concurrent hash table *libcuckoo* [LAKF14].

Midas is based on MVCC which requires garbage collection to release versions that are no longer reachable by any transaction. The current design of Midas only performs garbage collection on startup to reduce resource contention during runtime. Therefore, it could be investigated how online garbage collection can be integrated in serializable MVCC with reasonable overhead.

A Acronyms

ADR	Asynchronous DRAM Self-Refresh
ANN	Artificial Neural Network
API	Application Programming Interface
ASLR	Address Space Layout Randomization
BPRAM	Byte-Addressable Persistent RAM
COW	Copy-On-Write
CPU	Central Processing Unit
DAX	Direct Access
DBMS	Database Management System
DIMM	Dual In-line Memory Module
DMA	Direct Memory Access
DRAM	Dynamic RAM
FPGA	Field-Programmable Gate Array
HDD	Hard Disk Drive
IO	Input/Output
KVS	Key-Value Store
MMDB	Main-Memory Database
MOB	Memory Order Buffer
MOB	Memory Order Buffer
MVCC	Multiversion Concurrency Control
NVDIMM	Non-Volatile DIMM
NVM	Non-Volatile Memory
NVRAM	Non-Volatile RAM
PCM	Phase-Change Memory
PM	Persistent Memory
RAID	Redundant Array of Independent Disks
RAM	Random Access Memory
RCU	Read-Copy-Update
SCM	Storage Class Memory
SI	Snapshot Isolation
SRAM	Static RAM
SSD	Solid State Drive
STT-RAM	Spin Torque Transfer RAM
WPQ	Write-Pending Queue

List of Figures

2.1 System architecture with PMDK to manage NVRAM [Rud17]. 11
2.2 Architecture of memory subsystem [BCB12]. 13
2.3 A sitation where only one of two cached stores reaches durability. 14
2.4 The memory domain to be protected against power failures [Rud17]. 15

3.1 Interleaved notation of a write-write conflict. 21
3.2 Projected version of Figure 3.1. 22
3.3 A write-write conflict resulting in a lost update for t_2. 24
3.4 A write-read conflict resulting in an erroneous update for t_1. 24
3.5 A read-write conflict resulting in an inconsistent read for t_1. 25
3.6 Write skew due to transactions t_1, t_2 not seeing each others changes. 32
3.7 Transaction t_{RO} is read-only but not serializable. 33

4.1 Simplified example of expected system architecture. 46
4.2 Transactional data flow in the two-level store. 48
4.3 Components of the store and their interconnections. 52

5.1 System architecture layers of Midas. 58
5.2 Class diagram for the class `NVList`. 62
5.3 Class diagram for the class `NVHashmap`. 64
5.4 Class diagram of a transaction control block. 65

6.1 Logical flow of the benchmark application. 80
6.2 Logical flow of a single worker thread. 80
6.3 Transaction throughput for scenario S1. 82
6.4 Transaction throughput speedup for scenario S1. 82
6.5 Parallel efficiency for scenario S1. 82
6.6 Abort rate for scenario S1. 82
6.7 Transaction throughput for scenario S2. 83
6.8 Transaction throughput speedup for scenario S2. 83

6.9 Parallel efficiency for scenario S2. 83
6.10 Abort rate for scenario S2. 83
6.11 Transaction throughput for scenario S3. 84
6.12 Transaction throughput speedup for scenario S3. 84
6.13 Parallel efficiency for scenario S3. 84
6.14 Abort rate for scenario S3. 84
6.15 Transaction throughput for scenario S4. 85
6.16 Transaction throughput speedup for scenario S4. 85
6.17 Parallel efficiency for scenario S4. 85
6.18 Abort rate for scenario S4. 85

List of Tables

3.1 Shorthands for transactional operations in schedule notations. 21

4.1 API of the intended key-value store. 44

5.1 An overview of all components and their underlying data structures. 67
5.2 An overview of all procedures provided by Midas. 68

6.1 Types of transactions in terms of length. 78
6.2 Scenarios are combinations of database size and transaction length. 79

Bibliography

[ALR$^+$17] Mihnea Andrei, Christian Lemke, Günter Radestock, Robert Schulze, Carsten Thiel, Rolando Blanco, Akanksha Meghlan, Muhammad Sharique, Sebastian Seifert, Surendra Vishnoi, et al. Sap hana adoption of non-volatile memory. *Proceedings of the VLDB Endowment*, 10(12):1754–1765, 2017.

