# Fine-grained Transactions for NVRAM

Von der Fakultät MINT - Mathematik, Informatik, Physik, Elektro- und Informationstechnik der Brandenburgischen Technischen Universität Cottbus-Senftenberg genehmigte Dissertation zur Erlangung des akademischen Grades eines

Doktor der Ingenieurwissenschaften
(Dr.-Ing.)

vorgelegt von

Master of Science (M.Sc.)

Jana Traue

geboren am 20.05.1985 in Cottbus

| | |
|---|---|
| **Vorsitzender:** | Prof. Dr. H. T. Vierhaus |
| **Gutachter:** | Prof. Dr. J. Nolte |
| **Gutachter:** | Prof. Dr. W. Schröder-Preikschat |
| **Gutachter:** | Prof. Dr. A. Polze |

**Tag der mündlichen Prüfung:** 27.09.2018

# Abstract

For decades, there has been a distinction between fast, volatile and slow, non-volatile memory technologies in the storage hierarchy of computer systems. While volatile memories offer a byte addressable interface, persistence was limited to slower, block-oriented media. As a consequence, processing and storing information implies copying data from the persistent storage to volatile memory and vice versa.

In recent years, a new memory technology which is both byte-addressable and non-volatile has been announced: Non-Volatile Random Access Memory (NVRAM). Its unique combination of features allows processing of information exactly at the location where it is stored permanently, thereby eliminating the need for copies. Without the need to create and manage copies, data can be processed significantly faster.

The processing of persistent information can be disrupted by failures, like crashes. When changes are only partially performed, data might be left in an inconsistent state. Such a situation can be avoided by creating a copy before starting to change data. That backup is restored when a failure has been detected. Such a procedure relies on a highly specific order of its operations, because the backup can only be disposed after all changes have been carried out successfully. Since NVRAM removes the need for accessing block-oriented media, it removes a major bottleneck in the processing of persistent information, especially when only small amounts of data are modified. As a result, controlling the order of operations dominates the processing performance.

This thesis analyses transactional mechanisms for data on NVRAM which can guarantee that sequences of operations are either carried out as one unit or not at all. The idea is based on creating backups, as sketched above. Existing approaches for the traditional storage hierarchy use separate memory locations for backup and modified data. In a first step, the costs of applying existing procedures to NVRAM are identified. Afterwards, a novel approach which couples the versions tighter in space is presented. Enabled by the unique properties of NVRAM, this new approach is able to reduce the costs of ordering. The processing of persistent data can thereby be sped up significantly.

# Zusammenfassung

In der Speicherhierarchie von Computersystemen existiert seit Jahrzehnten eine Trennung zwischen schnellen, flüchtigen und langsamen, nicht-flüchtigen Speichermedien. Im Gegensatz zu Speichern der ersten Kategorie, welche byteweise adressierbar sind, bieten nicht-flüchtige Speicher bisher nur eine block-orientierte Schnittstelle. Infolge dessen sind bei der Verarbeitung und Speicherung von Informationen Kopieroperationen essentiell.

In den letzten Jahren wurde eine neue Speichertechnologie angekündigt, welche die bisher getrennten Welten vereint: Non-Volatile Random Access Memory (NVRAM). Dadurch, dass NVRAM zugleich byteweise adressierbar und nicht-flüchtig ist, ermöglicht er es, Daten an genau dem Ort zu verändern, an dem sie dauerhaft gespeichert werden. Ohne die Notwendigkeit von Kopieroperationen können persistente Daten im NVRAM hochgradig effizient verarbeitet werden.

Bei der Informationsverarbeitung können Fehler auftreten, beispielsweise wenn das System abstürzt. Sollten zum Zeitpunkt der Fehlers Änderungen nur teilweise ausgeführt worden sein, können die Daten einen ungültigen Zwischenstand repräsentieren. Hat man vor der Änderung eine Sicherungskopie angelegt, so könnte man nach der Erkennung eines solchen Zustandes zum alten Stand zurückkehren indem man die modifizierten Daten verwirft. Dabei ist es von enormer Wichtigkeit, dass die zeitliche Reihenfolge bei der Verwaltung der Kopie eingehalten wird: die alten Daten dürfen erst dann verworfen werden, wenn die neuen aktiv sind. Während bisher der Zugriff auf das nicht-flüchtige Medium die Verarbeitungsgeschwindigkeit dominiert hat, gewinnt mit NVRAM die Einhaltung von Reihenfolgen an Bedeutung.

In dieser Arbeit werden Transaktionsmechanismen für NVRAM untersucht, mit denen sichergestellt werden kann, dass Operationen ganz oder gar nicht ausgeführt werden. Als Grundlage dient dabei auch die Idee einer Sicherungskopie. Es werden zuerst die Kosten untersucht, die enstehen wenn das existierende Vorgehen auf NVRAM übertragen wird und Kopie und modifizierte Daten räumlich voneinander getrennt werden. Danach wird ein neuer Ansatz vorgestellt, der beide Versionen örtlich koppelt, was erst durch die einzigartige Kombination der Eigenschaften von NVRAM möglich wird. Diese Arbeit zeigt, dass mit dem neuen Ansatz die Kosten der Kontrolle von Reihenfolgen deutlich reduziert werden können, wodurch die Verarbeitungsgeschwindigkeit von persistenten Daten im NVRAM gesteigert werden kann.

# Contents

CHAPTER 1

# Introduction

Today's computers operate on two central types of memory. One is fast and allows for addressing of individual bytes. The other one is slow and features a more coarse grained interface, requiring the transfer of multiple bytes at once. A more widely known difference is that the former is volatile and loses its contents when disconnected from a power source. The latter one, instead, is non-volatile. Data stored on non-volatile media continues to exist even when disconnecting the system from a power supply. Tapes, disks, and flash drives are prominent instances of non-volatile memory. A processor is typically unable to read or write a single byte on these devices.

Commodity processors operate directly on byte-addressable memory, like registers, caches, and main memory. Data residing on other memories has to be transferred to one of the byte-addressable memories before processing. Transfers of data from one memory type to another one are nearly transparent to users, except for timing delays. In one scenario common these days, users experience the existence of transfers between different memory types. This scenario is explained next.

Modern mobile devices, like notebooks or tablets, are not turned off anymore. Instead, they are suspended, e.g., by closing the lid. Such a suspend puts the device into a low power state in which it may remain for a couple of days. Without such an energy saving mode, the runtime would be significantly shorter. When the user activates the device, for example by opening the lid, the old state is reactivated and everything is restored to its previous state. The reactivation includes the states of all processes and it may seem to a user like the system continued running for the whole time. To provide this illusion, the system copies all information from volatile memories to non-volatile media on suspend. It can then disconnect main memory from the power supply, resulting in enormous energy savings. The transfer between volatile and non-volatile memories is more noticeable when the system resumes. Continuing is only possible when essential information is back in volatile memory. As shown in Figure 1.1, latencies of non-volatile and volatile memories differ in orders of magnitude. Users experience this difference in latency when waiting for the system to be in a responsive state again, especially when gigabytes of main memory have to be restored.

In recent years, there has been an increasing number of research articles on the development of new memory technologies [24, 28, 51, 23, 33]. Although the technology details vary greatly, all promise a new memory technology which is non-volatile and byte-addressable at the same time: Non-volatile Random-Access-Memory (NVRAM). It can be seen as main memory which preserves its contents even when the power is turned off. Due to its combination of byte-addressability and non-volatility,

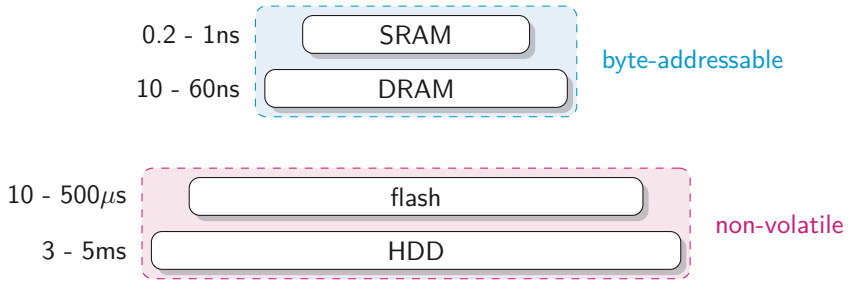



**Figure 1.1:** Today's layers of the storage hierarchy. Next to each layer are typical latencies, according to [25, 43, 58, 59, 61, 63, 65]. Notice the enormous latency gap between the upper, volatile memories and the lower, non-volatile memories.

NVRAM forms a new layer in the storage hierarchy, filling the gap between the byte-addressable and non-volatile worlds (see Figure 1.2). When suspending or resuming a system, as discussed before, there is no need to transfer gigabytes of main memory contents to another memory anymore, because main memory is already non-volatile.

Memories on layers on top of Dynamic Random-Access-Memory (DRAM) and NVRAM will continue to exist for a long time. Most importantly because NVRAM's latency is predicted to be close to DRAM, but caches and registers have to be way faster. The existence of these higher layers implies that volatile memories remain in future systems and there is still a need to copy their contents to a non-volatile memory when the power is turned off. Similar to suspend and resume, the transfers can be triggered lazily, exactly at the point when the power runs out. There is an enormous difference of the two settings with respect to failures. Imagine that you are editing a text document on a mobile device. After initiating a suspend, you experience unexpected behavior as a result of a failure in the software. Your system does not finish suspending, can not resume, and needs to restart. When you open your text document afterwards, you find the most recent state which was written to disk and, luckily, only recent changes are lost. With NVRAM, the suspend path becomes enormously more critical. The new technology eliminates the need for traditional non-volatile media, like disks. Consequently, an old version of data on disks does not necessarily exist anymore. A failure in the

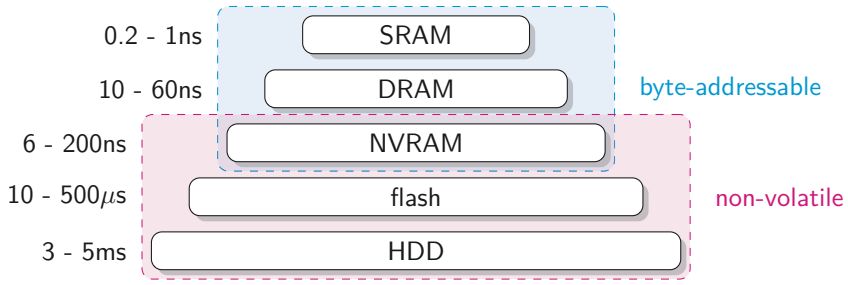



**Figure 1.2:** NVRAM forms a new layer in the storage hierarchy. Unlike any existing technology, it is byte-addressable *and* non-volatile at the same time. Projected latency for NVRAM ranges from 6 ns [59] to 200 ns [16].

process of writing data from volatile to non-volatile memories can not be recovered by restoring that old state. The failure may result in severe data loss. Another difference is that a suspend is commonly triggered in a non-critical situation, when enough energy remains to complete data transfers. In the case of a sudden power outage, there may not be sufficient capacity to transfer gigabytes of data to non-volatile disks. Suspend is therefore not an option for unexpected power loss. Although NVRAM may replace memory and thereby reduce the size of volatile memories significantly, modern systems may still have insufficient energy capacities to transfer their contents in the case of an unforeseeable power outage. Again, data loss can be a consequence.

An alternative to the reactive, lazy approach of preserving contents of volatile memories when the system fails is proactive copying. The system writes contents of volatile memories to non-volatile locations during normal operation, anticipating failures during that process. This scheme requires programmer support for detecting adequate points in time when such a transfer is appropriate. Think of a banking system transferring money. It can start by withdrawing the amount from the first account, then depositing it at the destination. Preserving the intermediate state where the money is in neither accounts, is inappropriate. A common solution is to save the old state prior to any of the modifications. If a failure happens while the money transfer is processed, the system reverts to that old state. Once all modifications are complete, the old state can be disposed as the system should continue with the new state even after subsequent failures. As a result, the modifications have transactional semantics: they are either carried out completely or not at all. This thesis focuses on efficient transaction mechanisms for NVRAM.

To get a first impression of the idea presented in this thesis, imagine you are organizing a home party. You bought two crates of drinks: one filled with water, the other one with beer. These two crates have to be carried to your apartment on the fourth floor. The lack of an elevator and your constitution prevent you from carrying more than one crate at a time. To get both up to the party location, you have to take the long stairways twice. If your party is small enough, you could mix the two crates, half water and half beer, carry only one and leave the other in the basement. If your guests drink an equal amount of both drinks and not too many in total, you saved one trip. Otherwise, if your prediction was off, you have to go down and up a second time as if you had not mixed the crates. The first approach is similar to common strategies for preserving the consistency of non-volatile data. Taking the steps resembles a transfer of data to NVRAM. NVRAM is the slowest layer in the new storage hierarchy when traditional non-volatile media is removed. A transfer from volatile memories to NVRAM is therefore time consuming. Existing strategies have to access NVRAM twice while processing: at first to preserve the old state, then to preserve the new state. This thesis investigates the opportunities of the second approach which mixes two types of information and stores old and new versions of data in the same location – in one crate.

The idea of using transactions has existed for a long time. Transactions are an essential part of databases, but also common in off-the-shelf computer systems which rely on disks. When transferring data to a byte-addressable memory for modifications, an old version remains on the non-volatile medium. Thereby, the two versions are inherently separated in space and can not be stored in the

same memory location. Recent work on transactions adapts ideas based on this spatial separation to NVRAM [16, 7, 29, 38]. Storing both versions close to each other in order to spare data transfers is a unique possibility offered by the new technology and has not been proposed before.

## 1.1 Outline

Due to its distinctive features, the variety of research projects for NVRAM is immense. First, several technology candidates exist, but none of them has reached the market yet. There are also different approaches towards the integration of NVRAM into today's and future systems. Chapter 2 presents an overview on recent work on NVRAM and shows that many projects present results for adaption of existing ideas to the new technology. Visionary and novel ideas, however, have gained less attention.

Transactions on NVRAM are the key aspect of this thesis. They are being used to preserve consistency of data even in the case of failures. Projects focusing on the challenges of consistency on NVRAM are heavily discussing changes to processor and cache designs. Hence, their solutions require NVRAM users to buy additional new hardware. The requirement of further hardware, which has not yet been agreed on, may inhibit the rise of NVRAM. This thesis follows a different approach and presents solutions for challenges arising in the near future. A detailed description of these settings and the deficits of existing approaches are given in Chapter 3.

Chapter 4 presents the central idea of this work: transactions based on cache line granularity. At first, a basic version is explained and use cases are discussed. The discussion reveals similarities of the new scheme and consistency preserving strategies for concurrent settings. However, comparing these approaches shows significant differences. In its pure form, the cache line transaction scheme is applicable in only limited settings. Extensions for overcoming these limitations conclude this chapter.

As already mentioned, NVRAM hardware is not yet available on the market. That situation presents a challenge for the implementation and evaluation of the cache line transaction mechanisms. Chapter 5 is concerned with the integration of approaches for NVRAM into existing systems. The foundation used by this work is a persistent memory framework. Its presentation is followed by a discussion of key aspects for implementing cache line transactions.

The implementation presented in Chapter 5 is used to compare cache line transactions to existing transaction schemes in the sixth chapter. An analysis of different settings shows that cache line transactions can indeed reduce the number of costly memory transfer operations. As a consequence, all the experiments result in a significantly reduced runtime when traditional transaction mechanisms are replaced by cache line transactions.

The contributions of this work are summarized in the final chapter. In addition to a presentation of the benefits of cache line transactions, it contains also critical reasoning of investigations that should be subject of future work.

CHAPTER 2

# The Many Faces of NVRAM Research

Despite the fact that Non-volatile Random-Access-Memory (NVRAM) is still a young topic, it spawned a variety of research areas. Many of them came to life because researchers envision ideas for a technology that they can not use, yet. Consequently, they specify their vision of future systems and explain the challenges they see. They also have to find ways to validate the feasibility of their ideas and often describe their methodology in detail. The different topics of NVRAM research are therefore tightly coupled. Even solutions for very small niches face the same challenges as other projects and need to take their ideas into account. The same applies for this thesis and therefore this chapter provides a short overview about the many faces of NVRAM research.

NVRAM research can be grouped as follows: **Technology** covers research that comes up with new memory technologies and is therefore focused on material science. **System Integration** contains ideas for integrating NVRAM into existing software and hardware interfaces. **Evaluation/Verification** contains ideas about how to validate ideas for a technology that is not yet available. **Visions** is about questioning whether existing concepts will still be relevant in a future with NVRAM.

Figure 2.1 provides a visual representation of these categories. Each leaf represents a research paper that covers an aspect of its parent category. A leaf's color indicates whether the main contribution of this paper is in its parent category. This picture underlines the claim that the fields are tightly coupled. Nearly all papers have to contain an evaluation on the not yet existing NVRAM and therefore describe their idea of an evaluation system, but it is not their most important contribution. The next sections describe the research categories in more detail and elaborates on the references shown in the figure.

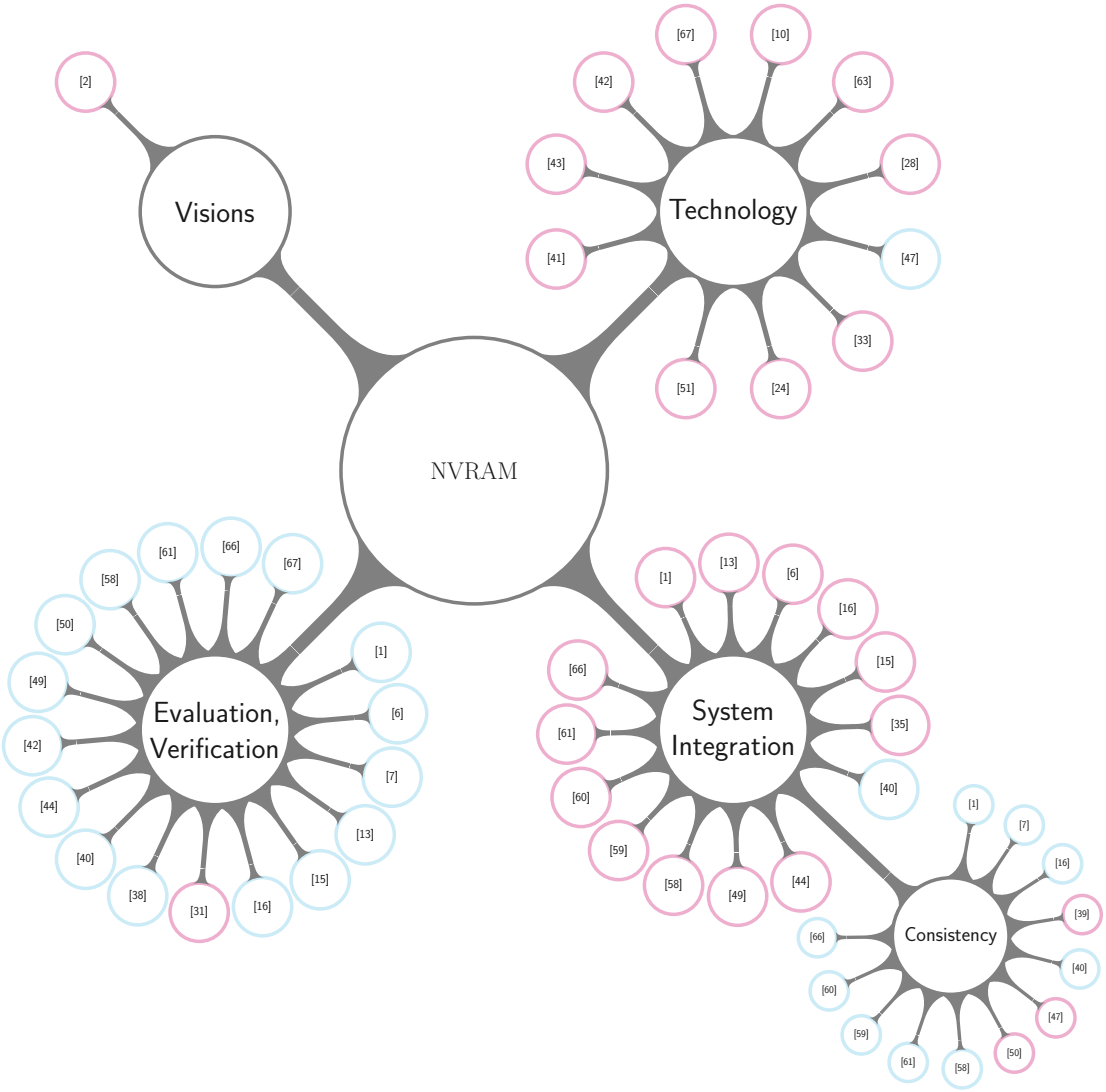



**Figure 2.1:** NVRAM research areas. References in a leaf contribute to the parent node's category. Magenta marks the major contribution of this reference.

## 2.1 Non-volatile Memory Technologies

Today's active NVRAM explorations were enabled by the discovery of new materials and effects based on their combination. These results raised hopes that non-volatile memories will be available some day. The technology aspect of NVRAM consists of the actual technology development, challenges, and opportunities.

**Memory technology** Research in the field of memory technology is actively ongoing since the "perfect" technology has not been found yet. Unfortunately, material science for non-volatile memories requires a detailed understanding of physics and chemistry and I am far from being an expert in any of these areas[1]. Hence, I can only sketch out the basics.

The common idea behind the emerging memory technologies is that a binary state can be represented by different resistance values of a memory cell [41]. A change of the resistance is triggered with a current pulse which causes different reactions. In the case of Ferroelectric RAM (FeRAM), the polarization of the material changes. For Magnetic RAM (MRAM), the magnetic orientation of one of two plates changes. Phase-Change Memory (PCM) [51][2] technologies use a material with two states: amorphous and crystalline. The current is used to heat the material and, depending on the temperature, one of the two states is activated. Resistive RAM (RRAM) [33] is based on two oxide layers and oxide ions migrating between them.

Some papers use other terms than the ones shown above. For example, Spin-Transfer-Torque MRAM (STT-MRAM) [24] is new, more compact form of MRAM. RRAM is based on the same basic principle as Memristors [23]. Table 2.1 shows my understanding of the basic principle behind the frequently used names.

In order to gain an understanding of the characteristics of non-volatile memory technologies, I collected latency numbers from articles that noted them down explicitly. Not all of these articles are primary sources, but the spreading of reported latencies shows that not all papers base their information on a

[1] Interestingly, even specialists are still uncertain why some of the mechanisms work, indicated by phrases like 'This effect is still not completely understood but is attributed to [...]' [51] and 'The nature of the switching phenomenon is believed to be due to [...]' [33]

[2] [28] use the acronym PRAM for Phase Change Random Access Memory

| Principle | Acronyms |
|---|---|
| Polarization | FeRAM |
| Magnetism | MRAM, STT-RAM, STT-MRAM |
| Phase Change | PCM, PRAM |
| Oxide Ion Migrations | RRAM, Memristor |

**Table 2.1:** Acronyms grouped by their basic principle for non-volatile memory technologies.

single source. In order to avoid a mix-up caused by the acronyms, I did not group the technologies and used the same term as the paper did. The resulting references can be found in Table 2.2.

A visual overview of the reported numbers of these articles is given in Figure 2.2. Despite the fact that the technologies base on different physical and chemical effects, they result in similar latencies which are close to DRAM's. The important consequence is that DRAM might be a suitable replacement for NVRAM when it comes to performance evaluations.

**Challenges** When a new technology is developed, it is typically not perfect in the first place. The limited endurance is one of NVRAM's most important challenges. If it is going to be used as main memory, the number of read and write accesses will be high. Wear leveling may help to avoid a single memory cell being accessed too frequently, resulting in an improved lifetime. Software for NVRAM should either perform wear leveling, rely on underlying software layers to perform this task, or assume that it will be performed by the hardware itself.

In addition to limited endurance, scalability is a challenge in the context of NVRAM [67, 63, 10] and it should be possible to build more dense memories than today.

**Opportunities** The most important feature of the new memory technologies in this work's context is non-volatility. Aside from preserving data, NVRAM offers other interesting opportunities. In contrast to DRAM, NVRAM does not need to be refreshed periodically. However, the energy required to read and write information will probably be higher. Whether it is possible to reduce the energy consumption of a system by replacing DRAM with NVRAM is investigated by Meza et al. [42]. In [67], Zhou et al. also discuss the possible energy savings and consider the overhead of wear leveling with respect to the energy consumption. Their newly developed memory controller increases the lifetime of NVRAM from 126 days to 22 years while consuming only 65% of DRAM's energy for the same workload.

| Technology | Min. Latency (ns) | Max. Latency (ns) | Sources |
|---|---|---|---|
| HDD | 3000000.0 | 5000000.0 | [43, 58, 63] |
| Flash | 50.0 | 1000000.0 | [9, 16, 25, 43, 58, 61, 63, 65, 65] |
| DRAM | 10.0 | 60.0 | [9, 16, 25, 43, 58, 59, 61, 63] |
| SRAM | 0.2 | 1.0 | [63, 65] |
| PCM | 10.0 | 120000.0 | [3, 9, 16, 25, 32, 43, 58], |
| | | | [59, 61, 63, 65, 28] |
| Memr. | 100.0 | 100.0 | [9, 58] |
| STT-RAM | 6.0 | 50.0 | [43, 59, 61, 65] |
| RRAM | 10.0 | 300.0 | [16, 43, 61, 65] |

**Table 2.2:** Latency numbers for different memory technologies.

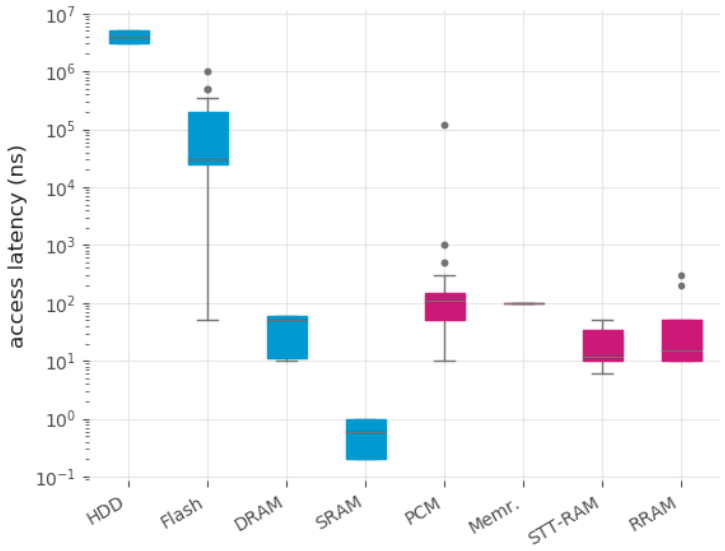



**Figure 2.2:** Access latencies according to literature. NVRAM technologies are depicted in magenta. Note the logarithmic scale on the y-axis.

Based on the above overview of the field of technology research, I conclude that DRAM can be used as a substitute for NVRAM for performance analysis. I expect wear leveling to be performed by the hardware and considering energy consumption is out of the scope of this work.

## 2.2 System Integration

Once NVRAM becomes available, it raises the question of how to integrate it. Is it compatible to any existing technology or do we need completely new hardware? The same applies for the software side: Can we continue to use existing software or do we have to develop new systems from scratch? The upcoming discussion covers the software and hardware side. The software side relies on hardware characteristics and primitives, especially for consistency management. Because this bridge between the two sides is the most important one for this thesis, it is discussed in more detail in a separate chapter (Chapter 3).

### 2.2.1 Software

Today's software is developed with a strict distinction between volatile data and non-volatile data in mind. Non-volatile data typically resides on disks and every programmer knows that avoiding disk access is crucial for the performance of an application. One has to carefully select the information

which is stored permanently and use appropriate algorithms[3]. Volatile data is treated less carefully. This is probably caused by the fact that restarts reset volatile data and somehow provide a "cure" for failures like memory leaks or concurrency bugs.

Should we treat NVRAM like volatile data? With focus on the persistence feature of NVRAM, which is the interesting one for this thesis, volatile data is not the best choice. NVRAM keeps its contents and, hence, simple reboots do no longer cure failures. Even worse, information that was previously automatically lost once the power was turned off is preserved with NVRAM. Having sensitive data like encryption keys or passwords automatically stored for a long time opens security holes and eases cold boot attacks. Consequently, research has suggested to either use NVRAM with existing interfaces for persistent data or to develop new abstractions.

#### Existing interfaces for persistent data

For decades, files and databases have been used for permanent information storage. Therefore, their interfaces are widely used and well known.

**File systems** File systems provide a hierarchical global name space to the whole system. A file created by one application is unique and identified by its name. The file can be shared among applications which have sufficient access rights. These three features - name space, access control, and sharing - remain relevant for file systems when NVRAM replaces disks. Significant changes lie in the implementation.

A disk based file system was forced to copy data from volatile main memory to the disk. Copying data to a disk required a system call in order to activate the disk driver. With NVRAM, the data is already at the location where it is permanently stored. Hence, copy operations and driver invocations are no longer needed. Volos et al. [60] claim that moving most of the file system implementation to user space libraries eliminates expensive system calls and increases the overall performance.

A disk based file system stores all of its meta data on disk. It is therefore optimized for the block-oriented disk interface. Since NVRAM is byte-addressable, there is no longer a block interface and implementations for the new, more fine-grained interface can improve the performance. PMFS [16], BPFS [13], and SCMFS [62] are examples for file systems for NVRAM which focus on efficient meta data management.

File systems have one big disadvantage: they do not care about the consistency of the data, only about meta-data. They have no knowledge about the internals of a file (and they should not have). This starts to become complicated when two processes operate on the same file simultaneously. For concurrent access and consistency of the actual data, a solution is offered by databases.

[3] Although files from disk can be mapped directly into an application's address space and the data therein can be used with the same interface as volatile data, these memory mappings are seldom used.

**Databases** In former times, main memory was scarce. The amount of data in a database easily exceeded main memory capacity. Hence, main memory was only treated as a cache for on-disk data and database systems were tailored for disks.

As main memory capacity increased, it became feasible to keep the whole database in main memory. In comparison to disk-based databases, data retrieval is way faster in such an in-memory database, because the disk access is no longer on the retrieval path. Since main memory was volatile, disks were still required to preserve the information across restarts. Algorithms writing recovery information to disk tried to minimize the number of disk accesses. As a consequence, a restart is expensive because all state in main memory has to be reconstructed from the minimal on-disk state.

NVRAM can be used as an additional cache for database systems as already outlined in 1987 [35]. Moreover, it could also eliminate the need for disks in main memory databases. The state is now preserved directly in main memory as explored by the database Sofort [49].

Main memory databases remove disk access from the data retrieval path. One step further, table structures can be eliminated, reducing a database to a hash-table. This allows for extremely fast access, but also for limited features as there is no structural information anymore. Key-value stores are such minimal databases and offer to store any data for a given key. The complexity is narrowed down to managing concurrent access and transactions which cover more than a single operation. In traditional systems, there is still the problem of durability when logging information has to be written to disk. The Echo [1] key-value store is designed for NVRAM and its authors investigate how to preserve the log in main memory instead of disks.

The aforementioned projects targeted main memory databases. What are better starting points for databases on NVRAM: disk or main memory based systems? DeBrabant et al. [15] analyzed the performance of existing disk-based databases and main-memory databases on top of NVRAM and conclude that neither is optimal. Ideas from both designs should be respected in database designs for NVRAM.

### Extended interfaces

What is the benefit of reusing existing interfaces on top of NVRAM as outlined above? First of all, legacy applications do not need to change. The faster access time of NVRAM in comparison to disks will probably cause a performance boost even in these cases. However, the old abstractions have been developed with disks in mind. Using them without any modification might not leverage the full potential of the new technology.

An alternative is to provide access to NVRAM by mapping it to special parts of an application's address space. The application may use libraries which provide persistent data structures as offered by [58]. Moreover, existing programming languages and run-time environments could support persistent variables and persistent heaps [59].

Being able to create, delete, and locate persistent objects is only a part of the tasks that arise with NVRAM. An operation that modifies persistent data could crash before it completed its work. It could leave the data in an inconsistent state - rendering it useless. This consistency issue is crucial for both existing and new interfaces for NVRAM and also the main topic for this thesis. Chapter 3 is therefore dedicated to a discussion of consistency with respect to existing projects.

### 2.2.2 Machine Integration

#### Storage Hierarchy

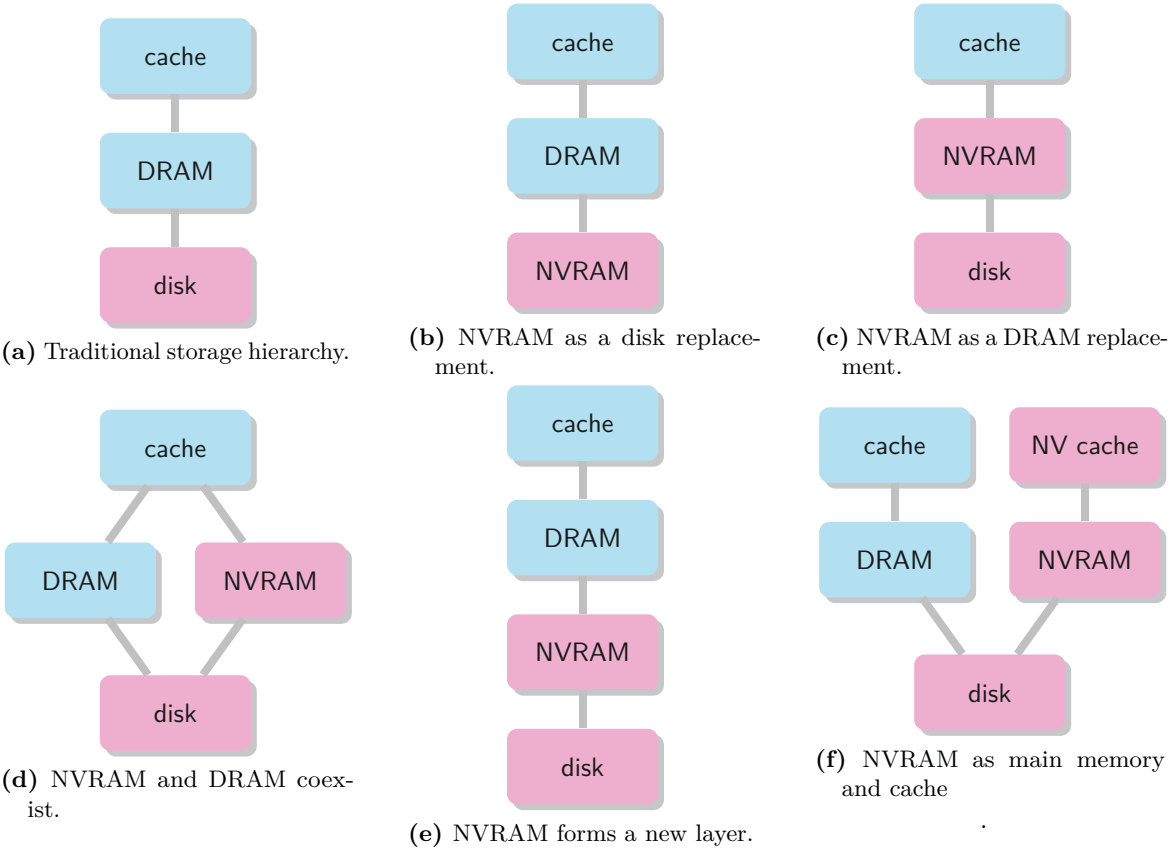
NVRAM unites the persistence of disks with the byte-addressability of main memory. As a consequence, there are many ways of integrating NVRAM into the existing storage hierarchy (Figure 2.3a) which differ in the expected access latency.

