



Leveraging Non-Volatile Memory in Modern Storage Management Architectures

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von
Lucas Lersch, M.Sc.
geboren am 6. November 1990 in Porto Alegre, Brasilien

Gutachter:

Prof. Dr.-Ing. Wolfgang Lehner
Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Lehrstuhl für Datenbanken
01062 Dresden

Prof. Dr.-Ing. Dr. h. c. Theo Härder
Technische Universität Kaiserslautern
Fachbereich Informatik
Lehrgebiet Informationssysteme
AG Datenbanken und Informationssysteme
67653 Kaiserslautern

Tag der Verteidigung: 30. Oktober 2020

Dresden, den 30. Oktober 2020

To my parents: Suzana and Nelson.

ABSTRACT

Non-volatile memory technologies (NVM) introduce a novel class of devices that combine characteristics of both storage and main memory. Like storage, NVM is not only persistent, but also denser and cheaper than DRAM. Like DRAM, NVM is byte-addressable and has lower access latency. In recent years, NVM has gained a lot of attention both in academia and in the data management industry, with views ranging from skepticism to over excitement. Some critics claim that NVM is not cheap enough to replace flash-based SSDs nor is it fast enough to replace DRAM, while others see it simply as a storage device. Supporters of NVM have observed that its low latency and byte-addressability requires radical changes and a complete rewrite of storage management architectures.

This thesis takes a moderate stance between these two views. We consider that, while NVM might not replace flash-based SSD or DRAM in the near future, it has the potential to reduce the gap between them. Furthermore, treating NVM as a regular storage media does not fully leverage its byte-addressability and low latency. On the other hand, completely redesigning systems to be NVM-centric is impractical. Proposals that attempt to leverage NVM to simplify storage management result in completely new architectures that face the same challenges that are already well-understood and addressed by the traditional architectures. Therefore, we take three common storage management architectures as a starting point, and propose incremental changes to enable them to better leverage NVM. First, in the context of *log-structured merge-trees*, we investigate the impact of storing data in NVM, and devise methods to enable small granularity accesses and NVM-aware caching policies. Second, in the context of *B+Trees*, we propose to extend the buffer pool and describe a technique based on the concept of optimistic consistency to handle corrupted pages in NVM. Third, we employ NVM to enable larger capacity and reduced costs in a *index+log* key-value store, and combine it with other techniques to build a system that achieves low tail latency. This thesis aims to describe and evaluate these techniques in order to enable storage management architectures to leverage NVM and achieve increased performance and lower costs, without major architectural changes.

ACKNOWLEDGMENTS

While this thesis carries a single name as its author, it is actually the result of the contributions and support of many people during the years that led to it.

First and foremost, it has been a privilege to be mentored and supervised by Prof. Dr.-Ing. Wolfgang Lehner. Not only has Wolfgang contributed significantly to my professional growth, but he has also been incredibly supportive on a personal level helping me to face the challenges and get through the obstacles I have faced along the way. Furthermore, it is truly inspiring to witness the positive impact that Wolfgang has in the life of its students.

Second, I would like to thank Prof. Dr.-Ing. Dr. h. c Theo Härder for reviewing this thesis. Prof. Härder was one of the people who inspired me to pursue a PhD degree, by granting me the opportunity to participate in his research group through an exchange program in 2012 during my bachelor's and later master's degree programs.

I am also immensely grateful for the opportunity to conduct this work in close collaboration with SAP, as part of the SAP HANA Database Campus. Ivan Schreter had a major role in supervising my work and teaching me a lot about software development. Arne Schwarz has been fundamental in leading the SAP HANA Database Campus and ensuring that the other students and I had the necessary tools, equipment, and environment required for us to focus on our research work. I would like to extend my gratitude to Alexander Böhm and Norman May for helping making the goals of SAP HANA Database Campus a reality. I would also like to thank the Intel colleagues Thomas Willhalm, Roman Dementiev, Otto Bruggeman and Heinrich Teiken for all of the support they provided me while I was conducting my research.

I would specially like to express my gratitude to two colleagues whom I also have the privilege of calling friends. First, it was while working with Caetano Sauer during my master's degree program that I was inspired to pursue a PhD degree. Not only has he graced me many times with insightful technical discussions, but Caetano has also been there to offer his support during my most difficult of times, to which I am most grateful. My gratitude to Caetano is also extended to his wife Irina and daughter Aria, for always being so kind and welcoming. Second, I would like to thank Ismail Oukid for enabling me to do my PhD as part of the SAP HANA Database Campus. Once I was given the opportunity, he continued to guide my work, even through our occasional heated debates. I look forward to working with Ismail again in the future.

I am also grateful to my colleagues at SAP, who have not only taught me a lot on a professional level, but also made my PhD years a lot more fun. Although I have shared

a limited time with Iraklis, Marcus, Michael R., Robert, David, and Elena, they all served as role models for ideal PhD candidates. I was privileged to start my PhD journey with Georgios, who always contributed to the socialization of the group, and Frank, who always was willing to help me by sharing his deep technical knowledge. Florian has shown me how important it is to insist on high standards. Matthias amazed me with his incredible broad knowledge of things. Thomas has supported me immensely not only on a professional level, but also personal one through engaging discussions. Stefan has been my favorite person to share both my frustrations and achievements during our daily afternoon breaks. Robin has made otherwise stressful days brighter with his sense of humor, good music taste, and by surprising me on how much denial a person can sustain about "The Last Jedi" being a horrible movie and a mistake in the "Star Wars" sequel trilogy. I also had the privilege to witness Tiemo and Michael B. transition from master's students to PhD candidates, and I look forward to seeing them achieve great things. It was also a pleasure to have met Mehdi and Jonas in the final months of my PhD and I wish them success.

On a personal level, I would like to thank my friends, both near and far, who have supported me and gave me the strength to endure this journey. As a representative of these friends, Gilson was a always close and supportive one. In supply-chain management, it is often said that the last mile is the most difficult one when delivering products. This seems to also have been the case of my PhD time, not only due to its own challenges, but also due to external events. I am grateful to Diana for having made this last mile a lot easier. Finally, I would like to thank all my family members who have supported me unconditionally throughout my whole life. I feel truly privileged to have had all the aforementioned people offering their support and encouragement, not only while I conducted my research and wrote this dissertation, but in my personal life as well.

Lucas Lersch
Heidelberg, 23. April 2021

CONTENTS

1	INTRODUCTION	13
1.1	Non-Volatile Memory	14
1.2	Challenges	15
1.3	Non-Volatile Memory & Database Systems	16
1.4	Contributions and Outline	17
2	BACKGROUND	19
2.1	Non-Volatile Memory	19
2.1.1	Types of NVM	19
2.1.2	Access Modes	21
2.1.3	Byte-addressability and Persistency	22
2.1.4	Performance	23
2.2	Related Work	25
2.3	Case Study: Persistent Tree Structures	30
2.3.1	Persistent Trees	31
2.3.2	Evaluation	34
3	LOG-STRUCTURED MERGE-TREES	45
3.1	LSM and NVM	45
3.2	LSM Architecture	46
3.2.1	LevelDB	47
3.3	Persistent Memory Environment	49
3.4	2Q Cache Policy for NVM	51
3.5	Evaluation	53
3.5.1	Write Performance	53
3.5.2	Read Performance	54
3.5.3	Mixed Workloads	56
3.6	Additional Case Study: RocksDB	57
3.6.1	Evaluation	58

4	B+TREES	61
4.1	B+Tree and NVM	61
4.1.1	Category #1: Buffer Extension	62
4.1.2	Category #2: DRAM Buffered Access	63
4.1.3	Category #3: Persistent Trees	63
4.2	Persistent Buffer Pool with Optimistic Consistency	64
4.2.1	Architecture and Assumptions	65
4.2.2	Embracing Corruption	67
4.3	Detecting Corruptions	69
4.3.1	Possible States	70
4.4	Repairing Corruptions	73
4.5	Performance Evaluation and Expectations	74
4.5.1	Checksum Overhead	76
4.5.2	Runtime and Recovery	79
4.6	Discussion	80
5	INDEX+LOG KEY-VALUE STORES	83
5.1	The Case for Tail Latency	84
5.2	Goals and Overview	85
5.3	Execution Model	85
5.3.1	Reactive Systems and Actor Model	85
5.3.2	Message-Passing Communication	86
5.3.3	Cooperative Multitasking	87
5.4	Log-Structured Storage	89
5.5	Networking	90
5.6	Implementation Details	91
5.6.1	NVM Allocation on RStore	91
5.6.2	Log-Structured Storage and Indexing	92
5.6.3	Garbage Collection	94
5.6.4	Logging and Recovery	97
5.7	System Operations	98
5.8	Evaluation	99
5.8.1	Methodology	99
5.8.2	Environment	99
5.8.3	Other Systems	100
5.8.4	Throughput Scalability	101

5.8.5 Tail Latency	103
5.8.6 Scans	104
5.8.7 Memory Consumption	106
5.9 Related Work	107
6 CONCLUSION	109
BIBLIOGRAPHY	113
A PIBENCH	125

1

INTRODUCTION

Storage managers are in the core of database systems. They are critical for achieving high performance in the overall system by enabling efficient access to slower storage devices which usually become the bottleneck. Furthermore, storage managers are also responsible for providing the A (atomicity) and D (durability) of the ACID properties [HR83], thus preventing data loss and corruption. As a consequence of being so critical, storage management has to be constantly revisited in order to properly leverage modern hardware and keep the pace with the ever growing data processing demands.

A good example of this constant revisit is the *The Five-Minute Rule* series, which surveys once per decade the advances in modern hardware and their implications in database systems. In the original paper from 1987 [GP87], Jim Gray and Gianfranco Putzolu considered metrics of DRAM and HDD, such as latency, bandwidth, and cost, to calculate the break-even point at which the cost of keeping a page in memory matches the cost of doing I/O to read the page from HDD. They introduced the rule of thumb that named the series: “*Pages referenced every five minutes should be memory resident*”. In 1997, Jim Gray and Goetz Graefe revisited the work in view of technological improvements of HDDs: ten-fold increase in access speeds, hundred-fold increase in capacity, and ten-thousand-fold decrease in costs [GG97]. In 2007, Goetz Graefe investigated the impact of a novel technology by the time: flash [Gra07]. While it was still in its early stages and many questions were still open, it was a promising technology. The main take-away was the prediction that flash would be used to fill the gap between DRAM and HDD, which has since been proven correct. More recently, in 2017, Appuswamy et. al. revisited the discussion in view of now well-established NAND flash-based solid-state storage devices (SSD) [ABGA17]. The initial challenges imposed by flash in 2007, namely reduced lifetime and higher costs, are long gone and, as a consequence, SSDs became the main storage device, thus relegating HDDs to a high-density storage medium for infrequently accessed data. The performance gap between DRAM and SSDs is reduced even further with the advent of PCIe NVMe SSDs, also referred to as *enterprise SSDs*. The work also scratches the surface of a novel storage technology rising on the horizon: non-volatile memory (NVM).

It is still too early to revisit the series in view of NVM. Even if the technology is currently available, it is not yet as established as DRAM, SSDs, or HDDs. Many works are exploring ways to properly leverage NVM in the context of database systems and answering open questions. The advancements made by these works will pave the way to the next chapter in the *The Five-Minute Rule* series. This dissertation aspires to be one of these works.

1.1 NON-VOLATILE MEMORY

Many different underlying technologies (PCM [PSU⁺70], carbon nanotubes [DVTBH13], ReRAM [Chu19], STT-MRAM [HYY⁺05, H⁺08], battery-backed DRAM) and names (non-volatile memory, NVRAM, storage class memory, persistent memory)¹ belong to a novel class of storage devices and are often used interchangeably. Although they might have different properties, they all refer to *byte-addressable* persistent media that blurs the boundary between main memory and storage. This is the main reason why history will not simply repeat itself with a faster media, as it was the case with flash-based SSDs. NVM has a higher potential of disrupting modern software stacks.

Works dating from as early as the 1980s entertained the idea of such a technology and its impacts in system architectures [Zak81, Cha78, Wri83, DGS80]. However, the assumption was mostly theoretical [Chu71] or unpractical due to technological limitations or prohibitive costs, existing only in the form of niche accelerators [Mic17, Tec18]. Recent technological advancements made this technology real and more accessible. More prominent is the 3D XPoint technology², developed by Intel and Micron [Mic20, Eva15]. Intel currently commercializes 3D XPoint both as regular Optane PCIe NVMe SSD [Int20b] and Optane DC Persistent Memory Modules (DCPMM for short) [Int20c]. The latter is the one we focus on in this work. As a result, there is renewed attention to this area of research as it promises to deliver the long-awaited features required to blur the boundaries between memory and storage. But how are these boundaries blurred? Or in other words, what does *byte-addressable* really mean? Since this term is often used with different meanings, it is appropriate to discuss and make explicit the definition used in this work.

Common consensus usually defines main memory (DRAM) as byte-addressable and storage (HDD and SSD) as block-addressable. This definition is often based on the fact that the programmer can access a single byte in DRAM. However, from this point of view, common file system interfaces also allow the programmer to read a single byte from a file. It is true that the underlying file system is actually reading a whole block (usually 4 kB) from storage into memory in order to provide access to that single byte, but that is also the case in modern CPU architectures, in which a block (usually much smaller and referred to as *cache line*) is read from memory into the CPU caches. In other words, conceptually there is no difference, as in both cases data is always transferred at larger granularities, be it a block, page, or cache line. Therefore, this definition is too loose and considered inadequate to define byte-addressability. Our definition focus more on the access path rather than on the access granularity. Whenever we refer to byte-addressability, we mean that NVM is attached directly to the same bus as regular DRAM and therefore it is also accessed through the CPU caches and shares the same virtual memory space.

In addition to byte-addressability, NVM also introduces other attractive characteristics. In comparison to modern DRAM, 3D XPoint is cheaper [Aco19] and prices are expected to drop within the next years. It is also denser, in the sense that the largest DRAM module commercially available is 128 GB while Intel has already announced a single DCPMM of up to 1 TB. The latency is higher than that of DRAM, but within the same order magnitude and, therefore, lower than NAND flash. The bandwidth, is better than NAND flash, but still far from that of DRAM. Finally, 3D XPoint also has a higher write endurance than modern NAND flash. All these advantages, however, come at a cost.

¹In this work we opted for using the term *non-volatile memory*, or NVM for short.

²An analysis has shown that 3D XPoint is based on *phase-change memory* (PCM) [Cho17].

1.2 CHALLENGES

The opportunities mentioned also entail challenges, following the “*no free lunch*” conjecture. In the case of NVM, it is beneficial to consider the challenges under two different scopes: *reading* from NVM and *writing* to NVM.