[BBG$^+$95] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *ACM SIGMOD Record*, volume 24, pages 1–10. ACM, 1995.

[BC16] Hans-J Boehm and Dhruva R Chakrabarti. Persistence programming models for non-volatile memory. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, pages 55–67. ACM, 2016.

[BCB12] Kumud Bhandari, Dhruva R Chakrabarti, and Hans-J Boehm. Implications of cpu caching on byte-addressable non-volatile memory programming. *Hewlett-Packard, Tech. Rep. HPL-2012-236*, 2012.

[BCGL11] Katelin Bailey, Luis Ceze, Steven D Gribble, and Henry M Levy. Operating system implications of fast, cheap, non-volatile memory. In *HotOS*, volume 13, pages 2–2, 2011.

[BG81] Philip A Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

[BG83] Philip A Bernstein and Nathan Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.

[BHC$^+$13] Katelin A Bailey, Peter Hornyack, Luis Ceze, Steven D Gribble, and Henry M Levy. Exploring storage class memory with key value stores. In *Proceedings of* *the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, page 4. ACM, 2013.

[Bla91] Andrew P Black. Understanding transactions in the operating system context. *ACM SIGOPS Operating Systems Review*, 25(1):73–76, 1991.

[Car83] Michael J Carey. *Multiple versions and the performance of optimistic concurrency control.* Computer Sciences Department, University of Wisconsin, Madison, 1983.

[CCA+11] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM Sigplan Notices*, 46(3):105–118, 2011.

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[CLR+09] Philip Carns, Sam Lang, Robert Ross, Murali Vilayannur, Julian Kunkel, and Thomas Ludwig. Small-file access in parallel file systems. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–11. IEEE, 2009.

[CM86] Michael J Carey and Waleed A Muhanna. The performance of multiversion concurrency control algorithms. *ACM Transactions on Computer Systems (TOCS)*, 4(4):338–378, 1986.

[CM14] Damiano Carra and Pietro Michiardi. Memory partitioning in memcached: An experimental performance analysis. In *Communications (ICC), 2014 IEEE International Conference on*, pages 1154–1159. IEEE, 2014.

[CNC+96] Peter M Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The rio file cache: Surviving operating system crashes. In *Acm Sigplan Notices*, volume 31, pages 74–83. ACM, 1996.

[CNF+09] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin

Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 133–146. ACM, 2009.

[Cor] Intel Corp. Persistent memory development kit. `http://pmem.io/pmdk`. Retrieved Apr 2018.

[Cou16] R. Courtland. Can hpe's "the machine" deliver? *IEEE Spectrum*, 53(1):34–35, January 2016.

[CRF09] Michael J Cahill, Uwe Röhm, and Alan D Fekete. Serializable isolation for snapshot databases. *ACM Transactions on Database Systems (TODS)*, 34(4):20, 2009.

[DFI$^+$13] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1243–1254. ACM, 2013.

[DHJ$^+$07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[DKDG15] Bailu Ding, Lucja Kot, Alan Demers, and Johannes Gehrke. Centiman: Elastic, high performance optimistic concurrency control by watermarking. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 262–275. ACM, 2015.

[DKK$^+$14] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[Dor] Dormando. `https://www.memcached.org/`. Retrieved Sep 2017.

[Eic86] Margaret H Eich. Main memory database recovery. In *Proceedings of 1986 ACM Fall joint computer conference*, pages 1226–1232. IEEE Computer Society Press, 1986.