NVRAM can be seen as a faster disk as in [6] (Figure 2.3b). In systems with disks, the performance was dominated by waiting for rotating elements to move and software management overhead was negligible. This changes with the significantly lower latency of NVRAM and Caulfield et al. [6] investigate in optimizations of the Linux I/O stack. However, treating NVRAM as a disk does not leverage its byte-addressability, which renders other storage hierarchy changes useful.

Its byte-addressability enables NVRAM to be used as main memory. It may replace volatile Dynamic Random-Access-Memory (DRAM) completely (Figure 2.3c) or co-exist in hybrid systems (Figure 2.3d). The latter option is often cited when NVRAM's latency is predicted to be higher than DRAM's. [15] uses DRAM as a cache for NVRAM. Instead of replacing any of the existing layers in the storage hierarchy, NVRAM can also introduce a new layer in between main memory and disks (Figure 2.3e). It can then be used as a disk cache, for example for file systems which frequently read and modify meta-data but seldom change any other data, as in [61].

If the characteristics of NVRAM are sufficient, it may also be used as a processor cache. Kiln [66] uses NVRAM for both: a cache and main memory (Figure 2.3f). The non-volatile cache buffers operations on persistent data in NVRAM. With the help of a new cache controller, Kiln prevents partial write-backs of operations on persistent data which are potentially harmful. The information that was not yet written back is preserved in the non-volatile cache and is still visible after a crash.

The last option of the non-volatile cache with a new cache control indicates that NVRAM may not be sold as modules that can be plugged into the DRAM slot of today's systems. Some designs enforce hardware changes which are explained next.




**(a)** Traditional storage hierarchy.

**(b)** NVRAM as a disk replacement.

**(c)** NVRAM as a DRAM replacement.

**(d)** NVRAM and DRAM coexist.

**(e)** NVRAM forms a new layer.

**(f)** NVRAM as main memory and cache

**Figure 2.3:** Different integrations of NVRAM into the storage hierarchy. Caches and DRAM are volatile. NV caches, NVRAM, and disks are non-volatile.

### System Modifications

At some point, it will be feasible to build NVRAM memories. Will they be compatible to existing technologies or will we need completely new systems? It seems desirable to simply plug a new, non-volatile memory module into an existing system and start to benefit from the new technology. As already hinted, this is not as easy as it seems on first thought. Especially consistency is hard to maintain without changes to today's systems, mainly because of the volatile, hardware-managed caches.

Data that is stored in a volatile cache is lost once the power runs out. If the contents are not written back to a persistent memory, the information is lost. While many projects handle this issue pro-actively and force data to be written back when an operation finishes, flush-on-fail reacts only when the failure occurs. Then, all data from volatile memories is written back to a dedicated region in the persistent memory and restored upon next boot.

The inventors of Whole-System Persistence [44] modified a computer system's power supply so that it triggers an interrupt when the voltage drops. All volatile data is then flushed by the interrupt

handler, using the remaining energy. This idea removes the burden of consistency management from programmers. No further care needs to be taken because the system restores the complete CPU state. There is no study about the reliability of this approach. Especially the question of how much energy is needed to flush all data is important and any failure in that projection or a hardware failure of the mechanism would render the data unrecoverable. Moreover, there is still a need for transactions for external components. The remaining energy is probably insufficient for completing all ongoing DMA transfers that devices may carry out and transactional mechanisms are still needed.

While flush-on-fail tries to avoid the loss of data, other projects accept it. Instead, they focus on the ordering of writes to NVRAM by modifying the cache controller. Today, a cache may evict cache lines in nearly any order, disturbing program order with respect to what is written to main memory. This controller can be modified in a way that it respects a foreseeable order as in BPFS [13]. These changes lead to a new consistency model which is introduced in Chapter 3.

For this thesis, I envision that existing systems are augmented with NVRAM with as little changes as possible and NVRAM and DRAM coexist. Since I do not expect a huge latency difference and ignore energy consumption, I assume that DRAM is used for sensitive information. It makes no difference whether information that is not persistent, meaning it is not reused after a power cycle, is placed in NVRAM or DRAM. The system offers interfaces to programmers so that they can place information in either of the two memory types. This interface will be similar to existing memory mappings and it does not matter whether the requests are handled directly by the Operating System (OS) or with the help of file systems. Disks are no longer essential but used for data exchange from system to system for economic reasons.

## 2.3 Evaluation and Verification

All projects that develop algorithms for the new NVRAM technologies face one serious challenge: the technology is not yet available on the market. Therefore, evaluations can not be performed on the actual target hardware. Nonetheless, some researchers run experiments on real hardware and some use software managed approaches.

### 2.3.1 Hardware based

The performance of NVRAM is not yet foreseeable. As shown in Section 2.1, it is sometimes predicted to be significantly slower than DRAM. Others claim that both technologies will differ only slightly in their performance. If one assumes that the latency of NVRAM is similar to DRAM, the performance of new algorithms can be compared on existing hardware by using DRAM as a substitute for NVRAM [1, 6, 7, 58].

DRAM can not resemble the persistence of NVRAM so easily. Experiments that target the persistence aspect and investigate in the correctness of algorithms, as [61, 44], may use DRAM modules that are slightly modified and also contain a persistent technology, typically flash. In the case of a power outage, the module has enough capacity to write the DRAM data to flash and restores it on the next boot.

A third option is only applicable on a small scale. Sensor nodes with FeRAM are sold by Texas Instruments and used in the sensor node project [40]. Of course, sensor nodes architectures differ greatly from commodity systems and this approach is not suitable for other projects.

Running experiments on real hardware is only sufficient when either real NVRAM can be used or DRAM is a sufficient replacement. In other cases, one has to fall back to software solutions, like simulations.

### 2.3.2 Software based

Access latency is often considered to be the most important difference between DRAM and NVRAM. The uncertainty of the real characteristics cause many researchers to introduce latency as a variable and repeat their experiments with varying access latency.

Lu et al. [39] consider only the delay of flush operations to be important. They introduce a manual delay in their implementation of the new flush operation. These flush operations controlled by developers of the application or a persistence framework. Ordinary write operations, which are normally translated directly to hardware instructions, can not be easily delayed. Mnemosyne [59] relies on a proprietary compiler with Software Transactional Memory (STM) support which invokes a software defined function for all writes within a transaction. The authors can therefore vary the delay by modifying an idle waiting loop in each of the operations.

Using Mnemosyne's approach requires an STM compiler, which is not used by all projects. Without the STM notifications, it is hard to vary the write latency. Dulloor et al. [16] built a hardware based simulator with the help of patched microcode and custom firmware. Again, it adds a manual delay to the writes, but does so on a low level. The resulting environment is also used by [49] and [15] which rely on Dulloor's PMFS.

Pelley et al. [50] collected memory traces of applications and chose an analytical model to predict the performance.

Using real hardware for evaluations is only possible based on the assumption that future hardware will change only slightly. Projects which suggest more drastic changes, for example a new cache controller, therefore rely on simulation software [38, 66, 13, 67, 42].

Nearly all publications report in their evaluations about the performance of software running on NVRAM, but only little is told about verification. This is probably due to the difficulty of verifying

software for NVRAM. There are two important aspects to consider: reordering by the hardware and power failures that may occur at any time. Yat [31] is a validation framework for NVRAM software that takes these two facts into account. The application to analyze is run as a virtual machine and selected events, like cache line flushes, are tracked. Afterwards, the authors determine the points in the execution trace where power outages are interesting and collect all possible orders in which the events may occur when the power runs out. They use this information to restart the application with exactly one selected order of events and analyze the recovery after a power outage.

As I expect the latency of NVRAM to be close to DRAM, performance evaluations can be performed on existing (volatile) hardware. Since I intend to rely only on existing hardware and hardware primitives, there seems to be no need for simulations. Still, simulations can be useful to get a detailed understanding of the results of performance evaluations. As shown in [57], performance of NVRAM applications can be predicted by counting the number of certain events with emulation software. This approach should be followed for correctness analysis, too.

## 2.4 Visions

The storage hierarchy did not change much for the last decades. Many algorithms and policies have been designed with a particular hierarchy in mind where main memory is scarce and persistence is limited to disks. NVRAM allows for drastic changes of the storage hierarchy when it is seen as a union of today's main memory and disks. So far, only a few researchers looked one step ahead [2] and questioned concepts that have been accepted as essential and useful for a long period of time, like virtual memory. Most of the other projects focus on problems of the intermediate systems which integrate NVRAM in existing architectures.

CHAPTER 3

# Consistency

The term consistency has various meanings in the different fields of computer science. In database systems, a consistent state of the database means that all integrity constraints hold. Integrity constraints are semantic rules, e.g., an employee's salary does not exceed the salary of the employee's supervisor. Within a transaction, it is possible to violate these rules. When all salaries are increased, it is allowed to adjust the employee first and the supervisor afterwards, even if this would cause a temporary inconsistent state. The relaxation is possible because such a temporary state is not allowed to be observed outside of this transaction, as defined by the ACID properties of transactions [55, pp. 175 sq.], [22, p. 120].

Consistency is also important in the field of shared memory programming. When a processor operates on a shared memory region, the memory consistency model defines when and which changes are observable by other processors which also use this memory region. The consistency model is not only a set of rules that define what becomes visible. It also defines a protocol based on a set of primitives and provides guarantees only for programs that adhere to this protocol. Consequently, the memory consistency model can be seen as a contract between hardware manufacturer, compiler, and programmers.

The notion of consistency models for shared memory breaks down to observability. So far, systems with Non-volatile Random-Access-Memory (NVRAM) do not bring any change to the existing shared memory consistency models. Although they introduce a different level for consistency requirements, it is similar to memory consistency. This difference is explained next.

## 3.1 Machine and Failure Model

### 3.1.1 Consistency in Today's Systems

Before the changes which NVRAM brings are described, current systems and their memory consistency are introduced. This description is followed by an explanation of the challenges which arise with NVRAM.

The example system, shown in Figure 3.1, consists of $N$ processors. For this discussion, only the system's buffers are important. Other aspects, like the exact topology of interconnects or the number

of cache levels, are not considered in detail. Each processor has a private buffer, for example registers and a store buffer. On the next level, all processors are connected by a shared buffer: a cache. The lowest level is main memory. For simplification, external disks are not considered and relevant data is assumed to be already in main memory.

The meaning and implications of memory consistency in systems without NVRAM is illustrated with the example of a simple address book residing in memory[1]. It contains two entries: Alice lives in Amsterdam and Bob in Berlin. A user starts a graphical interface that visualizes the address book on core 1. The newly created process starts and reads the current address book from main memory. The data is now installed in its local buffer so that the next iteration can be served from there and does not need to access main memory again.

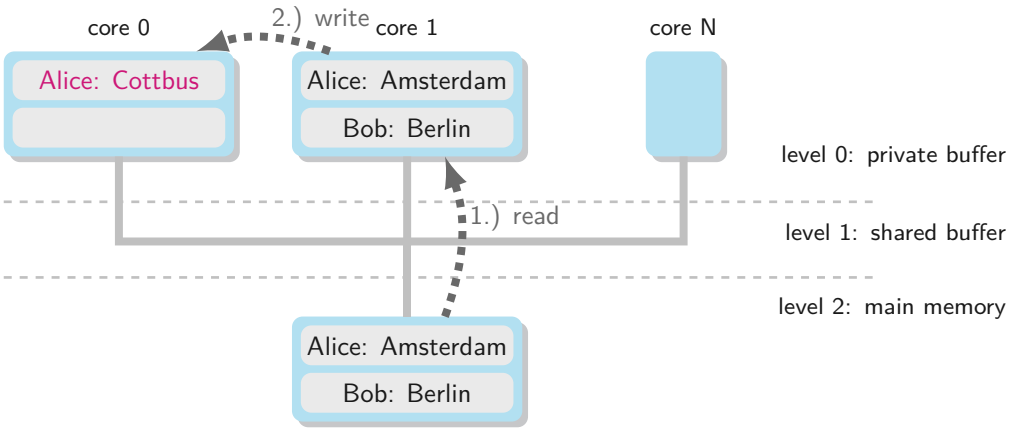
Next, Alice and Bob send a notification because they moved in together in the city of Cottbus. The address book update is performed on core 0 in its local buffer. Afterwards, the private buffers are inconsistent because core 1 is still using the old state. In this situation, the memory consistency model defines the steps which are necessary for the core 1 to observe the changes made by core 0.

On Intel systems, core 0 needs to flush the data explicitly from its private buffer with a so called store fence [26]. The data is now in the shared buffer, but core 1 also needs a fence, a load fence, in order to observe the changes. Programmers need to carefully place these fence operations. A missing fence can cause the process to read stale data - leading to inconsistencies.

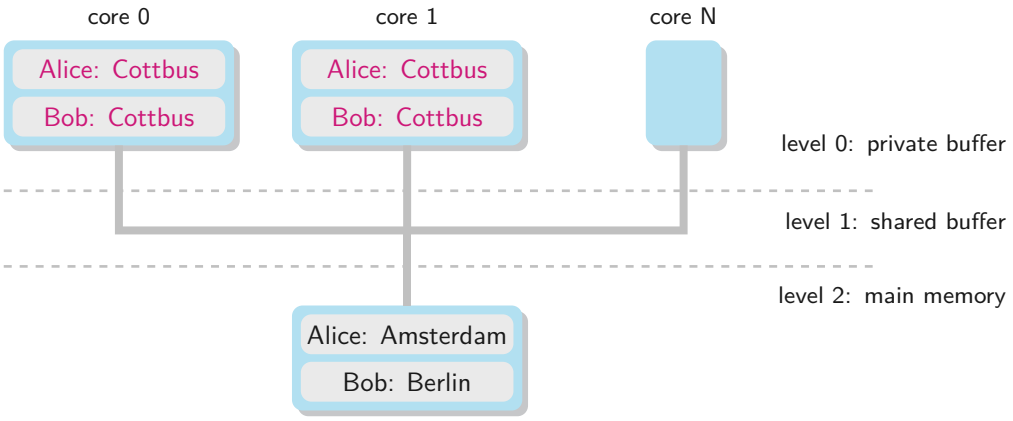
When fences are triggered correctly, core 1 is able to observe changes made by core 0 (Figure 3.1b). However, main memory contents have not yet been updated. The address book entries in main memory are still the old ones. Since all buffers are limited in their size, items have to be evicted at some point. Hardware policies choose data to evict and this information is written from the current buffer level to the next higher level, e.g., from a private buffer to the shared buffer. The hardware bases its decision on the overall system utilization, not on an individual process. This implies that a programmer can hardly foresee the point in time when data is evicted because it depends on the behavior of other processes, too.

In the address book example, it might happen that Bob's entry is evicted and written to main memory first, as shown in Figure 3.1c. It is possible to trigger a write-back to main memory explicitly, similarly to the flush of the private buffer. For the private buffers, the aforementioned fences cause a flush of the whole buffer. It might also be possible to flush a single entry, for example a cache line. Again, placing these flushes requires great care. The sequence in Listing 3.1 flushes both entries, but it is insufficient. After both entries are changed and before the first flush starts, the hardware may already evict Bob's entry, leading again to the situation in 3.1c.

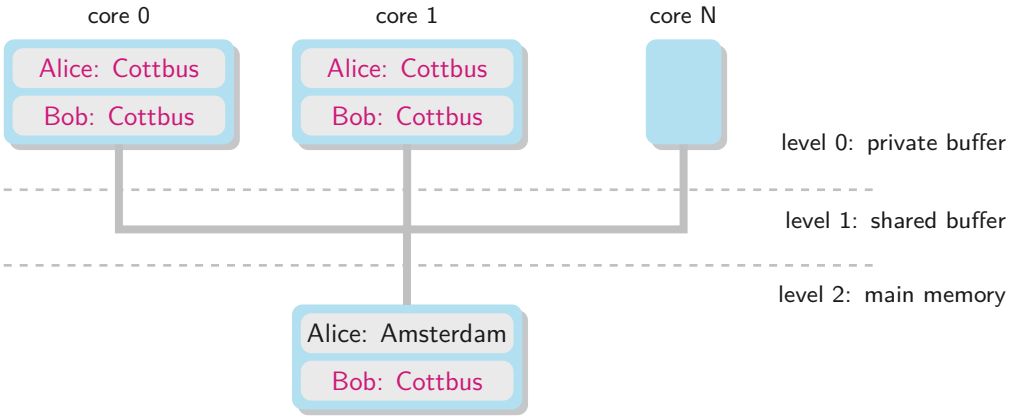
[1] The address book is a simplified version of data structure modifications. Many of them modify at least two elements. For example, inserting an element in the middle of a doubly linked list requires modifications of the element's new predecessor, its new successor, and the element itself. I favor the address book because the list example results in overly complex formulations distracting from the important points.




**(a)** Core 1 reads data from main memory and installs it in its private buffer (1). Afterwards, core 0 writes to this data set (2). The result are inconsistent private buffers.




**(b)** By applying the rules of the consistency model, both core 1 observes the changes of core 0. Note that main memory contents do not change.




**(c)** Bob's entry is evicted first from the higher levels and updated in main memory.

**Figure 3.1:** Schematic view of buffers in current systems and the problem of inconsistencies.

```
1 abook['Alice'] = 'Cottbus';
2 abook['Bob'] = 'Cottbus';
3 flush(abook['Alice']);
4 flush(abook['Bob']);
```

**Listing 3.1:** Flushing modifications at the end. Bob's entry might be evicted prior to the flush of Alice's change.

The situation can be avoided by flushing the first change before the second one starts, as shown in Listing 3.2.

The local buffers have to be flushed in order for data to be actually written to NVRAM. What happens if a failure occurs after the first flush? The change to the second entry was not yet performed and only one part of the whole operation is carried out. Many operations require their changes to be performed either completely or not at all, as discussed in the next section.

### 3.1.2 Systems with NVRAM

For this thesis, NVRAM modules are compatible to the interfaces of today's DRAM modules. They can be plugged into existing computer systems without further changes. The resulting system architecture differs only in the fact that main memory is now persistent. All other buffers remain volatile and, hence, their contents are lost when the power is turned off.

The power might be turned off during the system's normal operation and abort ongoing operations. In the address book example, the execution could be interrupted before core 0 flushes its changes. Core 1 will therefore not be able to see these changes. This does not matter, because the whole system will stop anyway and start all over once the power is turned on again. With NVRAM, one may reuse the data in the persistent main memory once the power returns. Hence, it is important to control when and what data is written to NVRAM.

As shown before, flushes exist in order to write data from higher buffer levels to main memory. These flushes need a detailed specification with respect to persistency. My requirements are as follows:

```
1 abook['Alice'] = 'Cottbus';
2 flush(abook['Alice']);
3 abook['Bob'] = 'Cottbus';
4 flush(abook['Bob']);
```

**Listing 3.2:** Flushing each modification individually assures that Alice' changes are written first.

**Atomicity** Cache lines are atomically written to NVRAM. The granularity of entries in the buffer on level one is cache lines. Both the hardware replacement policy and the existing flush primitives cause a whole cache line to be written to main memory. Once flushed, a cache line is either completely written to NVRAM or not at all.

**Ordering** Two cores may initiate a flush of the same cache line from their buffers at the same time. Again, cache lines are an atomic unit and it is impossible to write half the changes from one core and half the changes from the other core to NVRAM. One flush will precede the other.

The previously shown machine model featured only a single memory module. Modern systems are typically Non-Uniform Memory Access (NUMA) machines with multiple cores and also multiple memory modules as shown in Figure 3.2. In this example, core 0 modifies the address book. It starts to change the entry of Alice and flushes it. Then, it modifies and flushes Bob's data. Accessing, and hence flushing, data on a local memory module is faster than accessing a remote module. Consequently, the data of Bob might be written to main memory before Alice is updated on the remote module, even though the flushes have been used correctly.

So far, this issue has rarely been addressed in NVRAM research. Intel once announced to extend their instruction set with a new barrier operation for NVRAM called `pcommit`. A core that executes `pcommit` stalls until all previously flushed cache lines are completely written to NVRAM. Issuing a `pcommit` immediately after flushing Alice' address book entry would prevent the depicted situation in which Bob is updated in main memory first. Shortly after the instruction has been announced, it was already marked as deprecated [52].

**Optimizations** Up to this point, NVRAM is the only change for the target system in this thesis. Additional processor modifications are not required. Still, there is one optimization which is likely to be implemented by processor vendors.

Current flush instructions trigger a write-back from one buffer level to the next one. Moreover, they invalidate the data on the current level so that the next access has to address the next level. In the case of the address book entries, core 0 has to fetch Bob's entry from main memory after it has been flushed. Such an invalidation is useful for memory mapped I/O where a device may update the data directly in main memory. The cache would return stale data in this case.
In the context of NVRAM, an invalidation is not required when data is flushed. Having a special instruction for a flush without invalidation would therefore be sufficient. The resulting instruction reduces the number of main memory accesses and may improve the performance.

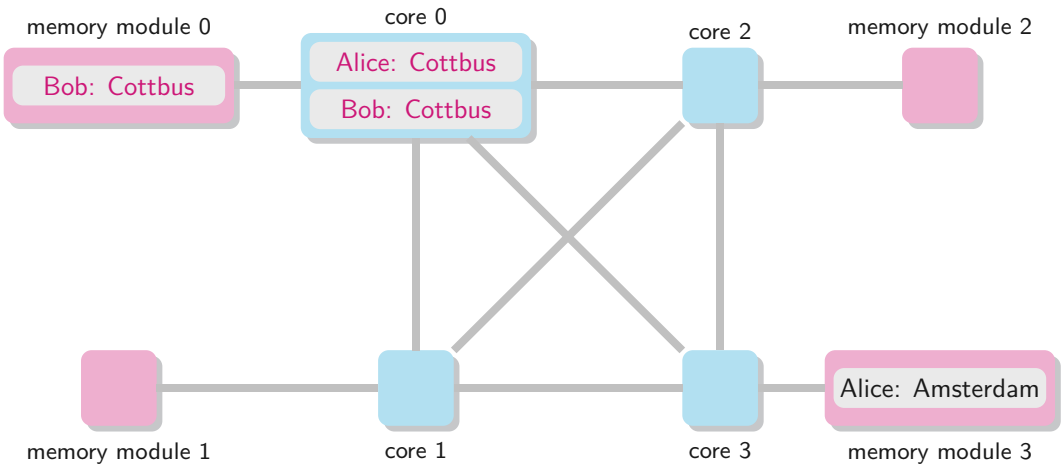



**Figure 3.2:** NUMA machine with four cores and four memory modules. Bob's address book entry resides on module 0 and Alice' entry on module 3. Both entries have been modified by core 0 and the modifications are in its local buffer. Core 0 triggered flushes of the modified cache lines. The path to the memory module hosting Alice' entry is longer than the path to the memory module hosting Bob. Therefore, Bob's change is already written to main memory while Alice' change is not yet updated in main memory.

### 3.1.3 Failure Scenarios

The most important difference which NVRAM brings is that data in main memory is preserved even when the power is turned off. When the user initiates a controlled shut-down, the power is turned off afterwards. The shut-down routine has sufficient time and resources to perform any action that it requires. An unexpected power outage, for example an empty notebook battery, is different. It may interrupt any ongoing operation, causing changes to be performed only partially. This new failure scenario is addressed in this work. In addition, software may still contain bugs, crash, or have data races. These issues are out of the scope of this thesis.

## 3.2 Persistency Models

Persistency models describe how consistency for persistent memory is approached. In all cases, the failure scenarios are power outages which cause a loss of data in volatile memories, especially CPU registers and caches.

Data loss avoidance strategies can either trigger at the exact moment of a power failure or proactively whenever persistent data is modified. One example from the reactive category is *Timely Sufficient Persistency* [45] which implies that data is moved to an adequately safe location in a timely manner. The aforementioned Whole-System Persistence [44] which writes CPU state and registers to NVRAM upon a power-outage signaling interrupt is an example implementation. It restores the saved data when the power is turned on again and the system continues where it left off.

An alternative to the reaction on failure is to act proactively and avoid data loss by controlling writes to NVRAM during normal operation. To ease the description and highlight the differences in the persistency models shown next, the address book example is used again. Alice and Bob moved in together and the address book needs an update. This time, it is important that both entries are updated at the same time. A partial update, which could be caused by a power outage, is to be avoided. Hence, the operations form a transaction that should be carried out either completely or not at all. Means to achieve transactional semantics are explained in Section 3.5. For now, an undo log serves as an example.

As its name indicates, an undo log is being used to revert to the old state in the case of a failure. Before the data is modified, its current state is written to the log. Then, the modifications are performed and, finally, the log entry is disposed. Listing 3.3 sketches this scheme.

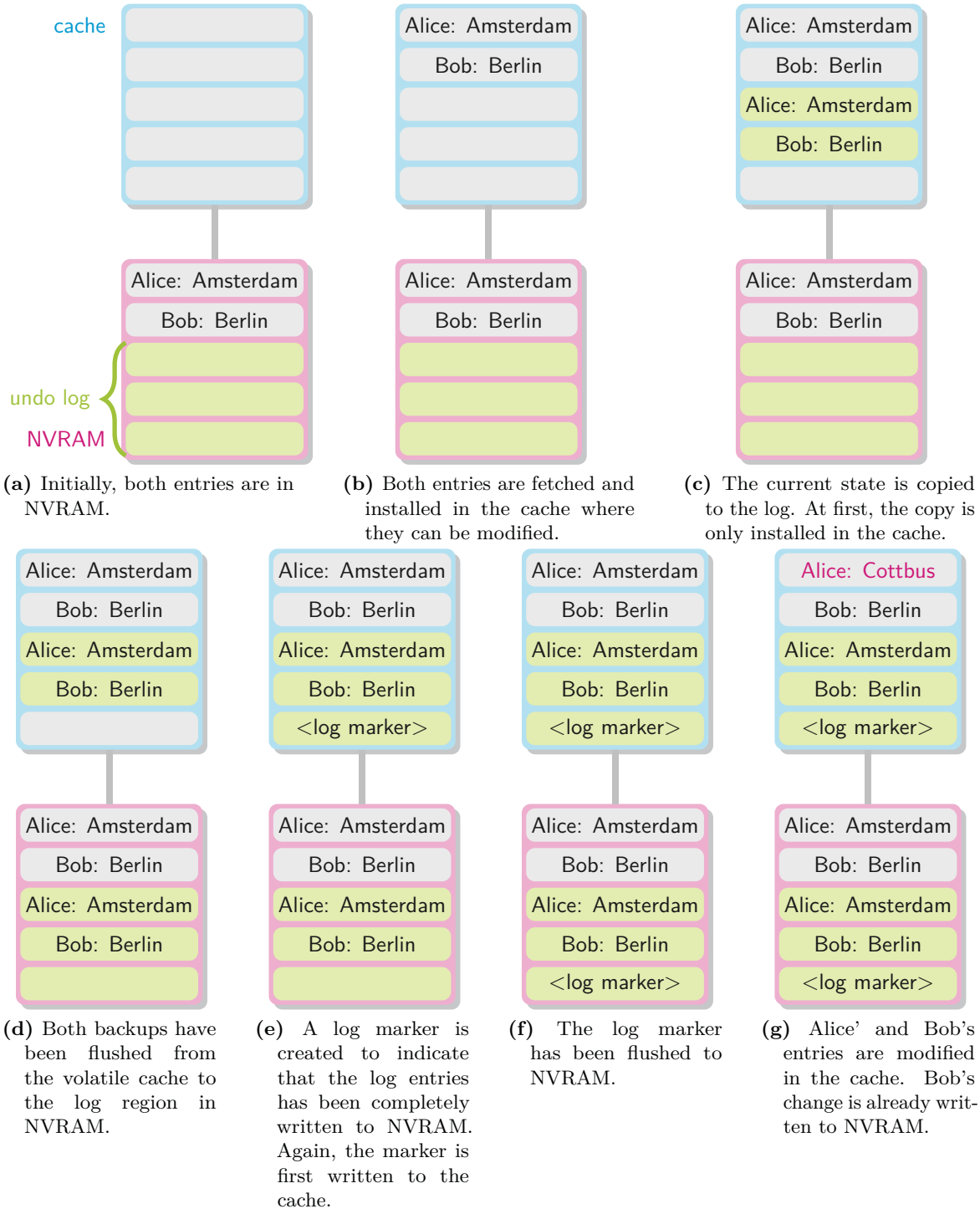
Figure 3.3 gives a graphical representation of the performed steps. Initially, the log is empty. After the log entry is written and flushed, the address book is modified. As shown in Figure 3.3g, the changes may be written to NVRAM only partially. If the power ran out at this point, the intermediate state is invalid. Before the data is used again, its old state has to be restored from the log entry. A typical log entry requires additional meta data, i.e., to indicate whether the log entry was completely written and a marker denoting that the transaction was complete. Since the exact details are highly implementation specific, a simple form of meta data is used here. A marker in the log entry tells whether it is valid. After all data is written to the log, the marker is set to valid. During commit, once all changes have been flushed, the marker is toggled and the log entry becomes invalid. The recovery routine ignores invalid log entries.

At this point, the address book example can be broken down to a sequence of write operations as shown in Listing 3.4. For future reference, a short form is given for each memory location, e.g., `LA` is the location of the log entry for Alice.

If all these operations were carried out in program order and each write would immediately update NVRAM, the system would be compliant to the model of *Strict Persistency* [50]. A more detailed

```
1 // preserve current state
2 log.log(abook['Alice'], abook['Bob']);
3
4 // modify
5 abook['Alice'] = 'Cottbus';
6 abook['Bob'] = 'Cottbus';
7
8 // commit
9 log.commit();
```

**Listing 3.3:** Basic steps for transactions using an undo log




**(a)** Initially, both entries are in NVRAM.

**(b)** Both entries are fetched and installed in the cache where they can be modified.

**(c)** The current state is copied to the log. At first, the copy is only installed in the cache.

**(d)** Both backups have been flushed from the volatile cache to the log region in NVRAM.

**(e)** A log marker is created to indicate that the log entries has been completely written to NVRAM. Again, the marker is first written to the cache.

**(f)** The log marker has been flushed to NVRAM.

**(g)** Alice' and Bob's entries are modified in the cache. Bob's change is already written to NVRAM.

**Figure 3.3:** A special region in NVRAM is used as an undo log, as marked in (a). Before data is modified, it is copied to the log in steps (b) to (d). Then, the log entry is marked as valid ((e) and (f)). Finally, the actual data is modified (g).

```
1  // preserve current state
2  write(log_alice);    // LA
3  write(log_bob);      // LB
4  write(log_marker);   // LM active
5
6  // modify
7  write(alice);        // A
8  write(bob);          // B
9
10 // commit
11 write(log_marker);   // LM inactive
```

**Listing 3.4:** Sequence of write operatoins in the address book example.

description of this model follows in the remainder of this section. In this case, caches can not hide latency as each write has to update main memory. Consequently, the run-time will increase drastically.

Researchers found out that it is not crucial to preserve the write order at all times [50, 27, 37]. Consider the logging of the current state of the address book. Alice's state is written to the log before Bob's entry is logged. Persisting these two writes in the opposite order is acceptable. The same applies for the updates of the entries in the address book. It should never happen that Alice' entry is updated before the old state is in the log. This situation is similar to shared memory programming where it was also found to be too expensive to make each and every store observable by other programmers. Inspired by relaxed memory consistency models in this field, the same researchers introduced relaxed persistency models for NVRAM. Figure 3.4 gives an overview of the resulting memory persistency models. *Timely Sufficient Persistency* [45] has already been introduced in the beginning of this sections. The other models are explained next[2].

## Strict Persistency

*Strict Persistency* [50] implies that all stores from the CPU are written immediately to NVRAM. A timing diagram for the address book example is shown in Figure 3.5a. Each store to cached data is followed by a write to NVRAM, which causes a large gap between the individual stores.

One way to implement *Strict Persistency* is to disable caching or use write-through caching. With no caching at all, stores and loads have to access main memory. Write-Through caching allows data to be installed in caches. Further reads can be answered from the cache and do not have to access main memory. A store updates the cache and also main memory. Bhandari et al. [5] analyzed the effects of write-through caching. As expected, the performance when using write-through caching is similar to using normal caching in a read-mostly workload. In a write-mostly workload, write-through

[2]There is no consensus on calling it 'persistence" or 'persistency". Due to the connection to memory consistency, I use 'persistency" when referring to memory models.

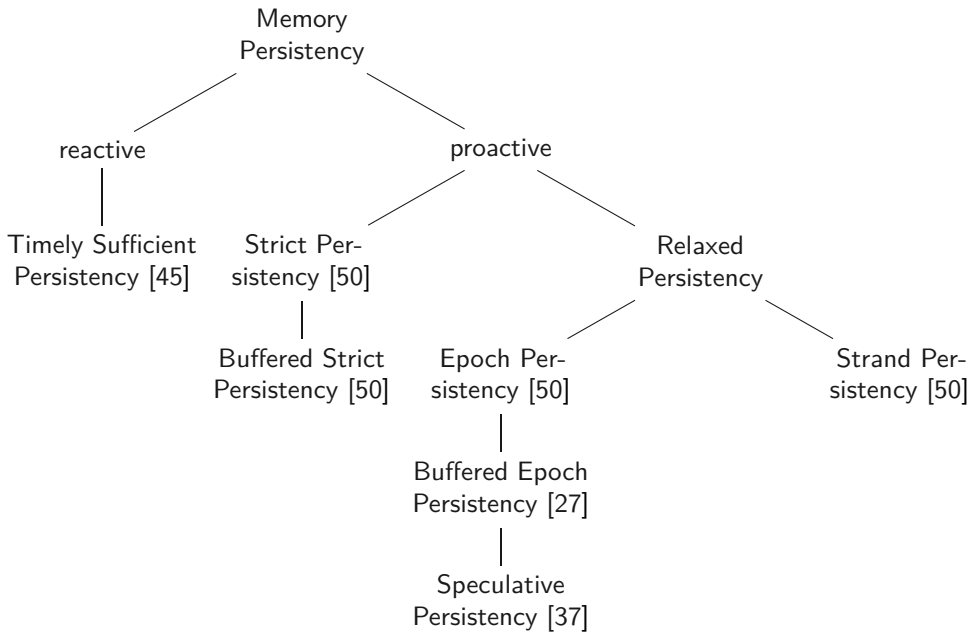


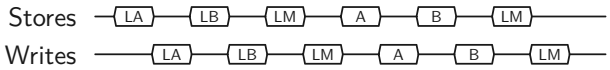

**Figure 3.4:** Persistency Models

behaves similar to a system with disabled caches. Although the results themselves are not surprising, the actual performance degradation is. Without caches, the run-time of the experiments increases by 750 to 1000 times.

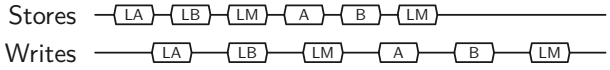
### Buffered Strict Persistency

In the *Strict Persistency* model, all stores are directly written to NVRAM. It implies a stall after each store because the system has to wait for the write. *Buffered Strict Persistency* [50] decouples stores and their corresponding writes in time. Still, each store triggers a write and their order has to be preserved, but the write can be delayed. Figure 3.5b illustrates the same writes as before according to *Buffered Strict Persistency*. The gap between stores is reduced, causing the sequence of stores to complete earlier.

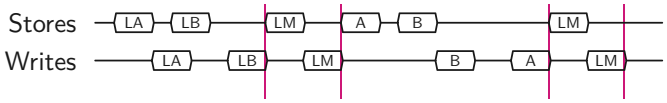
An implementation of this model may extend the existing cache hierarchy by a new cache for NVRAM writes, as described in [30]. This new cache is a FIFO buffer for stores to NVRAM and, hence, preserves the order of stores. The existing cache hierarchy needs a small modification: dirty cachelines that contain data from NVRAM are dropped when they are evicted. Otherwise their write-back to NVRAM would destroy the ordering guarantees.



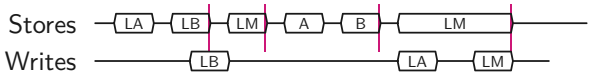

**(a)** Strict Persistency. Each store is immediately followed by a write to NVRAM. Thereafter, the next store may start.