In terms of reading, the challenge lies in investigating NVM performance characteristics in order to properly fit it into the storage hierarchy. For years, similar questions have been investigated by many works in the context of tape, HDDs, SSDs, and DRAM. In the case of byte-addressable NVM, these works only cover part of the problem. In the example of the *The Five Minute Rule* mentioned previously, the rule only dictates which pages should be evicted from DRAM, but it does not focus so much on which pages should be moved to DRAM. Fair enough, before NVM this was never really a concern, since there was no choice, as a page always had to be read from persistent storage to DRAM in order to be accessed. Even if a decision was made to evict a page immediately after reading it from storage, the high I/O cost was already paid. In the case of NVM, pages can be directly accessed without any additional I/O. Therefore, NVM introduces a new dimension that should be considered when optimizing data placement and buffer policies.

Writing directly to NVM is hard. Database systems rely on persistent data to always be consistent in some way. To achieve a desired degree of consistency, it is required full control of *what* and *when* is made persistent. For example, transactional recovery algorithms require log records to be persisted prior to the updated page (a.k.a. *write-ahead logging*). The challenge comes from the fact that, while systems have full control over data movement between memory and traditional storage, this is not the case with NVM. Since NVM can be accessed directly by the CPU, corruption and data loss may occur for many different reasons, such as updated data staying in the CPU cache and being lost at a power failure, or the CPU arbitrarily evicting (and therefore persisting) cache lines in the wrong order. In other words, as opposed to the traditional memory-storage interaction, there is no way of *pinning* a cache line to prevent it from being evicted by the CPU.

Figure 1.1 better illustrates the challenge with an example. The left-hand side shows the memory layout of a sorted array in NVM that spans two cache lines (indicated by the different shades of gray). The first position of the array indicates the current size, while each remaining record is represented by an integer. Three steps are required for inserting a new record **3**: move records **4-9** one position to the right in order to make space, insert record **3** in the appropriate position, increment the size from **5** to **6**. A failure might occur at any intermediate state of this operation and upon restart the sorted array may be found in one of different states illustrated on the right-hand side. In state ①, the operation completed successfully before the failure and the array is found in a consistent state. All the other states are inconsistent. In ②, the records **9** and **7** were moved one position to the right, the CPU arbitrarily evicted the second cache line, and the failure occurred. In ③, the insertion completed, but the cache lines were evicted and the power failure occurred before the size was updated. Finally, in ④, the insertion completed, the size was updated, but only the first cache line was evicted before the failure, while the second portion of the array remained in its original state. The example shows how such a simple and fundamental operation can easily lead to data being corrupted when NVM is used as persistent storage. To a certain extent, addressing this challenge is in the core of many modern works that propose architectures, data structures and algorithms in the context of NVM.

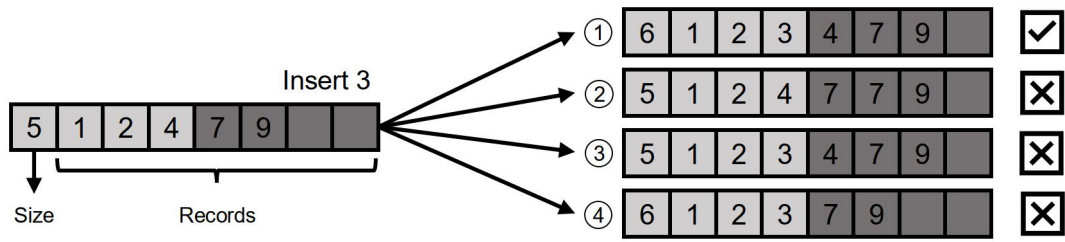


Figure 1.1: Possible states if a failure occurs while inserting a record in a sorted array in NVM (figure inspired by [CJ15]).

1.3 NON-VOLATILE MEMORY & DATABASE SYSTEMS

Many works investigate NVM in the context of databases. They either focus on individual components of the system, such as data structures [CJ15, OLN⁺16, ALML18], memory allocators [CCA⁺11, BCB16, OBL⁺17], logging schemes [HSQ14, WJ14, APP16], or on the database architecture as a whole. These works can also be categorized based on their initial assumptions and expectations regarding NVM. In this dimension, we observe that two categories can be identified: a *conservative* approach and a more *radical* approach.

Conservative approaches simply treat NVM as a faster SSD by accessing it through regular file system interfaces [GXH⁺11, PWGB13, EGA⁺18]. These approaches are commonly seen in industry, under the consideration that NVM is not well-established enough to justify expensive re-architecting efforts. The main advantage is that NVM is used transparently as a caching layer between DRAM and SSD, thus leveraging all devices in the modern storage hierarchy and avoiding the previously mentioned challenge of directly writing to NVM. Therefore, these systems are *NVM-agnostic*, as they behave exactly the same way, from an architectural point of view, both in the presence and absence of NVM.

On the other hand, **radical approaches** are *NVM-centric*, in the sense that they propose novel techniques with the goal of fully leveraging NVM characteristics to achieve higher performance. Recent works in the research community proposing novel database architectures fall in this category [Kim15, OBL⁺14, PAA⁺17]. Furthermore, an extreme assumption is that NVM performance and cost will lead to the extinction of modern DRAM and storage technologies, and drive the adoption of what is known as *single-level systems*. The notion of single-level systems is not new [KELS62] and refers to the idea of having no separation between volatile data and persistent data. While this assumption is far-fetched, it is made by many works [YWC⁺15, CJ15, WLL18, ALML18], as it opens interesting research questions. However, both conservative and radical approaches have drawbacks.

Conservative approaches have two main disadvantages. First, the lower latency, one of the major selling points of NVM, might not be observed, as access to persistent media may be dominated by kernel and file system operations, which become relatively more expensive when compared to I/O costs on modern SSDs. Second, it does not leverage byte-addressability and still imposes unnecessary data movement between DRAM and NVM. In the case of radical approaches, the main disadvantage comes from the fact that they are too optimistic regarding a quick and wide adoption of NVM and therefore the proposed architectures are very different from traditional systems. As a consequence, there is no

obvious way to migrate from a well-established traditional architecture to NVM-centric approaches, since they do not consider the modern storage hierarchy comprised of DRAM, SSD, HDD, and even tape. This becomes an issue since we consider that, as the time of writing and for the near future, NVM will not replace either DRAM or SSD, and therefore it must coexist harmonically with these other devices. Furthermore, the alternative of developing a complete new system from scratch is often unfeasible in practice due to high development costs. Therefore, we argue that **moderate approaches are required**.

Two principles should guide moderate approaches. First, they should focus on transforming systems designed for traditional devices (DRAM, SSD, HDD) into systems that leverage all aspects of NVM, mainly the byte-addressability and persistency. This is done to a certain extent by *hybrid data-structures* [OLN⁺16, XJXS17], which integrate NVM into regular memory data structures in order to make them persistent. While these data structures are often used as containers and building blocks, it is important to also look at this principle from the perspective of a more complete system. Second, the changes required should not be so disruptive that the architectural behavior of the system is completely changed. In other words, the architecture should not become dependant on the presence of NVM.

1.4 CONTRIBUTIONS AND OUTLINE

As mentioned in the beginning of this chapter, storage management in database systems is likely to be the area most impacted by the advent of NVM. Following the previously mentioned moderate approach, we investigate the impact of NVM in the three most common modern architectures for storage management: *LSM*, *B+Tree* and *index+log* [IC20]. For each one of them, we take the current state-of-the-art architecture as a starting point and explore potential use-cases, trade-offs, opportunities, and challenges in view of NVM. The goal is to leverage NVM as more than simply faster storage, while avoiding the, at times prohibitive, effort to design and implement a completely new architecture. In the following, we provide an overview of the organization of this dissertation by giving an overview of each chapter and their respective contributions³:

- **Chapter 2** lays the general technical background on NVM required for the understating of the next chapters. We also cover related work and conclude with our case study on persistent data structures [LHO⁺19], in which we revisit persistent tree structures proposed by prior work. Since these persistent trees were originally evaluated on emulation platforms, we re-implement and re-evaluate them on real NVM hardware as a way of setting the expectations of its real performance and behavior. The required background specific to each following chapter is introduced in the chapter itself.
- **Chapter 3** explores the synergy between NVM and **log-structured merge-tree (LSM)** systems [LOLS17, LOSL17]. We take LevelDB as a first case study and propose a persistent memory environment, *Pmemenv*, that enables LevelDB to directly access NVM. We also analyze the caching behavior and propose a *2Q Cache Policy for NVM* to better leverage the byte-addressability. Finally we complement with a more recent case study on RocksDB (a more modern system) and on real DCPMM.

³Many of the contributions have been peer-reviewed and published in the referenced papers.

- **Chapter 4** investigates opportunities in the context of **B+Tree** architectures. We propose an extended buffer pool with an optimistic consistency model [LLO19]. The optimistic consistency model relies on a checksum algorithm and recovery infrastructure common to most database systems, therefore requiring minor architectural modifications. This introduces a knob that enables the system to choose a trade-off between more NVM (lower costs and faster recovery) or more DRAM (faster performance). We present our algorithm and discuss all possible corner cases, as well as empirically evaluate the overhead and discuss end-to-end performance expectations.
- **Chapter 5** presents *RStore*, a multicore key-value store with an **index+log** architecture [LSOL20]. *RStore* focus on use-cases requiring low tail-latency, such as web caching, real-time systems, and metadata management. We introduce the building blocks of *RStore* and how to integrate NVM in order to enable high performance and lower costs. We show results of isolated evaluations to back our design decisions, as well as an end-to-end evaluation and comparison with similar modern systems.
- **Chapter 6** concludes our work by summarizing the key findings of each chapter and providing directions for future work.

The author of this thesis is responsible for the conception, implementation, evaluation, analysis, and the interpretation of the results for each of the contributions presented. The author is also the main author of the works upon which this dissertation is based on. Nevertheless, the author is thankful to all co-authors that contributed substantially to these works. Many of the ideas proposed had constant feedback and input through discussions with Wolfgang Lehner, Ivan Schreter, and Ismail Oukid. Xiangpeng Hao and Tianzheng Wang contributed with the full implementation of BzTree and with partial implementation of the framework used for the persistent trees evaluation discussed in Section 2.3. Thomas Willhalm contributed with suggestions and analysis of the results collected through hardware counters. Caetano Sauer contributed with clarifications about transactional recovery in the context of the buffer pool with optimistic consistency model in Chapter 4. Thomas Bach contributed with discussions about the probability of data loss due to checksum collisions, on that same chapter. The *RStore* system in Chapter 5 was built by the author on top of the Reflex project at SAP, idealized and implemented by Ivan Schreter. Reflex provides the following building blocks: message-passing communication, cooperative multitasking, memory management, and networking. The author contribution lies on the design, implementation, and evaluation of NVM in that system.

2

BACKGROUND

In this chapter, we cover the background on NVM required for the understanding of the following chapters. We also survey related work that has explored opportunities to leverage NVM in the context of database systems and storage management. Finally, we conclude with an empirical analysis of persistent data structures in order to better exemplify the challenges of NVM and solutions proposed, as well as to set realistic expectations of the performance of NVM.

2.1 NON-VOLATILE MEMORY

Non-volatile memory (NVM)¹ is a novel class of technologies that has been regarded as the next evolution step for persistent storage. The technologies in this class exhibit characteristics of both storage and main memory. More precisely, they provide persistency and high density while also being byte-addressable and offering a latency much closer to that of modern DRAM (albeit often higher). While these characteristics are common to most types of NVM technologies, they might vary slightly between them.

2.1.1 Types of NVM

There are two main groups of NVM technologies. The first group refers to technologies based on flash memory, commonly known as *NVDIMMs*. These devices are motivated by the observation that the recent improvements in throughput and capacity of modern NAND-flash SSDs have shifted the bottleneck from the storage device itself to the PCIe bus. Therefore, to avoid the high latency (tens of μ s) and limited bandwidth (tens of GBps) of PCIe, these devices are interfaced via the same memory bus used for DRAM modules. The *Storage Networking Industry Association* (SNIA) currently standardize two types of NVDIMM. The *NVDIMM-F* type defines a flash-only DIMM, similar to regular PCIe SSDs, but connected to the memory bus. The access latency of devices of this type is lower

¹Also referred to as *non-volatile RAM (NVRAM)*, *storage class memory (SCM)*, or *persistent memory (PM)*.

than that of PCIe NAND-flash SSDs, but still much higher than that of DRAM. They also present capacity and cost similar to those of modern SSDs. The main disadvantage is that NVDIMM-F devices are block-addressable rather than byte-addressable, which is one of the main advantages of other NVM devices. The second type is *NVDIMM-N*, which combines both DRAM and flash storage in a single module. Only the DRAM can be accessed during normal operation. With the aid of an on-chip battery or capacitor, the DRAM content is written to flash during a power failure, and restored during restart. NVDIMM-N devices have the advantage of achieving a performance similar to that of DRAM and of being byte-addressable, but have small capacities (in the tens of GB range) and high costs. Examples of these devices are the ones commercialized by Viking Technology [Tec18] and Micron [Mic18]. In general, NVDIMMs today are considered niche accelerators when compared to PCIe NAND-flash SSDs.

The second group of NVM includes devices based on materials other than NAND-flash, with the most prominent being *phase-change memory* (PCM) [LZY⁺10, PSU⁺70], *spin-torque transfer magnetic RAM* (STT-MRAM) [DWS⁺08, HYY⁺05, H⁺08], and *resistive RAM* (ReRAM) [GKC⁺11, SSSW08, Chu11]. PCM has its roots in research conducted in the 1960s [Ovs68] and is based on the property of certain materials, such as germanium, antimony and tellurium, to persist a phase change when the right current is applied to them. In industry, PCM has been researched by IBM, HGST, Micron, and Intel. STT-MRAM makes use of the spin property of electrons, as opposed to conventional semiconductor electronics that make use of the charge property. It relies on the magnetic retention property of certain materials to store information. Reads rely on sensing the resistant difference between two states of this material, while writes are based on the *spin-torque transfer* effect to change the orientation of the magnetization. Companies like Samsung, Qualcomm, Crocus Technology, Everspin, and Intel have demonstrated works on STT-MRAM technologies. Finally, ReRAM relies on applying high enough voltage to materials that do not conduct electric current. This will force a phenomenon called *dielectric breakdown*, which damages the material. In certain materials, this damage is reversible and the controlled defects caused can be used to represent bits in a binary system. HP Labs is the main company researching ReRAM. Table 2.1 compares the performance characteristics of single-level cell flash, DRAM, and the NVM technologies discussed².

Recent technological advancements lead by Intel and Micron resulted in the development of *3D XPoint*³ [Mic20]. 3D XPoint is based on PCM [Cho17] and is currently considered the NVM technology with the most potential to succeed and to be widely adopted. Intel commercializes 3D XPoint under the market name of *Optane*. Two lines of Optane products exist. First, Optane is sold in PCIe SSD format, similar to the NAND-flash variant, but offer latencies an order of magnitude lower. Second, Optane DC Persistent Memory Modules (DCPMM) are dual in-line memory modules (DIMMs) like DRAM, and thus are connected directly to the memory bus. While Optane SSDs offer a much lower latency than NAND-flash SSDs, they are still interfaced with PCIe and are block-addressable, while DCPMMs are byte-addressable. Furthermore, DCPMM is the NVM technology more accessible in the market and has received support both in Windows [Tob16] and Linux [Lin15] systems. For these reasons, in this work we focus on DCPMM and often use the more general term “NVM” to refer to it. Nevertheless, our findings are not specific to DCPMM and can be applied to other NVM technologies that exhibit access modes.