[FA15] Jose M Faleiro and Daniel J Abadi. Rethinking serializable multiversion concurrency control. *Proceedings of the VLDB Endowment*, 8(11):1190–1201, 2015.

[FAK13] Bin Fan, David G Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *NSDI*, volume 13, pages 385–398, 2013.

[FCP$^+$12] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. Sap hana database: data management for modern business applications. *ACM Sigmod Record*, 40(4):45–51, 2012.

[FFP16] Tobias Fiebig, Anja Feldmann, and Matthias Petschick. A one-year perspective on exposed in-memory key-value stores. In *Proceedings of the 2016 ACM Workshop on Automated Decision Making for Active Cyber Defense*, pages 17–22. ACM, 2016.

[FLO$^+$05] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems (TODS)*, 30(2):492–528, 2005.

[FNS$^+$16] Alessandro Fumarola, Pritish Narayanan, Lucas L Sanches, Severin Sidler, Junwoo Jang, Kibong Moon, Robert M Shelby, Hyunsang Hwang, and Geoffrey W Burr. Accelerating machine learning with non-volatile memory: exploring device and circuit tradeoffs. In *Rebooting Computing (ICRC), IEEE International Conference on*, pages 1–8. Ieee, 2016.

[FOO04] Alan Fekete, Elizabeth O'Neil, and Patrick O'Neil. A read-only transaction anomaly under snapshot isolation. *ACM SIGMOD Record*, 33(3):12–14, 2004.

[G$^+$81] Jim Gray et al. The transaction concept: Virtues and limitations. In *VLDB*, volume 81, pages 144–154, 1981.

[GFL$^+$] Manu Goyal, Bin Fan, Xiaozhou Li, David G Andersen, and Michael Kaminsky. libcuckoo - a high-performance, concurrent hash table. `https://github.com/efficient/libcuckoo`. Retrieved Feb 2018.

[GKP$^+$10] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. Hyrise: A main memory hybrid storage engine. *Proc. VLDB Endow.*, 4(2):105–116, November 2010.

[GmS92] Hector Garcia-molina and Kenneth Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 4:509–516, 1992.

[HJ14] Henry F Huang and Tao Jiang. Design and implementation of flash based nvdimm. In *Non-Volatile Memory Systems and Applications Symposium (NVMSA), 2014 IEEE*, pages 1–6. IEEE, 2014.

[HM93] Maurice Herlihy and J Eliot B Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM, 1993.

[HP86] Thanasis Hadzilacos and Christos H Papadimitriou. Algorithmic aspects of multiversion concurrency control. *Journal of Computer and System Sciences*, 33(2):297–310, 1986.

[HR83] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983.

[HR93] Theo Härder and Kurt Rothermel. Concurrency control issues in nested transactions. *The VLDB Journal—The International Journal on Very Large Data Bases*, 2(1):39–74, 1993.

[HSH$^+$08] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security Symposium*, 2008.

[JRRR17] Shubham Jain, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. Computing in memory with spin-transfer torque magnetic ram. *arXiv preprint arXiv:1703.02118*, 2017.

[JSG12] C. Jacobi, T. Slegel, and D. Greiner. Transactional memory architecture and implementation for ibm system z. In *2012 45th Annual IEEE/ACM International*

Symposium on Microarchitecture, pages 25–36, Dec 2012.

[Käs07] Christian Kästner. *Aspect-oriented refactoring of berkeley db.* PhD thesis, Master's thesis, University of Magdeburg, Germany, 2007.

[KKG+11] Jens Krueger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. Fast updates on read-optimized databases using multi-core cpus. *Proceedings of the VLDB Endowment*, 5(1):61–72, 2011.

[Kni86] Tom Knight. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 105–112. ACM, 1986.

[KPS+16] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. High-performance transactions for persistent memories. *ACM SIGPLAN Notices*, 51(4):399–411, 2016.

[KR81] Hsiang-Tsung Kung and John T Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[Lab] Redis Labs. `https://redis.io/`. Retrieved Sep 2017.