**(b)** Buffered Strict Persistency. Stores are not immediately followed by writes to NVRAM and can be delayed, but the order is preserved.




**(c)** Epoch Persistency. Before a new epoch starts (marked by a solid, vertical line), all stores from the preceeding epoch have to be written to NVRAM. Stores within an epoch can be written to NVRAM in any order.




**(d)** Buffered Epoch Persistency. Before any store from an epoch is written to NVRAM, all stores from preceeding epochs have to be written first. In contrast to Epoch Persistency, the writes can be delayed and do not start with each new epoch.

**Figure 3.5:** Timing behavior of selected persistency models. The upper line represents stores executed by the processing unit. On the lower line, writes to NVRAM are shown. `LA` and `LB` are the writes of Alice' and Bob's data to the log. They are followed by marking the log entry as active `LM`. Then, Alice' and Bob's entries are changed in `A` and `B`. Finally, the log is marked as invalid by writing the marker again `LM`.

## Epoch Persistency

Strict persistency models enforce that the order of writes to NVRAM matches the order of stores issued by the Central Processing Unit (CPU). Thereby, they prohibit performance optimizations such as reordering. In some situations, reordering is acceptable. When the address book entries are updated, it does not matter whether Alice' update precedes Bob's. It is of utmost importance to mark the log entry only after both updates are complete. The program can therefore be split into execution phases, which are called *Epochs* in *Epoch Persistency* [50].

An epoch is defined as a sequence of instructions for which the order of stores does not need to match the order of writes to NVRAM. A programmer has to denote the start of a new epoch explicitly. The address book example consists of four epochs, which are shown in Listing 3.5. In the first epoch, the data is logged. A new epoch starts before the log entry is marked as valid, because the marker has to be written only after the log entry is complete. Prior to the modifications, the system needs to complete writing the marker, which therefore requires the start of a new epoch. Finally, the last epoch starts before the log entry is disposed.

```
1  // preserve current state
2  BEGIN_EPOCH();       // epoch 0
3  write(log_alice);    // LA
4  write(log_bob);      // LB
5  BEGIN_EPOCH();       // epoch 1
6  write(log_marker);   // LM
7
8  BEGIN_EPOCH();       // epoch 2
9  // modify
10 write(alice);        // A
11 write(bob);          // B
12
13 BEGIN_EPOCH();       // epoch 3
14 // commit
15 write(log_marker);   // LM
```

**Listing 3.5:** Epochs applied to the address book example.

*Epoch Persistency* defines all stores from an epoch have to be written to NVRAM before the next epoch starts. A sequence of writes and stores for *Epoch Persistency* is shown in Figure 3.5c. In the first epoch, the hardware evicts the log entry for Alice' data immediately after it was changed by a store. When the next epoch starts, the system needs to wait only for Bob's log entry. In epoch 2, the address book is modified. None of the stores performed in this epoch is written to NVRAM before the next epoch starts. Therefore, the system stalls until both operations are complete. It can also be seen that the order of the writes for the address book updates differs from the order of the corresponding stores. Such a reordering is allowed for stores in the same epoch.

### Buffered Epoch Persistency

*Buffered Epoch Persistency* [27] relaxes *Epoch Persistency* even further. Instead of stalling when a new epoch starts, the stall is delayed. Delaying is possible up to the time when a conflict occurs. A conflict is the situation where an epoch accesses a memory location that has been updated in an older epoch. Since the new write operation overwrites the old data or creates new data based on the old information, the old state is written back first. During this write back, the order of epochs is preserved. If data from an even older epoch is not yet written back, it is completed first.

Sometimes, it is impossible to wait for a conflict to happen before data is written back. The buffers are still limited in size and there might be a need for earlier write-backs when data has to be evicted. In this case, the system also respects the order of epochs and writes older epochs before newer ones.

Figure 3.5d illustrates a conflict in *Buffered Epoch Persistency*. Only the log entry for Bob has been written back during normal execution. When the log entry is finally invalidated, the system detects a conflict, because there is dirty data from an older epoch - the store that marked the log entry as valid. Now, the epoch order is respected and the log entry for Alice is written because it belongs to an older

epoch. Afterwards, the old marker is written. This fulfills the requirements, because the log entry is only marked as valid after it has been completed.

Another scenario is not depicted in the figure. The system could evict the address book update for Alice at any point after its store, e.g., when Bob's entry is changed. Again, the ordering guarantees assure that the log entry is evicted first, then marked as valid. Afterwards, Alice' change would be written back to main memory.

An implementation of *Buffered Epoch Persistency* is given for the BPFS file system in [13]. The authors present a modified processor cache in which each cache line is tagged with the epoch context where it was modified last. In order to evict all cache lines from an epoch, this information is also tracked.

### Speculative Persistency

Stores within the same epoch can be reordered when their data is written to NVRAM. So far, it has been unacceptable to reorder stores from different epochs. Writing data from a newer epoch to NVRAM causes the system to stall until all data from older epochs is written. *Speculative Persistency* [37] reduces the overhead of such stalls by allowing them to proceed to some extent. The idea is inspired by speculative execution in a processor which tries to predict what will happen in the future. If the prediction was right, use the results of the speculation. Otherwise, hide the fact that some operations have been performed. Thereby, the utilization of the system is more efficient.

*Speculative Persistency* groups $n$ epochs in a so called speculation window. When epochs from the same window overlap in their working sets, the newer epoch does not need to wait for earlier ones to complete before it writes data. Its writes are performed speculatively. Similar to speculative execution, the system assures that its speculations are only observable when they have been correct.

An implementation is presented by Lu et al. in [37]. It requires hardware modifications, most importantly multiversioning caches, which are tightly coupled to a specific software protocol.

### Strand Persistency

So far, we considered only a single transaction which modified two address book entries. Of course, the system continues thereafter, for example with a change to calendar entries. These also require a transaction consisting of several epochs. Calendar changes are not in any connection to the address book changes. The persistency models as introduced above would enforce ordering of all epochs from these two transactions, causing them to be performed in serial order.

*Strand Persistency* [50] allows programmers to remove the ordering restrictions between epochs. A strand in this context is a group of epochs which depend on each other, for example the address book changes. Epochs from different strands are allowed to be written to NVRAM in any order. This

increases the throughput because unrelated strands do not need to wait for each other. The calendar update would form a new strand and its writes are allowed to be interleaved with writes from the address book update.

The hardware changes required for an implementation of *Strand Persistency* are shown in [36]. Similar to the previous implementations, it requires changes in the instruction set and the cache architecture.

Research came up with a variety of persistency models in recent years. Until now, a consensus on the best model has not yet been found. Before it is possible to agree on a solution, several other aspects have to be taken into account, too. One of these aspects is concurrency, which is covered next.

## 3.3 Concurrency under Epoch Persistency

The previous descriptions outlined the different memory persistency models. For simplicity, a discussion of concurrency was omitted so far. This gap is about to be filled. The order of the upcoming discussions is based on the chronological order of references. This way, argumentations and consequences are easier to follow.

In 2009, Condit et al. [13] were the first to introduce the term *epochs* for NVRAM. This early work also featured a discussion on concurrency. Two epochs which run on two different cores and share state are defined as *conflicting*. These conflicting epochs 'will be serialized" [13]. Unfortunately, their work lacks a definition of the term *serialized*.

*Serializability* is a common phrase in other contexts, for example databases. Concurrent transactions are serializable if any sequential execution of the transactions can be found so that the sequential and the concurrent execution yield the same results [55, p. 280]. Applying this definition to epochs yields the requirement that for conflicting epochs at least one sequential schedule exists and leads to the same results.

Although the definition of *serializability* sounds simple, the implications are immense. One of the most widely used concurrency control mechanisms in databases is snapshot isolation for multi-version concurrency control. Every once in a while, a new anomaly is being found and it shows that the current implementations are not serializable [4, 17]. New implementations aim for efficient solutions for the problem afterwards [18, 46].

In the context of epochs, consider two different threads each in its own epoch. The first one is the update of the address book entries for Alice and Bob who moved to Cottbus. The second epoch scans the address book and writes statistics: the number of entries for the city Cottbus and the federal state Brandenburg in which Cottbus is located. The result of the second epoch depicts whether Alice and Bob moved already. Therefore, if the move is observed, the first epoch has to be written to NVRAM prior to the counter from the second transaction. Both epochs conflict because they operate on the same memory region.

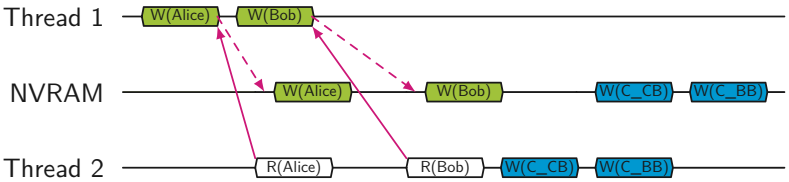
Condit et al. [13] define that upon a conflict, the conflicting data has to be flushed immediately. When the second epoch reads Alice' entry and it has already changed, Alice' entry has to be flushed. The same applies for Bob's data. Consequently, the epochs are written to NVRAM in serial order, as shown in Figure 3.6a.

What happens if the second epoch starts reading earlier? It may observe the old address for Alice and Bob and therefore not take them into account in its counter. Now, Alice' and Bob's entries are updated. From now on, both epochs write to different memory regions. As shown in Figure 3.6b, it is legal to write the epochs in any order to NVRAM, because the conflict can not be detected. The same may happen when both epochs overlap partially as in Figure 3.6c and only one of the two conflicts causes a write-back. Five years after the introduction of epochs, Pelly et al. [50] criticized this false assumption of serializability.

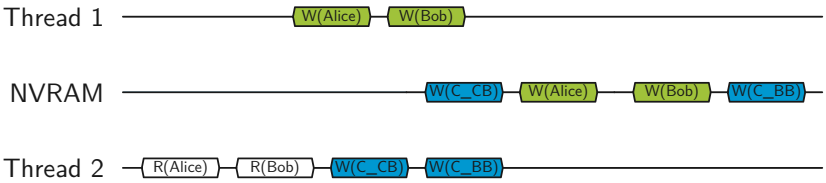
Pelley et al. [50] define epochs as not serializable. However, they still argue that the order of conflicting epochs is strict. Conflicts are caused by threads operating on shared data concurrently. Even without persistent memory, these threads need to employ a concurrency control mechanism such as locks. Since locks lead to sequential execution of the conflicting epochs, they are always ordered.

The idea of using locks to enforce ordering of epochs seems to be insufficient for Joshi et al. [27]. Locks cause the second conflicting epoch to wait until all writes from the earlier epoch are written to NVRAM. To reduce the resulting wait time, they introduce a dependency tracking mechanism for conflicting epochs. When a transaction observes changes made by another transaction, the second one is seen as dependent on the first one. This dependency is noted and respected in the subsequent epoch write-backs by the hardware. One complication in the dependency tracking scheme is a situation with circular dependencies. Joshi suggests to split the conflicting epochs into smaller epochs until the circle is resolved.

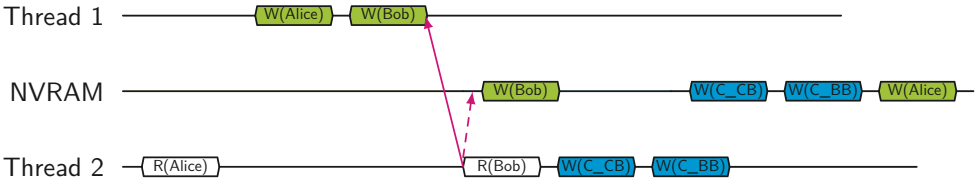
At this point, the short overview on concurrency in the context of memory persistency has already shown the field's complexity. All persistency models require hardware changes and concurrency makes these changes even more complicated. Concurrency control mechanisms which are already employed in the volatile world and memory persistency seem to influence each other. The underlying memory consistency may also contribute to persistency requirements and guarantees. So far, no consensus on the best solution for all these issues has been found. Not even all faces of the problem seem to be accepted. One of the most recent implementations of *Strand Persistency* was shown in [36]. Although the authors describe how to handle write conflicts, they seem to ignore that reads can also cause conflicts. It is therefore probably going to take a long time until research finds suitable persistency models. The first processors supporting these models are even further away. Therefore, this thesis concentrates on efficient persistency support on commodity hardware. The next section discusses the major cost factors which algorithms for NVRAM have to face.




**(a)** Epochs conflict when the epoch in thread 2 reads data from the epoch in thread 1. Due to the dependency, both epochs are serialized.




**(b)** Conflict between the epochs is not detected and they are not written to NVRAM in serial order.




**(c)** The first conflict of the epochs is not detected and they are not written to NVRAM in serial order.

**Figure 3.6:** Conflicting epochs with different timings.

## 3.4 Cost Models

Assume that two developers come up with two different strategies to preserve the consistency of a data structure on NVRAM. How can the performance of both strategies be compared to each other? Today, with only DRAM, runtime and memory consumption are the most important cost factors taken into account. NVRAM adds more aspects, because it requires flushes of volatile buffers in order to assure that data is written to a persistent location. Normally, these buffers are used for latency hiding. Flushing them brings a penalty - it is a stall for the pipeline. Because flushes are the major difference to today's algorithms, NVRAM research considers the number of flushes as an important cost factor [36, 62, 38].

When the number of flushes is being used to compare the performance of NVRAM algorithms, the consistency guarantees have to be kept in mind. In the case of the address book, one algorithm may guarantee the correctness of all entries at all times. The other algorithm might be derived from a context in which the persistent data structure was only used to preserve some kind of cache. It augments the data with a marker indicating whether an operation is pending. If it finds the marker set during recovery, an operation was interrupted, e.g., by a power outage. Because the information is only cached, it can be discarded completely and be rebuilt from other data sources. The second strategy does not provide the same consistency guarantees as the first one. Comparing the number

of flushes for both strategies would therefore be unfair. If not denoted otherwise, the upcoming discussions assume that all strategies provide the same consistency guarantees.

In addition to the number of flushes, execution time is often used to compare algorithms for NVRAM. Without a technology being available, the hardware characteristics are subject to speculations. To address this issue, simulations with varying latencies for the underlying memory are being performed and the resulting execution times are compared to each other. These results are meaningful if they show that one strategy outperforms others for all latencies. The simulations may also show that one strategy is better for a latency close to DRAM, while others outperform it when the latency is higher. This thesis is based on the assumption that NVRAM latency will be close to DRAM's and varying latencies are therefore not considered.

A final difference of the newly developed strategies which preserve the consistency of data structures on NVRAM is recovery. If the system crashes while an operation is in progress, it may need to perform some form of clean-up before the data can be used again. This recovery time can be traded for runtime and also consistency guarantees, as shown in Figure 3.7.

The next section gives a brief overview of consistency preserving strategies and discusses the resulting costs. Most of the introduced schemes existed long before NVRAM research began. Their recent optimizations for NVRAM are also explained.

## 3.5 Strategies for Preserving Consistency

With transactional semantics, a group of operations are either carried out completely or not at all. Partial execution of only a subset of operations in the group is prohibited. In the case of a failure, one of two widely used approaches can be taken. The first one is to roll back to the state prior to the start of the transaction. This is called *undoing* the transaction. Alternatively, the system could note down what was about to happen so that the actions can be repeated after the failure was handled. Repeating the actions means to *redo* them.

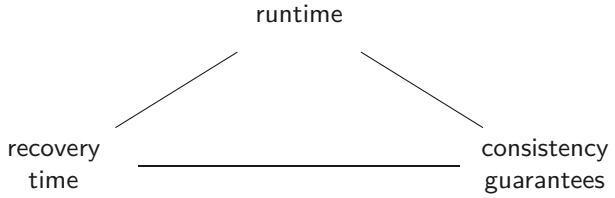



**Figure 3.7:** To reduce the costs when working on NVRAM, runtime can be traded for recovery time and consistency guarantees.

Gray [19] gives a metaphorical description of undo and redo schemes. Before Theseus entered the labyrinth to fight Minotaurus, Ariadne handed him a thread of wool. When he entered the labyrinth, Theseus unrolled the thread. After defeating Minotaurus, Theseus found his way back outside by following the thread. Thereby, he *undid* the action of entering the labyrinth. Not all effects of Theseus' actions are rolled back - Minotaurus is not brought back to life. The same applies in computer systems, where some side-effects can, or should, not be undone.

As a second example for transaction recovery in old myths, Hansel and Gretel are mentioned by Gray [19]. Similar to Theseus, they wanted to be able to undo the action of entering the forest by leaving a trace of bread crumbs. This trace also enabled their parents to follow Hansel and Gretel into the woods, hence, to *redo* their actions. Unfortunately for them, bread crumbs are not a stable medium and Gray calls Hansel's and Gretel's case the first log failure in history.

Both, the idea of undo and redo, require information to be logged. For NVRAM, this information has to be written to the non-volatile memory and therefore requires expensive flush operations. The next sections discuss the implications of both models with respect to the costs of running on NVRAM. In addition to the rather generic concepts of undo and redo, derivatives and alternatives are also shown.

### Undo Logging

In a system with undo logging, the first action to be performed is to write the current (valid) state to the log. Changes are then applied on the data's original location: in-place. Once all changes are complete, the log has to be disposed. In the case of a failure, the old state is restored by copying information from the log to the original location during recovery.

When changing two entries in an address book, one could write the whole address book to the log first. Alternatively, only the two entries can be logged. In the first case, data which is not modified by the transaction is written to the log. These writes are therefore unnecessary. For logging the two individual entries, additional information is required in order to identify the entries and restore the entries at their original location during recovery. Consequently, the granularity of log entries determines the amount of additional logging information.

As shown earlier, changes to more than one cache line are not performed atomically. The hardware may evict one line while the other one remains cached. This is also true for log entries. Therefore, the logged data has to be flushed before any modification of the data may start in-place. When the log entry covers more than one cache line, recovery has to be able to figure out whether the logging phase was interrupted. A common approach is to split logging in two phases: write the data first, flush it, write a valid marker, and flush the marker.

Undo logs are used in PMFS [16], Atlas [7] and by Kolli et al. [29]. Lu et al. [38] address the additional flush which is caused by writing the valid marker. Their idea of *eager commit* eliminates the need for

a marker by employing additional data in the individual cache lines of log entries. In the case of the address book, two cache lines are logged. When logging Alice' data, the total number of logged cache lines is written, too. This number is also recorded in the cache line for Bob's log entry. Moreover, each transaction has a unique identifier. If one of the log entries made it to NVRAM, recovery finds it after a failure. Since the total number of cache lines for this log entry is two, the system now knows that it should find another log entry. If it finds the second one for this transaction, the log entry was completely written. Otherwise, a partial log entry is found.

With the concept of eager commit, undo logging copies all modified data and some metadata to the log. All this information has to be flushed to make the log persistent before starting to modify the data. After all changes are complete, all modified data has to be flushed. Only after the second series of flushes the log entry can be discarded. For $n$ bytes of data, this sums up to copying $n + m(n)$ bytes of data to the log (where $m$ is the meta data size that depends on the data size). All in all, $2 * n + m(n)$ bytes are writes to NVRAM and flush sequences are called at 2 places.

### Redo Logging

In a redo-based system, changes are written to the log first and the original data remains untouched. Once all changes are complete, they are flushed and the log entry is marked as valid. The new data may stay in the log for a while. Only when log space has to be reclaimed, the changes are copied to the original location and flushed. Then, the log entry can be disposed. During recovery, complete log entries can be replayed on the actual data so that it is transferred to the new state.

The granularity of redo logging may vary from case to case. Sometimes, it is sufficient to denote a semantic information, like "transfer 100 dollar from account A to B". It could also be more abstract by recording each address and the value written to this location. The second, detailed version is employed by Mnemosyne [59] with the help of a special compiler. Mnemosyne also avoids writing a valid marker for log entries with a trick similar to eager commit: a single bit in each cache line is used to detect missing writes. They call this scheme *tornbit log*. Similar to undo logging, redo logging results in 2 places for flushes and writes $2 * n + m(n)$ bytes to NVRAM. The second flush sequence is deferred to log reclaiming. If the system has already evicted the corresponding cache lines, this second flush sequence does not need to wait and can immediately proceed. In the worst case, all cache lines are still dirty and the resulting wait time is similar to undo logging.

Ideally, redo logging has a better performance than undo logging because of the delayed flushes. However, each data access has to find the most recent version and therefore needs to scan the log. This overhead may eliminate the potential benefit of redo logging. The authors of REWIND [8] therefore let the user choose between undo and redo logging.

### Multiversioning

For undo logging as well as redo logging, the log is a memory location which is frequently written. Some memory technologies suffer from wearing effects and should not be written too often. Multiversioning can be seen as a form of logging, but not to a central location. Writes create a new version of the data. In the commit phase, the new versions are marked as valid, which means they become reachable. The old versions can be disposed immediately afterwards or later in a garbage collection cycle. In the case of a failure before committing, the memory region used for the new version is unreachable because the old one is still being used. The memory for the new version has to be reclaimed.

On first sight, the commit phase seems crucial for recovery. All pointers referring to the data which is changed, have to be adjusted. In most cases, this can not be performed atomically. It is probably due to this complication that multiversioning is only applied for trees so far [13, 58]. Imagine a change to a node in the tree. After a new version of the node is created, the parent node has to be changed so that it points to the new version. This change also creates a new version of the parent node and this scheme crawls up to the root. An explicit recovery is therefore not required, because only the last change - of the root - commits the transaction. What is left is memory reclamation.

While undo and redo logging may vary the granularity of the log information, multiversioning is tightly coupled to the data structure and best applied on object granularity. If a single bit is toggled, the logging schemes may write the address of this bit and its old state to the log. For multiversioning, a new copy of the whole object, e.g., a node in the tree, is created. This leads to additional copy cost. The advantage of multiversioning is that it does not need a second copy operation and writes only $o(n) + m(n)$ bytes to NVRAM. In this regard, $o(n)$ is the size of objects encapsulating the changed data. There is no overhead in the log, but additional data required to make the new version valid. This overhead, e.g., copying parent nodes in a tree, is $m(n)$.

Multiversioning requires at least two flush sequences: one for the new version to be written to NVRAM and another one for all the other data structures which are changed.

### Handcrafted Solutions

The logging schemes are generic and make optimizations for special use cases hard. File systems and key value stores are currently the target applications for many NVRAM projects. The central data structures for both applications are trees. Consequently, researchers came up with highly optimized tree variants [11, 64, 48, 34] which reduce the number of costly flush operations required for consistency.

As an example, consider a B+Tree - a tree which stores data only in leafs and these leafs are connected in a doubly linked list. Yang et al. [64] preserve only the linked list and reconstruct the remaining tree during recovery based on this list. Because these solutions are highly application dependent, their costs can not be predicted easily.

## 3.6 Summary

Existing hardware optimizations, like reordering and caches, are harmful for data structures which are stored in NVRAM. Research on the field of ordering requirements and the development of adequate persistency models is complex and far from reaching a consensus. Failures, like power outages, may result in incomplete operations and require transactional mechanisms to avoid the effects of partial updates. These mechanisms are either undo or redo logging, multiversioning, or highly specialized data structures. All of these schemes have to copy data prior to modifications. Avoiding duplicates is only possible with handcrafted solutions. Therefore, the three general schemes require to copy at least as much bytes as modified. Optimizations can only address the additional meta data used by the schemes.

The amount of data which has to be flushed depends on the working set of a transaction scheme. A log entry's size, for example, determines how many cache lines have to be flushed. Minimizing the working set therefore reduces the number of flushes. A further optimization target is the number of flush sequences. Initially, undo and redo logging require three flush sequences: one to write data to the log, one to validate the logged data, and one after applying the modifications in-place. The second flush sequence, which validates the log entry, has already been removed by prior research. How to reduce the number of flush sequences and flushes even further is subject of the next chapter.

CHAPTER 4

# Fine Grained Transactions with Cache Line Granularity

The storage hierarchy has been employed in computer systems for a long time. On top of main memory are multiple levels faster, but smaller caches as well as registers. Non-volatile Random-Access-Memory (NVRAM) may replace Dynamic Random-Access-Memory (DRAM) and take its place in the storage hierarchy. If the performance of NVRAM is similar to DRAM, the upper levels of caches will continue to exist. And, at first, they will remain volatile. As a result, modifications of data are first buffered in these upper memory levels. Changes become only persistent, when they are flushed from caches and written to NVRAM.

Programmers need to initiate cache flushes manually and with great care, because a missing eviction may harm the integrity of data. The opposite, too excessive use of flushes, degrades the performance since flushes are expensive. In addition to the programmer's burden of using flushes correctly, the hardware may flush data at hardly foreseeable times. Due to these implicitly triggered flushes, the order of writes to NVRAM may differ from the program order of writes. Explicit and implicit flushes have to be considered by software which operates on persistent, consistent data.

Transactional schemes can be used to preserve the consistency of data in NVRAM. These schemes store the state from before the modifications until all changes are complete. Prior work separates the old state from the modified version in space, e.g., by using a log. With dedicated, distinct memory regions, a transaction modifies at least two cache lines and these have to be flushed. This chapter shows how to reduce the number of flushes by keeping both backup and modified data in the same cache line.

The next section describes the prerequisites for this work and thereby defines the foundations in both hardware and software. Afterwards, the transaction mechanism for a single cache line is presented in Section 4.2. The similarity of the new cache line transaction mechanism and the existing hardware instruction Compare and Swap (CAS) is subject to Section 4.3. It addresses the question whether existing synchronization algorithms based on CAS can be adapted to cache line transactions. Thereafter follow two case studies which show how to built data structures using the cache line transaction mechanism to preserve the integrity of persistent data. Section 4.5 continues with extensions for the transaction mechanism that remove the limitation of allowing only a single cache line per transaction.

As each of the extensions brings different benefits and costs, guidance on selecting one of them is given at the end.

## 4.1 Prerequisites

The target setting for this work is a commodity computer with DRAM modules as main memory. NVRAM is expected to be connected in the same way as DRAM, so that at least one memory module contains non-volatile main memory (see Figure 4.1).

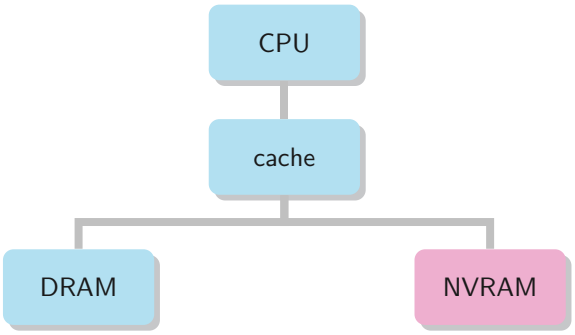



**Figure 4.1:** Schematics of the target system. NVRAM is connected in the same way as DRAM.

Software interacts with NVRAM by means of traditional load and store operations. This can be achieved by mapping NVRAM directly into address spaces where DRAM and NVRAM coexist, as shown in Figure 4.2.

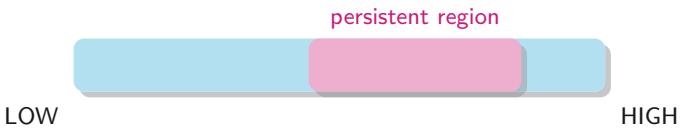



**Figure 4.2:** Address space of a process. NVRAM is available as a dedicated region in the address space.

The system is also expected to be built on top of an existing software stack. That means NVRAM is only an option in a legacy operating system. The Operating System (OS) has NVRAM support, e.g., to map persistent regions. but processes remain volatile and are restarted after a failure. Only the information in a persistent region may survive failures if flushes are used correctly.

For simplicity, the persistent memory region is expected to be at a fixed location so that processes can easily find and use it. Reality might be a bit more complex, because multiple regions may coexist. This implementation detail is not important for the concept presented in this thesis.

Another aspect which is out of this thesis' scope is integrity of references. When a reference to volatile data is stored in a persistent region, the information which is referred to is lost after a failure. Programmers have to avoid such references.

On the hardware side, there is one last important requirement: a cache line is written to NVRAM atomically: either entirely or not at all. Today, cache lines are already the granularity of interactions between cache and main memory. This interface is not expected to change. Internally, the NVRAM module may also contain volatile buffers where it stores data temporarily. In order to be able to flush these buffers upon a failure, the system needs additional capacity. Condit et al. [13] calculated that a capacitor of approximately $50\mu F$ is required to assure that a 64 byte cache line is atomically written to their target NVRAM hardware. They also argue that it is feasible to expect such a capacitor on an NVRAM module.

64 bytes are just one example for a cache line size. It is common on modern Intel systems, e.g., the Intel Core i5 [14]. Other hardware manufacturers implement different sizes, like 128 bytes on IBM POWER4 Central Processing Units (CPUs) [56]. Although the concepts presented in this thesis are independent from the actual hardware, they will benefit from larger cache lines.

The definition of atomicity of cache line writes is of utmost importance for this thesis. Note that it does not imply that writes to a cache line are atomic. Typical instructions fetch data from memory and store it in a register. Changes are made on the register's contents and, at some point, the information is written back to memory. Actually, the data is not written directly to main memory, but to a line in the cache. This thesis assumes that a register forms the smallest unit of transfers and a write of register values is therefore atomic up to the level of cache lines. It may not happen that a register is only partially transferred to a cache line. Multiple writes to the same cache line can be triggered before its contents are transferred to NVRAM. Once a cache line is evicted from the cache, either implicitly or explicitly, it is guaranteed that all bytes in that cache line manifest in NVRAM.

All of the upcoming solutions enforce a specific cache line layout and expect given bytes in a cache line to hold specific information. Programmers are responsible to structure their information so that it adheres to the required scheme.

If not denoted otherwise, all of the subsequent discussions consider power outages as their target failure scenario. When the power runs out, all data stored in volatile memory regions, like registers or caches, is lost. Only the information that as already been evicted from caches to NVRAM survives power outages.

## 4.2 Single Cache Line Transactions

Existing transactional mechanisms for NVRAM, like undo logs, require multiple cache flush operations. While a transaction is in progress, at least two states of the data coexist: the last active version and an in-flight version. Typically, these states are stored in different memory regions. Since both memory

regions are modified in a transaction, they have to be flushed to NVRAM, resulting in at least two flush operations. If the two states would reside on a single cache line, it is possible to cut the number of flushes down to one. This basic idea of cache line transactions is elaborated in more detail next.

In order to minimize the number of flushes, the two states of data modified in a transaction are stored in a single contiguous memory region: a cache line. The cache line is split into multiple parts: two slots for data and an additional part for meta data. One of the data slots is used to preserve the old state and the second one stores the in-flight version. Meta data indicates which of the two slots is currently active. A schematic illustration of the cache line layout is given in Figure 4.3.


**Figure 4.3:** Cache line layout. Two slots store data and the remaining part is for meta data.

When a transaction starts, the currently active data is copied to the other slot as the in-flight version. All changes are performed on the in-flight version. At commit time, the meaning of the two slots changes and the in-flight version is activated. The backup does not have to be copied back to the first slot, as a single bit in the meta data determines the currently active slot. The next transaction starts by inspecting the meta data and uses the new version of the data.

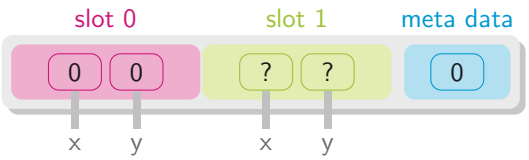
As an example, consider the following update of two variables:

```
x = 5;
y = x + 2;
```

This update shall be performed transactionally with cache line transactions as shown in Figure 4.4. Both values reside in a single data slot of the same cache line and are initially set to zero (a). In the first two steps (b and c), the contents of `x` and `y` are copied to the inactive slot. Then, the modifications are made in the inactive slot (d and e). Finally (f), the meta data is updated and the inactive slot containing the updated data becomes active. At this point, a subsequent transaction should use the updated data. This is guaranteed for sequential transactions in the running system, but failures require further care.

A transaction may fail, e.g., because of a power outage, at any time. As cache line write backs are unforeseeable, any of the states shown in Figure 4.4 may manifest in NVRAM, even the intermediate ones. At first, consider the effects when finding the states (a) to (e) after a failure. In these states, only the currently inactive slot has been changed. A new transaction that finds one of these states starts by inspecting the meta data. Because the meta data was not yet changed, the transaction continues with the old state and ignores the partial update. Only if the final state (f) with the updated meta data is found by a subsequent transaction, the new data is used. Consequently, premature write-backs cause no harm.

```
// initially x and y are set to 0
```




**(a)** Initial cache line contents. Slot 0 is active.

```
slots[!active].x = slots[active].x;
```


**(b)** A copy of $x$ is written to the currently inactive slot.

```
slots[!active].y = slots[active].y;
```


**(c)** A copy of $y$ is written to the currently inactive slot.

```
slots[!active].x = 5;
```


**(d)** The value of $x$ is changed in the currently inactive slot.

```
slots[!active].y = slots[!active].x + 2;
```


**(e)** The value of $y$ is changed in the currently inactive slot.

```
active = !active;
```


**(f)** The change is committed by toggling the active bit. Slot 1 is now active and the cache line has to be flushed.

**Figure 4.4:** Example for a cache line transaction. The values x and y are updated. The cache line stores both values in one of its slots. At the beginning of the transaction, they are copied to the second slot. Then, the second slot is updated and becomes active during commit.

One step is left: when the transaction has finished, the new data should be used, even after a failure. To make the changes persistent, the cache line has to be flushed before the next transaction starts. It is important to preserve program order for writing the active bit. It has to be the last action performed before the flush.

All in all, this scheme allows to modify data within the same slot of a cache line atomically with respect to persistence. Data can be a single large value or multiple fields. In contrast to other schemes which modify at least two cache lines, this idea requires only a single flush at the end.

## 4.3 Cache Line Transactions vs Compare and Swap

In the proposed transaction scheme, modifications of data in a cache line slot become atomic from the view of persistence. A transaction ends by toggling the valid bit in the final step. The subsequent flush makes the changes visible for future incarnations of the system. Partial updates are invisible after a power outage because the valid bit still points to the old version. Thereby, the scheme allows

for atomic updates of a small consecutive memory region. Since most modern processors feature instructions for atomic updates of a small consecutive memory region with CAS, one may ask: how do these approaches differ? In order to answer this question, a short overview of CAS is given before it is compared to cache line transactions.

CAS is a typical hardware primitive for synchronizing concurrent writes to a shared memory location. Imagine two threads on different cores of a shared memory machine. Both want to increment a shared data structure, e.g., a shared counter. Initially, the counter is set to 42. The first thread starts, reads the current value (42) and stores it in a local register. Then, the second thread is active and also reads 42 and stores it locally. Afterwards, the first thread increments the value in its register and writes the resulting 43 to the shared location. The second thread performs the same steps. At the end, the counter is set to 43, although two increments have been performed. Figure 4.5 summarizes this conflict.

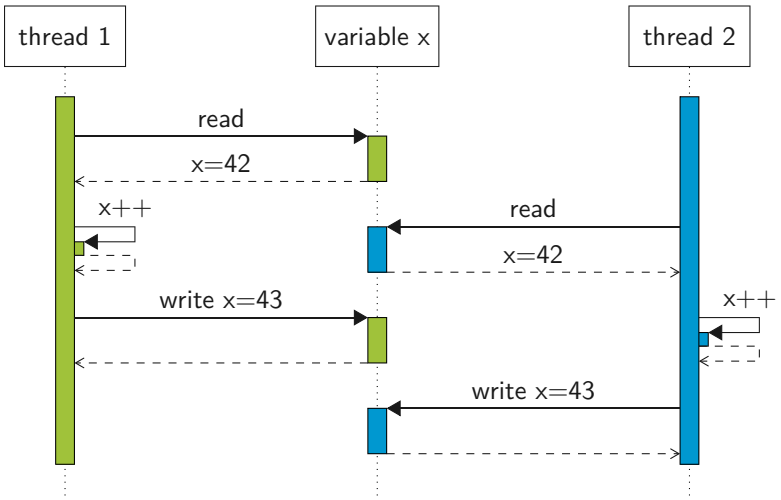



**Figure 4.5:** Concurrent updates of a shared variable. Thread 2 does not observe the changes made by thread 1 and overwrites them.