²In the real world, certain reported numbers might vary due format-specific functionalities, such as wear-leveling techniques employed in SSDs.

³Pronounced “three dee cross point”.

Parameter	SLC Flash	DRAM	PCM	STT-MRAM	ReRAM
Read Latency	25 μ s	50 ns	50 ns	10 ns	10 ns
Write Latency	500 μ s	50 ns	500 ns	50 ns	50 ns
Byte-addressable	No	Yes	Yes	Yes	Yes
Endurance	10^4 - 10^5	$>10^{15}$	10^8 - 10^9	$>10^{15}$	$>10^{11}$
Density	High	Low	Medium	Low	High
Cost	\$	\$\$\$\$	\$\$	\$\$\$	\$\$\$\$

Table 2.1: Comparison of flash, DRAM, and modern NVM technologies.

2.1.2 Access Modes

Enabling modern systems to access NVM requires changes at different levels. At the hardware level, Intel introduced support for DCPMM in the Cascade Lake processor family. The DCPMM can be exposed in two different modes. In the **memory mode**, DCPMM is treated as volatile memory. This mode has the benefit of significantly increasing the perceived amount of memory available, while being completely transparent to the operating system and applications. To amortize memory accesses due to the higher latency of DCPMM, the DRAM present in the system is used as a cache. Therefore, NVM is not treated as byte-addressable media as its contents must always be cached in DRAM instead of being directly accessed by the CPU. Furthermore, in *memory mode*, data on DCPMM becomes inaccessible after a restart, thus not leveraging the persistency of NVM.

In the **app direct mode**, DCPMM is exposed as a device, similar to a HDD or SSD. It then can be formatted and mounted with a file system. At the software level, applications have two ways of interacting with this file system. First, as shown on the left-hand side of Figure 2.1, applications can manage files through the regular file system interfaces, such as `read()` and `write()`. In this case, DCPMM is treated as a very fast block device and has the advantage that no modifications to the software are required. Like other storage devices, operations to files on DCPMM are done at a page granularity (usually 4 kB) and buffered in DRAM. The second way is shown on the right-hand side of Figure 2.1. If the file system is mounted with the special DAX flag, the application might map the file to its virtual memory address space with `mmap()`. The DAX flag indicates that the DCPMM is directly accessed and prevents intermediate buffering, such as the operating system page cache. As a consequence, load and store instructions done to the mapped address space are reflected directly to the DCPMM device. This has the advantages of avoiding expensive system calls, of reducing the transfer granularity from pages to cache lines (64 B), and of requiring only a single copy of the data, since it is not buffered in DRAM. It is worth noting that, while both DRAM and NVM share the same virtual memory address space, the application can distinguish their respective memory regions, and therefore control where to place data. Finally, while accessing persistent media through load and store instructions introduce challenges that require the software to be properly adapted, we consider this to be the only way to fully leverage NVM and achieve optimal performance. Therefore, in this thesis we only assume the scenario in which DCPMM is configured in *app direct mode* and files are mapped to the virtual memory address space of applications.

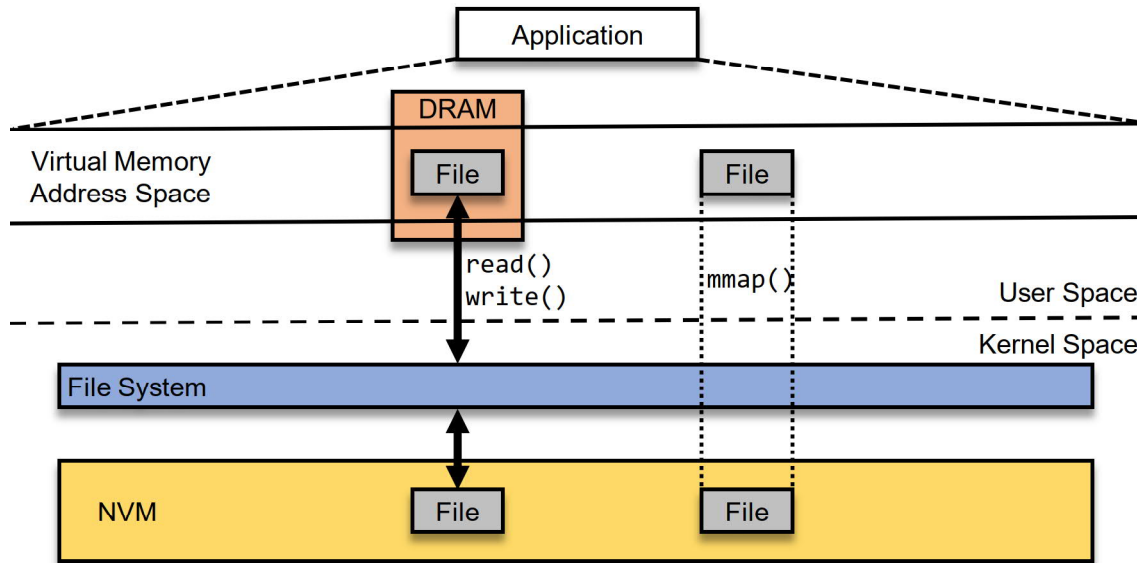


Figure 2.1: NVM Access.

2.1.3 Byte-addressability and Persistency

As previously mentioned, when configuring the DCPMM in the *app direct mode* and managing it through a file system with the *DAX* flag enabled, applications are able to directly map files in NVM to the virtual memory address space and access them through load and store operations. The *DAX* flag guarantees that the operating system or file system will not employ any intermediate buffering, thus ensuring that operations are always persisted in NVM. While the *DAX* flag removes the volatility from the access chain at the software level, there is still volatility to be considered at the hardware level, as the path from CPU registers to NVM is long and mostly volatile.

As shown in Figure 2.2, Intel CPUs employ store buffers and three levels of caches to hide the latency of memory accesses. Each physical core has its own L1 and L2 caches, while the L3 cache is shared between all the cores. Accessing NVM through load and store operations implies that data in NVM is transferred to the CPU caches, just like DRAM. This traffic happens in cache lines of 64 B. This leads to three problems. First, the CPU caches are volatile and therefore store operations are not guaranteed to be persistent after they were executed. Second, since most CPUs employ *out-of-order execution*, store operations might be reordered. Third, from the software point of view, cache lines are arbitrarily evicted and, therefore, persisted back to NVM. Traditionally the software has little control over the CPU caches, and, in such scenario, it is impossible to enforce store ordering and data durability, which are the main requirements to ensure consistency.

To aid software developers, Intel has introduced the hardware instructions **CLFLUSHOPT** and **CLWB**, which, when combined with **SFENCE**, can provide memory ordering and durability. While it is not possible to prevent a cache line from being evicted from the CPU cache, the **CLFLUSHOPT** and **CLWB** instructions eagerly force cache lines to be flushed and, therefore, persisted. While both instructions are used to flush cache lines, the only difference is that **CLFLUSHOPT** will also evict it from the CPU cache and **CLWB** will retain the cache line in the CPU cache. The **SFENCE** instruction is a memory barrier that serializes all pending

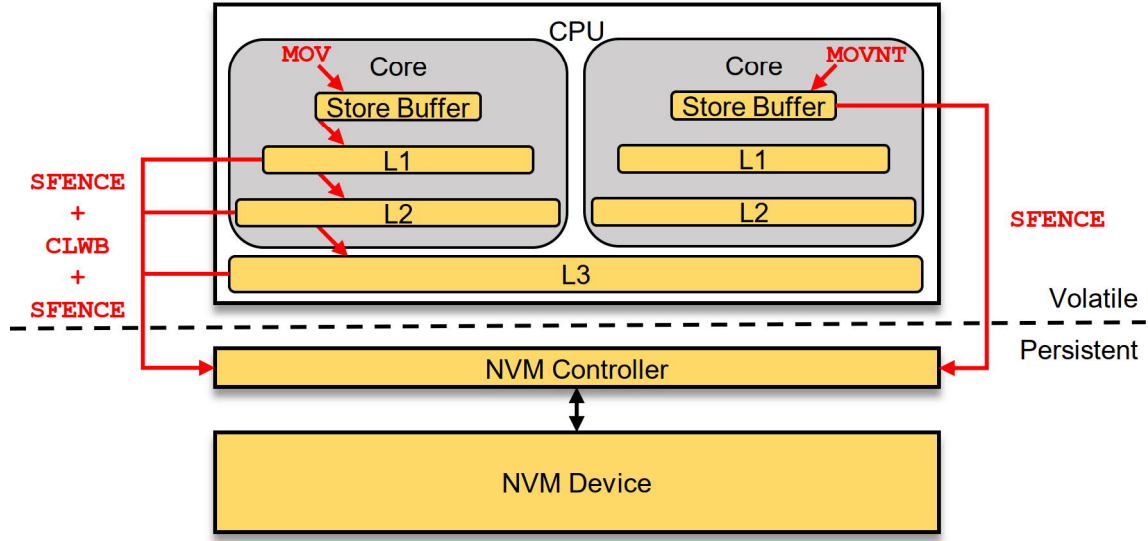


Figure 2.2: The volatility chain from the CPU to NVM.

stores, thus preventing undesired reordering. Therefore, as seen in the left-hand side of Figure 2.2, a typical pattern for persisting a store operation to NVM (represented by the MOV instruction) is to explicitly follow it by three instructions: SFENCE, CLFLUSHOPT or CLWB, and another SFENCE. The first SFENCE ensures that the stores completed, the CLFLUSHOPT or CLWB will trigger a cache line flush to persist the store, and the last SFENCE will ensure the previous CLFLUSHOPT or CLWB completed before resuming. In addition to these instructions, non-temporal stores (MOVNT) can also be used to bypass the cache by writing to a special buffer, which is evicted either when it is full, or when an SFENCE is issued. This is shown in the right-hand side of Figure 2.2. Non-temporal stores are useful when the CPU is simply writing to a new memory location without previously reading the data and loading its cache lines, such as when streaming data in a log-structured manner.

Finally, it is worth noting that, while cache line transfers between CPU caches and NVM happen in a granularity of 64 B and are atomic, current x86 CPUs only guarantee that 8 B stores are atomic, i.e., complete within a single CPU cycle⁴. While it is unlikely that a cache line will be evicted during a store operation larger than 8 B, algorithms and data structures should take that into consideration to avoid partial writes and guarantee correctness. This is usually achieved by atomically updating an 8 B variable to reflect larger stores, such as a pointer (in the case of *shadowing*) or the tail of a log. In the context of NVM, this 8 B unit is commonly referred to as **p-atomic**.

2.1.4 Performance

In order to give a good overview of the performance characteristics of DCPMM, we have measured its latency and bandwidth using *Intel Memory Latency Checker*(v3.9) [VKW⁺13]. The hardware used for the measurements is shown in Table 2.2. We first compare the latency of DCPMM to the latency of CPU caches and DRAM. The processor has 24 physical

⁴Not to be confused with atomic visibility, which can be achieved at larger sizes by the cache-coherency protocol when using hardware instructions such as compare-and-swap(CMPXCHG16B).

Processor	Intel Xeon Platinum 8260L CPU @2.40 GHz
Main Memory	96 GiB DDR4 @2666 MHz (6× 16 GiB modules)
NVM	Intel Optane DCPMM 1.5 TiB (6× 256 GiB modules)
Operating System	Linux 5.3.4-3

Table 2.2: Hardware used for measuring DCPMM performance characteristics.

Level	Sequential Access	Random Access
L1	1 ns	1 ns
L2	7 ns	7 ns
L3	20 ns	20 ns
DRAM	75 ns	85 ns
DCPMM	175 ns	300 ns

Table 2.3: Latency for accessing different levels of the memory hierarchy.

cores and 48 logical cores. The L1 caches are divided in *instruction cache* and *data cache*, each one having a capacity of 32 kB. Each L2 cache has a capacity of 1024 kB and the shared L3 cache has a capacity of 36 608 kB. The access latencies are shown in Table 2.3. The main observation is that DCPMM not only has higher latencies, but they differ significantly between sequential and random accesses. This is explained by buffering being employed internally in the DCPMM device, as reported by Intel [Int20d]. While traffic between CPU caches and DCPMM happens in a 64 B granularity, just like DRAM, internally the DCPMM employs buffering at a 256 B granularity. In other words, it is beneficial to collocate data on DCPMM in units of 256 B to explore spatial locality. It is also worth mentioning that the reported DCPMM latency refer to read operations. As previous studies have also noted [IYZ⁺19], precisely measuring the write latency of DCPMM is difficult. For supporting DCPMM, Intel CPUs rely on a feature called *asynchronous DRAM refresh* (ADR). The ADR triggers a hardware interrupt to the memory controller to flush write-protected data buffers in the case of a power failure. While stores that linger in the ADR domain are “safe”, currently there is no way to detect when they physically reach the DCPMM itself.

Figure 2.3 compares the bandwidth under two and six DCPMMs with a varying number of threads. The CPU supports six channels, each of which has two slots, one for DRAM and one for NVM. Therefore, the number of DCPMMs also indicates the number of memory channels. Similar to DRAM, adding more DCPMMs significantly increases the read bandwidth, while the impact for write bandwidth is relatively smaller. It is worth noting that in most cases using two threads is enough to saturate the DCPMM write bandwidth. Nevertheless, having as many DCPMMs as possible is not always better, as the optimal setup highly depends on the use-case. As an example, one might benefit from less DCPMMs in favor of more DRAM modules, since the amount of available memory slots is limited per CPU. In such case, this would imply trading the lower costs and increase in bandwidth of DCPMM for a more expensive setup and larger DRAM capacity, benefiting from its lower latency access. DCPMM exhibits peak sequential read and write bandwidth of 40 GB/s and 10 GB/s, respectively. These are respectively $\sim 3\times$ and $\sim 11\times$ times lower than those of DRAM. This gap widens even more for random read and write bandwidth to respectively $\sim 8\times$ (7.4 or 7.4 GB/s) and $\sim 14\times$ (5.3 GB/s) lower bandwidth than DRAM. Given the performance gap between DRAM and DCPMM, it becomes more important to leverage more memory channels (i.e., equip more DCPMMs) to reach the peak bandwidth.