[LAC14] Maysam Lavasani, Hari Angepat, and Derek Chiou. An fpga-based in-line accelerator for memcached. *IEEE Computer Architecture Letters*, 13(2):57–60, 2014.

[LAKF14] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *EuroSys*, 2014.

[LBD+11] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M Patel, and Mike Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proceedings of the VLDB Endowment*, 5(4):298–309, 2011.

[LGG+91] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. *Replication in the Harp file system*, volume 25. ACM, 1991.

[LIMB09] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 2–13. ACM, 2009.

[LKF$^+$13] Juchang Lee, Yong Sik Kwon, Franz Färber, Michael Muehle, Chulwon Lee, Christian Bensberg, Joo Yeon Lee, Arthur H Lee, and Wolfgang Lehner. Sap hana distributed in-memory database system: Transaction, session, and metadata management. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 1165–1173. IEEE, 2013.

[LKN14] V. Leis, A. Kemper, and T. Neumann. Exploiting hardware transactional memory in main-memory databases. In *2014 IEEE 30th International Conference on Data Engineering*, pages 580–591, March 2014.

[LLL$^+$15] Sheng Li, Hyeontaek Lim, Victor W Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G Andersen, O Seongil, Sukhan Lee, and Pradeep Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 476–488. ACM, 2015.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[LMM$^+$13] Juchang Lee, Michael Muehle, Norman May, Franz Faerber, Vishal Sikka, Hasso Plattner, Jens Krueger, and Martin Grund. High-performance transaction processing in sap hana. *IEEE Data Eng. Bull.*, 36(2):28–33, 2013.

[Mal17] Artur Malinowski. Using redis supported by nvram in hpc applications. *Computer Science*, 18(3), 2017.

[MBL01] Ethan L Miller, Scott A Brandt, and Darrell DE Long. Hermes: High-performance reliable mram-enabled storage. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 95–99. IEEE, 2001.

[MBM+06] Michelle J Moravan, Jayaram Bobba, Kevin E Moore, Luke Yen, Mark D Hill, Ben Liblit, Michael M Swift, and David A Wood. Supporting nested transactional memory in logtm. In *ACM Sigplan Notices*, volume 41, pages 359–370. ACM, 2006.

[McK07] Paul E McKenney. Memory ordering in modern microprocessors. *Interface*, 6:6, 2007.

[MGA16] Leonardo Mármol, Jorge Guerra, and Marcos K Aguilera. Non-volatile memory through customized key-value stores. In *HotStorage*, 2016.

[MH06] J Eliot B Moss and Antony L Hosking. Nested transactional memory: model and architecture sketches. *Science of Computer Programming*, 63(2):186–201, 2006.

[Mos81] John Eliot Blakeslee Moss. Nested transactions: An approach to reliable distributed computing. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE, 1981.

[Mos06] J Eliot B Moss. Open nested transactions: Semantics and support. In *Workshop on Memory Performance Issues*, volume 28, 2006.

[MS93] Jim Melton and Alan R Simon. *Understanding the new SQL: a complete guide.* Morgan Kaufmann, 1993.

[MS98] Paul E McKenney and John D Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[MTV02] Nick Murphy, Mark Tonkelowitz, and Mike Vernal. The design and implementation of the database file system, 2002.

[MV16] Sparsh Mittal and Jeffrey S Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1537–1550, 2016.

[MWMS14] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory oltp recovery. In *2014 IEEE 30th International Conference on Data Engineering*, pages 604–615, March 2014.

[NH12] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. *ACM SIGARCH Computer Architecture News*, 40(1):401–410, 2012.

[NHH+17] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. An analysis of persistent memory use with whisper. *SIGOPS Oper. Syst. Rev.*, 51(2):135–148, April 2017.

[NMK15] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 677–689. ACM, 2015.

[OBL+14] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. Sofort: A hybrid scm-dram storage engine for fast data recovery. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, page 8. ACM, 2014.