The problem of synchronizing concurrent activities has been addressed with a variety of solutions. One of them is the CAS instruction. It takes three parameters: a memory location, the current value, and the new value. Atomically, the hardware checks whether the data at the memory location is equal to the expected current value and if so, it writes the new value. Because the operation is atomic, no other thread may interrupt it.

In the case of the shared counter, both threads may use CAS to update the counter's value. As before, both want to write 43 as the new value and expect 42 to be still the existing value. Therefore, one thread will precede the other and successfully update the counter. For the second one CAS fails, because the counter does no longer hold the expected value of 42. The result of a CAS operation is signaled to the programmer who may act accordingly. Figure 4.6 illustrates this sequence.

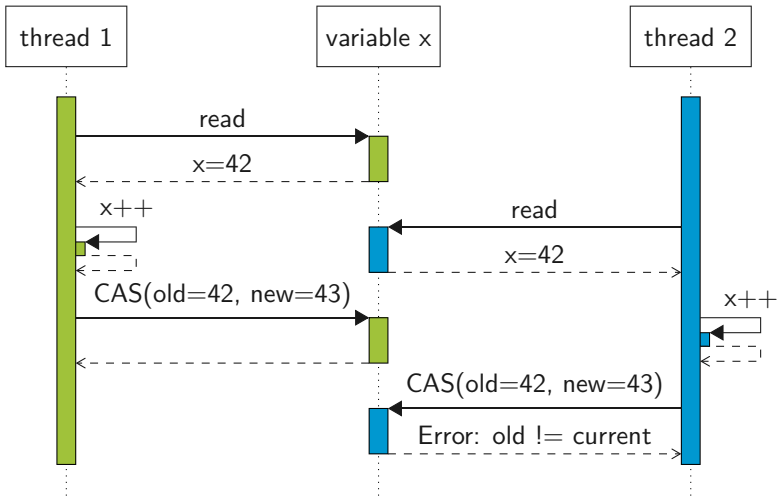



**Figure 4.6:** Concurrent updates of a shared variable using CAS. Thread 2 is able to observe the changes made by thread 1 and does not overwrite the other update.

Both CAS and cache line transactions allow for atomic updates. Their target settings are slightly different: CAS is for concurrent activities working on shared data. Cache line transactions are designed for sequential activities working on persistent data. Actually, these contexts are related to each other and the persistent setting can be seen as a special case of the concurrent setting.

When a persistent data structure is updated in NVRAM, the currently running activity starts a transaction. Before the changes are complete, a failure causes the activity to terminate. Later on, when the failure is gone and the system is running again, another activity also wants to use the persistent data. The following is an equivalent scenario with concurrent activities: one activity starts a transaction on shared data, is suspended and the second activity starts. One option for synchronizing these concurrent activities is to block the second one until the first one has finished, for example with a lock. If the first activity continues eventually, it releases the lock and allows the other one to start its changes. The first one may also fail and never release the lock. Exactly that may happen with NVRAM. The first activity is interrupted and when the second one is active after a restart, the first one will never continue (see Figure 4.7).

Up to this point, we identified the challenge of preserving consistency of persistent data as a special case of a concurrent synchronization problem. We can now consider whether synchronization schemes for the concurrent setting can be adapted to NVRAM. The common strategy of blocking synchronization is inapplicable, because waiting for the interrupted, hence aborted, activity to continue after a failure is not an option. An alternative is non-blocking synchronization, which is typically based on CAS. If CAS and cache line transactions are related to each other and the synchronization settings are similar, it might be possible to adapt non-blocking synchronization algorithms using CAS to the persistent setting with cache line transactions. Ideally, an existing non-blocking synchronization algorithm can be modified by replacing all occurrences of CAS with cache line transactions resulting in an algorithm

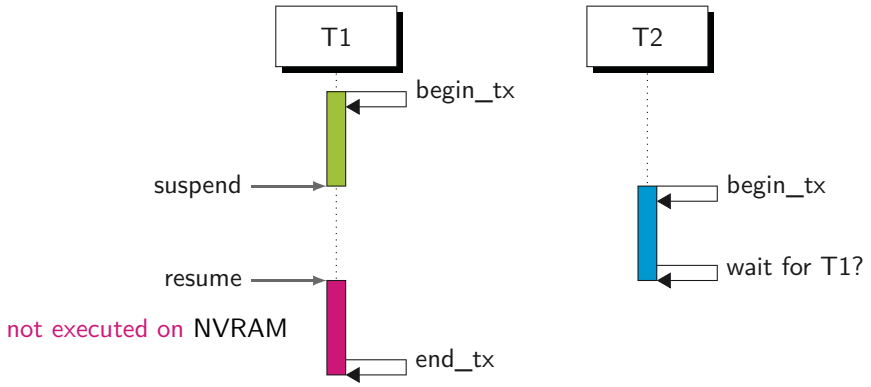



**Figure 4.7:** Transaction T1 starts and modifies persistent data. It is interrupted and another transaction T2 starts to operate on the same data. Whether the second transaction may wait for the first one to complete, depends on the setting. For concurrent threads, T1 may resume at some point. However, if T1 is a transaction on persistent data that is interrupted by a failure, T2 should not wait, because T1 will never continue.

which guarantees data consistency with respect to concurrency as well as persistence. This idea is subject of the upcoming subsections.

### 4.3.1 Non-blocking List

A practical example of non-blocking synchronization for linked lists was given by Harris [20]. Each element in the list stores a pointer to its successor and a data item. Imagine the following situation: The list already contains three elements: A, B, and C. Thread T1 wants to insert element D in between B and C. Thread T2 wants to insert element E, also in between B and C. This situation is depicted in Figure 4.8.

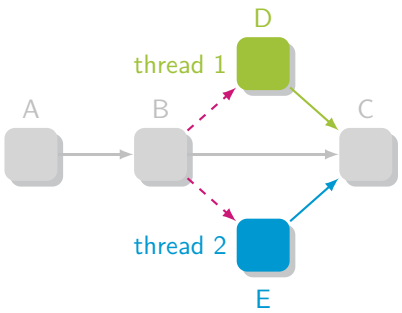



**Figure 4.8:** List on which two threads operate concurrently. One thread wants to insert element D and the other one wants to insert element E. Both have to adjust the pointer from B to point to their new elements.

The critical place is the update of B's next pointer. T1 wants to set it to D and T2 wants to set it to E. A non-blocking synchronization may use CAS and only one thread would successfully perform

its update. The other one has to retry the insert on the resulting list. Using CAS at this point is sufficient for synchronizing concurrent insert operations, but another problem remains.

The second problem is faced when one thread wants to remove element B and the other one wants to insert D in between B and C, as shown in Figure 4.9.

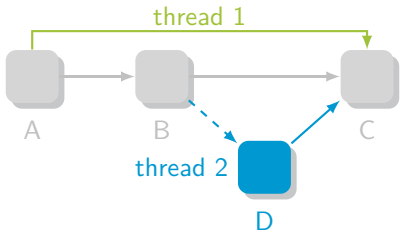


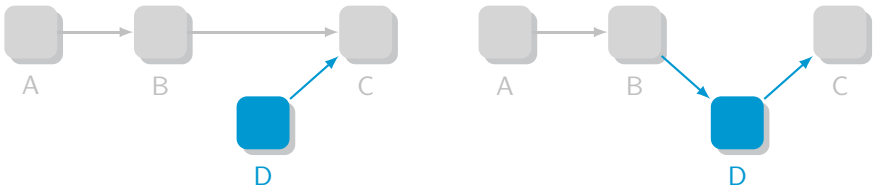

**Figure 4.9:** List on which two threads operate concurrently. One thread wants to delete element B and the other one wants to insert element D.

The thread inserting D is not informed of B's deletion by the other thread. Similar, the other thread does not see that B's next pointer is changed and C is no longer the new successor of A. Consequently, the deletion of B would also delete the newly inserted D, which is incorrect. Harris [20] suggests to mark the element B, which causes the inconsistency, prior to its removal. Practically, the least significant bit of B's next pointer is used and written with CAS. Now, both threads conflict when they want to update B and the conflict is resolved when B is updated using CAS.

What happens when this algorithm is applied for synchronization on a persistent list? Initially, the list containing elements A, B, and C is assumed to be valid in NVRAM. Thread 1 already created element D, set its next pointer to C and also flushed element D. This situation is depicted in Figure 4.10a.

At first, assume that thread 1 starts inserting element D, but fails due to a power outage. The new instance, thread 2, wants to delete element B. It finds the list in a state depending on the progress of thread 1. If thread 1 was interrupted before updating B's next pointer it finds the old, initial list (Figure 4.10a). The list is still valid and does not need further care. Thread 2 may therefore proceed as normal. Please note that garbage collection is triggered appropriately to dispose element D. Alternatively, thread 1 has already updated B's next pointer and this change was written to NVRAM. The update is therefore complete and thread 2 finds all four elements in the list, as shown in Figure 4.10b. All in all, atomic updates of the next pointers are crucial for consistency of the list when inserting an element. As long as these updates are performed with CAS, cache line transactions are not needed.

Now, the order of operations is inverted: the current thread wants to delete element B and fails. Afterwards, thread 2 inserts element D. Before B is actually deleted, it is marked as *to be deleted*. A traversal of the list after the crash may still find the element with the marker as in Figure 4.11a. It may now finish the deletion or remove the marker. If the deletion was able to perform a further step, it sets not only the marker, but also changes A's next pointer (Figure 4.11b). In that case, a




**(a)** Thread 1 started to insert element D. In the next step, B's next pointer will be set from C to D.

**(b)** Thread 1 inserted element D completely.

**Figure 4.10:** A persistent list initially containing elements A, B, and C. Thread 1 is about to insert element D in between B and C.

traversal does not find B after a crash. As before, B's memory needs to be reclaimed if it is not used anymore.


**(a)** Thread 1 started to remove element B by setting a marker.

**(b)** Thread 1 removed element D completely.

**Figure 4.11:** Removal of an element from a persistent list.

Both insert and delete operations rely on CAS for updates of pointers and the delete marker. Typically, pointer are small enough so that the hardware may modify them atomically. In that case, cache line transactions are not needed if the algorithm is adapted for a persistent list. Smart pointers encoding additional information, however, may exceed the hardware capacities and need cache line transactions.

The most important aspect in a persistent list is that changes have to be written to NVRAM explicitly, by means of flushes. Flushes have to be triggered after each CAS – a perfect situation for automatic adaption of the algorithm for a persistent setting. However, additional flushes are also required, for example after the new list element has been created and set to an initial state. Whether the need of these flushes can be detected automatically, is not yet clear. It will not be discussed further here, because of the absence of a need for cache line transactions in the transformed algorithm.

The previous discussion showed that it is not worthwile to adapt the non-blocking list based on CAS to cache line transactions on NVRAM. A second setting for the integration of cache line transactions in existing algorithms is analyzed next. Its intention is to overcome size limitation of CAS which is determined by the hardware and allows for Multi-Word CAS. Since cache line transaction restrict their users to a cache line slot, the ideas applied to an extension of CAS might also be suitable for cache line transactions.

### 4.3.2 Multi-Word Compare and Swap with Cache Line Transactions

Imagine a situation where multiple words need to be updated atomically. For illustration, augment a linked list with an additional `size` field depicting the number of elements which are currently in the list. Whenever data is inserted or deleted, this size field has to be updated in addition to pointers that need to be changed in list elements. The CAS hardware instruction does not allow for updates of multiple data regions. As a solution for this problem, Harris et al. [21] suggested a multi-word compare and swap which is based on the existing CAS operation.

#### Multi-Word CAS

The basic idea of Multi-word CAS is a helping mechanism. Prior to any change, a thread creates a descriptor and notes down what it is going to do. Each entry in the descriptor contains a memory location, an expected old value, and a new value. It follows an intermediate step in which a link to this descriptor is written to all the locations which have to be updated. Simultaneous writes by other threads are detected when one of the locations does not contain the expected old value. The conflict resolution is discussed later. For now, the thread succeeds with the intermediate step. It then starts to update all memory locations again, but this time the link to the descriptor is replaced by the new value. Once all these updates are complete, the whole operation is finished.

A conflict happens when a thread wants to update a memory location but the expected value was overwritten by another thread in the meantime. If the thread finds a descriptor instead of the expected value, the other activity is not yet complete. Instead of waiting for the other thread to complete its operations, the current thread starts to help the other one. Based on the other descriptor, it starts to perform all the updates denoted therein. Otherwise, if the thread does not find a descriptor, another thread has successfully updated the location. Therefore, the current thread's update fails and it has to revoke all of its changes, i.e., by replacing the links to the descriptor with the old values.

The proposed algorithm requires to differentiate a reference to a descriptor from other references and values. A solution is to reduce the range of values by dedicating one bit to identify references to descriptors. This idea can also be applied in the remainder of this discussion. Alternatively, an additional bit in the meta data of each cache line can be used to indicate that the slot holds a descriptor reference.

Can the idea of multi-word CAS also be used to extend cache line transactions from a single memory location to multiple ones? If a thread is interrupted by power outage, the future instance may help and finish the updates. To check the applicability of multi-word CAS, each CAS is replaced by a cache line transaction. As seen in the previous list example, additional flushes might be required, for example for the descriptor. These are discussed separately after a sketch of the basic algorithm.

### Multi-Word CAS with Cache Line Transactions

To illustrate the new algorithm, the same update as before is applied to two values:

```
x = 5;
y = x + 2;
```

This time, the variables reside on distinct cache lines which conform to the cache line layout required for transactions. As shown in Figure 4.12b both values are initially zero and the left slot of each cache line is valid.

Thread T1 starts and creates a descriptor (Figure 4.12a). It records 0 as the old value and 5 as the new value of x. For y, the entry contains 0 as the old value and 7 as the new one, respectively.

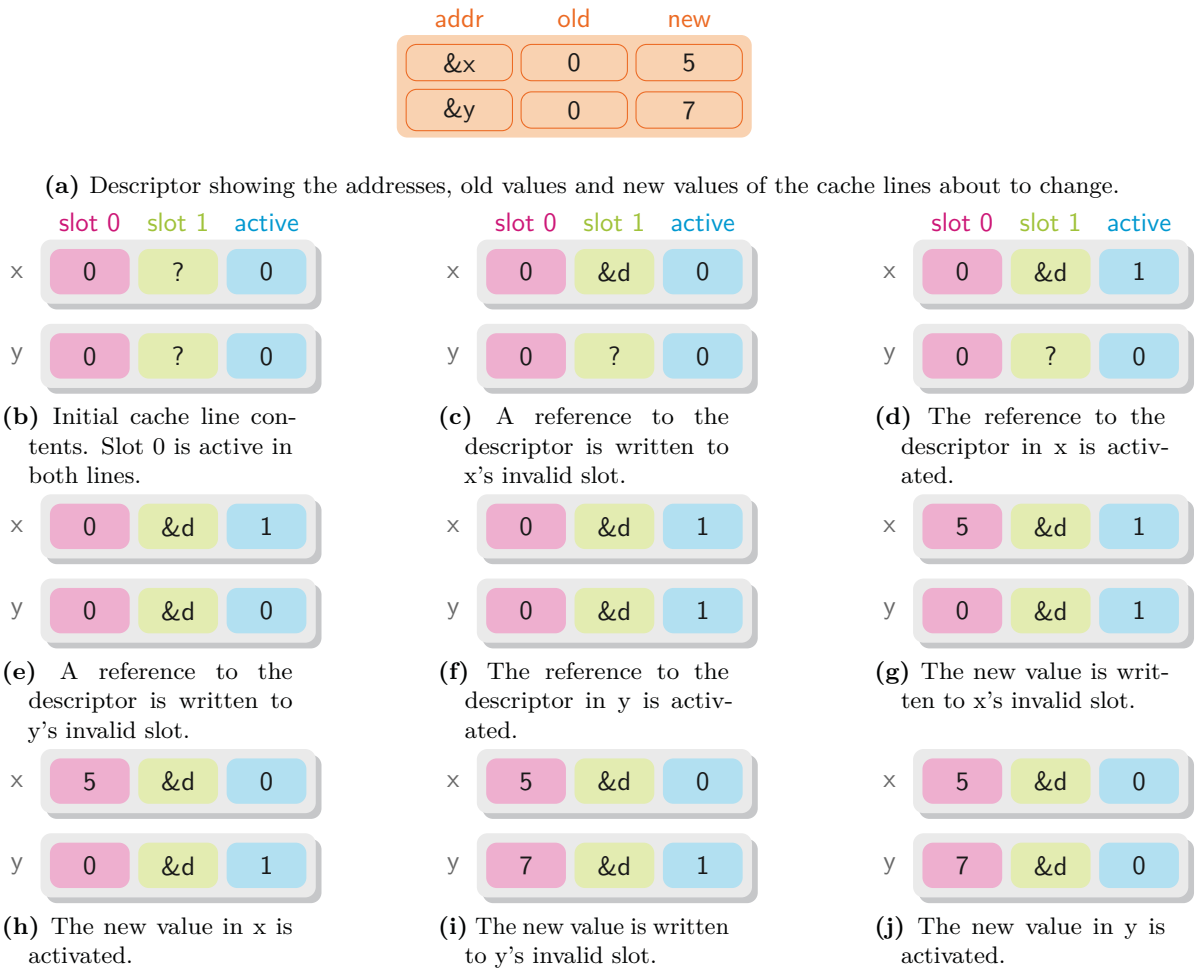
Now, x's value is replaced by a reference to the descriptor. According to the cache line transaction, the current value of x is copied to the inactive slot first. Then, it is overwritten with a reference to the descriptor (Figure 4.12c). Finally, the meta data is adjusted and the descriptor becomes valid (Figure 4.12d). The same procedure is applied to the cache line containing y. Its old value is copied, replaced by a reference to the descriptor, and activated, as shown in Figure 4.12e and 4.12f. Note, that the cache lines containing x and y have no yet been flushed. The necessity of flushes is discussed later on.

After all cache lines in the transaction have been linked to the descriptor, the next phase begins. During this phase, the transaction is carried out and the new value is written to the cache lines. Starting with x, the cache line transaction copies the descriptor reference to the inactive slot, replaces it with x's new value and activates the new one (Figure 4.12g and 4.12h). The same is repeated for y which contains now the new value in its active slot (Figure 4.12i and 4.12j). Finally, once all updates have been applied, the descriptor can be disposed.

### Failures

Up to this point, the algorithm does not flush any data to NVRAM. The descriptor has to be valid and persistent before a reference to it is written. Otherwise, the data would refer to a non-existing or partially created descriptor. Consequently, the creation of a descriptor is finalized by a flush of the corresponding memory region.

**Recovery** Assuming the system fails somewhere in the algorithm shown above: what happens afterwards? The new instance of the system may initiate a recovery phase. In that case, it is helpful if recovery can find all descriptors, e.g, because they are in a dedicated memory region. Additional meta data in a descriptor is required in order to signal whether it was successfully created so that a failure during the creation of a descriptor can be detected. Moreover, the descriptor has to be flushed before the marker can be set and the marker also needs to be flushed. When combined, all these




**(a)** Descriptor showing the addresses, old values and new values of the cache lines about to change.

**(b)** Initial cache line contents. Slot 0 is active in both lines.

**(c)** A reference to the descriptor is written to x's invalid slot.

**(d)** The reference to the descriptor in x is activated.

**(e)** A reference to the descriptor is written to y's invalid slot.

**(f)** The reference to the descriptor in y is activated.

**(g)** The new value is written to x's invalid slot.

**(h)** The new value in x is activated.

**(i)** The new value is written to y's invalid slot.

**(j)** The new value in y is activated.

**Figure 4.12:** Multi-word CAS adapted to use cache line transactions. Each of the cache lines adheres to the required layout. In addition, a persistent descriptor is created before the transaction starts to modify the first cache line.

steps resemble a logging approach. The dedicated memory region for the descriptors is a log and the information stored in the descriptors can either be used to redo or undo the operation. By storing undo information, redo information and a valid marker in the descriptor, the adapted multi-word CAS requires as least as many flushes for the descriptor as a log requires for the log entry. One flush is needed for each cache line in the descriptor. The logging equivalent is the log entry. Each cache line therein has to be flushed, too. Since logs typically contain either undo information or redo information, the combination of both in a descriptor leads to more flushes. Consequently, there is no benefit of the multi-word CAS at this point.

Without an explicit recovery phase, the system would continue after a failure and later access the cache lines containing `x` or `y` again. If it finds a reference to a descriptor in one of the variables, it follows the multi-word CAS algorithm and starts to help. The interrupted transaction is continued based on

the descriptor's contents. Only when the descriptor is flushed before one of the cache lines refers to it, helping is possible. Otherwise, the system may find a partial descriptor and can not continue. If the descriptor is flushed prior to referencing it in a cache line, there is no need for additional meta data for the validity of a descriptor. The recovery based approach needs such meta data because recovery may find an incomplete descriptor. Flushing the descriptor and eliminating recovery avoids that situation.

**Memory Reclamation** The system may fail when a descriptor exists, but no cache line refers to it. That may happen at the beginning of a transaction shortly after the descriptor has been created. The situation can also occur at the end of a transaction when all values have been successfully updated. In order to avoid memory leaks, a garbage collector is required to reclaim the memory used by the descriptor.

**Flushes** It has already been argued that the descriptor has to be flushed after its creation. Additionally, the final state of the modified cache lines has to be flushed before the descriptor can be disposed in order to assure that the new values are in NVRAM. In between these operations are multiple cache line transactions that write descriptors and the final values to the cache lines. According to the cache line transaction scheme, each of the individual transactions also needs a flush after the meta data has been updated. Each cache line needs to be flushed twice then: after the descriptor is written and after the final value is activated.

To reduce the number of flushes, the necessity of each flush has to be considered. The final flush is essential to activate the new values. The remaining flush is triggered after the descriptor is written to the cache line. If that flush is omitted, a problematic situation may arise. The first cache line is modified and its value is flushed to NVRAM. The second cache line is still cached. In NVRAM, there is still the old value without a link to the descriptor. If the system fails at this point, the interrupted transaction can not be detected, because none of the cache line in NVRAM refers to the descriptor. Hence, all the flushes are essential.

### Concurrency

The algorithm is similar to logging, but requires even more flushes. The initial inspiration, multi-word CAS, was designed as a synchronization mechanism for concurrent activities. Is this still a property of the adapted algorithm and does it bring a benefit when compared to traditional logging?

Undo logging for persistence is orthogonal to concurrency control. If the log is shared among all concurrent activities, it requires synchronization. Data modified in a transaction might be part of concurrent updates and also calls for extra synchronization. Extra synchronization is therefore required for concurrent activities that operate on persistent data using an undo log.

With redo logs, each activity creates a log entry containing all changes. Again, creating entries at a central log calls for synchronization. Applying the changes to the actual data later on can be done by a dedicated activity so that this write-back does not need further care. However, two activities may modify the same data concurrently. As before, each activity creates a log entry. Before the entries are marked as valid, a new phase has to be started and the log has to be scanned for conflicting entries. In the case of a conflict, one of the activities has to be aborted.

In the adapted multi-word CAS, each CAS operation has been replaced by a cache line transaction. Do cache line transactions support concurrency? The usage of CAS in the original multi-word points to parts of the algorithm where synchronization for concurrency is required. The first one are writes of descriptor references. According to the cache line transaction scheme, such a descriptor reference is written to the inactive slot first. Afterwards, the meta data is updated. For concurrent activities, these writes are not atomic and require further care.

One idea to solve the problem is to add a lock to a cache line's meta data, as shown in Figure 4.13. A cache line transaction would then start by acquiring the lock. A conflicting transaction on the same cache line that finds the lock has to wait. Now, the algorithm has become a blocking synchronization solution and therefore changed drastically. As argued before, blocking is impractical for persistence, because locks may never be released.

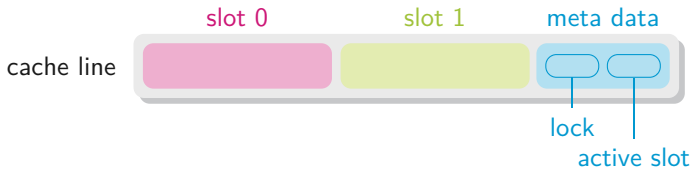



**Figure 4.13:** Cache line layout with an additional lock in the meta data to control concurrent access to the cache line.

There is also a non-blocking option to synchronize a cache line transaction, but only for the special case of the adapted multi-word CAS. If the adapted algorithm is being used for cache line transactions, even for those that manipulate only a single cache line, the slots in the cache line have fixed semantics. Only the left slot is used to store data and the right slot is only used for descriptor references, as illustrated in Figure 4.14. While the actual data may be larger than the granularity of CAS, the reference is not.

If CAS is being used to write the descriptor reference slot, the write becomes atomic and conflicts of concurrent activities can be detected. In order to apply CAS, the initial value of the reference slot has to be included in each item in a transaction's descriptor.

In summary, the adapted multi-word CAS is able to guarantee consistency for concurrent activities as well as the consistency for persistent data. The scheme relies on garbage collection for descriptors and

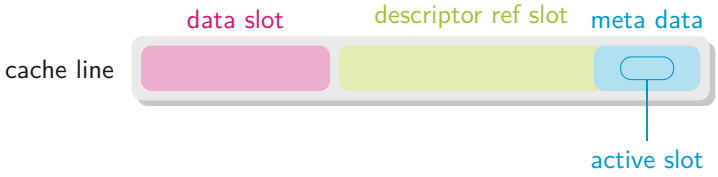



**Figure 4.14:** Cache line layout for multi-word CAS with cache line transactions. The left slot in the cache line holds data and the other one is used to store references to descriptors. Meta data is used to indicate whether the descriptor or the data is valid in order to handle failures.

uses more flushes than logging strategies. Its built-in concurrency control is a big plus in comparison to undo and redo logging which require further synchronization. Optimizations that reduce the number of flushes for the adapted multi-word CAS are discussed next.

### Optimizations

For each update, a descriptor in the multi-word CAS contains the location which is modified, the old value, and the new value. Therefore, it combines the information stored in undo log and a redo log. All this information has to be flushed after the descriptor is created. Log entries also have to be flushed, but since log entries contain less information, the descriptor in multi-word CAS requires more flushes.

In order to shrink the descriptors size, the information stored therein has to be critically examined. What happens if the redo information is removed? The new state of the data is recorded for the helping scheme and enables other transactions to continue an interrupted transaction. If it would be removed, helping is no longer possible and the lock-free property of the synchronization is lost. Hence, this information is crucial.

The descriptor size can also be shrunken by removing the undo information. This data seems to be redundant because each cache line also preserves its old state. Can the undo information be removed from a descriptor? In the original implementation, the undo data is used for two purposes. The first purpose is a roll-back of conflicting transactions. Multiple transactions may start at the same time and create their descriptors. Next, a reference to the descriptor is written to each memory location. Conflicts are detected in that step when one thread finds a reference to a descriptor from another thread. Ideally, the thread is able to help the other one. While helping, the thread may have to write a descriptor to a memory location that already contains a descriptor from the threads initial transaction. Starting to help would now lead to a deadlock and therefore one transaction has to be aborted. All of the memory locations to which a descriptor for the aborted transaction have been written, need a roll-back to the original value. With cache line transactions, it is sufficient to toggle the meta data's valid bit in each cache line to restore the old value. The undo information in the descriptor is not required for this scenario. In order to determine whether it can be removed, the second purpose of the undo information is considered next.

Multi-word CAS uses the undo information to write references to the descriptor atomically. The references are written with CAS which checks that no other thread has changed the value in the meantime. This check requires the presence of the old value which was valid when the transaction started. If concurrency is desired, conflicts can also be detected using CAS for the reference slots in the adapted algorithm. As stated before, this requires the undo information in a descriptor to hold not only the old value of the data, but also the old value of the descriptor reference slot. With only the descriptor references being required for conflict detection, the old state of the data is no longer required in the descriptor. Hence, the descriptor's size is reduced and fewer flushes are required.

#### Discussion

In comparison to logging, the adapted multi-word CAS does not require a dedicated recovery phase. Since recovery should be rare, its removal may not bring an enormous benefit in most settings. However, the lack of a recovery phase can be desirable when timing guarantees have to be given. Moreover, memory regions containing descriptors have to be reclaimed after failures, which can be seen as recovery, too. A big advantage of the adapted multi-word CAS is that concurrency control is already included. In contrast, the logging approaches require explicit synchronization of concurrent activities.

On the downside, multi-word CAS requires more cache line flushes than the logging schemes. With logs, each log entry and the data have to be flushed. With multi-word CAS, the descriptor and the data have to be flushed. Cache lines containing data have to be flushed when the descriptor is written and after the new data has been written. The goal of reducing the number of flushes can therefore not be reached.

The final question is: what is the benefit of the adapted multi-word CAS in comparison to the original multi-word CAS with cache line flushes? For the original algorithm to be applicable to persistent memory, it is necessary to flush each descriptor after it has been created. Whenever a reference to the descriptor is written to a memory location, a flush has to be triggered. Once the new data is written, another flush is needed. All in all, the number of flushes is equal in both algorithms. With the additional flushes, multi-word CAS would also work without cache line transactions. Their most important benefit is a coarser granularity. The original algorithm is limited to modifications which the hardware supports atomically. With cache line transactions, the granularity equals the size of the cache line slot used to store data. Algorithms which modify small consecutive memory regions in their transactions may benefit from the cache line scheme.

It is important to keep in mind that cache line transactions require the data layout to change. Data which can otherwise reside consecutively in memory is now mixed with meta data of the cache lines. The performance effects are not easily foreseeable. Even when no transaction is active, space is reserved for meta data and descriptor references. Hence, the cache pollution is higher and the hardware pre-fetch logic may have a harder time in predicting the access pattern. The performance of the algorithm without cache line transactions but with flushes could therefore be higher.

The previous discussions used the similarity of cache line transactions and CAS to find use cases for cache line transactions. With their helping mechanism, lock-free data structures may be a good starting point for the design of persistent data structures. However, this approach can not benefit from cache line transactions. A transformation of existing algorithms based on CAS for cache line transaction has not yet been found. However, it is possible to craft data structures using cache line transactions manually. The next section shows examples for persistent data structures fitting into a single cache line slot.

## 4.4 Data Structures For Cache Line Transactions

While the cache line transaction scheme has the advantages of being simple and using only a single cache line flush, it is limited to a special setting: only a single cache line slot can be modified in a transaction. Therefore, the data has to reside in a consecutive memory region of limited size. The next sections show how to build data structures adhering to these requirements.

### 4.4.1 Doubly Linked List

The elements in a doubly linked list typically store three items: a pointer to the previous list element, a pointer to the next element, and data. A list can either be direct or indirect, meaning it may store data in each list element or refer to external data. Figure 4.15 shows an indirect list, because the list elements reference external information.

Using pointers in list elements allows to give an upper bound for the size of each element: the size of three pointers. On a 64 bit system, the three 8 byte pointers sum up to 24 bytes. Two list elements consume 48 bytes and three elements 72 bytes, respectively. Consequently, at most 2 elements fit into a 64 byte cache line. If the cache line transaction scheme should be used to preserve the integrity of the list, the size limitation is even stricter, because the scheme limits the usable region to approximately half a cache line. Each list element would then reside on a separate cache line.

Cache line transactions enable atomic modifications within a single cache line. Transactions that cover more than one cache line are not supported. However, typical operations on a doubly linked list modify more than one list element. An example is the removal of an element which requires a change of the predecessor's next pointer and a change of the successor's previous pointer. As discussed before, each element probably resides on a separate cache line. Therefore, the list operations cover more than one cache line and can not be performed atomically with cache line transactions.

In order to use cache line transactions for a persistent doubly linked list, the list layout needs a transformation so that the whole list fits into a single cache line. The size limitation of the usable region in a cache line limits the number of elements which can be stored in the list. It is therefore possible to limit the width of pointers used to address list elements. With only 8 elements in a list, the

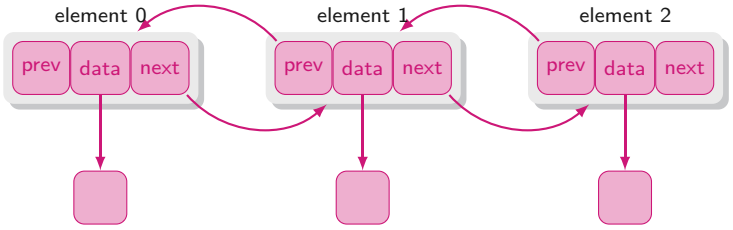



**Figure 4.15:** Three elements in a doubly linked list. Each element refers to its predecessor and its successor. Moreover, an element contains a pointer to data which is stored externally.

next and previous pointers can be reduced to 3 bits. Actually, 3 bits permit at most 7 elements in the list, because there is a need for `NIL` pointers when an element has no predecessor or successor. The idea of restricted pointer widths can also be applied to data pointers if the number of data elements is limited.

In order to discuss the applicability of this idea, a cache line size of 64 bytes is used. Is the number of list elements and the number of data items which can be used with such a cache line and the cache line transaction scheme acceptable? A pointer width of $x$ bits can be used to identify $2^x - 1$ elements (the remaining value is used for `NIL`). Assume that $x$ is the width of a pointer to a list element. The list may then contain up to $2^x - 1$ elements. Each element has two pointers to other list elements, which cover $x$ bits each. Therefore, each element requires at least $2 \times x$ bits for the list internal pointers. For all the $2^x - 1$ list elements, this sums up to $(2^x - 1) \times 2 \times x$ bits. An overview of the resulting memory consumption for list elements depending on the pointer width is given in Table 4.1. If 5 bits are used for each pointer, the list is able to store 31 elements which occupy 310 bits in total. Note that pointers to data are not covered yet.

| bits per pointer | number of list elements | memory consumption of all elements |
|---|---|---|
| $x$ | $2^x - 1$ | $(2^x - 1) \times 2 \times x$ |
| 1 | 1 | 2 bits |
| 2 | 3 | 18 bits |
| 3 | 7 | 56 bits |
| 4 | 15 | 120 bits |
| 5 | 31 | 310 bits |

**Table 4.1:** Space consumption for elements in a doubly linked list dependent on the width of pointers. Pointers to data elements are not considered here. If list element pointers are 5 bits wide, the list can store up to 31 elements. The pointers used by these list elements sum up to 310 bits.

According to the layout required for the cache line transactions, each cache line needs one bit in its meta data to index the currently valid slot. The remainder is split equally among the two slots. Therefore, the size of a slot is: $\lfloor (cacheline_{width} - 1)/2 \rfloor$. In the case of a 64 bytes = 512 bits cache line, a slot can store $\lfloor (512 - 1)/2 \rfloor$ bits = 255 bits. Using 5 bits per pointer leads to a list size of 310 bits, which exceeds the width of a cache line slot. At most 15 list items (using 4 bits for pointers)

may therefore be stored in the list.