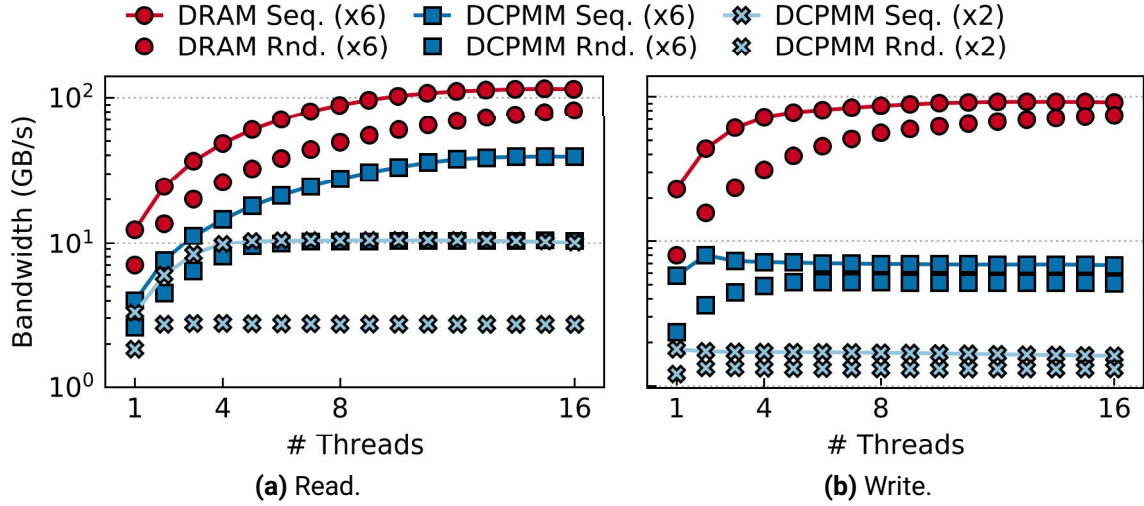


Figure 2.3: DRAM and Optane DCPMM bandwidth when a server is fully ($\times 6$ modules) and partially ($\times 2$ modules) populated.

2.2 RELATED WORK

The advances in the recent years raised the interest in NVM technologies both in academia and industry. Many interesting works proposed data structures, algorithms, and architectures in different contexts. While most of the contributions and the concepts explored can be used as building blocks for more complex systems, in this section, we limit to review the related work of more complete storage management systems in the context of NVM.

Storage Management in the NVRAM Era

Pelley et al. [PWGB13] explore the impact of NVM (referred to as NVRAM) in traditional database architecture. Rather than leveraging NVM in the context of the modern storage hierarchy comprised of DRAM, SSD, HDD, and even tape, the work assume that NVM will replace all existing persistent storage devices. Therefore, as a starting point, they use NVM as a drop-in replacement for disk and analyze how it impacts the database architecture. The first dimension considered refers to reading from NVM. Even if the performance of NVM is higher than that of other storage media, the authors assume that caching is still required due to its higher latency. Three alternatives are proposed: *Software Buffer*, *Hardware Buffer*, and *DRAM Replication*. The **Software Buffer** approach relies on caching hot pages in DRAM through a buffer manager, as in conventional databases. The **Hardware Buffer** alternative involves removing the buffer manager and rely solely on CPU caches to hide the higher latency of NVM. Finally, the **DRAM Replication** also relies on removing buffer management by having a complete replica of the database in DRAM to serve read requests. Like the authors, we consider that, while *Hardware Buffer* and *DRAM Replication* strategies can simplify the software stack by removing buffer management, they are impractical in most scenarios due to the limited capacity of CPU caches and to the large DRAM requirements, respectively. Therefore, having a portion of DRAM managed by a *Software Buffer* to cache hot pages is still the best strategy.

Since a *Software Buffer* in DRAM is used to read and modify pages, the second dimension considered by the authors refers to writing the pages back to NVM. The strategies considered are: *Traditional WAL/ARIES*, *In-Place Updates*, and *NVRAM Group Commit*. To enable **Traditional WAL/ARIES**, the authors introduce two operations, `persist_wal()` and `persistent_page()`, to enable atomic propagation of log records and pages to NVM, respectively. The authors note that, while it is a simple approach, the software overhead of logging and page flushing becomes the bottleneck if NVM is used as the main storage device. As an alternative, the **In-Place Update** strategy is proposed. It eliminates redo logging by employing a *force* strategy (i.e., flushing all updated pages at commit). Transactions employ private undo log records, which are persisted to NVM until the transaction commits, thus enabling the transaction to roll back after a failure. Finally, the authors also propose the **NVRAM Group Commit** strategy, which extends the **In-Place Update** by batching multiple transactions and applying their updates at once to a copy of the whole database. The authors consider **NVRAM Group Commit** to be the best strategy, as it amortizes the overhead associated with committing single transactions.

While we agree with the authors that *Software Buffer* is a good strategy, we make a few observations regarding writing to NVM. In the *Traditional WAL/ARIES* approach, the authors argue that logging and the asynchronous page flushing are the main source of overhead. We point out that, while the authors assume that shadow paging is used to atomically flush pages to NVM, this is not required, since the WAL suffices to guarantee the consistency of the database and recover pages in case of partial writes. The overhead of WAL can also be significantly reduced with better logging protocols, as discussed later in this section. Furthermore, the *In-Place Update* and *NVRAM Group Commit* are proposed to eliminate redo logging. However, both strategies employ shadow paging during commit time, which was previously pointed out as a main overhead in the context of asynchronous page flushing. Removing the redo log also implies lack of support for media recovery. Therefore, we consider that a well-implemented *Traditional WAL/ARIES* approach can reduce the mentioned overheads and perform better than the proposed alternatives.

Managing Non-Volatile Memory in Database Systems

Similar to the *Software Buffer* strategy discussed previously, van Renen et al. [vRLK⁺18] propose a buffer manager to leverage the benefits of NVM. The authors point that DRAM is still required to hide the higher latencies of NVM but that page-based DRAM caching leads to unnecessary memory traffic. Therefore, unlike Pelley et al. [PWGB13], they propose to leverage NVM in the context of a more complete storage hierarchy comprised of DRAM and SSD, and to reduce the traffic between DRAM and NVM from pages to smaller granularities, such as cache lines or “mini pages”. The authors also employ many state-of-the-art techniques, such as *pointer-swizzling* [GVK⁺14], and further discuss replacement strategies. The main advantages of the proposed buffer manager are its simplicity, which makes it easy to be integrated into existing systems, and its capacity to leverage other storage devices. Nevertheless, while the unnecessary traffic between DRAM and NVM is reduced, it is not completely eliminated, as NVM is never directly accessed by the CPU.

FOEDUS

The *FOEDUS* is a transactional system [Kim15] designed to leverage many-core servers and NVM. Like Pelley et al. [PWGB13], the author assumes a scenario in which NVM completely replaces other storage devices and coexists solely with DRAM. It proposes a B+Tree variation called *Master Tree* and relies on the concept of *dual pages*. Each page has two copies: a read-only state in NVM and a volatile state in DRAM. This technique resembles shadow paging, but it stores the address of both versions of a page in the incoming page pointers, rather than in a separate mapping data structure. *FOEDUS* employs a redo-only distributed logging scheme [TZK⁺13, ZTKL14]. A “*log gleaner*” process propagates the updates to the NVM pages by asynchronously replaying the log records. While *FOEDUS* employs many state-of-the-art techniques, the author assumes that NVM is accessed through the regular file system interface, thus not leveraging its byte-addressability and being the main drawback of the proposed architecture.

Write-Behind Logging

Motivated by the observation that write-ahead logging becomes the main bottleneck in many scenarios, Arulraj et al. propose an NVM-aware commit protocol called *write-behind logging* (WBL) [APP16]. They assume a database system consisting of only DRAM and NVM and multi-version concurrency control (MVCC) [LBD⁺11, KR79]. During transaction execution, changes are written to a DRAM copy of the database. At commit, dirty records are flushed individually to NVM, following a *force* strategy. Since MVCC is used, records are always written to a new location in NVM. Nevertheless, uncommitted updates might be propagated to NVM if a failure occurs during commit. To determine loser transactions, a group-commit interval is defined by two timestamps (c_p, c_d), such that all transactions with timestamp lower than c_p have successfully committed, and no transactions with timestamp higher than c_d have executed yet. These timestamps are written to a log after all the changes older than c_p have been propagated to NVM, hence “*write-behind*”. The recovery consists of reading the log to determine the (c_p, c_d) intervals that define the loser transactions. Following MVCC, records modified by loser transactions are then ignored by new transactions, later being removed by a garbage collection process.

The authors show benefits over write-ahead logging (WAL) in terms of memory consumption and reduced traffic to NVM. However, we note that WAL is a protocol for update-in-place strategies, which is not the case of MVCC. In other words, the multiple copies of a record stored in a MVCC system already resemble the before- and after-images stored in log records in WAL. Furthermore, while WAL can be eliminated in MVCC systems by using WBL, we point that WAL is often leveraged to enable other functionalities, such as remote replication, database auditing, and *partial transaction rollback*. Therefore, in addition to being limited to MVCC, WBL makes the support of these functionalities more cumbersome.

SAP HANA

SAP HANA [FML⁺12] is a modern database system that targets hybrid enterprise applications consisting of both transactional and analytical workloads. It follows an in-memory column-store architecture and exploits modern hardware, such as multiple cores, SIMD instructions, and large main memory and CPU caches. In order to store all database contents in memory, SAP HANA relies heavily on data compression. To enable a high compression rate, data has to be often coalesced and organized in efficient formats. Therefore, SAP HANA employs a *delta-main* approach: a small *delta* component ingests new records and updates and is periodically merged to a larger and read-only *main* component, which stores data in a compressed format. Durability is guaranteed through command logging [MWMS14] and transaction-oriented checkpoints [HR83]. SAP HANA can also leverage NVM to extend the capacity of the *main* component [ALR⁺17]. This is a natural fit, as the *main* component is read-only and, therefore, NVM can be accessed directly by the CPU just like DRAM. Not only DCPMMs are larger than DRAM modules, but they are also cheaper, leading to lower costs. While runtime performance is impacted by the higher NVM latency, the startup is significantly faster, as the *main* component of NVM is persisted across restarts, thus avoiding large load times from the storage device to memory.

Hyrise-NV

Hyrise-NV [SKD⁺16] is an NVM-aware version of the Hyrise in-memory database system [GKP⁺10]. Similar to SAP HANA, Hyrise is tailored to hybrid transaction and analytical workloads and employs the *delta-main* architecture. In addition to the *main* component, Hyrise-NV also persists other data structures, such as the proposed *nvBTree*, used as secondary indexes. Hyrise-NV relies on append-only strategies and multi-version storage to update data transactionally and guarantee consistency. Furthermore, since write-ahead logging is not employed, metadata of currently running transactions is persisted to enable undo in case of failure.

SOFORT

SOFORT [OBL⁺14, Ouk18] is a database system developed from scratch to leverage a hybrid DRAM-NVM environment. Similar to SAP HANA and Hyrise-NV, SOFORT is an in-memory column-store targeting both transactional and analytical workloads, also following the *delta-main* architecture. Differently than other systems that leverage traditional storage devices, SOFORT relies purely on DRAM and NVM and, therefore, is able to employ a *single-level* architecture. In other words, SOFORT only manages persistent objects in the virtual memory address space, rather than files in a file system. Primary data is stored in NVM and written/read directly by the CPU. Auxiliary data, such as secondary indexes, can be placed either in DRAM or in NVM, thus allowing the system to be configured either for high performance or for lower costs and faster restart.

SOFORT also supports ACID transactions through multi-version concurrency control (MVCC) [LBD⁺11, KR79]. Unlike other approaches that propose to eliminate logging but

still employ it in a smaller granularity, SOFORT is able to achieve that goal by employing a *force* strategy and storing the transaction table in a persistent data structure. Therefore, “*transaction objects*” are persisted and are used to discard operations tagged with the MVCC timestamps of transactions that were in-flight when the failure occurred.

The main limitation of SOFORT is the lack of support for other storage devices (HDD and SSD) in a more complete storage hierarchy. While the proposed architecture may have benefits in a future where NVM will replace other storage technologies, for the time being, modern storage technologies, such as SSDs, still have to be considered for most use-cases. Finally, as also noted by the authors, it is impractical to adapt existing systems to follow the same architecture, as it would require heavy software rewrite and architectural changes

Logging

Many of the works discussed so far make the observation that, when leveraging NVM as the main storage device in a database, logging (typically WAL) becomes the main bottleneck. Therefore, these works employ different flavors of *force* strategies with the goal to completely eliminate logging from the critical path of transactions [PWGB13, APP16, SKD⁺16, OBL⁺14, Ouk18]. Other works go in the different direction of leveraging NVM to reduce the overhead of logging, rather than eliminating it [CKKS89, AJ89, FHH⁺11, GXH⁺11, HSQ14, WJ14]. We focus on two of these works.

Huang et al. [HSQ14] make the observation that NVM is more expensive than other storage devices (HDD and SSD) and, therefore, restricting the use of NVM to improve logging performance yields a higher ratio of transactions per dollar than simply replacing all storage with NVM. The authors propose NV-Logging to improve the traditional ARIES logging protocol [MHL⁺92]. The main idea is to reduce the software overhead associated with centralized log buffers by introducing decentralized transaction-private log buffers. During transaction processing, log records are created in private DRAM buffers. Upon commit, the transaction flushes the log records to arbitrary locations in NVM in the form of “*log objects*”. To enforce the global LSN order, pointers to these “*log objects*” are then appended to a “*log entry index*” implemented as a circular buffer. While the “*log entry index*” still acts as a contention point, its critical-section is much shorter, since only fixed-length pointers have to be appended, rather than arbitrarily large log records.

Unlike NV-Logging, Wang et al. [WJ14] propose fully-distributed logging for multi-core multi-socket environments. They note that, while distributed logging reduces the overhead of logging by spreading log records across multiple physical logs, they are prohibitive due to the overhead introduced by dependency tracking and additional I/O required to guarantee global order of log records. Fortunately, NVM can be leveraged to significantly reduce these I/O costs and enable distributed logging. The authors investigate both page-level and transaction-level log space partitioning and employ a *global sequence number* based on a logical clock [Lam78] to uniquely identify log records. The empirical evaluation shows that transaction-level partitioning is more favorable, as it avoids cross-socket communications.

We agree that leveraging NVM to improve the logging infrastructure will be mandatory for database systems, and that completely replacing all storage by NVM is currently unfeasible

in most scenarios due to its higher cost when compared to SSD and HDD. We also note that, this approach not only is more cost effective, but it also limits the changes in the code base to the log manager, thus not requiring drastic changes in the whole database architecture. Nevertheless, we consider that there are other opportunities to leverage NVM in database systems, such as NVM-aware buffer management policies.

NOVA

While the works previously discussed were mostly in the context of database systems, NOVA [XS16] explores NVM in the context of file systems. The goal of NOVA is to enable efficient access to NVM through the abstraction of regular file system interfaces. The main motivation is to enable applications to leverage NVM without any changes, rather than specifically tailoring them to manage NVM through the virtual memory space. NOVA is a file system in the user-space that follows a hybrid DRAM-NVM architecture and relies on log-structured writes to provide strong consistency guarantees. The authors claim that it is able to achieve 22% to $216\times$ throughput improvement compared to state-of-the-art file systems, and $3.1\times$ to $13.5\times$ improvement compared to file systems that provide equally strong data consistency guarantees. We consider that, while specialized applications such as database systems can benefit more from directly managing NVM, other applications that do not require high performance can easily leverage some of the performance benefits of NVM through file systems like NOVA.