[OBS99] Michael A Olson, Keith Bostic, and Margo I Seltzer. Berkeley db. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191, 1999.

[OL17] Ismail Oukid and Wolfgang Lehner. Data structure engineering for byte-addressable non-volatile memory. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1759–1764. ACM, 2017.

[OLK+15] Ismail Oukid, Wolfgang Lehner, Thomas Kissinger, Thomas Willhalm, and Peter Bumbulis. Instant recovery for main memory databases. In *CIDR*, 2015.

[ONO16] K. Oe, T. Nanri, and K. Okamura. Feasibility study for building hybrid storage system consisting of non-volatile dimm and ssd. In *2016 Fourth International Symposium on Computing and Networking (CANDAR)*, pages 454–457, Nov 2016.

[Oraa] Oracle. Getting started with berkeley db. `http://docs.oracle.com/cd/E17076_05/html/gsg/C/index.html`. Retrieved Apr 2017.

[Orab] Oracle. Java® platform, standard edition version 10 - api specification. `https://docs.oracle.com/javase/10/docs/api/java/lang/String.html#hashCode()`. Retrieved Apr 2018.

[PCW14] Steven Pelley, Peter M Chen, and Thomas F Wenisch. Memory persistency. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 265–276. IEEE, 2014.

[PHR$^+$09] Donald E Porter, Owen S Hofmann, Christopher J Rossbach, Alexander Benn, and Emmett Witchel. Operating system transactions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 161–176. ACM, 2009.

[PWGB13] Steven Pelley, Thomas F Wenisch, Brian T Gold, and Bill Bridge. Storage management in the nvram era. *Proceedings of the VLDB Endowment*, 7(2):121–132, 2013.

[Ree78] David Patrick Reed. *Naming and synchronization in a decentralized computer system.* Massachusetts Institute of Technology, 1978.

[RO92] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[Rud17] Andy Rudoff. Persistent memory programming. *; login: The Usenix Magazine*, 42(2), 2017.

[SBD$^+$16] David Schwalb, Girish Kumar BK, Markus Dreseler, S Anusha, Martin Faust, Adolf Hohl, Tim Berning, Gaurav Makkar, Hasso Plattner, and Parag Deshmukh. Hyrise-nv: Instant recovery for in-memory databases using non-volatile memory. In *International Conference on Database Systems for Advanced Applications*, pages 267–282. Springer, 2016.

[SCB$^+$14] Mohammad Sadoghi, Mustafa Canim, Bishwaranjan Bhattacharjee, Fabian Nagel, and Kenneth A Ross. Reducing database locking contention through multi-version concurrency. *Proceedings of the VLDB Endowment*, 7(13):1331–1342, 2014.

[SFW$^+$15] David Schwalb, Martin Faust, Johannes Wust, Martin Grund, and Hasso Plattner. Efficient transaction processing for hyrise in mixed workload environments. In *In Memory Data Management and Analysis*, pages 112–125. Springer, 2015.

[SGC$^+$09] Richard P Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P Wright. Enabling transactional file access via lightweight kernel extensions. In *Proccedings of the 7th conference on File and storage technologies*, pages 29–42. USENIX Association, 2009.

[SM09] Margo I Seltzer and Nicholas Murphy. Hierarchical file systems are dead. In *HotOS*, 2009.

[Spi09] Diomidis Spinellis. User-level operating system transactions. *Software: Practice and Experience*, 39(14):1215–1233, 2009.

[SS90] Margo I Seltzer and Michael Stonebraker. Transaction support in read optimized and write optimized file systems. In *VLDB*, pages 174–185, 1990.

[ST97] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[Ste05] Lex Stein. Stupid file systems are better. In *HotOS*, 2005.

[SXH$^+$10] Liang Shi, Chun Jason Xue, Jingtong Hu, Wei-Che Tseng, Xuehai Zhou, and Edwin H.-M. Sha. Write activity reduction on flash main memory via smart victim cache. In *Proceedings of the 20th Symposium on Great Lakes Symposium on VLSI*, GLSVLSI '10, pages 91–94, New York, NY, USA, 2010. ACM.