For 15 list elements, the list internal structure already occupies 120 of 255 bits in the cache line slot. The remaining 135 bits can be used for pointers to data elements. Each list item can now use 135/15 bits = 9 bits for pointers to data items, allowing for 512 unique data items. However, the list is not yet complete. It needs a pointer to the current list head element. This additional meta data limits the amount of memory which can be used to store pointers to data items. Instead of 135 bits, only $135-5$ bits remain for pointers to data. Each of the 15 list elements can therefore use only $(135-5)/15 = 8$ bits to refer to one of 256 data elements. In total, the list occupies two list pointers and one data pointer per list element and a head pointer:

$$\begin{aligned} width_{list} =& width_{list\_elem} * nr_{list\_elems} + width_{head\_pointer} \\ =& (2 \times width_{list\_pointer} + width_{data\_pointer}) * (2^{width_{list\_pointer}} - 1) + width_{list\_pointer} \end{aligned}$$

With the specific sizes $width_{list\_pointer} = 4$ bits and $width_{data\_pointer} = 8$ bits, the sum of bits is:

$$\begin{aligned} width_{list} =& (2 \times 4 + 8) \times (2^4 - 1) + 4 \\ =& 16 \times 15 + 4 \\ =& 240 + 4 \\ =& 244 \end{aligned}$$

All in all, the list consumes 244 out of 255 bits in a cache line slot. Remaining bits can be used to store the list's size, if needed.

A visual representation of the resulting list is shown in Figure 4.16. The layout adheres to the example which used 4 bits for list pointers and 8 bits for data pointers. In addition to the 15 list elements, the list's head and size are stored. All this information is stored in 248 bits which form a slot in the cache line. The cache line contains two slots, each with a list, and the remainder is used for meta data.

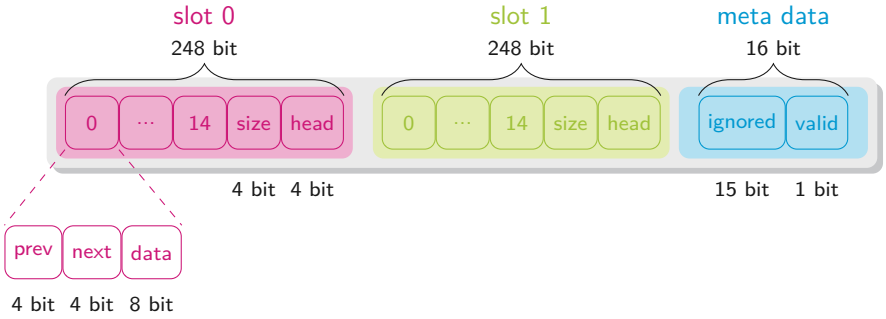



**Figure 4.16:** Doubly linked list in the cache line transaction scheme with a 64 byte cache line. It can store up to 15 list elements and may refer to 256 data items.

This subsection introduced a doubly linked list with a limited number of list elements and a restricted set of values stored in the list. For 64 byte cache lines, the list may contain up to 15 elements and manage up to 256 different data values. All operations on such a list can be carried out with cache line transactions and therefore require only a single cache line flush. The effect for situations where the restrictions are acceptable, is immense. As shown in the beginning of this section, other lists may store each element on an individual cache line. List operations which modify multiple elements change multiple cache lines in that case. At least one flush per modified cache line is required to make the changes persistent. Consistency preserving strategies increase that number of flushes even further. The next subsection shows how to built a hash table based on the cache line list.

### 4.4.2 Hash Table

A hash table is an associative container which maps keys to values. In one of its simplest forms, these tables use a hash function to determine the hash of a key. The result is an index into an array where the value is located. Depending on the hash function, keys may collide and result in the same index. Therefore, the array contains not only a single value, but a list of key-value pairs.

With slight modifications, the cache line list shown in Section 4.4.1 can be used to build a hash table, as illustrated in Figure 4.17. A hash table does not need a doubly linked list — a linked list is sufficient and the list elements do not need a pointer to the previous element. Each list element has to store a key and a value. Such a key-value pair can be stored in the data region of a list element. For the specific example of 64 byte cache lines, an element in the doubly linked list used 16 bits: 4 bits for the previous pointer, 4 bits for the next pointer, and 8 bits for data. With no need for the previous pointer, 12 bits can be used for key-value pairs. When splitting this region equally for keys and values, 6 bits are available for each key and for each value. This allows for 64 different keys and 64 different values in the whole hash table.

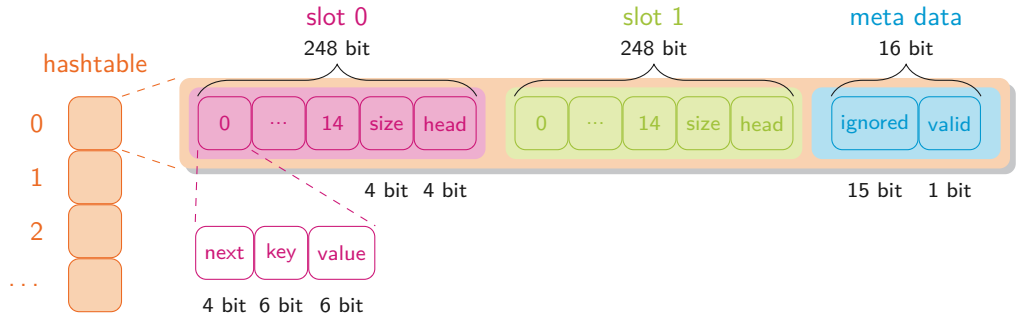



**Figure 4.17:** Hash table constructed from linked lists in cache lines. The hash table stores an array consisting of cache lines. Each cache line contains a linked list that can be managed with a single cache line transaction and adheres to the basic cache line layout.

Operations on a hash table modify only one of the lists. Since the list modification can be performed with only one cache line flush, the hash table operations also require only one flush.

The capacity of the hash table is limited, as the cache line determines the maximum number of keys and values per index. Can this limitation be overcome by using multiple levels of lists? Figure 4.18 sketches the idea of storing a doubly linked cache line list in each hash table slot. The items in such a list refer to cache line lists holding key-value pairs as before.

Due to the indirection, the modified design allows to store more key-value pairs per index. In the specific example in Figure 4.18, the first level, doubly linked list stores up to 15 references to second level lists, each containing at most 15 key-value pairs. In comparison to 15 key-value pairs supported by the initial design, 225 are an enormous capacity increase. Can another level of indirection increase the capacity even further? Even more, is it possible to use the basic scheme and the list design as a building block for other, more complex data structures?

The basic hash table contains one cache line per index. When an element is inserted, the hash function determines the index and only the cache line for this index is modified. In the case of the multi-level hash table, the answer is not as simple. It depends on whether all data structures are allocated statically or dynamically. In the first case, when all cache lines are allocated in advance, the first level list does not need any transactions. Since all second level lists exist already, the is no insert or delete operation on the first level anymore, rendering these lists read-only. Otherwise, if second level lists are created on demand, the first level lists are modified.

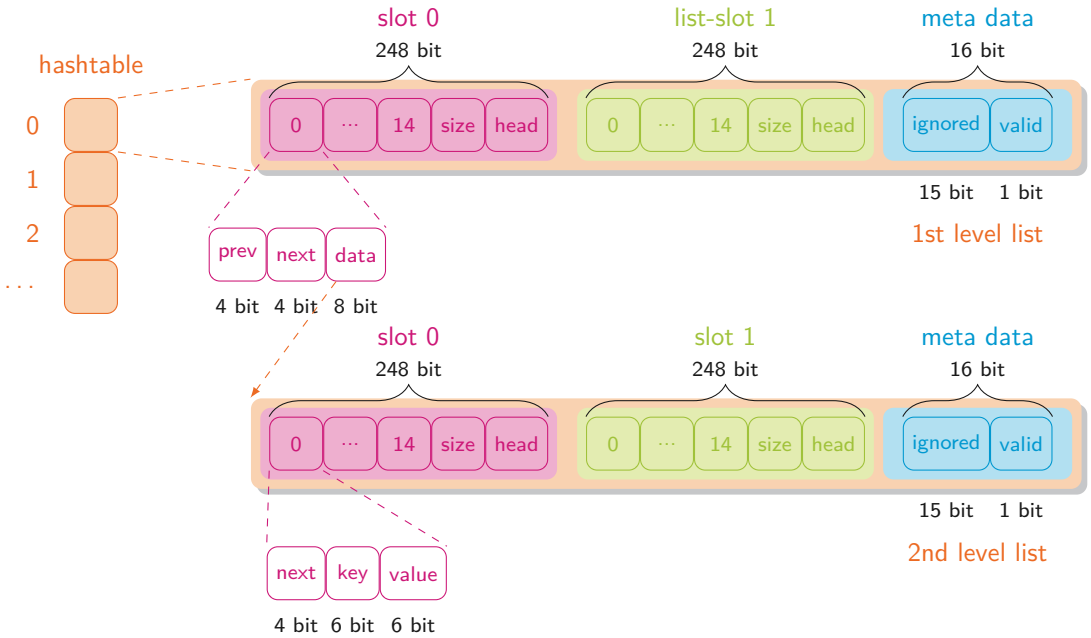



**Figure 4.18:** Using multiple levels of lists to extend the hash table's capacity. Each index slot in the hash table is represented by a doubly linked cache line list. An item in this list refers to a cache line holding a list of key-value pairs.

When inserting a key-value pair into a dynamically allocated, two level hash table, a new second level list has to be created when the existing second level lists do not have any free slots. Now, a new

second level list has to be created, which includes memory allocation, initialization, and adding the new list to the matching first level list. In addition, the key-value pair has to be inserted in the newly created second level list. There are at least three cache lines modified during this operation: at least one for memory allocation, another one containing the first level list, and one for the second level list. Since cache line transactions support only transactions on a single cache line so far, they can not guarantee transactional semantics for the dynamically allocated hash table. Although inserting ordering primitives, like fences, at the correct places might be sufficient, this situation shows the limitations of the basic cache line transaction scheme.

In addition to dynamic allocation, another typical hash table operation modifies more than one cache line in a transaction: re-hashing. When the hash function is not ideal for a use case, it may happen that the majority of indices in the table is unused, but some are heavily frequented. Such a situation can be resolved by switching to another hash function. As a consequence, the existing table has to be reconstructed using the new hash function. Similar to the other operations, re-hashing should be performed transactionally, but it will most likely modify more than a single cache line. Again, cache line transactions can not address this problem and, consequently, they can not be used as building blocks for more complex data structures so far.

The basic cache line transaction scheme supports transactions covering a limited, consecutive memory region within a single cache line. Compared to logging, cache line transactions feature two prominent benefits. The first benefit is the single cache line flush which is used to complete a transaction. Logging, instead, requires more than one flush because log information and modified data reside in distinct memory regions. Secondly, cache line transactions have implicit recovery. There is no need to roll back when a transaction is interrupted. Instead, the system automatically continues with the version that was valid before the interrupted transaction had started.

Cache line transactions do not only have benefits. Their size limitations, described earlier, might not be acceptable in all scenarios. Some transactions, such as re-hashing a hash table, may have to cover more than a single cache line. Possibilities to extend the cache line transaction scheme so that it covers multiple cache lines while still keeping the number of flushes at a minimum are shown next.

## 4.5 Multi Cache Line Transactions

The basic cache line transaction scheme supports transactions covering a limited consecutive memory region within a single cache line. In contrast to other transaction schemes based on logging, such a transaction needs only a single flush — at commit time. Moreover, recovery is implicit and no recovery routine is required. Is it possible to extend the mechanism so that it covers multiple cache lines while preserving the benefits of the simple scheme?

At first, consider the effects of applying single cache line transactions from Section 4.2 to a situation that is out of its scope. Because the situation does not fit to the limitations of the scheme, it provides

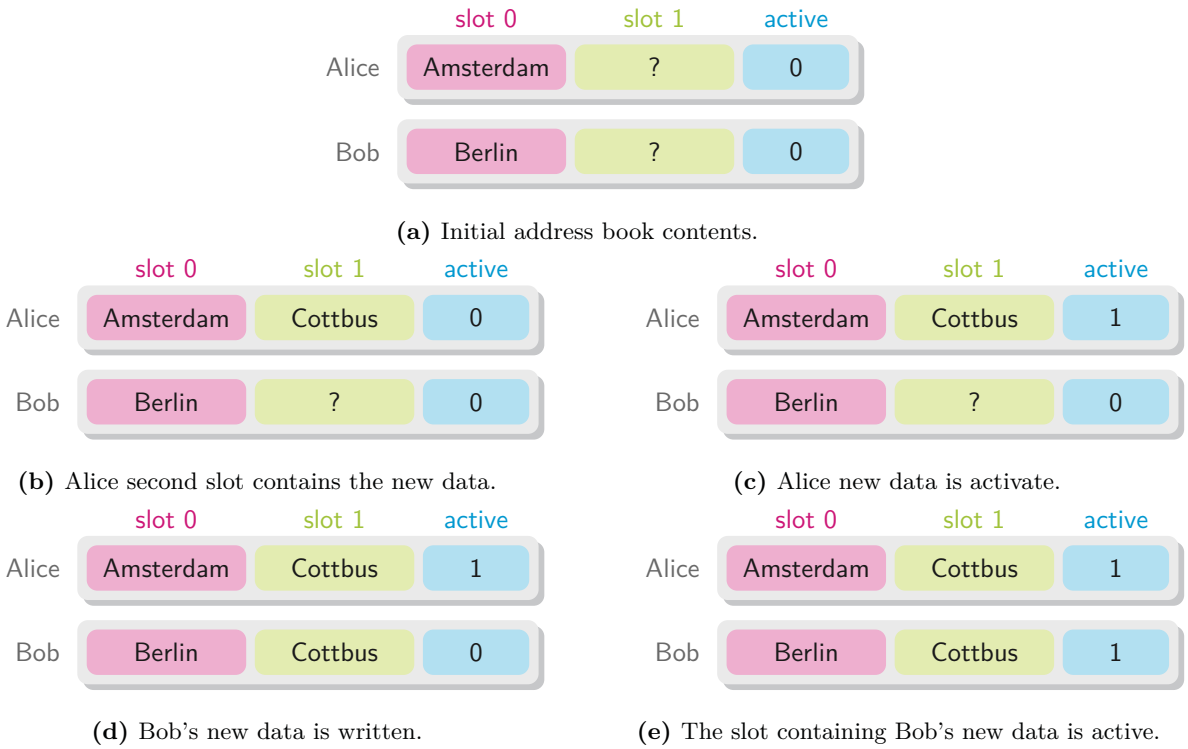
a first impression of the challenges which multi cache line transaction schemes have to face. The example relies on a persistent address book with two entries, one for Alice and one Bob. Both move in together in Cottbus and their address book entries should be updated transactionally. Each of these entries resides on its own cache line and its data layout conforms to the single cache line transaction mechanism. Initially, the old state is active in both entries, as depicted in Figure 4.19a. In the next step, the transaction may proceed by updating Alice' entry and activating it (Figures 4.19b and 4.19c). The same steps are then performed for Bob's cache line as shown in Figures 4.19d and 4.19e). Finally, the cache lines are flushed to NVRAM.

Two challenges have to be addressed with multi cache line transactions. First, flushes of multiple cache lines are sequential and not performed in a single instruction. The system may fail after flushing the cache line containing Alice' entry. As a result, Alice has been updated, but Bob's change can not be observed. Since transactions are prohibited to manifest partially, this is a violation of transactional semantics. Second, the hardware may trigger flushes on its own. Installing new data in a cache may evict one of the two cache lines before they are explicitly flushed. Such an eviction may cause a write-back of Bob's cache line before Alice' line is flushed. If the system fails before Alice' flush is triggered, Bob's update would be completely written to NVRAM while the state of Alice is still the original one. Completing Bob's update, but not the one for Alice, is none of the states performed by the transaction in Figure 4.19. Early, automatic evictions may therefore disrupt program order.

The address book example shown above presents two challenges for multi cache line transactions: early evictions and the absence of atomicity when flushing multiple cache lines. They are addressed with an explicit recovery routine in this work. Implicit recovery, as in single cache line transactions, is desirable but not crucial. An explicit recovery phase brings additional efforts, like the design of a recovery algorithm and its execution time. However, recovery is only triggered in response to a failure, which should be an exceptional situation. All of the upcoming solutions accept the overhead of a recovery routine in the rare case of failure. Cache line flushes, on the other hand, happen during normal transaction processing. Their influence on the overall performance is therefore enormous and the number of flushes should be kept at a minimum.

For a closer look at recovery, consider the address book once more. Intermediate states exist when only one cache line transaction is complete, but the other one is not. What is an adequate reaction to the detection of such an invalid state? Because the old version of the data is preserved in the cache lines, the already completed cache lines could be rolled back to their previous state by simply resetting the active bit. However, the system needs to know on which cache lines the roll-back should be performed. Keeping track of cache lines modified by a transaction is not required so far. All of the upcoming solutions restore the previously active state in an explicit recovery phase. Recovery is started when the system resumes after a failure. It has to address the following challenges:

- Detect whether a transaction has been interrupted.




**(a)** Initial address book contents.

**(b)** Alice second slot contains the new data.

**(c)** Alice new data is activate.

**(d)** Bob's new data is written.

**(e)** The slot containing Bob's new data is active.

**Figure 4.19:** Modifications of two cache lines using the single cache line transaction scheme. At first, the cache line containing Alice' address book entry is updated and activated. Afterwards follows Bob's update. A flush of both cache lines when they reached the state in 4.19e completes the transaction. The intermediate states in 4.19c and 4.19d could also manifest due to cache evictions. Multi cache line transactions have to address these invalid states.

- Roll-back all the changes made by the interrupted transaction which have already been written to NVRAM. In order to perform the required roll-backs, the system has to be able to find all modified cache lines.

Transactions that modify multiple cache lines need to group the changes so that they are either completely carried out or do not manifest at all. Logging approaches address the problem of atomicity by buffering changes in a separate memory region. As already discussed, this decoupling results in additional flush operations which harm the performance. The previously shown adaption of Multi-Word CAS also allows for multiple cache lines to be modified in a single transaction. Similar to logging, it requires an additional persistent descriptor and, hence, flushes. In comparison to logging, the number of flushes in the adapted Multi-Word CAS is even higher, but the algorithm had concurrency control built in. Extensions of the basic cache line transaction scheme for transactions on multiple cache lines are shown next. Again, the goal is to keep the number of cache line flushes at a minimum. Concurrent operations are not yet considered.

One could use the meta data in the cache lines to group cache lines belonging to the same transaction. For example, augment the meta data by a *dirty* bit that is set for all the cache lines which are modified

in a transaction. This idea has two side effects. At commit time, the dirty bits are set in all of the modified cache lines. Once the data is flushed, it is no longer dirty and the bits have to be cleared. This has to be done in a separate step which takes time to clear the dirty bits, but also traverses the transaction's working set for a second time. In addition, recovery has to detect whether the most recent transaction already started its commit. If it did not start, all the dirty lines should be recovered. Otherwise, if commit has been started, some of the dirty bits may already be cleared and the new values could be found in NVRAM. If the remaining dirty cache lines are recovered, the transaction would only partially be rolled back, leading to inconsistent data. Additional information about the transaction's state is therefore needed. Alternatively, the system needs other information which does not have to be reset on a transaction commit.

A final topic has to be covered before introducing the actual algorithms. Is it acceptable to initiate a roll-back for the most recent transaction even if it has been completed? From the point of persistency, it makes no difference whether the transaction fails at commit time or afterwards, as long as a new transaction has not been started. If a new transaction would have began, the new one would be the most recent one and recovery would consider this one. Therefore, reverting the most recent completed transaction should be tolerable in most cases. Such a roll-back might be unacceptable if there are side effects. Imagine a system that informs an external component about the completion of a transaction. If the transaction is rolled back after a failure, the external component may still assume that the transaction has been completed. In the case of such a side-effect, a roll-back is unacceptable.

All in all, to keep the number of flushes at a minimum, a multi cache line transaction mechanism should have two properties. First, the amount of persistent information that is kept separately from the modified cache lines should be as small as possible, because it has to be flushed. Second, persistent state which has to be reset at commit time should be avoided. Otherwise, each cache line has to be flushed twice. Additional functionality, such as using multiple cache lines in a transaction, implies extra cost. The three different multi cache line transaction mechanisms shown next vary in their costs. They require specific hardware support, limit the size of usable data within a single cache line, or induce additional flushes. An introduction of these extension is given next. Each description starts by showing the cache line layout and a sequence of steps performed in a transaction. The normal operation of transactions is followed by an overview of the recovery routine. A final paragraph summarizes the costs for each variant. After all three multi cache line transaction schemes are introduced, guidance is given for choosing one of them according to the programmers needs.

### 4.5.1 MCL-TXID: Transaction Identifier in Meta Data

#### Cache Line Layout

The first solution groups cache lines which have been modified by a transaction by extending each cache line's meta data. In addition to the information used by the single cache line transaction scheme, a link to a transaction is stored in each cache line, as shown in Figure 4.20. As cache lines modified in

the same transaction share the same link, it is the base of grouping in this scheme. A simple form of that link is an integer unique for each transaction. Think of a transaction identifier, like a sequence number, which is increased whenever a new transaction starts, as discussed next.

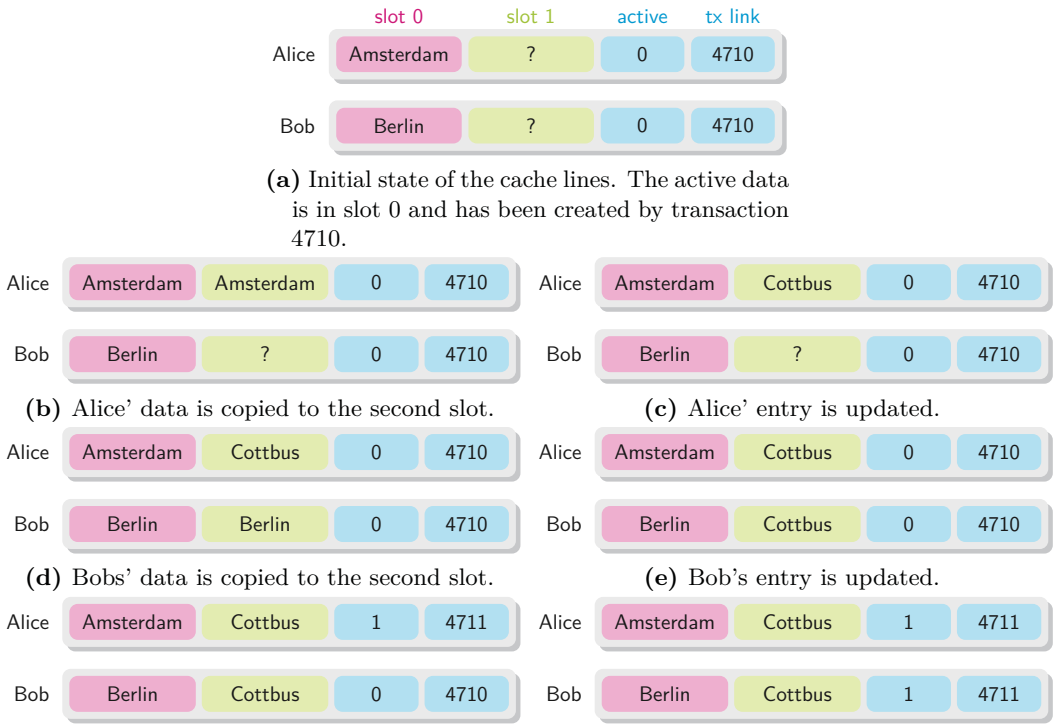

**Figure 4.20:** Cache line layout for MCL-TXID. Meta data does not only contain an active bit, but also a link to the most recent transaction which modified the cache line.

### Transaction Processing

Upon beginning a new transaction, a unique transaction link is acquired. The upcoming steps are similar to the single cache line transaction scheme. When a cache line is accessed for the first time in a transaction, a copy of its active state is written to the inactive slot. All modifications are then performed on the inactive slot. During commit, the new slot is activated. This activation is different from the basic cache line transaction mechanism, because the data is now also linked to the modifying transaction. In addition to toggling the valid bit, the transaction link is written. These two writes, active bit and transaction link, have to be carried out atomically, both fields at the same time. The necessity of that atomic write for the meta data is discussed in the description of the recovery routine. The final step for each cache line is a flush.

A visual application of this algorithm to the address book example is given in Figure 4.21. It starts with the address book entries for Alice and Bob in their initial state. The cache lines are already linked to an earlier transaction, but which one does not matter in this example. At first, Alice' current information is copied to the second slot in (b) and modified in (c). These changes are followed by preparing Bob's cache line in (d) and modifying it in (e). Finally, the commit phase activates the new data and links Alice' cache line to transaction 4711 (f). It can now be flushed. The same is then repeated for Bob (g) and followed by a flush of Bob's cache line. Instead of flushing directly after the activation of a new version, all flushes could also be postponed and triggered at the end.

One aspect has not been mentioned so far: the need for a persistent transaction identifier. The transaction started by acquiring a unique transaction identifier. This identifier has to reside in NVRAM and needs one flush per transaction. It is required for the recovery mechanism presented next.




**(a)** Initial state of the cache lines. The active data is in slot 0 and has been created by transaction 4710.

**(b)** Alice' data is copied to the second slot.

**(c)** Alice' entry is updated.

**(d)** Bobs' data is copied to the second slot.

**(e)** Bob's entry is updated.

**(f)** Alice' change is committed by activating the new slot and linking the cache line to transaction 4711.

**(g)** Bob's change is committed by activating the new slot and linking the cache line to transaction 4711.

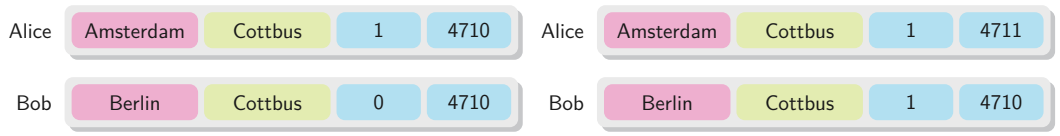
**Figure 4.21:** MCL-TXID uses additional meta data per cache line. The active bit indicates which of the two slots currently contains valid data. Each cache line refers to the most recent transaction operating on this line by a transaction identifier stored in the `tx link` field. This example shows the steps for a transaction with the identifier 4711. Information irrelevant for the example is denoted by a question mark.

### Recovery

Recovery routines can either perform an unconditional roll-back of the most recent transaction or revert the changes only if the transaction was not yet complete. For simplicity, consider that even if the most recent transaction committed successfully, its effects are still reverted. Recovery has to find all cache lines modified by the most recent transaction in the first step. If increasing numbers were used as transaction links, recovery could scan all cache lines and select those with the highest transaction identifier. For each cache line found by the recovery routine, the transaction has already activated the new version, due to the atomic write of active bit and transaction link. Now, the original slot containing the active data has to be restored by toggling the active bit and invalidating the transaction link. Afterwards, the cache line has to be flushed to assure that all roll-backs are complete before the next transaction starts.

The atomic writes of active bit and transaction link are crucial during normal operation and during

recovery. Without an atomic write, either the active bit or the transaction link is written first. An early flush may then write data which can not be recovered. Two examples are shown in Figure 4.22. In both cases, new data has already been activated, but not yet linked. Situation (a) represents an incomplete transaction, because Bob's state has not been modified. The correct response to failure is to undo Alice' changes, but it can not be found by the recovery routine because the transaction identifier is missing. If Bob's meta data is not written atomically, the resulting situation (b) is found during recovery. Although the state is valid – both changes are complete – this situation is harmful for consistency. The unconditional recovery would find Alice' entry and revert it, but can not find Bob's. Consequently, the transactional semantics are violated.




**(a)** Alice' new data is activated, but not yet linked to transaction 4711.

**(b)** Bob's new data is activated, but not yet linked to transaction 4711.

**Figure 4.22:** Two invalid states which can be the result if active bit and transaction link are not written atomically during transaction processing.

Atomic writes during recovery are used for activating the old version and linking to an invalid transaction. Recovery may find Alice' cache line for the most recent transaction in the state depicted in Figure 4.23a: slot 1 containing the location of Cottbus has been activated by transaction 4711. Since transaction 4711 is the most recent transaction, the active bit in Alice' data has to be toggled to reactivate Amsterdam. Figure 4.23b shows the situation when only this bit is toggled. A failure at this point will be followed by a new recovery phase later on. Again, Alice's cache line has been modified by the most recent transaction and recovery would toggle the active bit once more, thereby restoring an invalid state. One idea to avoid this situation is to write an invalid link to each cache line which has been recovered. It may also be acceptable to write additional information about the ongoing recovery process to NVRAM so that later recovery may continue where the first one left off. As argued earlier, recovery is rare and such an overhead is acceptable.

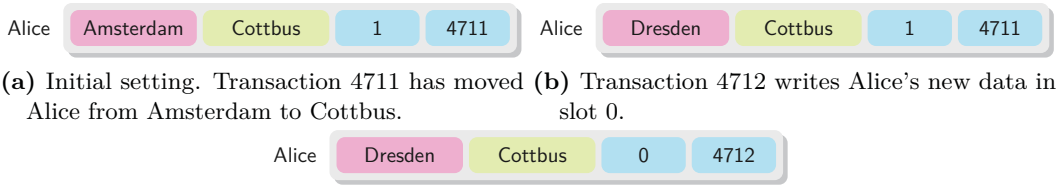

**(a)** Alice' entry as found during recovery. Slot 1 has been activated by transaction 4711.

**(b)** Alice' old state is reactivated by setting the active bit to 0.

**Figure 4.23:** Steps performed during recovery.

The necessity of storing the identifier for the most recent transaction can be seen by inspecting a subsequent transaction. Assume that transaction 4712 wants to update Alice's address book entry and set it to Dresden. The sequence of actions for this step is illustrated in Figure 4.24. At first, Alice' location is copied to the inactive slot and updated in (b). Finally, the new slot is activated and the cache line is linked to transaction 4712 in (c). Crucial is the second step where the new data is

written, but the cache line is still linked to transaction 4711. Based on the information in the cache lines, recovery would identify transaction 4711 as the most recent one and toggle the valid bit in Alice' cache line. Although transaction 4712 has not yet committed, recovery would activate its changes, leading to an inconsistent state.




**(a)** Initial setting. Transaction 4711 has moved Alice from Amsterdam to Cottbus.

**(b)** Transaction 4712 writes Alice's new data in slot 0.

**(c)** Slot 0 is now activated by transaction 4712.

**Figure 4.24:** Transaction 4712 updates Alice' cache line. The situation in b illustrates the need for additional meta data. Recovery may find this cache line and identify transaction 4711 as the most recent one. Since transaction 4712 has already started and recovery would restore the in-flight data from 4712.

Avoiding incorrect activation of subsequent modifications requires the additional persistent transaction identifier denoting the most recent completed transaction. If recovery finds a cache line referring to a newer transaction, the new one has not been complete and needs a roll-back. Otherwise, if both match, the transaction was complete and recovery is finished. Whenever the persistent transaction identifier is incremented, it has to be flushed to NVRAM explicitly. Imagine a flush of the persistent link to transaction 4712 is omitted and transaction 4713 starts. The persistent identifier could still refer to transaction 4711, denoting that 4712 is still in progress. All cache lines from transaction 4712 would now be reverted, although it was complete.

#### Summary

In this variant, each of the cache lines changed by a transaction has to be flushed once. One additional flush per transaction is required to update the persistent transaction identifier pointing to the most recent completed transaction. In addition, there is the requirement of an atomic write for the meta data, which consists of an active bit and a transaction identifier. The amount of usable memory in each cache line is limited by the meta data: $width_{usable} = \frac{width_{cache\_line} - width_{active} - width_{tx\_link}}{2}$

The next variant removes the need for the additional persistent transaction identifier. It thereby reduces the number of flushes at the cost of extended atomicity requirements.

### 4.5.2 MCL-FIXED: Fixed Slots

The previous variant augmented each cache line's meta data with a transaction identifier for grouping cache lines from the same transaction. A recovery routine can find all cache lines that may require a roll-back with the help of that transaction identifier. Whether recovery is necessary, is determined by a persistent transaction identifier. This identifier induces one flush per transaction in addition to the

flushes of all modified cache lines. The version presented in this section does not rely on external data and reduces the number of flushes.

### Cache Line Layout

Multi cache line transactions have to perform two essential tasks for each cache line: copy data to preserve the old state and link the cache line so that the old state can be restored in the case of a failure. The basic scheme and the previously shown scheme use an active bit to toggle the meaning of two cache line slots. When a transaction starts when slot 0 is active, a transaction will use that slot as a backup and write new data to slot 1. During commit, slot 1 becomes active. This approach removes the `active` bit. Instead of toggling its meaning, one cache line slot is reserved for the in-flight version and the other one holds the backup of the last active version, as shown in Figure 4.25.


**Figure 4.25:** Cache line layout for MCL-FIXED. The cache line consists of two slots and meta data. One slot always holds a backup and the other one stores the in-flight version.

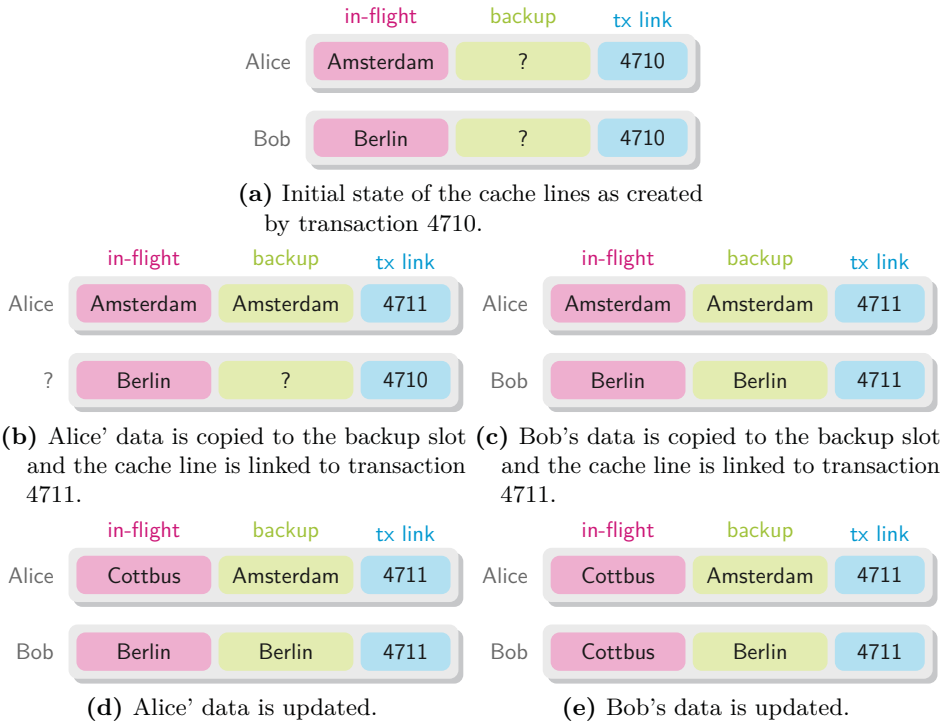
### Transaction Processing

When a transaction modifies a cache line for the first time, it creates a backup in the backup slot and also links the cache line to the modifying transaction atomically. Now, it can proceed with the modification in the in-flight slot. Finally, the cache line is flushed.

The address book update with this variant of multi cache line transactions is shown in Figure 4.26. At first, Alice' data is written to the backup slot and, atomically, the cache line is linked to the transaction (b). This atomic write requires both fields, backup and transaction link, to be consecutive in memory. After the same steps have been applied to Bob's cache line (c), the modifications are performed on the in-flight slots (d and e). Now that all changes are complete, the cache lines are flushed to NVRAM.

### Recovery

Recovery finds the most recent transaction and all its cache lines by inspecting all transaction links. As there is no additional information, it can not be told whether the transaction was complete. At this point, one has to decide whether unconditional roll-backs are acceptable or not. The latter choice brings one additional flush per transaction, because an additional transaction identifier is needed in




**(a)** Initial state of the cache lines as created by transaction 4710.

**(b)** Alice' data is copied to the backup slot and the cache line is linked to transaction 4711.

**(c)** Bob's data is copied to the backup slot and the cache line is linked to transaction 4711.

**(d)** Alice' data is updated.

**(e)** Bob's data is updated.

**Figure 4.26:** Address book update using MCL-FIXED. Note that an atomic write is used to update the backup and transaction link.

this case. Once recovery has identified all the cache lines which should be rolled back, the backup is copied to the in-flight slot in each cache line. These changes are then flushed to NVRAM.