2.3 CASE STUDY: PERSISTENT TREE STRUCTURES

Perhaps one of the most prolific categories of works in the context of NVM is comprised of persistent data structures, such as persistent hash tables [LHWL20, DHK⁺15, NCC⁺19, NIK⁺17, SDUP15, ZH18, ZHW18] and persistent trees [ALML18, CGN11, CJ15, HKWN18, LLS⁺17, OLN⁺16, VTRC11, YWC⁺15, LXCW19]. Works in this category take as a starting point volatile memory data structures common in existing programming languages (such as `std::map` in C++). NVM is leveraged to make these data structures persistent, but still maintaining a similar behavior to their volatile counterparts. Therefore, persistent data structures are simpler when compared to the architecture of storage management system that employs concepts such as buffer management, ACID transactions, and garbage collection. While they may be inappropriate as a standalone storage manager, they are still relevant as building blocks for more complex systems. As an example, one could use a persistent hash table to implement a persistent version of a lock manager in a relational database system. The techniques employed by these persistent data structures can be generalized and applied in other scenarios. Therefore, we consider that looking at their design decisions can aid the reader in better understanding the challenges of NVM.

	Architecture	Node structure	Concurrency
wBTree	NVM-only	Unsorted	Single-threaded
NV-Tree	NVM-only (optionally hybrid)	Unsorted leaf nodes; inconsistent inner nodes	Locking
BzTree	NVM-only	Partially unsorted leaf; sorted inner nodes	Lock-free (PMwCAS [WLL18])
FPTree	DRAM (inner nodes) + NVM (leaf nodes)	Unsorted leaf nodes	Selective (HTM + locking)

Table 2.4: Comparison of key design decisions of the persistent tree structures analyzed.

2.3.1 Persistent Trees

We limit the scope of persistent data structures to persistent tree variants, since these are the most commonly used ones in database systems. We pick four candidates considered to cover a good portion of the design space: **wBTree** [CJ15], **NV-Tree** [YWC⁺15], **BzTree** [ALML18], and **FPTree** [OLN⁺16]. In the following, we give an overview of each one of these persistent tree data structures. Table 2.4 summarizes their key design decisions.

Write-Atomic B+Tree (wBTree)

wBTree [CJ15] is a persistent, single-threaded B+Tree that achieves high performance by reducing cache line flushes and writes to NVM. Traditional B+Tree nodes are sorted for faster binary search. However, as Figure 2.4a shows, keeping a node sorted requires a shift of data to make place for the new key, which might leave the node in an inconsistent state upon crashes, and incurs more (expensive) NVM writes. wBTree solves this problem with unsorted nodes proposed in prior work [CGN11]. Figure 2.4b illustrates the idea. A bitmap is used to indicate if each slot contains a valid (green box in the figure) record or not (red box). The new record is inserted into a free slot (out-of-place), and the bitmap is atomically modified using 8 B writes to set the validity of the inserted record. Using unsorted nodes reduces the number of NVM writes and eases implementation, but requires linear search for lookups, which might be more expensive than a binary search. Nevertheless, as we will see later, the use of unsorted nodes is a common and effective design in NVM trees.

To enable binary search (thus reducing NVM accesses), wBTree uses an indirection slot array in each node, as shown in Figure 2.4c. Each entry of the array records the index position of the corresponding key in sorted order, i.e., the n -th array element will “point” to the n -th smallest key by recording the key’s index into the key-value slots. In the example, after inserting key 5, in step 3 the bitmap needs to be modified so that the third element records the position of key 7, which is stored as the second element (index 1) in the key-value storage area. One bit (left-most box in the figure) in the bitmap is reserved to indicate the validity of the array. wBTree relies on the atomic update of the bitmap to achieve consistency, and on logging for more complex operations such as node splits. After inserting the record out-of-place in a free slot, the indirection slot array is flagged as invalid and updated, as shown in step 3 of Figure 2.4c. In case of a failure, the indirection

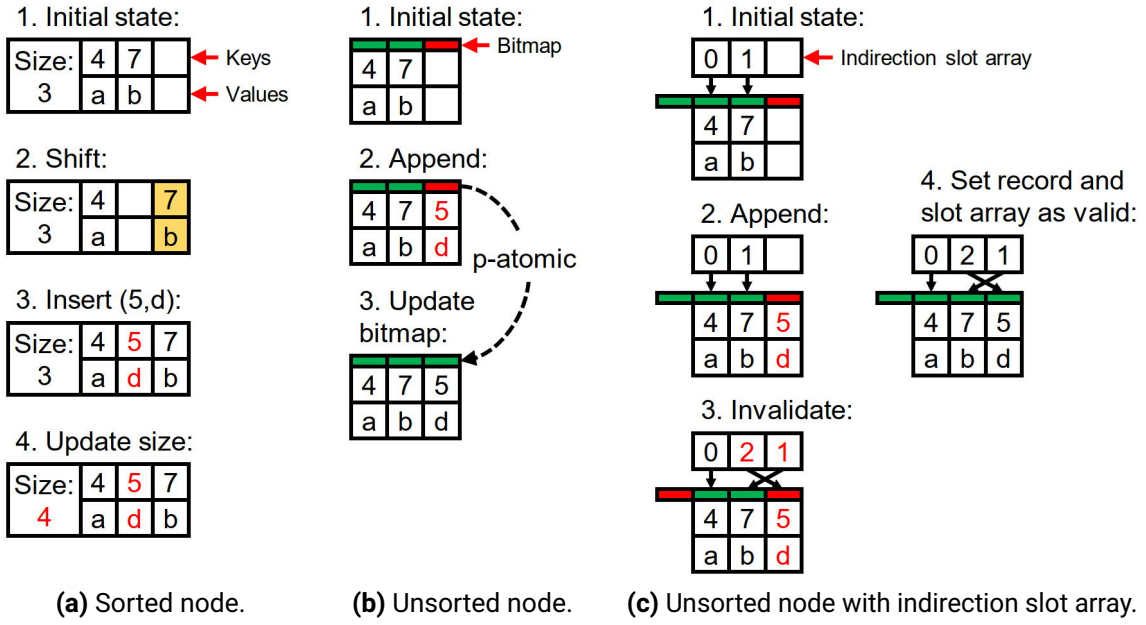


Figure 2.4: Comparison of inserting key 5 with value d. Unsorted node reduces writes but requires linear search for lookup, which can be avoided by using an indirection slot array.

slot array will be detected as invalid and reconstructed upon recovery. Finally, the bitmap is atomically updated to set both the indirection slot array and the new record as valid. This last step imposes that the bitmap is not larger than 8 B, the p-atomic unit. When the indirection slot array is smaller than 8 B, the bitmap could be removed as the indirection slot array can be atomically updated and serve as the validity flag.

NV-Tree

NV-Tree [YWC⁺15] proposes the concept of *selective consistency*, which, as shown in Figure 2.5, enforces the consistency of leaf nodes and relaxes that of inner nodes. This design simplifies implementation and reduces consistency costs by avoiding cache line flushes. Inner nodes, however, have to be rebuilt upon restart, since the copy in NVM might be inconsistent and unable to guide lookups correctly. We note that inner nodes could also be placed in DRAM since their consistency is not enforced. Similar to the wBTree, NV-Tree also uses unsorted leaf nodes with an append-only strategy to achieve fail-atomicity. Figure 2.6 shows an example of an insertion in an NV-Tree leaf node. The record is directly appended with a positive flag (or a negative flag in case of a deletion) regardless of whether the key exists or not. Then, the leaf counter is atomically incremented to reflect the insertion. To lookup a key, the leaf node is scanned backwards to find the latest version of the key: if its flag is positive, then the key exists and is visible; otherwise, the key has been deleted. The inner nodes are stored contiguously to abstract away pointers and improve cache efficiency. However, this implies the need for costly rebuilds when a parent-to-leaf node needs to be split. To avoid frequent rebuilds, inner nodes are rebuilt in a sparse way, which may lead to high memory footprint. As inner nodes are immutable (except parent-to-leaf nodes) once they are built, threads can access them without locking and only need to take locks at the leaf and their parents level when traversing the tree.

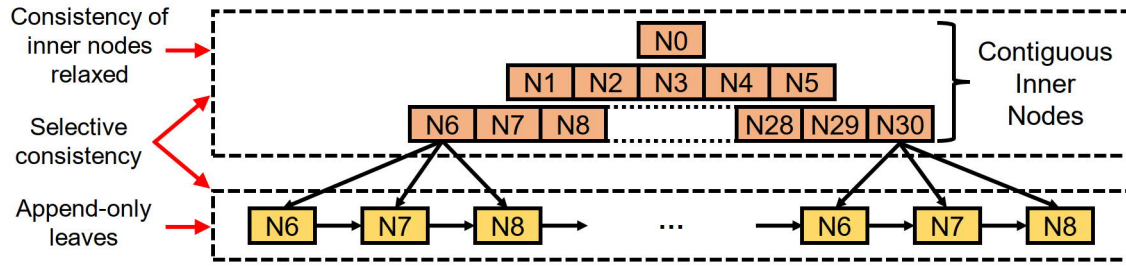


Figure 2.5: NV-Tree overview.

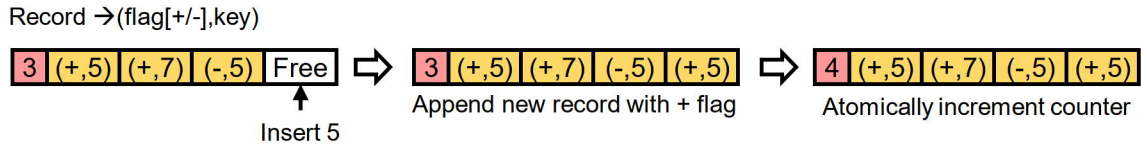


Figure 2.6: Insertion of a record in an NV-Tree node.

BzTree

BzTree [ALML18] is a lock-free B+Tree for NVM that uses persistent multi-word compare-and-swap (PMwCAS) [WLL18] to handle concurrency and ease implementation. PMwCAS is a general-purpose primitive that allows atomically changing multiple arbitrary 8 B NVM words in a lock-free manner with crash consistency. To achieve this, PMwCAS uses a two-phase approach. In Phase 1, it uses a descriptor d to collect the “expected” and “new” values for each target word, persist the descriptor, and atomically installs (using single-word CAS) a pointer to the descriptor on each word. If Phase 1 succeeded, Phase 2 will install the new values; otherwise the operation is aborted with all changes rolled back.

BzTree uses PMwCAS for insert, delete, search, scan, and structural modification operations which may need to change multiple NVM words. Because of the use of PMwCAS, while being lock-free, BzTree implementation is easier to understand than typical lock-free code. PMwCAS ensures that any multi-word changes are done atomically and recovery is transparent to BzTree, removing the need for customized logic for logging and recovery.

As Figure 2.7a shows, BzTree stores both inner and leaf nodes in NVM. Inner nodes are immutable (copy-on-write) except for updates to existing child pointers; leaf nodes can accommodate inserts and updates. Figure 2.7b shows the situation in which a node L splits and a new pointer must be inserted in the parent node P . Inserting to a parent node causes it to be replaced with a new one that contains the new key (P'). Then, an update in the grandparent node G is conducted to point to the new parent node P' . Splits can propagate up to the root and grow the tree. Records in inner nodes are always sorted, while records in leaf nodes are not. Initially, records are inserted to the free space serially. Periodically leaf nodes get consolidated (sorted) and subsequent inserts may continue to insert into the free space serially. After searching the sorted area (using binary search), the tree must linearly search the unsorted area to get a correct result. The design rationale is that inner nodes are not updated as often as leaf nodes and should be search-optimized; leaf nodes, however, need to be write-optimized.

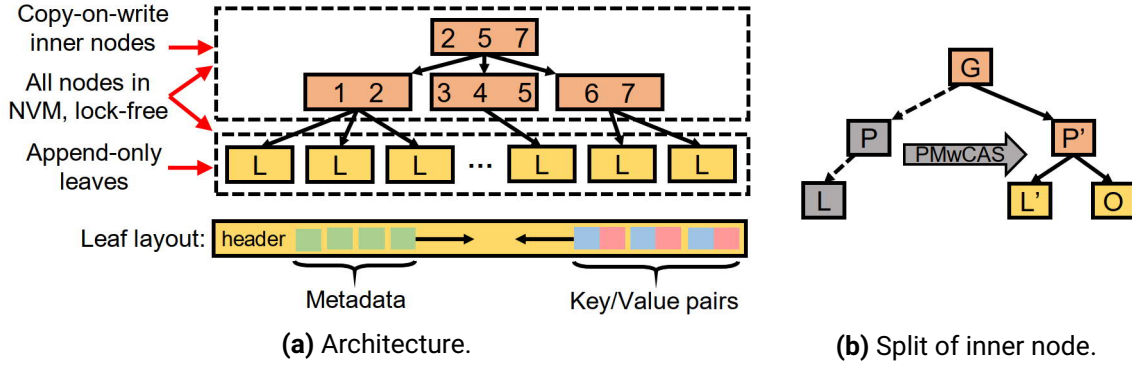


Figure 2.7: BzTree overview.

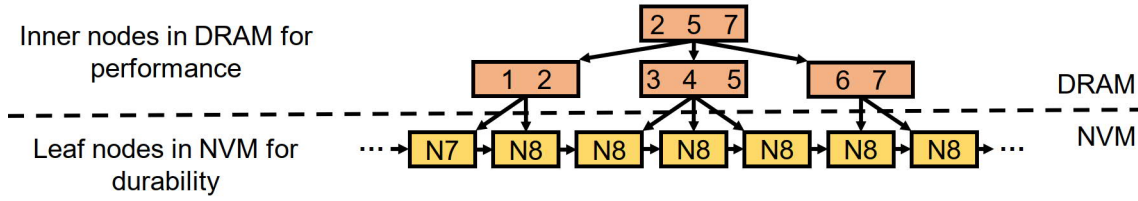


Figure 2.8: FPTree overview.

Fingerprinting Persistent Tree (FPTree)

Unlike the other trees being evaluated, FPTree [OLN⁺16] uses both DRAM and NVM to achieve near-DRAM performance. As Figure 2.8 shows, it stores inner nodes in DRAM, and leaf nodes in NVM. This way, FPTree accelerates lookup performance while maintaining persistence of primary data (leaf nodes), as only leaf accesses are more expensive during a tree traversal compared to a fully transient counterpart. The rationale behind is that while losing leaf nodes leads to an irreversible loss of data, inner nodes can always be rebuilt from leaf nodes. Since the inner nodes must be rebuilt upon recovery, FPTree trades recovery time for higher runtime performance.