[Twe98] Stephen C Tweedie. Journaling the linux ext2fs filesystem. In *The Fourth Annual Linux Expo*, 1998.

[TZK$^+$13] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32. ACM, 2013.

[VNP$^+$14] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, page 14. ACM, 2014.

[VTR+11] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, Roy H Campbell, et al. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, volume 11, pages 61–75, 2011.

[VTS11] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight persistent memory. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 91–104. ACM, 2011.

[WAL+17] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment*, 10(7):781–792, 2017.

[WBR+12] Johannes Wust, Joos-Hendrick Boese, Frank Renkes, Sebastian Blessing, Jens Krueger, and Hasso Plattner. Efficient logging for enterprise workloads on column-oriented in-memory databases. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, pages 2085–2089, New York, NY, USA, 2012. ACM.

[WJFP17] Tianzheng Wang, Ryan Johnson, Alan Fekete, and Ippokratis Pandis. Efficiently making (almost) any concurrency control mechanism serializable. *The VLDB Journal*, 26(4):537–562, 2017.

[WNZ+16] Xingbo Wu, Fan Ni, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, Zili Shao, and Song Jiang. Nvmcached: An nvm-based key-value cache. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, page 18. ACM, 2016.

[WR11] Xiaojian Wu and A. L. Narasimha Reddy. Scmfs: A file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 39:1–39:11, New York, NY, USA, 2011. ACM.

[WRPK02] An-I Wang, Peter L Reiher, Gerald J Popek, and Geoffrey H Kuenning. Conquest: Better performance through a disk/persistent-ram hybrid file system. In *USENIX*

Annual Technical Conference, General Track, pages 15–28, 2002.

[WS92] Gerhard Weikum and Hans-Jörg Schek. Concepts and applications of multilevel transactions and open nested transactions, 1992.

[WSSZ07] Charles P Wright, Richard Spillane, Gopalan Sivathanu, and Erez Zadok. Extending acid semantics to the file system. *ACM Transactions on Storage (TOS)*, 3(2):4, 2007.

[WZ94] Michael Wu and Willy Zwaenepoel. envy: A non-volatile, main memory storage system. *SIGPLAN Not.*, 29(11):86–97, November 1994.

[WZT^{+}15] Yandong Wang, Li Zhang, Jian Tan, Min Li, Yuqing Gao, Xavier Guerin, Xiaoqiao Meng, and Shicong Meng. Hydradb: A resilient rdma-driven key-value middleware for in-memory cluster computing. In *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*, pages 1–11. IEEE, 2015.

[XFJ14] Yuehai Xu, Eitan Frachtenberg, and Song Jiang. Building a high-performance key-value cache as an energy-efficient appliance. *Performance Evaluation*, 79:24–37, 2014.

[XFJP14] Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Characterizing facebook's memcached workload. *IEEE Internet Computing*, 18(2):41–49, 2014.

[XS16] Jian Xu and Steven Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *FAST*, pages 323–338, 2016.

[YADA17] Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Reetuparna Das, and Todd M. Austin. Cold boot attacks are still hot: Security analysis of memory scramblers in modern processors. *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 313–324, 2017.

[ZSLH16] Jie Zhou, Yanyan Shen, Sumin Li, and Linpeng Huang. Nvht: an efficient key-value storage library for non-volatile memory. In *Big Data Computing Applications and Technologies (BDCAT), 2016 IEEE/ACM 3rd International Conference on*, pages 227–236. IEEE, 2016.

[ZWT13] Omer Zilberberg, Shlomo Weiss, and Sivan Toledo. Phase-change memory: An architectural perspective. *ACM Computing Surveys (CSUR)*, 45(3):29, 2013.

[ZZYT17] Xuan Zhou, Xin Zhou, Zhengtai Yu, and Kian-Lee Tan. Posterior snapshot isolation. In *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*, pages 797–808. IEEE, 2017.