Because the information inspected by recovery is not changed during the recovery itself, it can be repeated. A failure during recovery can trigger another recovery using the same algorithm as the interrupted one without any harmful side effects. In the previous variant, the active bit made recovery much more complicated.

### Summary

All in all, this variant requires as many flushes as cache lines have been modified and is therefore optimal with respect to the number of flushes. If unconditional roll-back is unacceptable, a persistent transaction identifier induces one further flush per transaction. However, backup and transaction link have to be written atomically, which requires hardware support. The exact requirements depend on the width of user data and the width of transaction links. If the cache line is fully utilized, backup and link would occupy more than half a cache line. The usable data in each cache line depends on the size of the transaction link stored in the meta data: $width_{usable} = \frac{width_{cache\_line} - width_{tx\_link}}{2}$

The following, third variant does not need additional persistent data outside of cache lines to detect whether the most recent transaction is complete. Moreover, it reduces the required hardware support

for atomicity. On the downside, less user data can be stored on each cache line.

### 4.5.3 MCL-2TXIDS: Two Transaction Identifiers

MCL-TXID relies on an external identifier to determine whether cache lines from an incomplete transaction exist. The counter can be eliminated with MCL-FIXED, but to the cost of strict hardware requirements for atomic writes. This section presents a third alternative which does not need additional persistent data and comes without strict hardware requirements, but needs additional non-volatile meta data.

#### Cache Line Layout

Both of the previous variants extended the existing meta data with a link to a transaction. This information can be used to find the modified cache lines during recovery and undo the changes therein. Since MCL-TXID writes the backup prior to the link, recovery does not know whether the backup slot contains data from a newly started transaction which is not linked to any of the cache lines yet. It needs the additional persistent information for a decision. The same problem is addressed in MCL-FIXED with an atomic write of backup and transaction link. In summary, the central issue is that two fields have to be updated atomically.

With the single cache line mechanism, two values in a cache line slot can be updated atomically. Their initial values are copied to the second slot where they are modified. When all changes are complete, the new values are activated by updating the meta data and activating the second slot. Instead of adding a transaction link to the existing meta data, this variant stores it in the cache line slots, as depicted in Figure 4.27.

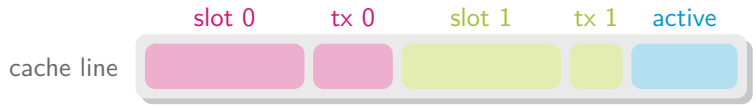


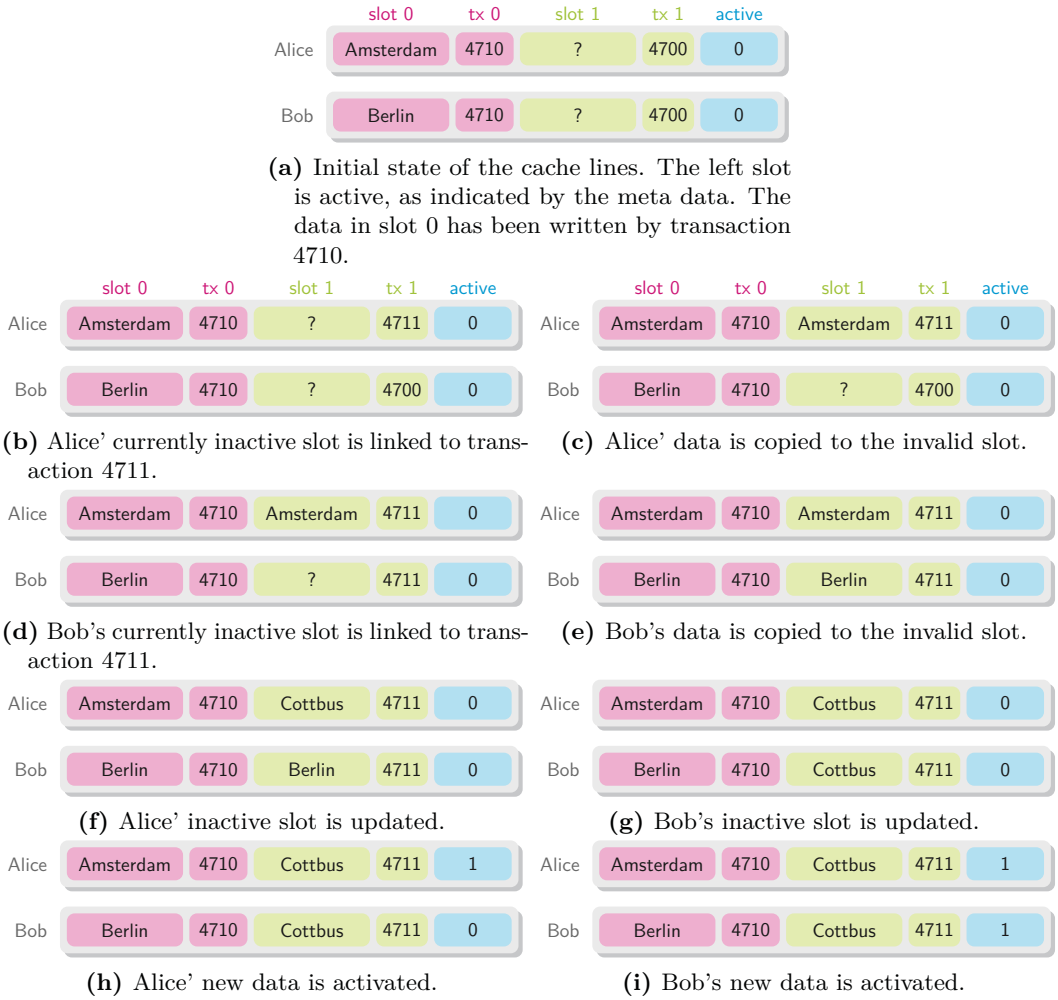

**Figure 4.27:** Cache line layout for MCL-2TXIDS. It is based on the single cache line transaction mechanism. The transaction link is stored in each cache line slot, additionally to user data.

#### Transaction Processing

The first step for each cache line in a transaction is to link the currently inactive slot to the current transaction[1] Then, the currently valid data is copied to the inactive data slot where it is modified. In the last step, the new slot is activated and the cache line is flushed. Figure 4.28 shows this sequence of actions applied to the address book example. Initially, the left slot is active in both cache lines. The

[1] Once linked to the transaction, the cache line has to be flushed. See Appendix B for more details.

transaction starts by setting the transaction link in Alice' inactive slot (b). Afterwards, the current data is copied to the inactive data slot (c). The same is then repeated for Bob (d and e). Changes of the user data are then performed on both inactive slots (f and g). In the final step, the new slots are activated by toggling each valid bit and flushing the cache lines in (h) and (i).




**(a)** Initial state of the cache lines. The left slot is active, as indicated by the meta data. The data in slot 0 has been written by transaction 4710.

**(b)** Alice' currently inactive slot is linked to transaction 4711.

**(c)** Alice' data is copied to the invalid slot.

**(d)** Bob's currently inactive slot is linked to transaction 4711.

**(e)** Bob's data is copied to the invalid slot.

**(f)** Alice' inactive slot is updated.

**(g)** Bob's inactive slot is updated.

**(h)** Alice' new data is activated.

**(i)** Bob's new data is activated.

**Figure 4.28:** Address book update using MCL-2TXIDS.

### Recovery

When the recovery routine searches for cache lines modified by the most recent transaction, it has to inspect both transaction links in each cache line. All cache lines containing a link to this transaction are found during the search. By inspecting these cache lines, the recovery procedure can determine whether it needs to undo changes.

If the most recent transaction finished its commit phase, the slot referring to that transaction is active in all of this transaction's cache lines. All of these cache lines are found and no roll back is needed.

If the transaction did not even start its commit phase, all of the cache lines found by recovery refer to the most recent transaction in their inactive slot. In this case, none of the changes will affect the processing of subsequent transactions and recovery is finished.

The most complex situation is when a transaction has started its commit phase, but was unable to complete it. This situation can be detected when the transaction identifier is found in an inactive slot of a cache line and in an active slot of another cache line at the same time. Recovery can tell exactly how many cache lines have been modified by the interrupted transaction, because all of its cache lines are already linked. A roll-back has to be performed for those cache lines where the new slot has already been activated. The active bit has to be toggled for these cache lines to restore the old state. Then, the new state has to be flushed explicitly in order to avoid changes from subsequent transactions to manifest before recovery is completed.

Recovery modifies only the active bit in some cache lines' meta data and only if a roll-back is needed. If recovery is interrupted, the next initiated recovery finds the same transaction identifiers and therefore selects the most recent one again. Some of the cache lines may have already been restored to their old valid state by the interrupted routine. When the new routine finds these cache lines, it detects that the most recent transaction identifier is inactive in there and starts to roll-back the remaining cache lines from that transaction. Consequently, recovery does not need to detect that recovery has been interrupted. The highest transaction id has been found and can be incremented for the next transaction.

#### Summary

This variant of multi cache line transactions works without additional persistent information and triggers one flush per modified cache line. Moreover, it requires no special hardware support. The cost of this scheme is the additional space occupied by transaction links. Instead of a single link, as used by MCL-TXID, two links have to be stored. Therefore, the width of user data per cache line is limited to: $width_{usable} = \frac{width_{cache\_line} - width_{meta} - 2 \times width_{tx\_link}}{2}$ Whether this is acceptable depends on the user requirements. Section 4.5.4 shows how to select the appropriate algorithm according to the user's needs.

### 4.5.4 Comparison

Three extensions of the simple cache line transaction scheme which support changes of multiple cache lines in one transaction have been presented in this section. Table 4.2 summarizes the different properties of the three variants. It summarized the following aspects:

**Flushes per TX** Each of them has to flush all the cache lines modified by the transaction. The additional persistent transaction identifier essential for MCL-TXID induces one additional flush per transaction.

**Atomicity Requirement** Both MCL-TXID and MCL-FIXED require hardware support for atomic writes. MCL-TXID has to write the active bit together with a transaction identifier. The sum of this information is probably smaller than the sum of user data and transaction identifier sizes atomically written in MCL-FIXED. MCL-2TXIDS does not need additional hardware atomicity guarantees.

**Usable Data per Cache Line** The size of data that is available for user information is limited by the size of meta data in each variant. MCL-FIXED adds a transaction identifier to the meta data and removes the active bit. MCL-TXID also needs a transaction identifier, but also relies on the active bit. The difference in MCL-TXID and MCL-FIXED is only a single bit. In contrast, MCL-2TXIDS stores an active bit and two transaction links. Therefore, this variant is the most restrictive with respect to usable data in each cache line.

| Variant | Flushes per TX | Atomicity Requirement | Usable Data per Cache Line |
|---|---|---|---|
| MCL-TXID | $nr_{mod\_cl} + 1$ | $w_{active\_bit} + w_{tx\_link}$ | $\frac{w_{cl} - w_{active\_bit} - w_{tx\_link}}{2}$ |
| MCL-FIXED | $nr_{mod\_cl}$ | $w_{usable\_data} + w_{tx\_link}$ | $\frac{w_{cl} - w_{tx\_link}}{2}$ |
| MCL-2TXIDS | $nr_{mod\_cl}$ | - | $\frac{w_{cl} - w_{active\_bit} - 2 \times w_{tx\_link}}{2}$ |

**Table 4.2:** Comparison of the multi cache line transaction variants. The variable $w$ is short for *width* of the data denoted in the index. The label *cl* is short for *cacheline* and *mod_cl* refers to the number of cache lines modified in the transaction.

When choosing a multi cache line transaction mechanism, one should consider the variants in reverse order as presented in this thesis. MCL-2TXIDS uses a minimum number of flushes without additional hardware requirements. However, it enforces the most strict size constraints. As long as the data fits into the available space, MCL-2TXIDS is the best choice. In scenarios where the size of usable data in a cache line slot exceeds MCL-2TXIDS's capabilities, MCL-FIXED should be preferred to MCL-TXID because of its fewer flushes. When the amount of data exceeds even MCL-FIXED, MCL-TXID may still work.

One detail has not yet been discussed: transaction identifiers may overflow. The exact process of handling overflows is subject to implementations.

## 4.6 Summary

As long as upper, faster levels in the storage hierarchy remain volatile, storing data in NVRAM requires explicit cache line flushes. These flushes are expensive and their number should be kept at a minimum. The main objective of this work is the development of a transaction mechanism with a

minimum number of flushes. A transaction mechanism which triggers only a single flush per cache line has been introduced in this chapter. The mechanism allows for atomic updates of data that fits into a cache line slot. Its atomicity guarantee regards persistence, not concurrency. The atomic CAS for concurrent activities has been compared to cache line transactions to figure out whether existing algorithms become persistent when CAS is replaced by cache line transactions. Such a transformation has proven to require additional efforts and further investigation.

The single cache line transaction scheme can be used for data which fits into a cache line slot. Such data might be semantically connected information, like the meta data for files stored in a file system. Even more complex data structures, like the exemplary doubly linked list and hash table, can be built with a single cache line transaction.

Complex scenarios may require transactions which modify more than a single cache line. Three different solutions for this setting have been presented. The solutions differ in their cost and choosing an adequate one depends highly on the use case. Guidance for the decision process has been provided.

All in all, the single cache line transaction scheme and two of its extensions require only one flush per modified cache line. The remaining extension added only one additional flush. Its performance penalty depends on the ratio of transactions and number of cache lines modified in each transaction. Performance analysis is subject to the next sections. But first, implementation aspects of cache line transactions are shown in Chapter 5. The subsequent evaluations demonstrate the benefits of the single cache line flush.

CHAPTER 5

# Developing for Non-Existing NVRAM Hardware

The crux of Non-volatile Random-Access-Memory (NVRAM) research is that the target hardware is not yet available. Hence, many of the aspects can only be speculated on and experiments have to rely on simulations. In the context of NVRAM, different properties have to be considered. On the one hand are physical properties, like latency and power consumption. Since this work assumes NVRAM latency to be close to Dynamic Random-Access-Memory (DRAM) and does not consider energy consumption, these physical properties do not need special analysis. Logical features, specifically byte-addressability and persistence, on the other hand, form the base of this work. Their resemblance is therefore essential.

Software is going to access NVRAM with a special persistent memory interface. General requirements and implementation aspects of such an interface are discussed in the next section. Moreover, the simulation of the logical NVRAM properties on top of existing technology is addressed. Many of the challenges of NVRAM integration and simulation have already been addressed by persistent memory frameworks. The implementation of cache line transactions has been integrated in one existing persistent memory framework built by Intel. Its solutions for the NVRAM challenges are addressed in Section 5.2. Important aspects of the cache line transaction implementation are shown in Section 5.3.

## 5.1 Integrating NVRAM into the Software Stack

As described earlier, this work assumes NVRAM can be mapped directly to virtual address spaces where it can be used with conventional load and store instructions. The fulfillment of this assumption has several consequences. A brief overview of challenges on the system level is given next. Its description includes the interface applications can use to access NVRAM. Details of application level issues are discussed afterwards.

### 5.1.1 System Level

In the desired setting, a computer user buys an NVRAM module and plugs it into an existing system. The contents of this memory module become part of the physical address space of the system. During boot, the Operating System (OS) has to detect the NVRAM memory region, because it needs special treatment – different from volatile memory. The system's firmware, e.g., the BIOS/UEFI, informs the OS about available memory regions and their types. It is one of the software parts of the system that need to be updated before NVRAM can be used. An OS may then scan the provided memory map and identify regions in the physical address space belonging to NVRAM.

The OS has to keep track of DRAM and NVRAM separately. NVRAM may already contain data if it has been used before. Once the system has decided whether the data is usable, it has to built or rebuilt management data, which may also be stored in NVRAM. In the end, the NVRAM should be available in virtual address spaces, but not as a whole. Management data should be available only in privileged system parts, rendering it desirable to map only selected parts of NVRAM into an address space. Consequently, persistent memory should be split into multiple regions which can be used in virtual address spaces. Access protection is needed for these regions to prevent applications from using kernel data and also for separating data belonging to different applications or users.

A privileged subsystem has to manage persistent regions. It is contacted by applications whenever they want to create, map, unmap, or delete a region. Each region therefore needs an identifier. The management subsystem has to keep track of regions, provide a name space and control access rights. Responsibilities are similar to those of file systems. On the application level, it is therefore assumed to have a file system like API featuring:

- creation of a persistent region with a given name and file system like access rights
- mapping of a region into the address space
- unmapping of a region
- deletion of a region

Real NVRAM hardware is not yet available for this work and a persistent management subsystem does not yet exist. In order to investigate algorithms for NVRAM, the expected API has to be simulated. A close resemblance can be found with direct access file systems, like `tmpfs` on Linux. Direct access means that this file system is backed by main memory instead of files and therefore omits overhead like page caches. Loads and stores on a memory mapped file therefore directly hit main memory, as expected with NVRAM.

### 5.1.2 Application Level

From an application's perspective, a persistent region can be mapped into its address space if the identifier of the region is known and the process is authorized. One important aspect of this mapping is the target virtual address. Is a persistent region always mapped to the same virtual address or may the address differ? The former case allows to store direct, absolute pointers to persistent objects. Even an object in a persistent region may refer to another object from a different region by its virtual address. This form of direct addressing is probably the most effective one. However, it requires all processes to agree on a fixed, virtual address for each region. In the past, fixed addresses for data have been proven to be a nightmare from a security point of view. Consequently, it is unlikely that a persistent region will always be mapped to the same virtual address. If the virtual address of a persistent region may differ whenever it is mapped, pointers to persistent objects can not be absolute. Instead, the current start of the region has to be used as a base for relative addressing.

NVRAM is going to introduce challenges on the system level and also on the application level. The interface offered from the system to applications is probably similar to the existing file interface. Based on the abstraction of files, it is already possible to write application software even without NVRAM. Intel's persistent memory framework Persistent Memory Development Kit (PMDK) is built on top of the file interface and can already be used today[1]. Its solutions for challenges on the application level are explained next.

## 5.2 PMDK

Two of the most cited persistent memory frameworks introduced by academia are Mnemosyne [59] and NV-Heaps [12]. Both of them provide application level access to persistent memory and could therefore be the basis of the implementation of cache line transactions. Unfortunately, Mnemosyne relies on an Intel compiler and the compiler's license restricts its use cases. The source code of NV-Heaps is neither open source nor became available when contacting one of the authors. In addition to the academic projects, industry also came up with a persistent memory framework which is used for the implementation in this thesis: PMDK.

PMDK is an open source project, developed by a team of Intel employees. Andy Rudoff started to describe the basics of this project back in 2013 [54]. Shortly afterwards, when the project website[2] went online, the project was called Non-Volatile Memory Library (NVML). This is also the name Andy Rudoff used in a subsequent article [53]. As of December 2017, the project is known as PMDK, because the old name did not reflects the fact that the projects contains multiple libraries. Some of the different C libraries in PMDK are for block-oriented access to NVRAM, for persistent logs in

[1] It has been possible to use persistent heaps in memory mapped files for a long time, but the alternative of serializing data when writing to disk is commonly used today. Serialization brings additional overhead which memory mapped files do not have. In comparison to disk access times, serialization overhead is insignificant. This is probably the most important reason why software stuck to the well known interface instead of using the more efficient one.

[2] `http://pmem.io`

NVRAM, and access to persistent objects. The focus of the upcoming description is on the persistent object interface and its C++ bindings.

PMDK is best explained with an example. Listing 5.1 shows a persistent queue developed for PMDK. Most of the implementation is similar to a queue implementation for volatile memory. The queue consists of queue elements, each of them storing a value and a pointer to the next queue element. Moreover, the queue has a head and tail pointer, which are not explicitly shown in the listing. In order to push a new value to the end of the queue, a new list element has to be allocated. Once it is initialized, its is connected to the queue by adjusting pointers. The differences of the PMDK version are the data types `persistent_ptr<>` and `p<>`, the `transaction` wrapper, and memory allocation. These aspects are discussed next.

```
1  // list which stores unsigned integers
2  class persistent_queue {
3      // a queue element
4      struct queue_entry {
5          persistent_ptr<queue_entry> next;
6          p<uint64_t> value;
7      };
8  // ...
9
10     // append a new element to the queue.
11     void push(pool_type& pool, uint64_t value) {
12         transaction::exec_tx(pool, [&] {
13             // allocate persistent memory for the new element
14             auto new_elem = make_persistent<queue_entry>();
15
16             // initialize element
17             n->value = value;
18             n->next = nullptr;
19
20             // queue empty?
21             if (head == nullptr) {
22                 head = tail = n;
23             }
24             else {
25                 tail->next = n;
26                 tail = n;
27             }
28         });
29     }
30 };
```

**Listing 5.1:** Example usage of PMDK's C++ bindings, adapted from [53]. This persistent queue consists of `queue_entry` elements, each storing a value and a pointer to the next element. The implementation is similar to a volatile version, except for the red parts: persistent data types, a pool reference, transaction wrapper, and memory allocation.

A PMDK pool is a consecutive region of persistent memory, backed by a file. An application may map a pool to its address space. Figure 5.1 shows an application which is currently using two different pools.

An instance of the persistent queue is spread among these pools (queue meta data and first element reside in `pool0` while the second element is in `pool1`). A pool can either be located in NVRAM or on a traditional disk. With pools on disks, the library can be used even on systems without NVRAM. In that case, changes of persistent data are automatically synced to disk by the framework with a system call. For experiments where persistence across power outages is not required, the file can be located in main memory, e.g., using `tmpfs`. Instead of a system call, a cache flush is sufficient to write changes back to the file. This is important for the implementation of cache line transactions in the next section.

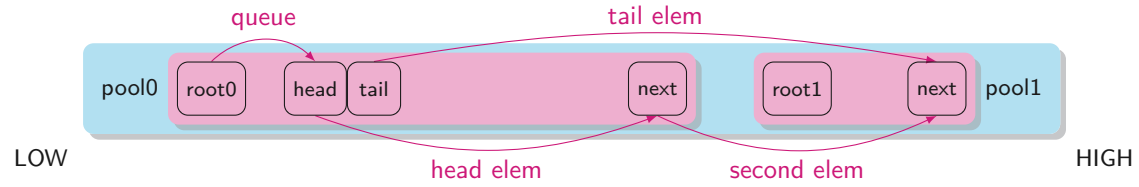



**Figure 5.1:** Address space of an application using two PMDK pools. The root object in the first pool refers to a queue. In that queue are currently two elements, head and second, to which the queue's head and tail refer. Since persistent pointers contain a pool identifier, it is possible have cross-pool references, such as the link from the first to the second element.

With PMDK, special, persistent data types, like `persistent_ptr<>` and `p<>`, have to be used. As their name indicates, `persistent_ptr<>` denotes a pointer to a persistent object. These pointers are relative to the start of a pool. Internally, they store a pool identifier and an offset within the pool, enabling cross-pool pointers, like the tail of the queue in Figure 5.1. When accessing a persistent pointer, the virtual address to which the pool is mapped to has to be found. PMDK keeps a hash table of currently mapped pool identifiers and virtual addresses in each process. While `persistent_ptr<>` is different from a normal pointer because of its internal addressing scheme, the other data type, `p<>` does not affect the layout of the underlying data. Both data types hide implementation details from programmers when persistent data is modified. This is explained in more detail later on when transactions are covered.

When a pool is mapped by an application, how does the application know where the objects in that pool are? Each pool contains a special root object for the purpose of finding existing persistent data. As seen in Figure 5.1, each pool has such a root object. The persistent queue can be reached from the root object in the first pool. By definition, one root object exists in each pool. Other persistent objects have to be allocated explicitly. Listing 5.1 shows that persistent allocations use the `make_persistent<>` call instead of `new`. The allocation returns a persistent pointer to the newly created object.

Another aspect of PMDK are transactions. Memory allocation and all changes in the `push()` method of Listing 5.1 are wrapped and executed in a transaction. Any failure during the execution of the wrapped code causes the transaction to abort and to revert all changes. With `exec_tx`, a new transaction is started. The persistent data types hide that each change of the underlying data in

a transaction is registered by the framework so that it can be rolled back in the case of an abort. Aborts can be triggered programmatically, by exceptions, or by failures like power outages. When the transaction scope is left without an abort, the changes are committed.

Internally, PMDK uses an undo log for its transactions. Each modification of data in a transaction triggers the log manager. If this is the first access to this data in the active transaction, a new log entry is created. Otherwise, the log entry can be omitted. Therefore, the log manager has to keep track of the logged regions and handle corner cases, e.g., when an already logged region overlaps with a new one.

So far, a brief introduction into PMDK has been given. Please note that the project is under construction and frequently updated. It will probably undergo several more changes once actual NVRAM hardware is available. Its current state has been used to implement cache line transactions. The important aspects of such an implementation are the subject of the upcoming section.

## 5.3 Cache Line Transactions for PMDK

Two categories of cache line transactions have been implemented. The first one is the basic scheme where only one cache line is modified by a transaction. Its explanation follows next. The other one, multi cache line support, requires additional bookkeeping, as shown afterwards. Only one of the design variants of the multi cache line transaction scheme has been implemented so far. It is the one closest to the single cache line transaction scheme. In the design phase, this multi cache line variant has been identified as the preferable version for most use cases, because it keeps the number of cache line flushes at a minimum.

### 5.3.1 Single Cache Line Transactions

In order to support cache line transactions, a specific data layout has to be enforced. The C++ template class, shown in Listing 5.2, supports the programmer's enforcement of the layout. It stores two elements of a given type and additional meta data. Great care has to be taken to assure that the start of an instance of the helper class is aligned to cache line boundary and that the whole information fits into a single cache line. Compiler intrinsics can help for aligning statically allocated data. Persistent data is allocated at runtime, rendering the issue of alignment more complicated. It is discussed later in Section 5.3.2. Static assertions check at compile time whether the information does not exceed the size of a cache line.

The cache line transaction scheme defines a protocol for transactions. Prior to any write, the data has to be copied from the active slot to the inactive one. Therefore, modifications have to be preceded by a call of `begin_write()`. Then, the inactive slot can be used for modifications, until `end_write()` is called. At the end of a write, the new data is activated and the cache line is flushed.

```
// helper class to enforce cache line layout for single cache line transactions
template <typename data_t>
struct
__attribute__((aligned(sizeof(cache_line))))
single_tx_p
{
    data_t slots[2]; // two data slots
    bool active : 1; // meta data

    // copy the currently valid data
    void begin_write() {
        slots[!active] = slots[active];
    }

    // activate the new slot
    void end_write() {
        active = !active;
        flush(this);
        fence();
     }
};
```

**Listing 5.2:** Structure of the helper class for single cache line transactions. This data structure is aligned to cache line boundary with the compiler intrinsic. It contains two slots for user data and an additional Boolean for meta data.

As mentioned in the previous section, PMDK pools can be either backed by files on disk or in main memory. Flushing cache lines is only sufficient to assure persistence for the latter case where data resides in main memory. For persistent files on disks, `msync()` has to be issued. Whenever data has to be synced with its backing store, PMDK determines at runtime whether `msync()` or `clflush` has to be used. It is also possible to configure the behavior manually and a user can disable syncing completely when starting a `PMDK` program. This can be useful for benchmarks which determine the impact of syncing primitives.

The flush used by the cache line transactions, shown in Listing 5.3, adheres to the behavior of PMDK. On main memory, a cache flush is triggered and on disks, a system call is used. Whether syncing is completely disabled, is handled by the PMDK's routines internally. After flushing the cache line, a fence is triggered to order flushes. Flushes issued before the fence can not be reordered with flushes triggered after the fence.

### 5.3.2 Multi Cache Line Transactions

Similar to the single cache line transactions, the multi cache line version also enforces a specific cache line layout. The corresponding helper class does not vary greatly from the one used for single cache line transactions. More important are differences arising from explicit recovery in this scheme.

```
// sync the cache line to the backing store
// PMDK's routines check whether syncing is enabled or disabled
void flush(void* addr) {
    int is_pmem = pmem_is_pmem(addr, sizeof(cache_line));
    if (is_pmem == 1) {
        // calls clflush
        pmem_persist(addr, sizeof(cache_line));
    }
    else {
        // calls msync
        pmem_msync(addr, sizeof(cache_line));
    }
}
```

**Listing 5.3:** Implementation of flush routine. PMDK decides whether clflush or msync are required.

During recovery, the system has to find all of the cache lines which have been used by the most recent transaction. Such a check has to be performed when the next process accesses the persistent memory region for the first time. For the PMDK implementation, recovery is triggered when a persistent pool is opened. At the time of opening one pool, it is not clear whether another pool was also involved in an interrupted transaction. The illustration of the persistent queue in Figure 5.1 spread among multiple pools shows how such a situation may come up. The prototypical implementation of the cache line transaction scheme limits transaction to cache lines residing in only one PMDK pool.

A dedicated memory region in each pool is used to store cache lines using the multi cache line transaction scheme. Its manager keeps track of used and free lines and is able to allocate more lines on demand. All of the required book keeping information is stored in persistent memory and modified in transactions. The layout of a pool using multi cache line transactions is depicted in Figure 5.2. Each root object refers to a `multicltx_manager` which is responsible for a set of persistent cache lines. A bitmap indicates which of the cache lines are currently in use. In case all of the cache lines are in use, another region with its own manager has to be allocated and linked to the existing one's `next` pointer.

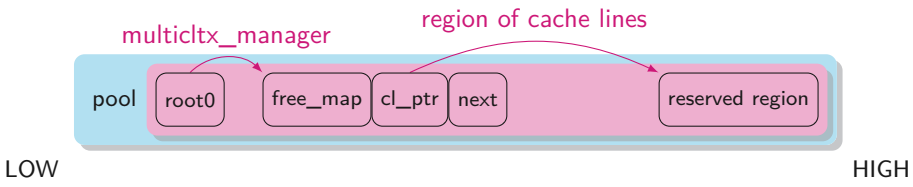



**Figure 5.2:** Management of cache lines used by the multi cache line transaction scheme in a persistent pool. The root object refers to a manager which keeps track of a region of cache lines in the `cl_ptr` field. In addition, a bitmap indicates which lines are currently used. Another manager can be linked to the existing one with the help of the `next` pointer.

During recovery, all cache lines explicitly managed by the scheme can be found in one of the allocated regions. They are scanned and those which need to be recovered are rolled back by activating the previously valid slot.

### Alignment

One issue arises in the context of memory allocation using PMDK. The previously shown `make_persistent<T>()` can be used to dynamically allocate persistent memory. All allocations of persistent memory, starting with the root object, are dynamic. Hence, alignment hints for static allocations, given to the compiler, are ignored. Even worse, PMDK does not feature a dynamic allocation routine which allows for alignment hints. Unless a routine similar to `memalign()` becomes available, allocation has to be handled manually.

As shown earlier, the multi cache line transaction scheme needs to store its cache lines in a fixed region so that they can be found during recovery. The memory for this region is allocated with the array version of `make_persistent<T[]>()`. This call may return a memory region which is not aligned to cache line boundary. Internally, the region's manager handles alignment so that its users operate only on properly aligned data.

The layout helper structure for single cache line transactions shown in Listing 5.2 used compiler intrinsics for proper alignment. Such a hint is only respected when allocating data statically and ignored for persistent data. The manager developed for multi cache line transactions can be used for allocations of properly aligned cache lines. A derived manager, and another reserved region of cache lines, is used for cache lines using single cache line transactions. In contrast to the multi cache line manager, it does not need to perform recovery checks when the pool is opened.

### Persistent Pointers

In each of the cache line transaction schemes, data has to fit into a cache line slot. A system with 64 byte cache lines allows fewer than 32 bytes per slot, because there is also meta data in the cache lines. Persistent pointers in PMDK contain a 64 bit pool identifier and a 64 bit offset – 16 bytes in total. While one persistent pointer can be stored in a cache line slot, a second one already exceeds the available space. For now, storing two persistent pointers in a cache line for cache line transactions is not supported.

In the future, one may consider to change the data types used by persistent pointers. When 64 bit offsets are completely utilized, the pool can only be mapped to address 0 on a 64 bit system. This is unlikely to be useful in the near future where volatile memories remain. The full utilization of 64 bit for pool identifiers also seems to be unlikely in the near future. However, the effects of such restrictions should be considered more carefully first.

### 5.3.3 Application Example

To conclude the implementation aspects of cache line transactions for PMDK, consider the example in Listing 5.4. It demonstrates the differences for using volatile data, persistent data based on PMDK's data types and cache line transactions. The example uses a persistent vector holding two integers. Both of these integers are incremented and these increments should be performed transactionally. Since the persistent data has to be connected to a pool's root object, it is stored directly in the root. Allocation of that data is omitted for simplicity.

Line 18 creates a volatile vector so that its normal usage can be shown afterwards. The two subsequent lines increment the vector's fields. These two steps are also the heart of any upcoming modification of persistent data. These changes are encapsulated by PMDK's transaction wrappers. At first, the typical implementation using PMDK is shown. When write access to the persistent object is requested, the data is written to the undo log and registered in PMDK's management data. At commit time, when the scope of the transaction is left, the modified data is flushed.

The cache line transaction schemes are similar. When the user requests write access, the transaction scheme copies data from the active to the inactive slot internally. It returns a temporary helper object used by the subsequent operations. Its destructor finishes the transaction on that cache line and activates the new version. The scope is used so that temporary objects are created and destroyed automatically.

With the single cache line transaction scheme, a transaction is not allowed to modify more than one object. In contrast, the multi cache line scheme might be used to modify more than one persistent object. The transaction wrapper couples changes of all objects in that transaction, so that they use the same transaction identifier.

This chapter presented the current implementation of cache line transactions on top of the PMDK persistent memory framework. By using the existing file system interface for access to persistent memory, the implementation can be used on commodity hardware. This fits perfectly to the target setting where NVRAM is plugged into commercial off-the-shelf hardware with no further hardware changes. Experiments presented in the next chapter do therefore not need a special NVRAM simulator. The upcoming evaluation demonstrates the benefits of cache line transactions in comparison to logging schemes.

```cpp
// data types
using base_t = std::pair<int, int>; // volatile type
using pmdk_t = typename pmdk::p<base_t>; // pmdk version
using single_p = typename cltx::single_p<base_t>; // single cl tx
using multi_p = typename cltx::multi_p<base_t>; // multi cl tx

// persistent object
struct root_t
{
    // ...
    // instances of the persistent data types shown above
    pmdk_t pmdk_obj;
    pmdk::persistent_ptr<single_p> single_cltx_obj;
    pmdk::persistent_ptr<multi_p> multi_cltx_obj;
};

// volatile data
base_t volatile_obj;
volatile_obj.first = volatile_obj.first + 1;
volatile_obj.second = volatile_obj.second + 1;

// pmdk
pmdk::transaction::exec_tx(pool, [&] {
    auto& obj = root->pmdk_obj.get_rw();
    obj.first = obj.first + 1;
    obj.second = obj.second + 1;
});

// single cltx
{
    auto obj = root->single_cltx_obj->get_rw();
    (*obj).first = (*obj).first + 1;
    (*obj).second = (*obj).second + 1;
}

// multi cltx
cltx::tx_manager::exec_tx([&] {
    auto obj = root->multi_cltx_obj.get()->get_rw();
    (*obj).first = (*obj).first + 1;
    (*obj).second = (*obj).second + 1;
});
```

**Listing 5.4:** Usage example of the persistent data types. The underlying data types of this is example is a pair of two integers (`base_t`). A root object of the persistent pool, holds a PMDK version of a pair, a version for the single cache line, and one for the multi cache line transaction scheme. When persistent data is modified, the changes are encapsulated in a transaction. Within these transactions, temporary helper objects hide most of the details, like marking the beginning and end of write operations.