FPTree uses fingerprints to accelerate search. They are one-byte hashes of in-leaf keys, placed contiguously in the first cache line of the leaf node. FPTree also uses unsorted leaf nodes with in-leaf bitmaps [CGN11], such that a search iterates linearly over all valid keys in a leaf. A search will scan the fingerprints first, limiting the number of in-leaf key probes to one on average, which significantly improves performance. FPTree applies different concurrency control methods for the tree's transient and persistent parts. It uses *hardware transactional-memory* (HTM) and node-grained latches for inner and leaf nodes, respectively. Such *selective concurrency* design solves the apparent incompatibility of HTM and persistence primitives required by NVM such as cache line flushing instructions which always cause HTM transactions to abort directly.

2.3.2 Evaluation

To complement this case study, we look at the impact of different design decisions on the final performance. It is worth noting that all these data structures were proposed before

NVM was available, and, therefore, they were originally evaluated on different emulation platforms. While a qualitative comparison was possible, a quantitative comparison was impractical. Therefore, as our first contribution, we evaluate the tree structures discussed on real NVM hardware (DCPMM). We consider that this performance evaluation can serve to lay down the performance expectations of DCPMM. Furthermore, since the persistent trees discussed previously were proposed by several authors, they made different assumptions about NVM and its programming model, such as how persistent memory is allocated. As our second contribution and to enable a fair comparison, we have re-implemented all of them according to their original design, while assuming the programming model for memory allocation defined by the PMDK library [Int14]. The environment used for the benchmarks is the same as previously shown in Table 2.2. Finally, as our third contribution, we devised a benchmark framework, **PiBench**, targeted at the specific use-case of persistent data structures. As a contribution to the community and to enable future work to achieve a fair comparison and reproducible results, we made PiBench available both as an open-source project⁵ and as a web application⁶. Further details are shown in Appendix A.

Index Implementations

We highlight important details for implementing the evaluated trees, especially changes we made either to make them compatible with PMDK’s programming model such that they can be performed on real NVM, or due to necessary details not covered by the original works.

wBTree. wBTree originally uses undo-redo logs for failure atomicity [CJ15]. We improved it with more efficient micro-logs used by FPTree [OLN⁺16] and implemented it using the same code template as FPTree’s to reduce the impact of different implementations. We also changed wBTree to use PMDK persistent pointers instead of volatile pointers.

NV-Tree. The original paper [YWC⁺15] did not cover concurrency, so we implemented latch coupling. We changed NV-Tree to use PMDK persistent pointers and align records in leaf nodes to 8 B boundaries; for 8 B keys and values, the size of a record is 24 B with the validity flag. This is 7 B more than necessary, but gives better performance. Since the consistency of inner nodes is not enforced, we placed them in DRAM to improve performance.

BzTree. Splits in BzTree may propagate to upper levels, replacing all the nodes along the path (copy-on-write inner nodes). We prepared all the nodes on the split path and issued a final PMwCAS at the highest level to atomically swap in the new nodes. For this to work, we increased the size of PMwCAS descriptor size from 4 to 12 to accommodate enough memory word changes and new allocations.⁷

FPTree. The original paper [OLN⁺16] proposed two versions: a single-threaded version and a concurrent version. We focus on the concurrent version since we are most interested in multi-threaded experiments. However, we note that optimizations in the single-threaded version, such as allocating leaf nodes in groups, could be applied to all trees.

⁵<https://github.com/sfu-dis/pibench>

⁶<https://pibench.org/>

⁷These strategies were not presented in the original work [ALML18], but has been confirmed by one of the original authors in private communications.

Workloads

We evaluate the indexes with individual operations (lookup, insert, update, delete, scan). All experiments are run under a uniform key distribution. Scans are performed by selecting a random initial key and then reading the following 100 records in ascending sorted order. Each run starts with a new tree loaded with 100 million records with 8 B keys and 8 B values. We then measure and report the tree performance during the run phase, in which 100 million operations are executed by a specified number of threads. The numbers reported here refer only to the run phase, excluding the load phase. We use the list of operations completed in every time window (100 ms) of a single run to calculate the average throughput (depicted as the bars and points) as well as the standard deviation (depicted as the error bars).

Single-threaded Throughput

Figure 2.9 shows the single-threaded performance under workloads consisting of a single type of operation. We analyze and discuss the performance of each operation individually.

Lookup. The first important observation is that the persistent trees that place inner nodes in DRAM (FPTree and NV-Tree) have higher throughput due to the lower latency of DRAM. FPTree's fingerprints further reduce cache line accesses in leaf nodes to two in most cases: one for the fingerprints and bitmap, the other for the potentially matched record. This contrasts with NV-Tree which uses append-only leaves and requires scanning on average half of the leaf entries to determine if a record exists and is valid. BzTree employs a hybrid of sorted and unsorted leaf node format, so it needs to search the unsorted area linearly if the key is not found in the sorted area. It is worth noting that the performance benefits of lookup operations are transported to the other operations as well, as they must perform a lookup prior to additional work.

For insert, update, and delete, we note that all trees enforce the consistency and durability of single operations using out-of-place writes (possibly within a node) and atomically flipping a validity bit to "commit" the operation (for BzTree, this is delegated to PMwCAS). Therefore, these operations must always force the changes to NVM using `CLWB`, making it impossible for the CPU caches to amortize the high write latency of NVM. This explains the lower throughput and the increased standard deviation of these operations when compared with their lookup counterparts.

Insert. We observe insert performance is directly affected by (1) the amount of flushes per insert, (2) the needed maintenance work per insert, and (3) the overhead of node splits. Table 2.5 summarizes the amount of flushes needed by each operation. For all trees, each insert entails at least one flush for the record being inserted. FPTree and wBTree keep an 8 B bitmap per node to indicate which records are valid and enable the slot of invalid records to be reused. FPTree also requires flushing the fingerprints, leading to a total of three flushes per insert. In addition to the bitmap, wBTree keeps a slotted array per node to keep the order of records and a single validity bit to indicate the validity of this slotted array. Therefore, three additional flushes are required by the wBTree (slotted array, validity bit, validity bitmap), to a total of four flushes per insert. NV-Tree requires one additional flush to update the size of the node, to a total of two flushes. BzTree uses

Tree/Operation	Insert	Update	Delete
FPTree	3	3	1
NV-Tree	2	2	2
wBTree	4	3	1
BzTree	15	10	7

Table 2.5: Number of cache line flushes per operation.

two double-word PMwCAS operations per insert to reserve space in the leaf node and make the new insertion visible to other threads, respectively. Each PMwCAS incurs at least three flushes [WLL18]. In total, BzTree incurs 15 flushes per insert. If the current PMwCAS conflicts with another on-going PMwCAS, it might incur more flushes as it helps finish the other operation first. We attribute BzTree’s low insert performance mainly to the high number of flushes. Finally, in BzTree, FPTree and wBTree, a node split might propagate all the way up to the root level. However, for NV-Tree the inner nodes must always be completely rebuilt whenever a split happens in the parent-to-leaf level. When splitting a leaf node, two new nodes are allocated to split the records of the node that became full, causing the higher amount of NVM writes. This operation becomes expensive in comparison to other trees, which has also an impact in the throughput standard deviation.

Update. As opposed to inserts, an update only operates on an existing key. Overall, the standard deviation for updates is lower than that of inserts, due to the absence of allocations and splits. NV-Tree performs updates slower than inserts, as it handles updates as a deletion followed by an insertion. wBTree updates are faster than inserts since each update requires one fewer flush (3 vs. 4 in Table 2.5), as record order in the node does not change (the key is not updated). Thus, the slotted array can be updated atomically without flushing its validity bit, as only the offset of the updated record changes, while the others remain the same. BzTree’s update is faster than its insert operation, due to the absence of allocation and splits, but it still needs many flushes, leading to lower throughput.

Delete. The throughput of delete operations follows a similar trend to those of lookups. The reason is that deletion for FPTree and wBTree is basically a lookup followed by flushing the validity bitmap to invalidate the record deleted. There is no deallocation or merging of nodes implemented, as data structures are more likely to grow rather than shrink. This is also the approach taken by implementations of the C++ Standard Template Library. In contrast to FPTree and wBTree, NV-Tree requires two flushes per deletion, one for a tombstone and one for the node size. For BzTree, the process is similar, but it uses a PMwCAS to mark records invisible which requires multiple flushes, leading to lower performance.

Scan. Range scans start at a random initial key and read the following 100 records. wBTree is the only one that directly returns records in sorted order using its indirection slotted arrays. All the other trees must perform an additional sorting and filtering step to return the requested records. We can conclude that reading less from NVM (e.g., FPTree) does not compensate the overhead of sorting and filtering.

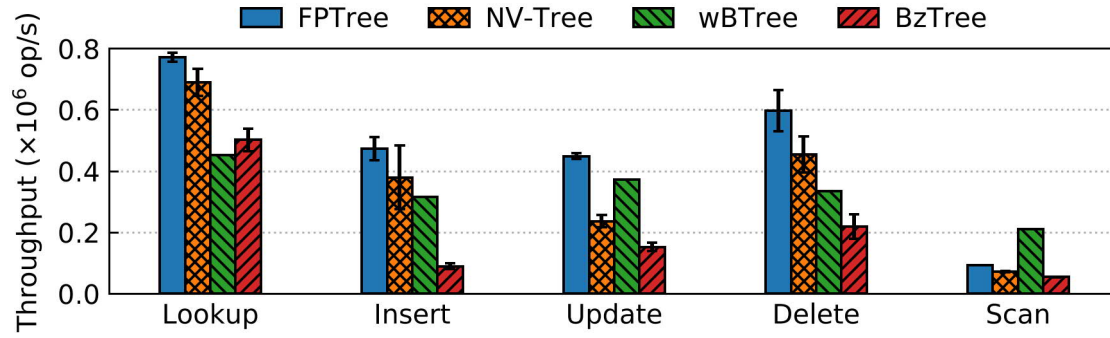


Figure 2.9: Single-thread throughput under uniform distribution. Placing inner nodes in DRAM helps much in traversal performance for FPTree and NV-Tree; wBTree performs the best in range scan as it does not require an extra sorting step.

Multi-threaded Throughput

We now evaluate the performance under multiple threads. We include wBTree’s single-thread performance for reference as it does not support concurrency. Since we dedicate one thread to collect the statistics, we scale the number of worker threads until 23 out of the 24 physical cores available in the CPU. Furthermore, we experiment with 32 and 47 threads to show the behavior of the trees under hyperthreading.

Figure 2.10 depicts the throughput of individual operations under multiple threads and uniform distribution. It shows a similar trend to the single-threaded experiments. All the evaluated trees scale as expected for lookup, insert, update, delete and scan operations using 1–23 threads (no hyperthreading). With hyperthreading (shaded areas in Figure 2.10), all trees maintain or slightly improve the throughput compared to using 23 threads. In particular, FPTree is able to leverage hyperthreading significantly better than other trees in lookup operation.

Figure 2.11 shows the throughput of individual operations under the skewed distribution (skew factor 0.2). Since a skewed workload accesses a small subset of keys multiple times, only the first insert/delete for a given key would succeed and all the subsequent insert/delete operations for the same key would simply be a lookup. Therefore, we omit these operations under skewed workloads. The results here showed similar pattern to the ones with the uniform workload: all trees exhibit higher throughput and largely scale under all operations, except BzTree and FPTree’s update operation, which respectively scales up to 8 and 16 threads and performs worse as we add more threads. There are two main reasons for BzTree’s behavior. First, because of the use of PMwCAS, a memory word may store a pointer or actual value. Each NVM read is instrumented to check the type of the word value, adding additional overhead. Second (and more importantly), the update operation employs an optimistic approach that retries a PMwCAS until success; it is well known that optimistic approaches are vulnerable to high contention. FPTree does not scale beyond 16 threads for a similar reason: it uses HTM (Intel TSX, which is an optimistic approach) for traversing the inner nodes and acquiring leaf latches. A skewed workload will incur more conflicts at the leaf level, hence more HTM aborts and lower throughput.

For lookup operations, as we vary the skew factor from 0.1 to 0.5 in Figure 2.12a, we see the throughput dropping for FPTree and BzTree, as lower contention (e.g., skew factor 0.5)

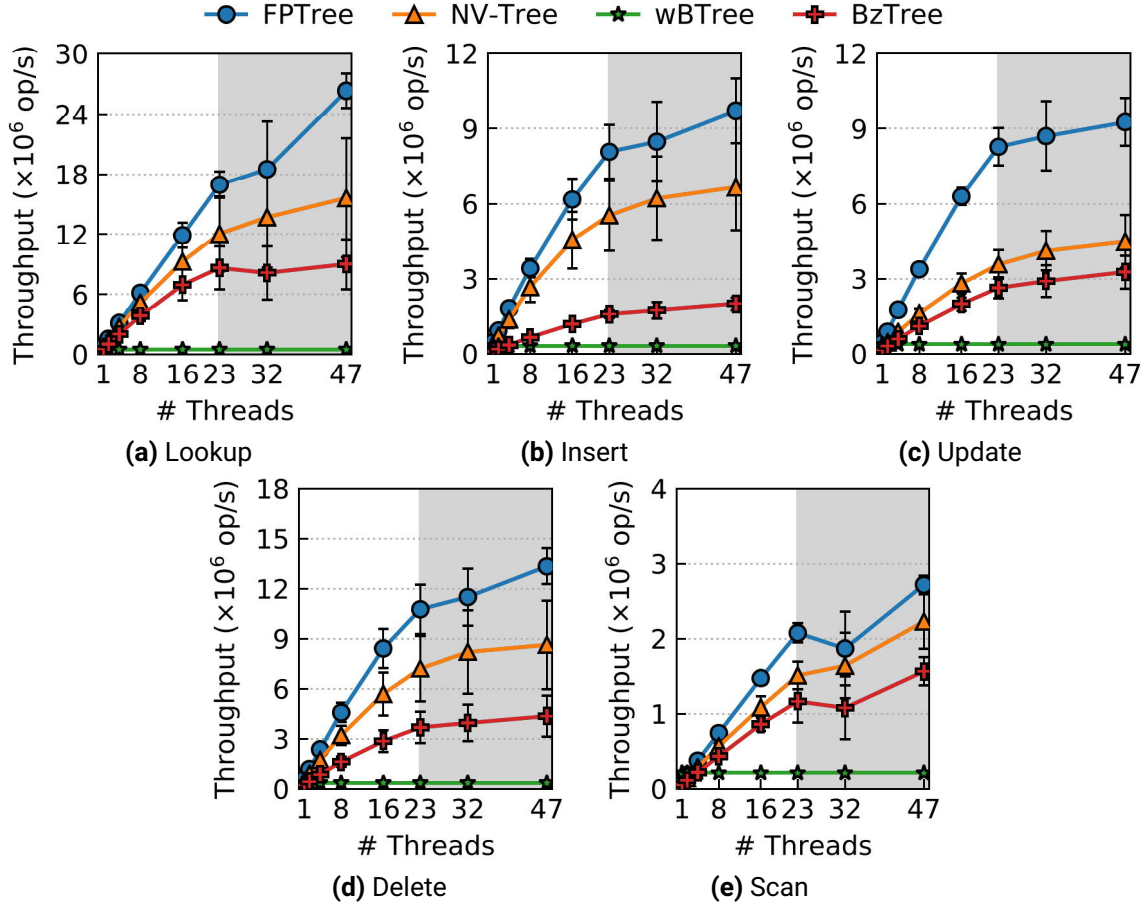


Figure 2.10: Throughput under uniform distribution. FPTree and NV-Tree leverage DRAM and perform generally better than pure NVM trees (BzTree and wBTree). All the trees maintain their throughput with hyper-threading (beyond 23 threads). wBTree’s single-thread throughput is shown for reference as it does not support concurrency.

leads to accesses to more keys and therefore more cache misses and NVM accesses. NV-Tree does not show obvious changes when we ease the contention. We attribute this behavior to the fact that it needs to acquire node locks even for read-only workloads, causing extra inter-core communications and traffic on the memory bus which is often unscalable for read-only workloads on multiple cores [TZK⁺13, WK16]. Update operations exhibit a different trend in Figure 2.12b, as we ease the contention, the throughput increases, although lower contention leads to larger NVM footprint in general. These results highlight two factors that affect performance under skewed workloads: (1) the amount of NVM accesses and (2) contention level. Both factors impact performance, and as we add more concurrent threads, contention takes over to become the major factor, contrasting with the single-threaded case where NVM footprint is the dominating factor.

Impact of Programming Model

Persistent data structures face challenges in handling persistence, recovery and concurrency, which can be resolved using a sound programming model enforced by NVM

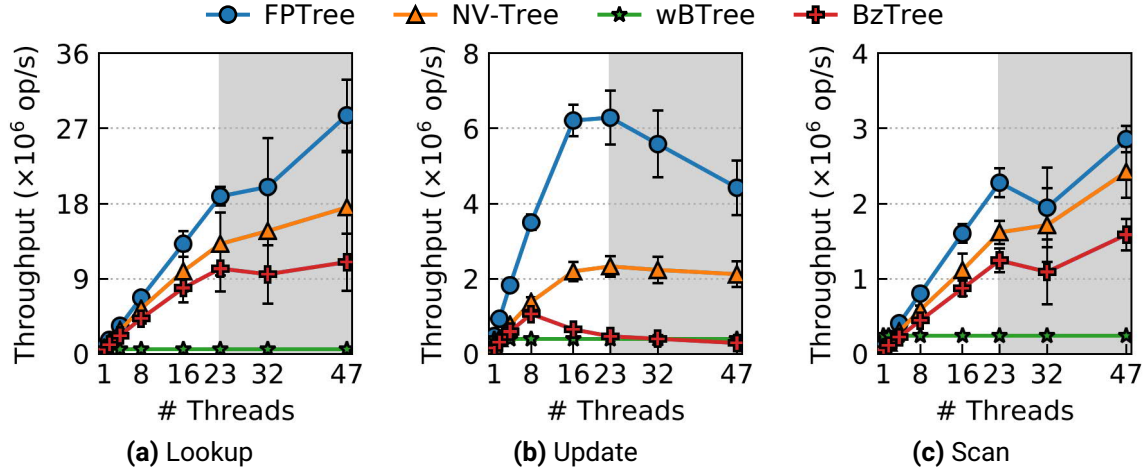


Figure 2.11: Throughput under the skewed distribution (skew factor 0.2). FPTree and BzTree do not scale for updates due to their use of optimistic concurrency control.

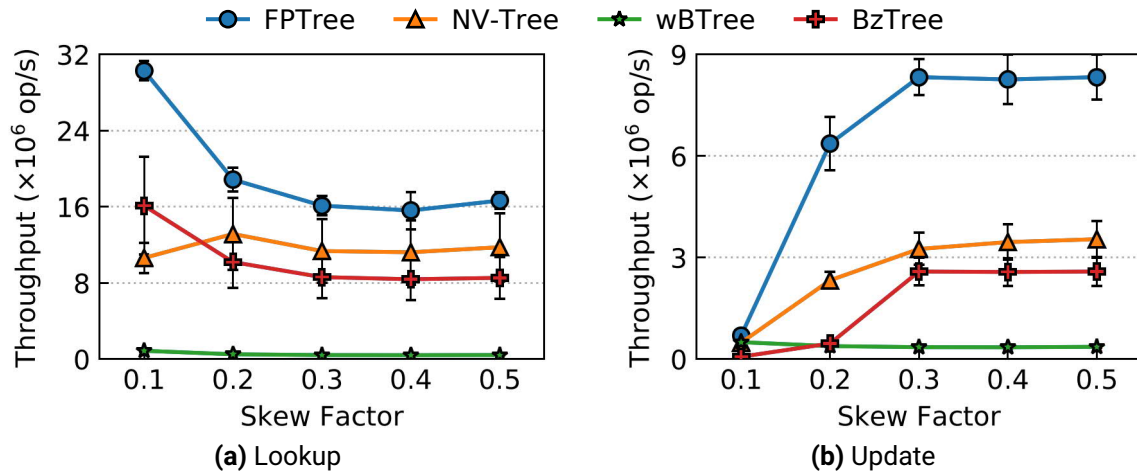


Figure 2.12: Throughput under varying skew factors with 23 threads. Higher skew factor means lower contention.

programming libraries, such as PMDK. Specifically, this boils down to the use of persistent pointers, memory alignment, and an NVM-aware allocator, which, as we show next, incur space amplification and performance overheads.

Space amplification. Similar to in-memory data structures [ZAP⁺16], persistent data structures may occupy a significant amount of memory (NVM and/or DRAM), due to various design decisions to optimize performance (e.g., alignment) and conform to the required NVM programming paradigm, in particular the use of 16 B persistent pointers [Int14]. We quantify this effect in Figure 2.13 by plotting the amount of memory consumed by each tree. We insert 100 million records of 8 B keys and 8 B values; this corresponds to ~ 1.5 GB of raw data. Any consumption beyond this amount would be the metadata, extra alignment or other allocations (e.g., during a split) needed by the tree. We use statistics from jemalloc for DRAM (`stats.allocated`), and the `pmempool` tool for NVM. The NVM consumption is precise, and DRAM consumption is an upper bound of the real consumption, as jemalloc also records other allocations made by our benchmark framework.

As shown in Figure 2.13, all the trees in fact use more than 50% of the space needed for the raw data; NV-Tree/BzTree use respectively $\sim 2\times/10\times$ the raw data size. This is partially due to the relatively small key/value sizes used (8 B) and the alignment requirement (typically 8 B) in all the trees. Although both FPTree and NV-Tree place inner nodes in DRAM, NV-Tree requires more DRAM because inner nodes are rebuilt sparsely to amortize rebuild costs. This means that it will have a different DRAM-NVM ratio depending on the fill ratio of its inner nodes. BzTree and wBTree consume a negligible amount of DRAM as they are pure NVM-based. BzTree’s memory consumption is cumulative (of all nodes ever created) and the highest, due to its use of copy-on-write for inner nodes. We note, however, that this is the worst case for BzTree, and in realistic workloads with fewer inserts, inner nodes will not change as often, which should result in lower NVM consumption.

NVM allocation overhead. Compared to their DRAM counterparts, persistent allocators need to issue cache-line flush instructions, handle recovery and run on slower NVM. To understand their behavior, we run an experiment using jemalloc on DRAM and PMDK allocator (which is based on jemalloc) on DRAM and NVM. Each thread issues 1024 allocation requests, each of which allocates 1 kB of memory. Figure 2.14 shows the time needed to finish the test. As we increase the number of threads, no allocator scales, and due to the extra complexity associated with PMDK allocator (e.g., the use of cache-line flush instructions and fences), PMDK allocator is $2.9\text{--}4.4\times$ slower than jemalloc on DRAM. On NVM, the PMDK allocator can be up to $\sim 8\times$ slower than itself running on DRAM. These results signify the high cost of NVM allocations and indicate that persistent data structures should carefully handle their interactions with NVM allocators. The insert operation interacts with allocators the most, and we observed non-trivial allocation overhead in all the evaluated trees. In particular, we found that BzTree spends more than $\sim 41\%$ of CPU cycles on NVM allocation in insert operations. Compared to other trees, its use of copy-on-write adds more burden on the NVM allocator because an inner node cannot be updated in-place when a new key is added to it. We observed similar trends in the other trees.

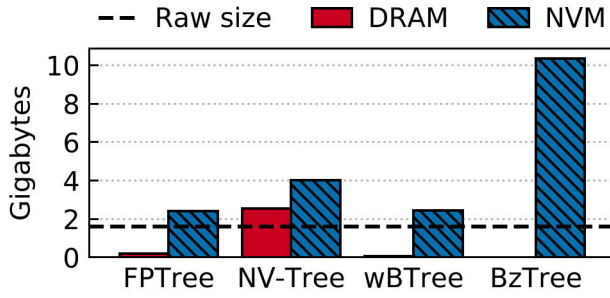


Figure 2.13: Memory consumption after inserting 100 million records of 8 B keys and 8 B values.

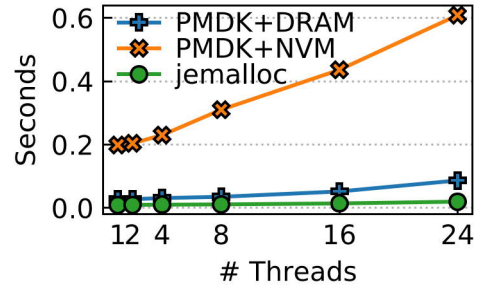


Figure 2.14: Allocation of 1024 blocks of 1 kB with different allocators.

Recovery Time

An important aspect of persistent data structures is their ability to recover consistently and (near) instantly after a failure or clean shutdown. We test recovery time by loading in each tree a fixed number of records and then killing the process. Table 2.6 shows the time in seconds for each tree to recover from a crash after loading 50 million and 100 million records. wBTree, FPTree and NV-Tree enforce consistency of each single operation, so recovery consists basically of rebuilding transient data. Both FPTree and NV-Tree place inner nodes in DRAM and thus these must be rebuilt upon recovery. As expected, the time for rebuilding inner nodes after inserting 100 million records is about $2 \times$ the time with 50 million records (recovery time scales linearly as all leaf nodes must be read). wBTree and BzTree reside fully in NVM. Therefore, upon recovery they simply need to open an existing NVM pool and retrieve the root object. From the root object, all the remaining objects allocated in the pool can be discovered and reached. BzTree relies on PMwCAS to always transform the tree from one consistent state to another, without needing a customized recovery procedure. After opening the existing pool, BzTree delegates its recovery process to PMwCAS, which completes its own recovery phase by rolling forward or backward PMwCAS operations that were in-progress when the crash happened. This translates into scanning the PMwCAS descriptor pool [WLL18] which is fixed-sized (100 kB in our experiments). Moreover, the amount of in-progress PMwCAS operations at any point in time is bounded by the number of concurrent threads. Therefore, we see a very small difference in recovery time under different initial sizes.

These result show that NVM-only trees are able to recovery near instantly (sub-second recovery time), at the price of lower runtime performance; placing more components in DRAM may improve runtime performance at the cost of longer recovery time. Nevertheless, we note that the recovery time of hybrid trees could be improved, in case of clean shutdowns, by spilling inner nodes to NVM and copying them back to DRAM upon recovery.

Future of NVM

To conclude our case study, we note that NVM is still at an early stage and yet to become mainstream. Nevertheless, our experience in this evaluation enabled us to identify two areas where improvements of the current NVM technologies could have a major impact.

Initial Size (# of records)	FPTree	NV-Tree	wBTree	BzTree
50 million	1.77s	4.15s	0.036s	0.153s
100 million	3.56s	8.45s	0.037s	0.186s

Table 2.6: Recovery time. NVM-only trees (BzTree/wBTree) achieve sub-second recovery time. DRAM-NVM hybrid trees (FPTree/NV-Tree) trade recovery time for performance.

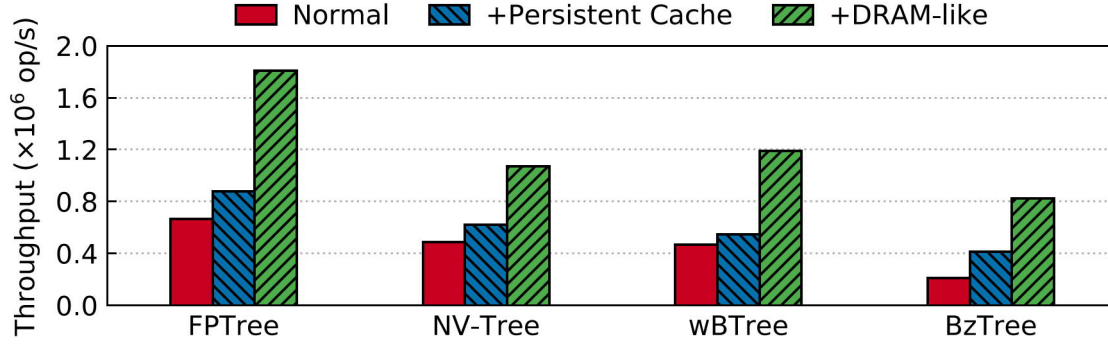


Figure 2.15: Impact of persistent caches and DRAM-like performance on NVM in a single thread workload with 50% reads, 25% inserts, and 25% updates with uniform key distribution.

Persistent CPU caches. Modern CPUs rely on sophisticated, fast volatile caches for good performance. This introduces the main challenge of carefully flushing cache lines to NVM while trying to reduce the amount of flushes and NVM accesses. We consider that enabling CPU caches to become persistent (e.g., by protecting them against power failures with a capacitor) is the natural next step to simplify software development and increase performance [WJ14, IKK16]. In such scenario, applications could completely relinquish the use of instructions such as CLFLUSHOPT and CLWB. However, guaranteeing the ordering of writes using SFENCE may still be required.

DRAM-like NVM devices. A second advancement would be approaching the performance characteristics of DRAM. The NVDIMM-N devices discussed in Section 2.1.1 already offer DRAM performance, but its high cost and DRAM’s scalability issues make it prohibitive in large scale. Devices based on new materials are much cheaper but still lag behind DRAM in terms of performance. This gap might be closed either by reducing the cost of flash-based NVM or by enhancing the performance of cheaper alternatives (e.g., via innovations in materials or more sophisticated caching mechanisms).