CHAPTER 6

# Evaluation

The cache line transactions developed in this work aim to reduce the number of cache line flushes for transactions on persistent data. The first part of the evaluation confirms that this goal has been achieved.

The goal of reducing the number of cache line flushes arose from the fact that flushes are expensive. Minimizing the number of flushes should therefore bring a performance improvement. However, there is a price to pay, because the transaction schemes change the layout of data. The effects of that change are most prominent for multi cache line transactions on arrays. Normally, the data would be consecutive in memory. The transaction scheme disrupts that layout by storing a second version and meta data in between items. As a result, the access pattern differs from the one used on volatile data. With logging, the access pattern also changes, because in addition to the actual data, a second memory region holding the log is frequently accessed. Before the results are presented, Section 6.1.1 presents a methodology for counting the number of cache line flushes. Section 6.1.2 describes how the runtime effects of different transactions schemes have been determined.

The methodology explanation is followed by results for micro benchmarks in Section 6.2. These settings are simplistic and allow precise predictions of the resulting behavior. Cache line transactions outperform logging when used for such small data structures. However, effects of the modified data layout can hardly be demonstrated. Hence, Section 6.3 puts cache line transactions in a more complex scenario in order to investigate the effects of a changed working set.

## 6.1 Methodology

### 6.1.1 Counting the Number of Cache Line Flushes

Cache line flushes can be triggered in two different ways, either by manually using a flush instruction or automatically by the hardware. At first, this section considers the number of flush instructions triggered by an application. Automatic flushes are discussed afterwards.

Static analysis of a binary is insufficient to determine the number of flush instructions triggered by an application. As loops and dead code are not taken into account, the results would not represent meaningful numbers. Instead, the dynamic behavior of the program has to be analyzed. For a proof of concept, this work is only interested in monitoring one dedicated process performing an

experiment. Simultaneously running processes should not influence the result. The same applies for kernel activities, which may also use flush instructions. A flush triggered by the kernel is probably used for communication with other hardware and not relevant for the transactions on persistent memory.

It is impossible to determine the number of flushes by running a program directly on current hardware. Although several events can be monitored, the performance counters for caching are limited to determining hit and miss rates. Instead, simulation is used here. A variety of x86 simulators exists. Most of them are full system simulators, meaning that they resemble a whole system, include a kernel and multiple user space applications. These simulators could be modified to trace flush instructions, but it would be hard to determine which process or subsystem triggered them. In addition to the whole system simulation mode, gem5 [1] has a so called system call emulation mode which can be used to analyze the behavior of an individual process. Unfortunately, this mode is not yet compatible to C++17 which Persistent Memory Development Kit (PMDK) and the current cache line transaction implementations depend on.

For this work, Intel's Pin tool[2] for dynamic instrumentation has been used. Binaries can be instrumented at runtime with Pin, removing the need to compile a dedicated version of the program for counting flushes. The analysis can use the same binaries as other evaluation scenarios. A dedicated Pin tool which counts cache line flushes has been developed. In addition to counting all flushes triggered by a process, the programmer can also mark regions of code which should be taken into account. Thereby, the number of flushes used in a setup phase can be distinguished from flushes triggered by the remaining algorithm.

In addition to counting cache line flushes, Pin can also be used to collect other information. It can create a trace of all memory references issued by a process. Such a trace can be used with a cache simulator to determine the number of flushes triggered automatically by the eviction policy. The effects of cache sizes, cache line sizes, and eviction policies can be assessed with such tools in the future.

### 6.1.2 Runtime

The number of flushes issued by a process has an impact on the overall execution time. As of today, a cache flush instruction triggers a write back of the corresponding line to main memory and also invalidates the line in the cache. A subsequent access to that line has to fetch data from main memory again. If the second access happens shortly after the flush, the cache miss induces stalling. Otherwise, if the data is not needed for a while, the data might get evicted from the cache anyways. In that case, the effects of invalidations are less severe.

[1] `http://gem5.org`
[2] `https://software.intel.com/en-us/articles/pintool`

Intel recently introduced a new instruction for cache line flushes which does not invalidate the line and causes fewer cache misses: `CLWB`. This instruction can now be found in their handbooks [26], but is not yet available for this thesis. In order to determine the influence of invalidations, the runtime is measured in two different settings. At first, the existing cache line flushes are triggered. Second, the cache line flushes are disabled. Without flushes, the results provide a lower bound for the runtime on future processor generations with the optimized cache flush routine.

## 6.2 Micro-Benchmarks

Micro benchmarks are performed for two scenarios. In the first one, the single cache line transaction scheme is applicable, because the modified data is small enough. The second scenario uses persistent data spread among multiple cache lines, so that the single cache line transaction scheme can not be used.

All experiments have been carried out on an Intel Core i5-5287U which uses 64 byte cache lines and runs at 2.9 GHz.

### 6.2.1 Incrementing a Pair of Integers

In this setup, a pair of integers is incremented transactionally. This example has already been used to illustrate the usage of transactions in Section 5.3.3. In short, the example works as follows: start a transaction, increment the two integers, and end the transaction:

```
1 // begin transaction
2     obj = persistent_pair.get_rw(); // request write access
3     obj.first = obj.first + 1;
4     obj.second = obj.second + 1;
5 // end transaction
```

The experiment covers the following settings:

**base** A volatile implementation which should not trigger any cache flush at all.

**cltx-single** This version stores the pair in a single cache line and uses the single cache line transaction scheme. It should trigger only one flush.

**cltx-multi** The multi cache line version does not differ from the single cache line transaction scheme as the pair of integers fits into a single cache line. It should therefore also trigger only one flush.

**pmdk** A persistent pair of integers based on PMDK's undo log. It is expected to log the pair and flush the log entry. Then, after the data is modified, it is expected to also flush the new data. Hence, at least two flushes should be seen.

**undo** Instead of relying on PMDK's logging mechanism, a dedicated undo log is used. Inspired by logging in PMFS [16], it splits the log into cache lines. Each cache line holds one log entry of up to 40 bytes. These log entries are flushed before the original data is modified. At commit time, another cache line containing a commit record is written to the log and also flushed. The pair of integers is small enough to fit into one log entry. Consequently, the undo log should trigger one flush for the log entry, one flush after modifying the pair, and a third flush for the commit record.

### Cache Line Flushes

As shown in Figure 6.1, most expectations are met[3] Only the PMDK implementation exceeds the expected number of flushes, by far. Detailed investigation revealed that 20 of the 24 flushes are triggered when write access to an object is requested in a transaction. This is the point in time when the object is added to the log. Instead of using a dedicated log region, PMDK allocates a region of persistent memory at this time. Its underlying best fit heap lies in persistent memory and the created headers and footers are flushed. Moreover, a redo log is used to perform the changes to the heap atomically and this redo log also triggers flushes. Unfortunately, these actions are not covered in PMDK's project description and the reasoning for this infrastructure is unclear.

The manually crafted undo log shows that it is possible to reduce the number of flushes drastically, while still relying on logging. As expected, this log triggers more cache line flushes than the cache line transaction schemes. The undo log implementation uncovered further benefits of the cache line transactions. When logging data, the source of the original has to be written to the log as meta data. A persistent pointer has to be used for the source address, because the pool might be mapped to a different start address during recovery. The cache line flush, however, is triggered with volatile addresses. Mixing volatile and persistent addresses within transactions results in conversion overhead. Cache line transaction schemes do not need to track a persistent location of the modified data and spare the overhead of transforming addresses. The severity of the resulting overhead can be seen in the upcoming runtime analysis.

### Runtime

Of all the different configurations, the volatile *base* version has to deliver the lowest runtime. It is the pure version with no overhead for persistence. The single cache line transaction mechanism uses the cache line layout with two slots to store the pair. It has to copy the data from one slot to the other when a transaction starts. At commit time, it toggles the active bit and flushes the cache line. The flush is followed by the next iteration of the benchmark which also copies slot contents in the same cache line. Due to the previous flush, the next iteration works on a cold cache line, because a

[3] Please consider also Appendix B.

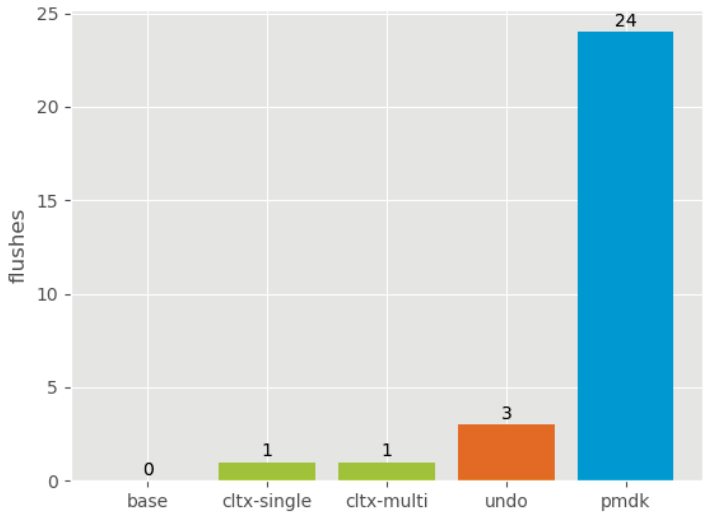



**Figure 6.1:** Number of cache line flushes triggered when two integers stored in a pair are incremented. A volatile pair is depicted as *base*. It does not trigger any flush. Both the single cache line (*cltx-single*) and the multi cache line transaction scheme (*cltx-multi*) store the pair in a single cache line and need only one flush. PMDK's transaction (*pmdk*) results in 24 flushes. The *undo* log implementation triggers 3 flushes.

flush currently invalidates the cache line. Therefore, the data has to be fetched from main memory. This fetch should result in additional 100 ns for main memory access in comparison to the volatile version. Out of all the transaction schemes, *cltx-single* is expected to perform best. The multi cache line transaction has to keep track of the transaction counter and needs to write it to the cache line. It should be the second best transaction scheme, because it also issues only one cache line flush. The remaining two transaction schemes have to create log entries, convert volatile and persistent pointers, and use more than one cache line flush. As the previous analysis of cache line flushes revealed, PMDK performs many more steps than required in the undo log. This should reflect in the runtime difference of the two logging schemes.

Runtime has been measured on an Intel Core i5-5287U running at 2.9 GHz using Google's micro-benchmark library[4]. Results depict the median of at least $680,000$ runs with a standard deviation of at most 0.2 %.

Figure 6.2 shows the resulting execution times for modifying two integers stored in a pair. The *base* version takes 38 ns. For the other transaction schemes, consider the higher bars first. As expected, the memory access latency is introduced in the single cache line transaction scheme, resulting in 157 ns. Multi cache line transactions induce additional runtime, but take only half the time required for undo

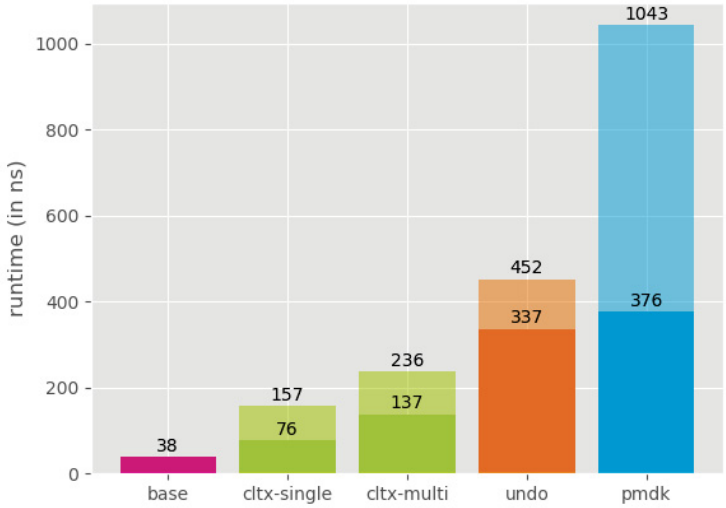



**Figure 6.2:** Runtimes of the pair micro benchmark. The larger bars represent the runtime when cache line flushes are enabled. For solid bars, cache line flushes have been disabled. Since the volatile *base* version does not trigger any flush, the two settings do not differ in their runtime. PMDK's default behavior results in the longest execution time. Out of the three transaction schemes for persistent data, *cltx-single* performs best.

logging (452 ns). PMDK execution time sums up to 1043 ns.

The lower bars in Figure 6.2 represent the runtime when cache line flushes are disabled. Without flushes, invalidations and the resulting cache misses are reduced. Even with optimized cache flushes, the runtime will not be lower than these results. As expected, the different transaction schemes perform in the same order as before: single cache line transactions being the best, PMDK being the worst. There is an enormous runtime improvement when flushes are disabled in PMDK. It indicates that the cache flushes do not only cause cache misses on subsequent access, but might also render the memory bandwidth a bottleneck.

### 6.2.2 Modifying a Linked List

The second set of benchmarks considers an algorithm using more than one cache line. It inserts 10 elements into a linked list. The list itself stores a pointer to the first element as well as the list's current size. Each element stores data and a pointer to the next element in the list. All elements are pre-allocated and inserted to an initially empty list.

Two types of experiments have been performed: one inserts at the head, the other at the tail of the list. Both experiments differ in their access pattern. Inserting at the head is write intensive, resulting

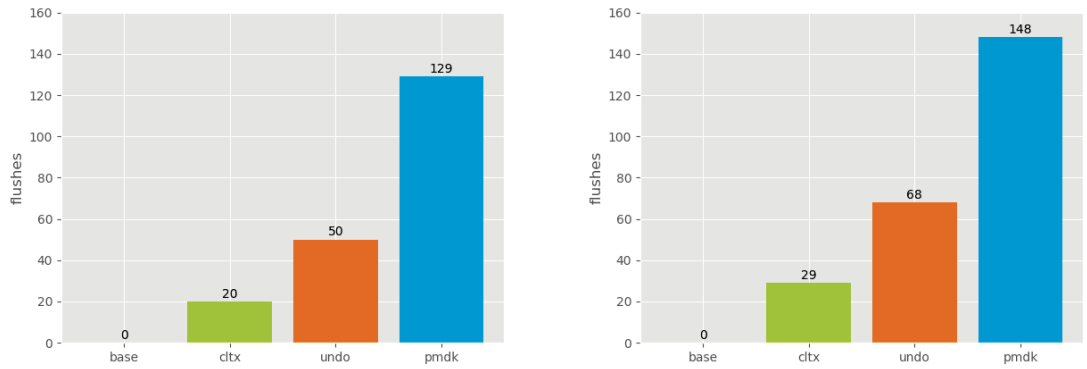
[4] `https://github.com/google/benchmark`

in updates of the list's head and size and the new element's next pointer. In the other scenario, the tail of the list has to be found by traversing and, hence, reading the list.
Similar to before, the experiments have been carried out with different settings. *Base* depicts a volatile implementation of the list. Because the list is spread among multiple cache lines, the single cache line transaction scheme can not be applied.

### Cache Line Flushes

The base version operating on volatile data should not trigger any flush. With the cache line transaction scheme, the list's pointer to the header element and the size can be stored in one cache line. This is updated, and flushed, for each insertion. Another flush is triggered for updating the next pointer of an existing list element - either the new one (insert at head) or its predecessor (insert at tail). All in all, the 10 inserts should trigger 20 flushes.

The logging schemes have to copy the list header and one list element to the log. Each of these log entries should trigger one flush. Another flush is also used when these two items are completely modified. Then, the commit record is flushed. An insert should therefore trigger 5 flushes - at least 50 for all the 10 inserts. When inserting at the tail, one more write operation is performed to set the new tail element's next pointer to `null`. Except for the first element, there should be one additional flush for each element in the cache line transaction scheme, resulting in a total of 29 flushes. Similarly, the undo log has to log and flush one more entry 9 times, resulting in 68 flushes. The measured number of flushes is shown in Figure 6.3. Again, PMDK requires far more flushes than expected, as in the previous experiment.




**(a)** Insert at the head of the list.

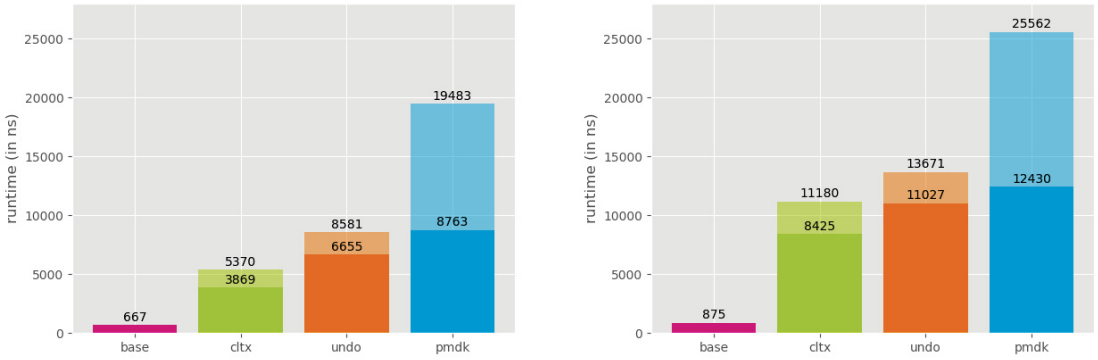
**(b)** Insert at the tail of the list.

**Figure 6.3:** Number of cache line flushes triggered when 10 elements are inserted in a linked list. A volatile list is depicted as *base*. It does not trigger any flush. Multi cache line transactions (*cltx*) triggers 20 flushes when inserting at the head, 29 when inserting at the tail. PMDK's transaction (*pmdk*) results in 129 and 148 flushes. The *undo* log implementation triggers 50, respectively 68 flushes.

### Runtime

For the runtime measurement, it is expected that the volatile version is the fastest. It should be followed by the cache line transaction scheme, undo logging, and PMDK, in that order. Each insertion has to adjust the list's header and flushes it. The next insert will therefore result in a cache miss. When inserting at the head, only this miss should be relevant. Although the previously inserted element has also been flushed, it is not accessed by the following insert. All in all, the 10 cache misses should be observable by a runtime increase of 1000 ns when compared to the volatile version, because each cache miss causes a memory access time of about 100 ns. When inserting at the tail, the recently flushed element is also accessed, because the new element has to be linked to it. Two misses instead of one should result in twice the runtime increase.

Results of this experiment are shown in Figure 6.4. The order of the resulting runtimes for the different transaction schemes fits and cache line transactions outperform logging. The actual runtimes are higher than expected. For cache line transactions, the expectation was to add 1000 ns to the base runtime. Instead, there is an increase of 4500 ns. Frequent flushes followed by access to the same data seem to trigger a bottleneck in the memory hierarchy.




**(a)** Insert at the head of the list.

**(b)** Insert at the tail of the list.

**Figure 6.4:** Runtime for inserting 10 elements in a linked list. To insert an element at the tail, the existing list has to be traversed. This results in additional reads of previously flushed data and causes cache misses. Consequently, the runtimes are higher than when inserting at the head.

The experiments shown so far considered limited workloads with small data sets. In the next section, the idea of cache line transactions is integrated into a real application.

## 6.3 A Persistent Game of Life

Due to its novelty, the number of real world applications for Non-volatile Random-Access-Memory (NVRAM) is very limited so far. Only two fields of applications are commonly targeted by research: main memory data bases and file systems.

In-memory databases, also called main memory databases, have been developed under the assumption that the whole database can be stored in main memory. With all data in memory, there is no need for disk access during queries. Only durability requires disk access, when main memory is still volatile. NVRAM removes the need for disk interactions from a main memory database. In addition to omitting writes at runtime, recovery might be instantaneous because the state does no longer have to be loaded from disk. Both main memory databases and key value stores are currently being adapted to NVRAM [49, 1]. However, main memory databases are not ideal for analyzing the effects of cache line transactions. A typical database is highly optimized and designed for concurrent workloads. Keep in mind that cache line transactions have not yet been developed for concurrency. In the worst case, they may harm the scalability of a multi-threaded application. Moreover, changes to even a single data structure in a main memory database may affect the runtime behavior drastically. Explanations of the effects require a detailed understanding of all the database internals. For the sake of determining the effects of data layout changes imposed by the cache line transaction schemes, integrations into databases are not ideal. Nonetheless, such an integration should be part of future investigations.

Aside from databases, persistent file systems may seem like a candidate for experiments with cache line transactions. The meta data contained in a file system might be small enough to render the transaction scheme applicable. However, the effect of a change in the file system is very small. Think about the persistent pools in PMDK. Only creating, opening, or closing a pool involves the file system. The majority of time is spent while the pool is mapped and does not interact with the file system. Instead, one should invest in optimizing work which is performed most of the time, on application level data structures.

So far, neither main memory databases nor file systems have been reported to be suitable for analyzing the effects of cache line transactions' layout changes. A suitable setting would access data using cache line transactions not only frequently but also in an amount that influences caching. Having a deterministic access pattern would also be a plus.

One category of applications that can benefit from NVRAM are numerical simulations, like a long running simulation of heat distribution. It is typically performed by multiple compute nodes, each operating on its own part of the whole system, with only little communication with other nodes. In order to avoid data loss, checkpoints are created periodically. Today, such a checkpoint has to be written to disk. Storing data directly in persistent main memory removes the need for creating checkpoints on disk. Consistency of this data, however, has to be guaranteed, as failures are expected to happen.

Although not directly numerical, Conway's Game of Life shows a behavior similar to that of numerical applications. It uses a two dimensional board of cells and performs simulation of evolution in discrete steps on that board. In each of these steps, the application computes whether the cells change their state, depending on the state of neighboring fields. Overpopulation, for example, may cause a cell to die. All in all, the simulation has to access each of the board's cells, compute its new state and apply the changes. The board structure is simple and the access pattern predictable. When the board is small enough, the application can benefit from processor caches. A change to the board's structure will therefore affect caching. Parallel simulation is possible by dividing the board into multiple regions which can be processed individually. The major difference to sequential processing is that the state of cells at the border of regions has to be exchanged.

When implementing a Game of Life simulation, the new state of cells has to be buffered. It can not be applied directly to the board, because the computation needs the field's old state when processing neighbors. One solution is to store two boards, one holding the current and one holding the new state, as shown in Figure 6.5. All changes are written to the currently inactive board – the right board in the figure. A step is finished when all cells have been processed. In order to apply all changes, it is sufficient to activate the currently inactive board, e.g., by swapping a pointer.

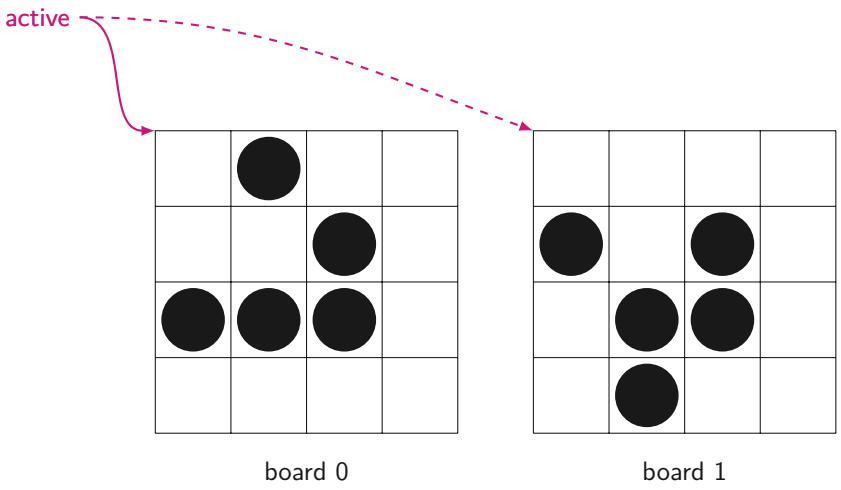



**Figure 6.5:** Game of Life implementation using two boards. The left board represents the state in step $x$. All changes performed in step $x$ are written to the right board. Finally, the right board is marked as active and used for the calculation in step $x + 1$.

This Game of Life protocol is similar to cache line transactions, but at a much larger scale. The built-in transactional processing eases a port to persistent memory. One could store the two boards directly in NVRAM. After a new state has been calculated, this board has to be flushed so that the changes arrive in NVRAM. Afterwards, the new board has to be activated, e.g., by swapping a persistent pointer. This pointer is the only data structure requiring additional transactions. However, an alternative approach exists.

The previously discussed implementation of Conway's Game of Life used two boards: one for the current state and one for the new state. It therefore doubles the memory consumption. Think of the extreme case of a huge board containing only dead cells. Although no changes are ever performed, the algorithm reserves space for two of the huge boards. If the number of changes in a step is small, it might be more adequate to spare memory by buffering the changes in another data structure instead of using two boards. An alternative implementation for persistent memory can perform a step by calculating and buffering changes in volatile memory. Figure 6.6 shows the same board as used in Figure 6.5 in step $x$. Instead of a copy, there are references to cells which die and cells which spawn in the current step. These changes are applied in the last phase of each simulation step. Listing 6.1 summarizes the basic procedure of a simulation step.

While the first approach stores two boards with a size of 16 fields, the second one needs only one board plus additional 4 references for the changing cells. If the same glider is used on a board with 1000x1000 cells, the first approach reserves a whole additional board while the second one still uses an overhead of 4 references.

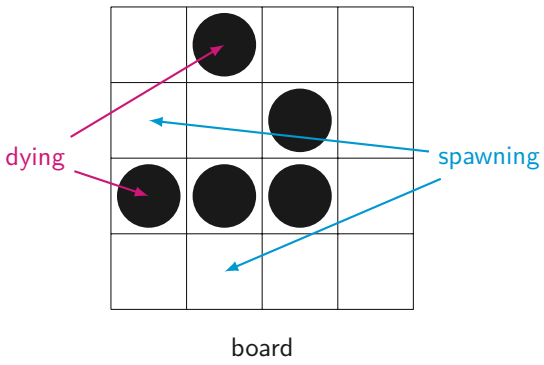



**Figure 6.6:** Game of Life implementation using a single board and buffering only the coordinates of fields which change their state. Once all changes in a step are computed, they are applied before the next step starts.

Another important difference is that the second approach may buffer changes in volatile memory. It needs a transaction to apply the changes to the persistent board after each step to preserve the integrity of the board. While the first approach needs transactions only to swap the pointer to the active board, this second version has a need for complex transactions. It is therefore used in the upcoming evaluation.

The implementation of Conway's Game of Life used for the evaluation stores its board in persistent memory. Each step buffers the computed changes in volatile memory and applies them transactionally to the persistent board. A simulation has different parameters. One parameter is the size of the board. Small boards fit into processor caches so that main memory is seldom accessed. Larger boards, exceeding cache sizes, impose the penalty of main memory access. Another parameter is the number of changes performed in a step. It depends on the population of the initial board. Imagine a board

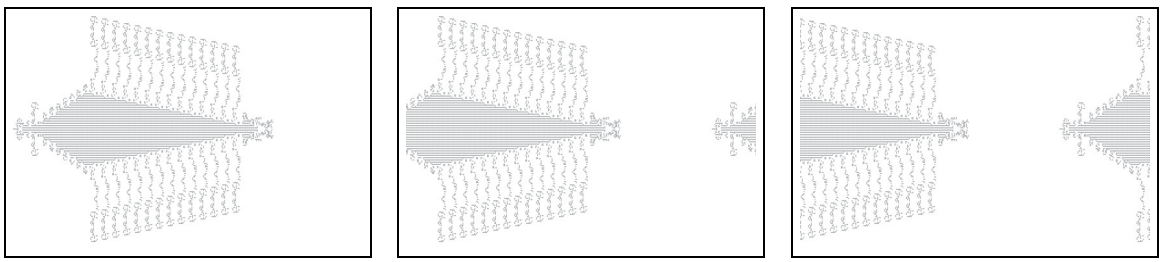
```
// calculate changing cells by reading the whole board
// and buffer the changes
for (auto cell: board.get_cells())
{
    if (is_spawned(cell)) spawning.push_back(cell);
    else if (is_dying(cell)) dying.push_back(cell);
}

// apply changes transactionally
begin_tx();
    for (auto cell: spawning) board.spawn(cell);
    for (auto cell: dying) board.die(cell);
end_tx();
```

**Listing 6.1:** Basic simulation loop. At first, all changes are determined. Then, in a transaction, these changes are applied to the board.

where all cells die after five steps. Once all cells are dead, there are no further changes and the simulation turns into a read-only pattern. The initial board can be used to influence the read/write ratio of the simulation.

The experiments use three implementations. As before, a volatile implementation is used to determine the *base* costs. The second version is a `PMDK` implementation which stores the board in persistent memory and uses `PMDK`'s undo logging for transactions. Cache line transactions *cltx* are the third implementation variant. They change the board's layout and store each cell on an individual cache line, according to the multi cache line transaction scheme.

Two boards have been used. The first one resembles a space ship, shown in Figure 6.7. The whole ship moves to the left, but its structure remains throughout the whole simulation. In a second board, a smaller glider travels from one corner of the board to another one, as depicted in Figure 6.8. In contrast to the space ship, the glider triggers only a very small number of changes in each step. It results in a read intensive workload. When writes are rare, the cache line transaction scheme can not benefit from its faster transaction processing. Instead, its disadvantage of spreading the data layout is more prominent. Without changes, PMDK may even outperform the cache line transactions.

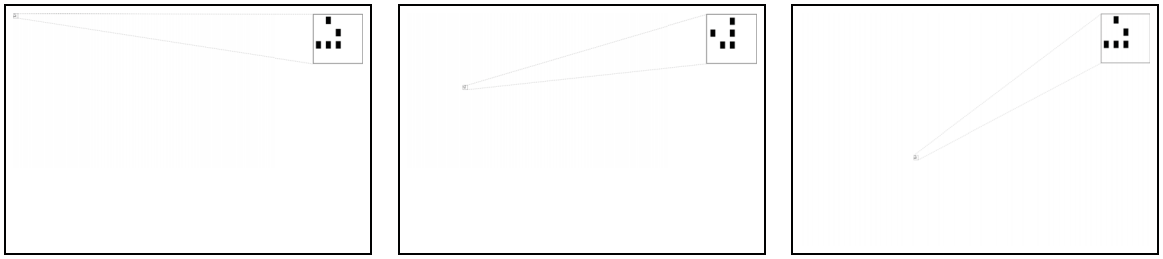


**(a)** Initial board.

**(b)** Board after 250 simulation steps.

**(c)** Board after 500 simulation steps.

**Figure 6.7:** Game of Life simulation of a space ship. It is a write intensive workload because many cells change their state in each of the simulation steps.



**(a)** Initial board.

**(b)** Board after 250 simulation steps.

**(c)** Board after 500 simulation steps.

**Figure 6.8:** Game of Life simulation of glider. The board's size is equal to the space ship settings. The glider is so small that it can hardly be seen. Therefore the image contains a zoomed view into the glider's state. A glider travels from the top left to the bottom right corner. Only a couple of cells are changed in each step and the workload is read intensive.

At first, consider the results of experiments using the read mostly glider board (Figure 6.9). It shows the runtime measured when simulation 1, 2, 4, ..., 1024 steps of the Game of Life. All in all, the overhead per step does not change when more steps are being performed. A higher number of steps makes the differences of the schemes more clear. Simulating 1024 steps on a volatile board takes 61 ms. Cache line transactions increase the runtime to 218 ms – 3.57 times the base version. With PMDK's default transactions, the simulations takes 255 ms. This equals 4.18 times the base version.

The few transactions on this board flush only a few cache lines. It makes therefore little difference whether cache line flushes are enabled or disabled – the impact of invalidations is small.

It is surprising that cache line transactions outperform PMDK in this scenario. Each cell's state is represented by a byte in the PMDK version. The board consists of approximately $80,000$ cells, resulting in a 77 KB memory footprint. Cache line transactions spread the board and store each cell on an individual cache line. As a result, the board's size increases to nearly 5 MB. Such a difference in the working set should result in a drastically better performance for PMDK. However, PMDK was developed for being user friendly. Internally, there are many levels of indirections and abstractions. It is worthwhile trading increased memory consumption for faster execution time in this scenario.

When cache line transactions outperform PMDK in the read mostly workload, their faster transaction processing is likely to improve the write intensive workload, too. As shown in Figure 6.10, it takes 67 ms to simulate on a volatile board. Cache line transactions increase the runtime to 232 ms and PMDK requires 292 ms. In comparison, cache line transactions result in 3.46 times the runtime of the volatile version and PMDK in 4.36 times, respectively. Interestingly, there is only a small difference when compared to the glider board. To verify that the workloads have been characterized correctly, the number of cache lines flushes in each setting is analyzed next.

The glider board has been characterized as a read mostly workload. Contrary to the space ship, there are only a couple of changes in each step. Consequently, there should be significantly fewer flushes.

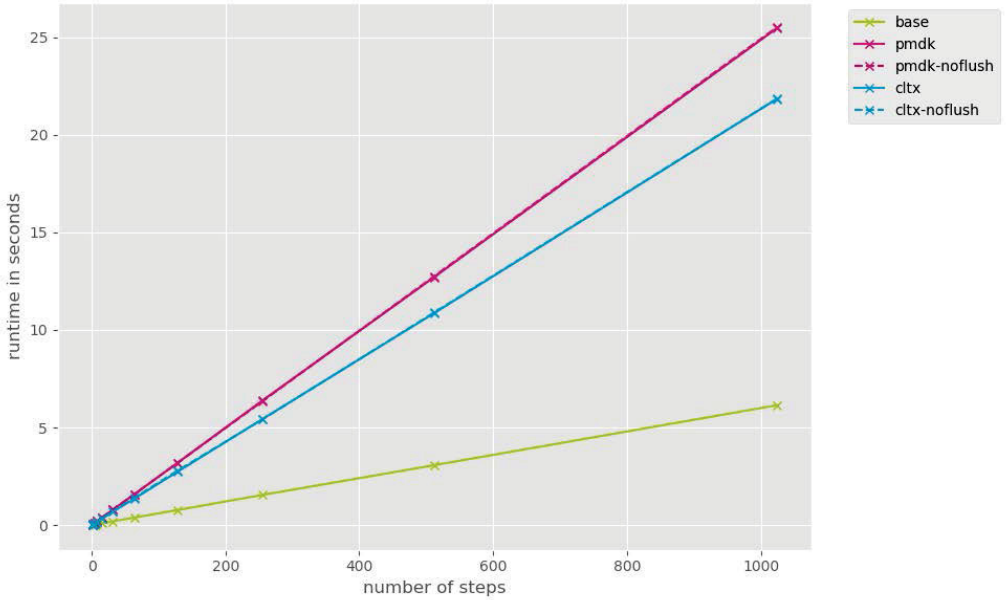



**Figure 6.9:** Runtime for simulation of the glider board. The volatile *base* version is the fastest. It is followed by the multi cache line transaction scheme *cltx* even for this read mostly workload. A *pmdk* implementation is slower, although only few transactions are being used. Because the workload is read mostly, it makes little difference whether cache flushes are enabled or disabled.

An inspection of the number of flushes is performed to validate that the workload characterization is correct.

In both settings, the basic structures – space ship and glider – continue to exist throughout the whole simulation. On average, the number of flushes should therefore be constant. For both boards, a simulation of 10 steps has been analyzed with the pin tool for counting cache line flushes.

Exactly four cells change their state in a simulation step for the glider. In addition to the field of cells, there is a persistent step counter which is incremented in each step. The sum of five changes fits the number of flushes reported by the cache line transaction scheme: 50 flushes for 10 steps — 5 per step. PMDK uses on average 19 flushes for a single simulation step. The density of living cells is much higher on the space ship. Cache line transactions flush on average $4,000$ cells in one simulation step and PMDK uses $10,000$ cache line flushes. These results confirm that logging induces at least twice as many flushes as cache line transactions because the logged data is stored in a separate memory location. However, it does not explain the small difference in runtime in both workloads.

A simulation step of the space ship board with PMDK triggers $10,000$ cache lines flushes, while a board containing a glider requires only 19 flushes per step. Despite this enormous difference in the number of flushes, the runtime for simulating 1024 steps is as close as 292 ms and 255 ms. This outcome seems to be contrary to the initial assumption of the impact of the cache line flushes on transaction processing. A second curiosity is that the runtimes remain similar, no matter whether cache line

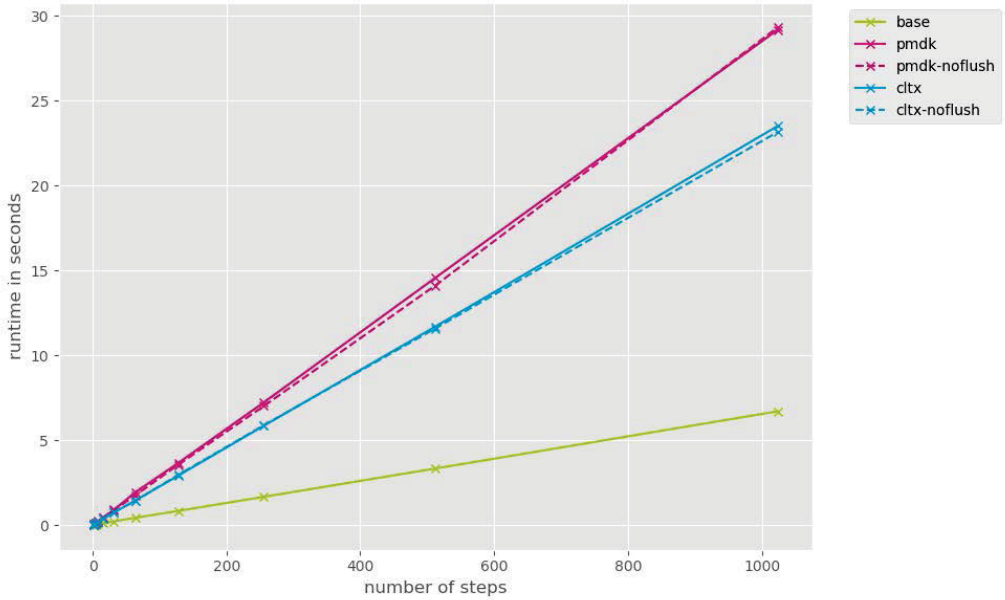



**Figure 6.10:** Runtime for simulation of the space ship board. The volatile *base* version is the fastest. It is followed by the multi cache line transaction scheme *cltx* which outperforms the *pmdk* implementation.

flushes are enabled or disabled. The explanation can be found in the simulation's implementation.

In each simulation step, all of the board's cells are read in the first phase. Reading starts with the top left corner and continues in the same row before it switches to the next one. Then, the algorithm iterates over all spawning cells and flushes the corresponding cache lines. Afterwards, the list of dying cells is traversed and these changes are written and flushed, too. The next step also starts by reading the board. It inspects the board's top left cells at first. If the board is sufficiently large, these cells are not cached anymore and the consequence is a cache miss. The invalidation following a cache line flush is therefore insignificant. Consequently, this Game of Life implementation does not seem to benefit from caching at all. This is the reason why both workloads do not vary as much as initially expected and why the invalidation triggered by flushes has nearly no effect.

All in all, this experiment has shown that the overhead of PMDK is so immense that cache line transactions outperform PMDK every time. Even though the Game of Life is a very simple example, it can already point out how complex performance analysis of algorithms on NVRAM might become. An analysis of algorithms for concurrent applications, with atomic Compare and Swap (CAS) instructions, concurrent memory access, and cache coherence protocols, is even more complex.

This chapter has demonstrated that cache line transactions can be used to reduce the number of cache line flushes triggered by an application. Micro-benchmarks have shown that the performance impact of using the single cache line transaction scheme in contrast to logging is immense. The basic mechanism is not applicable in all scenarios. Modifying more than one cache line in a transaction calls for the

multi cache line transaction mechanism. In comparison to logging, the multi cache line transaction scheme also reduces the number of cache line flushes and results in faster execution times.

Of course, there is a price to pay for fewer cache line flushes: the data layout changes. Due to the cache line layout enforced by the transaction schemes, the working set of an application increases. An analysis of an example where the layout is crucial for cache utilization has shown that cache line transactions are beneficial despite their layout changes. One reason for this unexpected outcome is the overhead caused by internals of the PMDK framework. Improvements for the framework and an outlook on future work are given in the upcoming, final chapter.

CHAPTER 7

# Conclusion

In the near future, a new byte-addressable, non-volatile memory will become available: Non-volatile Random-Access-Memory (NVRAM). Its unique combination of features unites worlds which have been separated for a long time. Although the technology is not yet available, NVRAM is heavily discussed and analyzed by recent research projects. Several projects have already proposed changes to processor and cache designs for a future with NVRAM. They have, however, not yet reached a consensus. In the near future, it will be likely that NVRAM finds its way into existing systems, hand in hand with common off-the-shelf hardware. If its benefits are convincing in these settings, chances of being widely accepted and finding additional funding for further research increase. The near term future with NVRAM has been subject to this thesis.

A major concern with NVRAM are power outages which cause the remaining volatile memories in a system to lose their contents. In order to avoid data loss, transactional mechanisms can be employed. Their basic idea is a multi-versioning approach which stores one version of data prior to any modification and another one once all changes are carried out. For a short period, the old and new states of data coexist. Existing mechanisms, as used by Intel's Persistent Memory Development Kit (PMDK), are inspired by the traditional storage hierarchy and separate the two versions spatially, in distinct memory regions. As a consequence, these schemes have to use at least two flush instructions to assure that both versions are written to their persistent location. Flush instructions become critical for the performance of applications on NVRAM because they enforce ordering an heavily out-of-order system.

This work presented the novel idea of preserving the two versions used by a transaction in a single cache line. The basic form of cache line transactions supports only one cache line per transaction. It is the most restrictive, but also the most beneficial version. Each transaction triggers the minimum of one flush. Except for one bit, no additional meta data is required. Other transaction mechanisms have to store meta data, like source locations or transaction identifiers. Such information is not required for the pure form of cache line transactions.

Although it has been sketched how to construct more complex data structures based on single cache line transactions, their restrictions might be too strict for some use cases. This has been addressed with extensions of the basic scheme that support the modification of multiple cache lines in a transaction. None of the three presented variants of multi cache line transactions uses as much meta data as logging, because there is still no need to keep track of source locations. However, additional meta data

is stored in volatile or non-volatile memory. Two approaches also require hardware support which may not be provided on all platforms. The number of cache line flushes is still at a minimum.

The cost of cache line transactions are changes to the data layout, which result in larger working sets. This price has to be paid even when no transactions are active and data is only read. The contrary approach of logging, on the other hand, does not enforce such a penalty in read-only workloads. Still, multi cache line transactions have outperformed logging even in read-mostly workloads.

Future systems with NVRAM can benefit greatly when logging is replaced by cache line transactions. For now, their usage is a bit tedious, mostly because of the data layout requirements. Closer integration into persistent memory frameworks and compiler support may remove this burden from programmers.

However, a few concerns remain. The experiments rely on Intel's PMDK framework for persistent memory. Its exact details are scarcely documented. Most importantly, it has shown a significant overhead which can not fully be attributed to the transactional logging mechanism. These overheads should gain focus in upcoming projects. It would also be interesting to observe the effects of integrating cache line transactions into main memory databases.

A tool for counting the number of cache flush instructions triggered by an application has been developed for the evaluation of this work. During the development phase, this tool helped to detect implementation failures when only two cache line flushes have been triggered but 2 MB of data have been modified. It is a first step for verification and validation of software for non-volatile memory. This is going to become more and more important when NVRAM replaces existing layers in the storage hierarchy. While today, on Dynamic Random-Access-Memory (DRAM), a reboot typically cures systems from the effects of failures, this may no longer be an option with NVRAM.

Finally, I would like to pick up a topic which has only briefly been considered in Chapter 4: the adaption of lock-free synchronization algorithms to NVRAM. These concurrent synchronization schemes typically employ a helping mechanism. Before a thread starts to modify data, it denotes the changes it is about to perform. Initially, this information is designed so that other threads avoid waiting for the thread to complete its changes by helping. However, the information can also be seen as a redo information for activities following a power outage. Combining concurrency control and consistency with respect to persistence seems highly promising for future, scalable systems with multiple processing cores.

# APPENDIX A

# Acronyms

| | |
|---|---|
| **CAS** | Compare and Swap |
| **CPU** | Central Processing Unit |
| **DRAM** | Dynamic Random-Access-Memory |
| **FeRAM** | Ferroelectric RAM |
| **MRAM** | Magnetic RAM |
| **NUMA** | Non-Uniform Memory Access |
| **NVRAM** | Non-volatile Random-Access-Memory |
| **OS** | Operating System |
| **PCM** | Phase-Change Memory |
| **PMDK** | Persistent Memory Development Kit |
| **RRAM** | Resistive RAM |
| **STM** | Software Transactional Memory |
| **STT-MRAM** | Spin-Transfer-Torque MRAM |

# APPENDIX B

# Errata

The original description of the multi cache line transaction scheme MCL-2TXIDS is missing one cache line flush. This chapter presents the corrected algorithm. Since the scheme has been used in the evaluation, updated results are also presented.

The MCL-2TXIDS cache line layout added a transaction identifier to each cache line slot in order to group cache lines which are modifed in the same transaction. In the first step of each transaction, the cache lines are linked to the current transaction by writing the identifier. Thereafter, each of the cache lines has to be flushed. Then, data from the currently active slot can be copied to the inactive slot where it is modified. At the end, the new slot is activated and the cache line is flushed once more. All in all, each cache line is flushed twice.

It is impossible to perform recovery when the first flush is omitted. Consider the example shown in Section 4.5.3. The address book entries for Alice and Bob are updated in transaction 4711. In the situation shown in Figure B.1(a), the transaction is nearly complete. Both updates have been performed and the new slots have been activated by writing 1 to the meta data. In the next two steps, the cache lines are flushed. If however, a failure happens before the flushes are triggered, the contents in Non-volatile Random-Access-Memory (NVRAM) are unforeseeable. In particular, it may happen that one of the cache lines has already been flushed by a hardware triggered cache eviction.

In the situation depicted in (b), the cache line containing Alice' data has been written to NVRAM, but Bob's change is still cached. If none of the changes to Bob's cache line have been flushed in the current transaction, Bob's cache line is not linked to transaction 4711 and recovery may decide that all changes made by transaction 4711 are complete. In order to avoid this mistake, each of the cache lines has to be flushed after it has been linked to the current transaction.

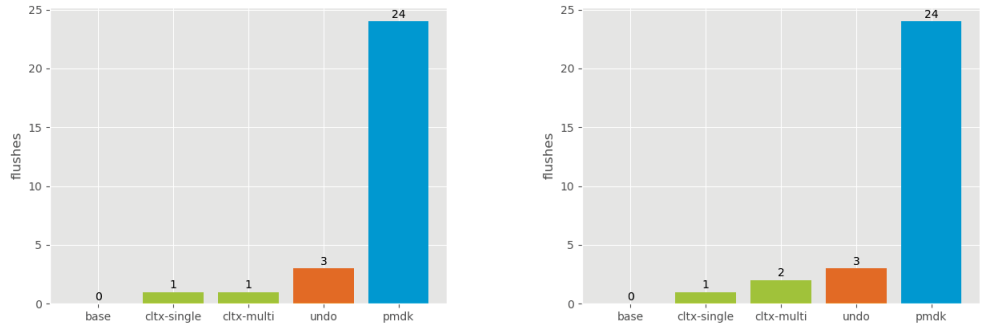

**(a)** Both cache lines have been updated, but the software did not yet trigger a flush.

**(b)** Possible NVRAM contents before the software triggers the first flush. Due to a hardware cache eviction, Alice' data has already been updated in NVRAM.

**Figure B.1:** Address book update using MCL-2TXIDS.

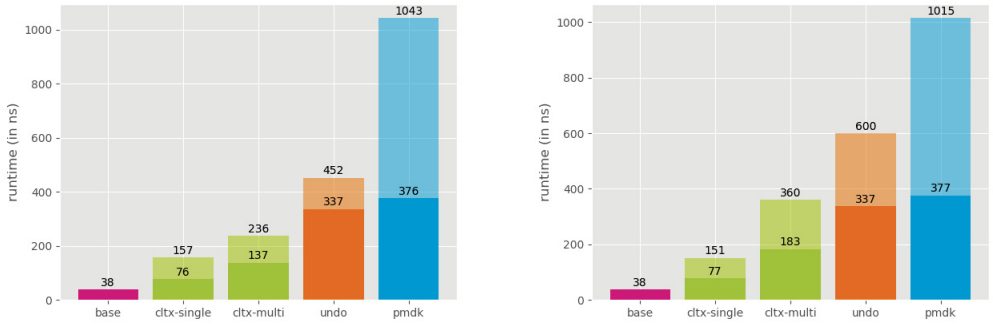
Evaluations using the multi cache line transaction mechanism have been repeated and the results are presented in the next figures. The number of flushes for the multi cache line transaction scheme increases by one flush per cache line in each transaction. Runtimes are also slightly affected.

The experiments have also been repeated for the Game of Life examples. As shown earlier, the number of cache line flushes does not affect the runtime of the algorithm. Hence, updating the cache line transaction scheme does not change the presented results.



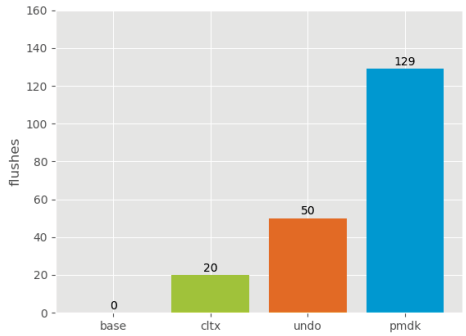

**(a)** Original data. **(b)** *cltx-multi* with additional flush.

**Figure B.2:** Number of cache line flushes triggered when two integers stored in a pair are incremented.

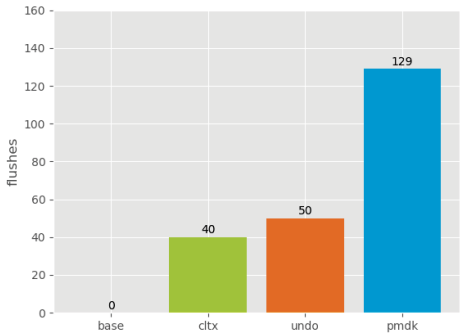



**(a)** Original data. **(b)** *cltx-multi* with additional flush.

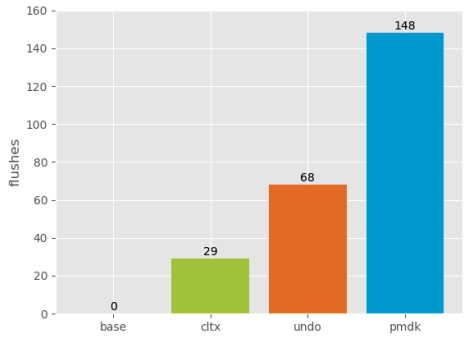
**Figure B.3:** Runtimes of the pair micro benchmark.



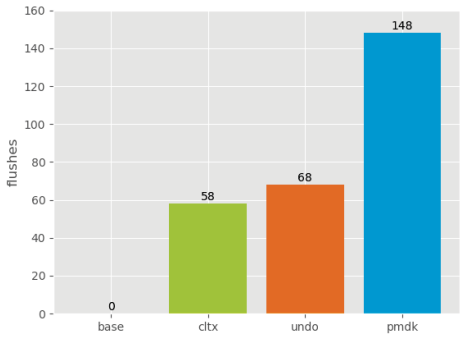

**(a)** Insert at the head of the list (original).




**(b)** Insert at the head of the list (additional flush).

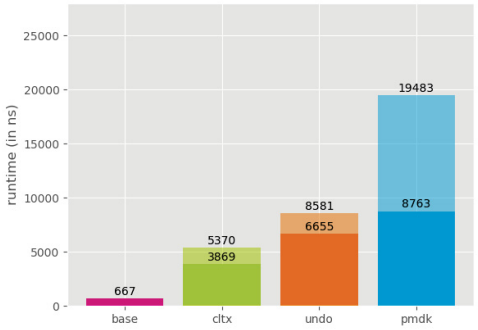



**(c)** Insert at the tail of the list (original).

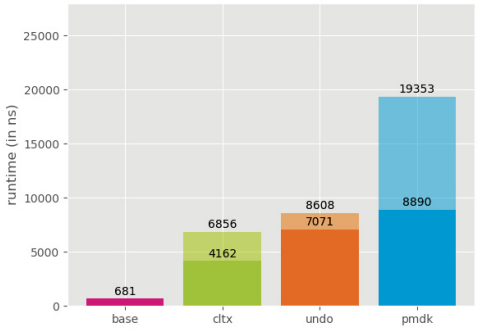



**(d)** Insert at the tail of the list (additional flush).

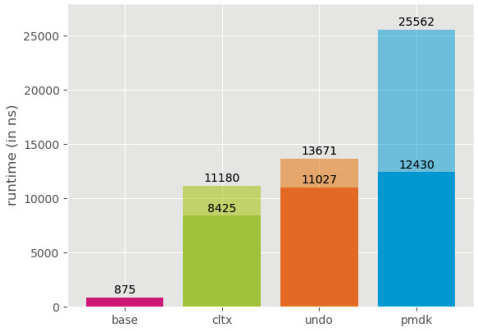
**Figure B.4:** Number of cache line flushes triggered when 10 elements are inserted in a linked list.



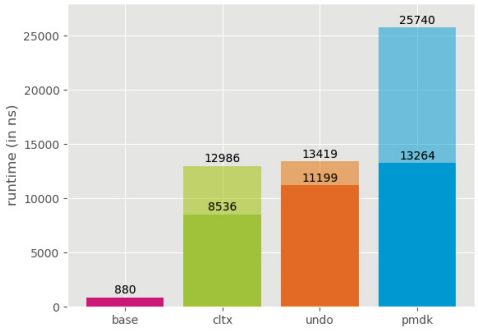

**(a)** Insert at the head of the list (original).




**(b)** Insert at the head of the list (additional flush).




**(c)** Insert at the tail of the list (original).




**(d)** Insert at the tail of the list (additional flush).

**Figure B.5:** Runtime for inserting 10 elements in a linked list.

# Bibliography


[1] Katelin A. Bailey et al. 'Exploring Storage Class Memory with Key Value Stores'. In: *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*. INFLOW '13. Farmington, Pennsylvania: ACM, 2013, 4:1–4:8. ISBN: 978-1-4503-2462-5. DOI: 10.1145/2527792.2527799. URL: http://doi.acm.org/10.1145/2527792.2527799.

[2] Katelin Bailey et al. 'Operating System Implications of Fast, Cheap, Non-Volatile Memory.' In: *HotOS*. Vol. 13. 2011, pp. 2–2.

[3] F. Bedeschi et al. 'A Multi-Level-Cell Bipolar-Selected Phase-Change Memory'. In: *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*. Feb. 2008, pp. 428–625. DOI: 10.1109/ISSCC.2008.4523240.

[4] Hal Berenson et al. 'A Critique of ANSI SQL Isolation Levels'. In: *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*. SIGMOD '95. San Jose, California, USA: ACM, 1995, pp. 1–10. ISBN: 0-89791-731-6. DOI: 10.1145/223784.223785. URL: http://doi.acm.org/10.1145/223784.223785.

[5] Kumud Bhandari, Dhruva R. Chakrabarti and Hans-J. Boehm. 'Implications of CPU Caching on Byte-addressable Non-Volatile Memory Programming'. In: 2012.

[6] Adrian M. Caulfield et al. 'Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories'. In: *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 385–395. ISBN: 978-0-7695-4299-7. DOI: 10.1109/MICRO.2010.33. URL: http://dx.doi.org/10.1109/MICRO.2010.33.

[7] Dhruva R. Chakrabarti, Hans-J. Boehm and Kumud Bhandari. 'Atlas: Leveraging Locks for Non-volatile Memory Consistency'. In: *SIGPLAN Not.* 49.10 (Oct. 2014), pp. 433–452. ISSN: 0362-1340. DOI: 10.1145/2714064.2660224. URL: http://doi.acm.org/10.1145/2714064.2660224.

[8] Andreas Chatzistergiou, Marcelo Cintra and Stratis D. Viglas. 'REWIND: Recovery Write-Ahead System for In-Memory Non-Volatile Data-Structures'. In: *Proc. VLDB Endow.* 8.5 (Jan. 2015), pp. 497–508. ISSN: 2150-8097. DOI: 10.14778/2735479.2735483. URL: http://dx.doi.org/10.14778/2735479.2735483.

[9] F. Chen, M. P. Mesnier and S. Hahn. 'A protected block device for Persistent Memory'. In: *2014 30th Symposium on Mass Storage Systems and Technologies (MSST)*. June 2014, pp. 1–12. DOI: 10.1109/MSST.2014.6855541.

[10] Kaimeng Chen, Peiquan Jin and Lihua Yue. 'A Survey on Phase Change Memory-Aware Cache Management'. In: *International Journal of Multimedia and Ubiquitous Engineering* 11 (1 2016), pp. 293–310. URL: http://www.earticle.net/article.aspx?sn=268418.

[11] Shimin Chen and Qin Jin. 'Persistent B+-trees in Non-volatile Main Memory'. In: *Proc. VLDB Endow.* 8.7 (Feb. 2015), pp. 786–797. ISSN: 2150-8097. DOI: 10.14778/2752939.2752947. URL: http://dx.doi.org/10.14778/2752939.2752947.

[12] Joel Coburn et al. 'NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories'. In: *SIGPLAN Not.* 46.3 (Mar. 2011), pp. 105–118. ISSN: 0362-1340. DOI: 10.1145/1961296.1950380. URL: http://doi.acm.org/10.1145/1961296.1950380.

[13] Jeremy Condit et al. 'Better I/O Through Byte-addressable, Persistent Memory'. In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: ACM, 2009, pp. 133–146. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629589. URL: http://doi.acm.org/10.1145/1629575.1629589.

[14] Intel Corporation. *5th Generation Intel Core Processor Family, Intel Core M Processor Family, Mobile Intel Pentium Processor Family, and Mobile Intel Celeron Processor Family*. Tech. rep. 2015.

[15] Justin DeBrabant et al. 'A prolegomenon on OLTP database systems for non-volatile memory'. In: (2014).

[16] Subramanya R. Dulloor et al. 'System Software for Persistent Memory'. In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. Amsterdam, The Netherlands: ACM, 2014, 15:1–15:15. ISBN: 978-1-4503-2704-6. DOI: 10.1145/2592798.2592814. URL: http://doi.acm.org/10.1145/2592798.2592814.

[17] Alan Fekete, Elizabeth O'Neil and Patrick O'Neil. 'A Read-only Transaction Anomaly Under Snapshot Isolation'. In: *SIGMOD Rec.* 33.3 (Sept. 2004), pp. 12–14. ISSN: 0163-5808. DOI: 10.1145/1031570.1031573. URL: http://doi.acm.org/10.1145/1031570.1031573.

[18] Alan Fekete et al. 'Making Snapshot Isolation Serializable'. In: *ACM Trans. Database Syst.* 30.2 (June 2005), pp. 492–528. ISSN: 0362-5915. DOI: 10.1145/1071610.1071615. URL: http://doi.acm.org/10.1145/1071610.1071615.

[19] Jim Gray et al. 'The transaction concept: Virtues and limitations'. In: *VLDB*. Vol. 81. 1981, pp. 144–154.

[20] Timothy Harris. 'A pragmatic implementation of non-blocking linked-lists'. In: *Distributed Computing* (2001), pp. 300–314.

[21] Timothy L. Harris, Keir Fraser and Ian A. Pratt. 'A Practical Multi-word Compare-and-Swap Operation'. In: *Distributed Computing*. Ed. by Dahlia Malkhi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 265–279. ISBN: 978-3-540-36108-4.

[22] Andreas Heuer. *Objektorientierte Datenbanken*. Addison-Wesley, 1997.

[23] Yenpo Ho, Garng M. Huang and Peng Li. 'Nonvolatile Memristor Memory: Device Characteristics and Design Implications'. In: *Proceedings of the 2009 International Conference on Computer-Aided Design*. ICCAD '09. San Jose, California: ACM, 2009, pp. 485–490. ISBN: 978-1-60558-800-1. DOI: 10.1145/1687399.1687491. URL: http://doi.acm.org/10.1145/1687399.1687491.

[24] M. Hosomi et al. 'A novel nonvolatile memory with spin torque transfer magnetization switching: spin-ram'. In: *IEEE InternationalElectron Devices Meeting, 2005. IEDM Technical Digest.* Dec. 2005, pp. 459–462. DOI: 10.1109/IEDM.2005.1609379.

[25] Taeho Hwang, Jaemin Jung and Youjip Won. 'HEAPO: Heap-Based Persistent Object Store'. In: *Trans. Storage* 11.1 (Dec. 2014), 3:1–3:21. ISSN: 1553-3077. DOI: 10.1145/2629619. URL: http://doi.acm.org/10.1145/2629619.

[26] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2*. 325383-064US. Oct. 2017.

[27] Arpit Joshi et al. 'Efficient Persist Barriers for Multicores'. In: *Proceedings of the 48th International Symposium on Microarchitecture*. MICRO-48. Waikiki, Hawaii: ACM, 2015, pp. 660–671. ISBN: 978-1-4503-4034-2. DOI: 10.1145/2830772.2830805. URL: http://doi.acm.org/10.1145/2830772.2830805.

[28] S. Kang et al. 'A 0.1- $\mu$m 1.8-V 256-Mb Phase-Change Random Access Memory (PRAM) With 66-MHz Synchronous Burst-Read Operation'. In: *IEEE Journal of Solid-State Circuits* 42.1 (Jan. 2007), pp. 210–218. ISSN: 0018-9200. DOI: 10.1109/JSSC.2006.888349.

[29] Aasheesh Kolli et al. 'High-Performance Transactions for Persistent Memories'. In: *ASPLOS*. 2016.

[30] Chun-Hao Lai, Jishen Zhao and Chia-Lin Yang. 'Leave the Cache Hierarchy Operation As It Is: A New Persistent Memory Accelerating Approach'. In: *Proceedings of the 54th Annual Design Automation Conference 2017*. DAC '17. Austin, TX, USA: ACM, 2017, 5:1–5:6. ISBN: 978-1-4503-4927-7. DOI: 10.1145/3061639.3062272. URL: http://doi.acm.org/10.1145/3061639.3062272.

[31] Philip Lantz et al. 'Yat: A Validation Framework for Persistent Memory Software.' In: *USENIX Annual Technical Conference*. 2014, pp. 433–438.

[32] Benjamin C. Lee et al. 'Architecting Phase Change Memory As a Scalable Dram Alternative'. In: *SIGARCH Comput. Archit. News* 37.3 (June 2009), pp. 2–13. ISSN: 0163-5964. DOI: 10.1145/1555815.1555758. URL: http://doi.acm.org/10.1145/1555815.1555758.

[33] Myoung-Jae Lee et al. 'A fast, high-endurance and scalable non-volatile memory device made from asymmetric Ta2O5- x/TaO2- x bilayer structures'. In: *Nature materials* 10.8 (2011), pp. 625–630.

[34] Se Kwon Lee et al. 'WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems'. In: *FAST*. 2017.

[35] Tobin J. Lehman and Michael J. Carey. 'A Recovery Algorithm for a High-performance Memory-resident Database System'. In: *SIGMOD Rec.* 16.3 (Dec. 1987), pp. 104–117. ISSN: 0163-5808. DOI: 10.1145/38714.38730. URL: http://doi.acm.org/10.1145/38714.38730.

[36] Shuo Li et al. 'SPMS: Strand based persistent memory system'. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017* (2017), pp. 622–625.

[37] Y. Lu et al. 'Improving Performance and Endurance of Persistent Memory with Loose-Ordering Consistency'. In: *IEEE Transactions on Parallel and Distributed Systems* PP.99 (2017), pp. 1–1. ISSN: 1045-9219. DOI: 10.1109/TPDS.2017.2701364.

[38] Y. Lu et al. 'Loose-Ordering Consistency for persistent memory'. In: *2014 IEEE 32nd International Conference on Computer Design (ICCD)*. Oct. 2014, pp. 216–223. DOI: 10.1109/ICCD.2014.6974684.

[39] Youyou Lu, Jiwu Shu and Long Sun. 'Blurred Persistence: Efficient Transactions in Persistent Memory'. In: *Trans. Storage* 12.1 (Jan. 2016), 3:1–3:29. ISSN: 1553-3077. DOI: 10.1145/2851504. URL: http://doi.acm.org/10.1145/2851504.

[40] Brandon Lucia and Benjamin Ransford. 'A Simpler, Safer Programming and Execution Model for Intermittent Systems'. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA: ACM, 2015, pp. 575–585. ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2737978. URL: http://doi.acm.org/10.1145/2737924.2737978.

[41] Jagan Singh Meena et al. 'Overview of emerging nonvolatile memory technologies'. In: *Nanoscale research letters* 9.1 (2014), p. 526.

[42] Justin Meza et al. 'A case for efficient hardware/software cooperative management of storage and memory'. In: (2013).

[43] S. Mittal and J. S. Vetter. 'A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems'. In: *IEEE Transactions on Parallel and Distributed Systems* 27.5 (May 2016), pp. 1537–1550. ISSN: 1045-9219. DOI: 10.1109/TPDS.2015.2442980.

[44] Dushyanth Narayanan and Orion Hodson. 'Whole-system Persistence'. In: *SIGPLAN Not.* 47.4 (Mar. 2012), pp. 401–410. ISSN: 0362-1340. DOI: 10.1145/2248487.2151018. URL: http://doi.acm.org/10.1145/2248487.2151018.

[45] Faisal Nawab et al. 'Procrastination Beats Prevention: Timely Sufficient Persistence for Efficient Crash Resilience.' In: *EDBT*. 2015, pp. 689–694.

[46] 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*. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, pp. 677–689. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2749436. URL: http://doi.acm.org/10.1145/2723372.2749436.

[47] Ren Ohmura, Nobuyuki Yamasaki and Yuichiro Anzai. 'A Design of the Persistent Operating System with Non-volatile Memory'. In: *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*. EW 10. Saint-Emilion, France: ACM, 2002, pp. 149–152. DOI: 10.1145/1133373.1133401. URL: http://doi.acm.org/10.1145/1133373.1133401.

[48] Ismail Oukid et al. ‘FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory’. In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD ’16. San Francisco, California, USA: ACM, 2016, pp. 371–386. ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2915251. URL: http://doi.acm.org/10.1145/2882903.2915251.

[49] Ismail Oukid et al. ‘SOFORT: A Hybrid SCM-DRAM Storage Engine for Fast Data Recovery’. In: *Proceedings of the Tenth International Workshop on Data Management on New Hardware*. DaMoN ’14. Snowbird, Utah: ACM, 2014, 8:1–8:7. ISBN: 978-1-4503-2971-2. DOI: 10.1145/2619228.2619236. URL: http://doi.acm.org/10.1145/2619228.2619236.

[50] S. Pelley, P. M. Chen and T. F. Wenisch. ‘Memory persistency’. In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. June 2014, pp. 265–276. DOI: 10.1109/ISCA.2014.6853222.

[51] S. Raoux et al. ‘Phase-change random access memory: A scalable technology’. In: *IBM Journal of Research and Development* 52.4.5 (July 2008), pp. 465–479. ISSN: 0018-8646. DOI: 10.1147/rd.524.0465.

[52] Andy Rudoff. *Deprecating the PCOMMIT Instruction*. https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction. Accessed: 2017-10-23.

[53] Andy Rudoff. ‘Persistent Memory Programming’. In: *;login: The USENIX Magazine* 42.2 (2017), pp. 34–40.

[54] Andy Rudoff. ‘Programming models for emerging non-volatile memory technologies’. In: *;login: The USENIX Magazine* 38.3 (2013), pp. 40–45.

[55] Gunter Saake, Can Türker and Ingo Schmitt. *Objektdatenbanken*. International Thomson Publishing, 1997.

[56] J. M. Tendler et al. *POWER4 system microarchitecture*. Tech. rep. 2002.

[57] Jana Traue et al. ‘Using Emulation Software to Predict the Performance of Algorithms on NVRAM’. In: *Proceedings of the 7th International ICST Conference on Simulation Tools and Techniques*. SIMUTools ’14. Lisbon, Portugal: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2014, pp. 142–146. ISBN: 978-1-63190-007-5. DOI: 10.4108/icst.simutools.2014.254796. URL: https://doi.org/10.4108/icst.simutools.2014.254796.

[58] Shivaram Venkataraman et al. ‘Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory’. In: *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*. FAST’11. San Jose, California: USENIX Association, 2011, pp. 5–5. ISBN: 978-1-931971-82-9. URL: http://dl.acm.org/citation.cfm?id=1960475.1960480.

[59] Haris Volos, Andres Jaan Tack and Michael M. Swift. ‘Mnemosyne: Lightweight Persistent Memory’. In: *SIGPLAN Not.* 47.4 (Mar. 2011), pp. 91–104. ISSN: 0362-1340. DOI: 10.1145/2248487.1950379. URL: http://doi.acm.org/10.1145/2248487.1950379.

[60] Haris Volos et al. ‘Aerie: Flexible File-system Interfaces to Storage-class Memory’. In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys ’14. Amsterdam, The Netherlands: ACM, 2014, 14:1–14:14. ISBN: 978-1-4503-2704-6. DOI: 10.1145/2592798.2592810. URL: http://doi.acm.org/10.1145/2592798.2592810.

[61] Qingsong Wei, Jianxi Chen and Cheng Chen. ‘Accelerating File System Metadata Access with Byte-Addressable Nonvolatile Memory’. In: *Trans. Storage* 11.3 (July 2015), 12:1–12:28. ISSN: 1553-3077. DOI: 10.1145/2766453. URL: http://doi.acm.org/10.1145/2766453.

[62] Xiaojian Wu, Sheng Qiu and A. L. Narasimha Reddy. ‘SCMFS: A File System for Storage Class Memory and Its Extensions’. In: *Trans. Storage* 9.3 (Aug. 2013), 7:1–7:23. ISSN: 1553-3077. DOI: 10.1145/2501620.2501621. URL: http://doi.acm.org/10.1145/2501620.2501621.

[63] Fei Xia et al. ‘A Survey of Phase Change Memory Systems’. In: *Journal of Computer Science and Technology* 30.1 (2015), pp. 121–144. ISSN: 1860-4749. DOI: 10.1007/s11390-015-1509-2. URL: http://dx.doi.org/10.1007/s11390-015-1509-2.

[64] Jun Yang et al. ‘NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems’. In: *Proceedings of the 13th USENIX Conference on File and Storage Technologies*. FAST’15. Santa Clara, CA: USENIX Association, 2015, pp. 167–181. ISBN: 978-1-931971-201. URL: http://dl.acm.org/citation.cfm?id=2750482.2750495.

[65] S. Yu and P. Y. Chen. ‘Emerging Memory Technologies: Recent Trends and Prospects’. In: *IEEE Solid-State Circuits Magazine* 8.2 (2016), pp. 43–56. ISSN: 1943-0582. DOI: 10.1109/MSSC.2016.2546199.

[66] Jishen Zhao et al. ‘Kiln: Closing the Performance Gap Between Systems with and Without Persistence Support’. In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-46. Davis, California: ACM, 2013, pp. 421–432. ISBN: 978-1-4503-2638-4. DOI: 10.1145/2540708.2540744. URL: http://doi.acm.org/10.1145/2540708.2540744.

[67] Ping Zhou et al. ‘A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology’. In: *SIGARCH Comput. Archit. News* 37.3 (June 2009), pp. 14–23. ISSN: 0163-5964. DOI: 10.1145/1555815.1555759. URL: http://doi.acm.org/10.1145/1555815.1555759.