Figure 2.15 shows the potential impact of these advancements on the tree structures considered. We emulate persistent CPU caches (+Persistent Cache) by removing all cache line flushes from the code path, and emulate fast NVM (+DRAM-like) by placing the NVM memory pool in a DRAM-backed file system (`tmpfs`). While persistent CPU caches improve throughput by $1.32/1.27/1.17/1.94\times$ for FPTree/NV-Tree/wBTree/BzTree respectively, the main benefit is probably in terms of simplifying the programming model which will also lead to fewer bugs and savings in development and code maintenance costs. The biggest gains are achieved by increasing the raw device performance, which further improves the throughput by a factor of $2.05/1.72/2.17/2.00$ for FPTree/NV-Tree/wBTree/BzTree respectively. This shows that indexes are highly sensitive to device latency and bandwidth.

3

LOG-STRUCTURED MERGE-TREES

One of the main challenges of NVM is the lack of control developers have over when records are persisted, making it hard to ensure consistency when records are updated in-place. On the other hand, in log-structured architectures, records are always written to a new location and never overwritten. This property significantly alleviates the challenge of guaranteeing atomic and consistent writes to NVM. Therefore, log-structured systems are a very attractive first candidate for exploring opportunities to leverage NVM. This chapter explores use-cases of NVM in log-structured merge-trees, a storage management architecture implemented by many modern systems, such as LevelDB, RocksDB, SQLite, Cassandra, and Bigtable. More precisely, in this chapter we investigate two main questions:

- What is the impact on LSMs if we replace all persistent storage by NVM?
- Do LSMs still benefit from DRAM caches when NVM is used as persistent storage?

3.1 LSM AND NVM

The log-structured merge-tree (LSM) [OCG096] was originally proposed in the context of HDDs with the goal of leveraging the faster sequential writes. The write-optimized nature of LSMs make them appealing to systems that experience high write and data ingestion rates. Systems that must ingest an event log and query the ingested data with acceptable response time are common examples.

In the context of SSDs, while sequential writes are still usually faster than random writes, the performance gap is smaller. Nevertheless, LSMs gained a new notoriety due to the reduced *write amplification* to persistent media, a major concern due to the limited erase cycles of SSDs. This has a direct impact in lowering the performance degradation and increasing the device lifetime. While NVM reduces the gap between random and sequential accesses even further and it has a longer endurance and lifetime, it can still benefit from sequential writes and reduced write amplification, even if to a smaller extent. However, LSMs have an important characteristic that makes them particularly appealing in the context of NVM: it is easy to guarantee atomicity and durability of arbitrarily large writes.

As previously introduced, when directly accessing NVM through *load/store* operations, the developer cannot prevent a cache line from being arbitrarily evicted by the CPU. Furthermore, store operations can also be re-ordered by the compiler or the CPU. Under these circumstances, it becomes challenging for database systems to guarantee a certain level of consistency, since that requires a careful control of when data is written to persistent storage. In contrast to update-in-place systems, maintaining consistency and atomicity in an LSM is easier, as records are always written to a new location in a log-structured manner, only requiring a small atomic update to reflect the update (such as updating the tail of the log or a catalog entry pointing to the new location). Log-structured writes also enable *non-temporal stores* to be employed to directly stream data from the CPU to NVM, bypassing the CPU cache. Furthermore, the synergy between NVM and LSM is made even more explicit by the observation that reading from NVM is faster than writing, while writing to LSM is faster than reading. In other words, they complement each other nicely.

3.2 LSM ARCHITECTURE

Figure 3.1 illustrates the general architecture of an LSM. Writes in an LSM are made to an in-memory data structure (C_0) and made durable by logging. When C_0 reaches a certain threshold size, it is merged with a lower level persistent component C_1 (which can be, for instance, a tree data structure) and newer writes are made to a brand new C_0 instance. This can be generalized to a hierarchy of multiple persistent components $C_1..C_n$, in such a way that the size of components grows exponentially in relation to the preceding component in the hierarchy. The persistent components are immutable. The lookup operation has to search each component, from the most recent to the oldest, until the record with a given key is found. A read-only cache holds frequently accessed blocks of the persistent components in memory, thus reducing the amount of I/O required. It is also common to employ Bloom filters at each component to minimize the amount of I/O by avoiding searching a component that does not contain a given record. Delete operations are done by inserting a special *tombstone* record to indicate the absence of a record with a given key. Range scans are usually more expensive than in a B+Tree, as all components must be searched, since records in the range can be present in any of them.

In order to prevent slowdown when a merge is triggered during normal processing, an eager rolling-merge process between components usually runs in the background. Merging components is required to reclaim space and keep a predictable performance by alleviating the read penalty introduced by multiple components. The merge process picks blocks of adjacent components that have overlapping records, eliminates duplicates by removing older versions of records, and sequentially write the results to a new location, following the log-structured nature of the LSM. There are many approaches for merging components, such as *leveling* and *tiering*, and picking the right one, as well as tuning the scheduling policy of merge processes (which portions of which components to merge), is a critical aspect of LSMs. To summarize, LSMs achieve a better write performance when compared to update-in-place strategies, since random writes are converted into sequential ones and the amount of data written to persistent storage is reduced (i.e., decreasing write-amplification).

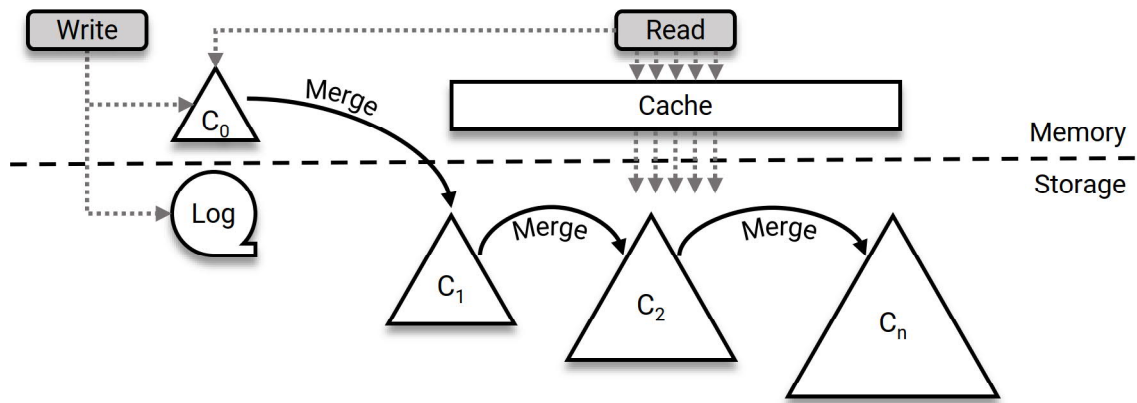


Figure 3.1: General architecture of a log-structured merge-tree (LSM).

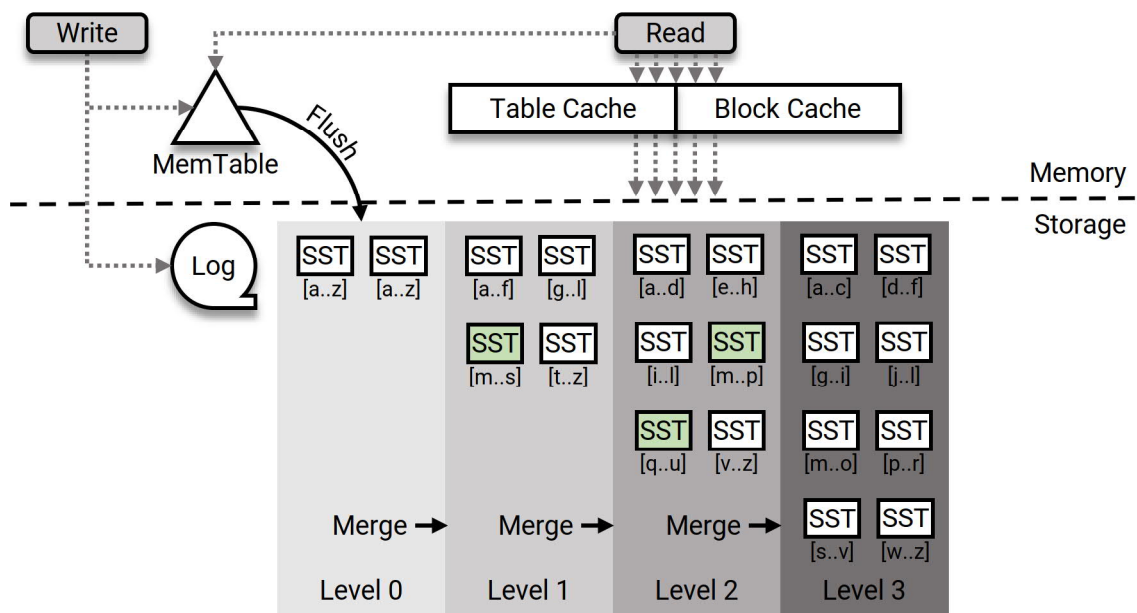


Figure 3.2: LevelDB architecture represented with the first 4 levels for simplicity.

3.2.1 LevelDB

LevelDB is an open-source, embeddable, persistent key-value store originally developed by Google. Keys and values are stored as byte arrays and sorted by key. It defines a basic interface including *Put(key,value)*, *Get(key)*, and *Delete(key)* operations. While LevelDB implements the conceptual architecture of an LSM, relevant implementations details and specific nomenclature is defined for the sake of better understanding of the upcoming sections. Figure 3.2 shows LevelDB specific architecture.

LevelDB uses a skip-list as its in-memory data structure, called **MemTable**. The persistent components are organized as levels starting from *Level 0* (most recent) with an increasing number to older levels. Each level is composed of files called **Sorted String Tables (SST)**. The layout of an SST is represented in Figure 3.3. An SST has a fixed size of 2 MB and consists of four types of blocks. A **data block**, usually 4 kB, contains keys and values in

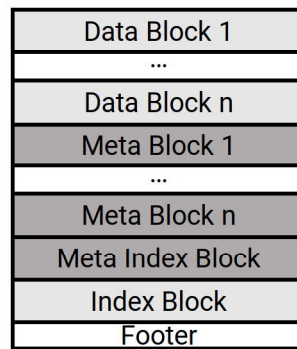


Figure 3.3: Sorted String Table layout.

sorted order (possibly compressed). Additionally, an SST has a single **index block** used to locate the data block of a given key (conceptually similar to a B+Tree of height 2). Optionally, there can be **meta blocks** to store information such as Bloom filters, in which case there will also be a **meta index block** to locate them. Finally, each SST has a small **footer** containing information such as checksum, unique identifier, total size and relative offset of blocks. Therefore, the footer is always read prior to searching a record in an SST.

Whenever the *MemTable* reaches a threshold size (4 MB by default), it is compressed and converted into one or more SSTs of *Level 0*. With the exception of *Level 0*, SSTs of the same level do not have overlapping key ranges, meaning that at most one SST per level has to be read (as shown by the key range covered by each SST in Figure 3.2). When the number of SSTs in *Level n* reaches its threshold, they are merged with the SSTs of *Level n+1* that have overlapping key ranges to generate new SSTs. As an example, in Figure 3.2, the SST in *Level 1* marked in green will be merged with the two SSTs in *Level 2* marked in green, since they have overlapping key ranges.

Finally, LevelDB has 7 levels by default, with *Level 0* containing a maximum of 4 SSTs, *Level 1* containing 10 SSTs, and each level after containing a maximum of factor 10 the number of SSTs in the previous level. A system catalog keeps track of information on all levels, e.g., current SSTs and their key ranges. The system catalog is atomically updated to reflect changes, such as removing and adding SSTs after a merge operation.

Block Cache

By default, LevelDB uses memory-mapped files and the page cache of the operating system for improved performance. These features are orthogonal and we have ignored them to better isolate the behavior of LevelDB's own caching component, since most database systems also rely solely on their own caching.

In addition to the *MemTable*, LevelDB implements two DRAM read-only caches: **table cache** and **block cache**. The table cache is used to hold entries containing metadata about SSTs and index blocks (possibly meta blocks) of SSTs recently accessed. The block cache holds exclusively the data blocks from SSTs. In order to improve concurrency, both caches are composed of 32 shards (default), and each shard implements least recently used (LRU)

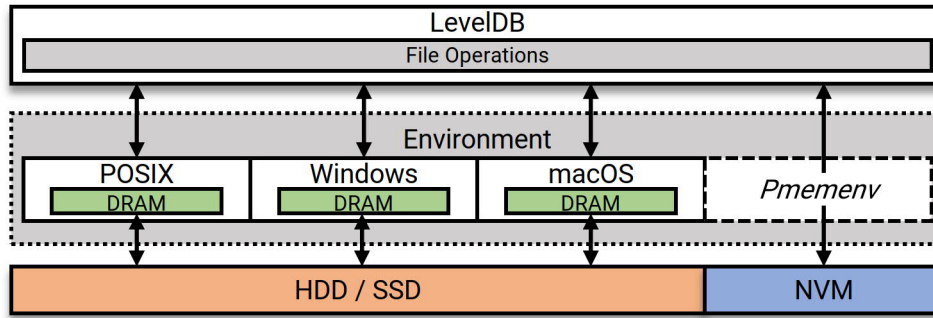


Figure 3.4: LevelDB environments for accessing storage.

as the default block replacement policy. A read operation must first check if the given key is present in the MemTable. If not, then it will locate the candidate SST in the next level based on the SST key ranges contained in the system catalog. Once the relevant SST is found, the table cache and block cache are searched for the corresponding index block and data block, respectively. If any of these blocks is not found, they are read from storage into the cache before resuming the operation.

3.3 PERSISTENT MEMORY ENVIRONMENT

In the context of NVM, the first question to be answered is: *what is the impact on LSMs if we replace all persistent storage (usually SSD) by NVM?* Our first contribution is the investigation of this impact. To that aim we extend LevelDB to provide direct access to NVM. As shown in Figure 3.4, LevelDB originally accesses storage by issuing **file operations** (open, create, rename, delete, read bytes, write bytes) to an abstract **environment**. The goal of this abstract interface is to provide easy portability to different operating systems by only implementing the required file operations, as well as allowing custom behaviors on file operations (such as monitoring). We rely on this interface to implement a custom environment to directly access NVM: *Pmemenv*.

Pmemenv is implemented with the aid of the Intel's *Persistent Memory Development Kit* (PMDK) [Int14] and it acts as a lightweight file system in the user space. It is tailored specifically for accessing and managing LevelDB's files directly in NVM. This contrasts with usual interface, where the application has to go through the kernel space to access persistent storage. *Pmemenv* has two main advantages over general purpose file systems. First, it enables zero-copy reads, meaning that data can be read directly from NVM without loading it to DRAM. Second, it enables read and write operations to NVM at a cache-line granularity (64 B) instead of whole block (usually 4 kB).

PMDK enables users to create a persistent memory pool by creating a file an NVM-aware file system and mapping the file to the application virtual memory, allowing direct access to the underlying storage. The application can then manage objects in the persistent memory pool through the PMDK allocator interface, which hides the complexity required to properly enforce the order of write operations. PMDK also uses a lightweight logging scheme to guarantee the fail-safe atomicity of persistent allocations. A persistent pointer

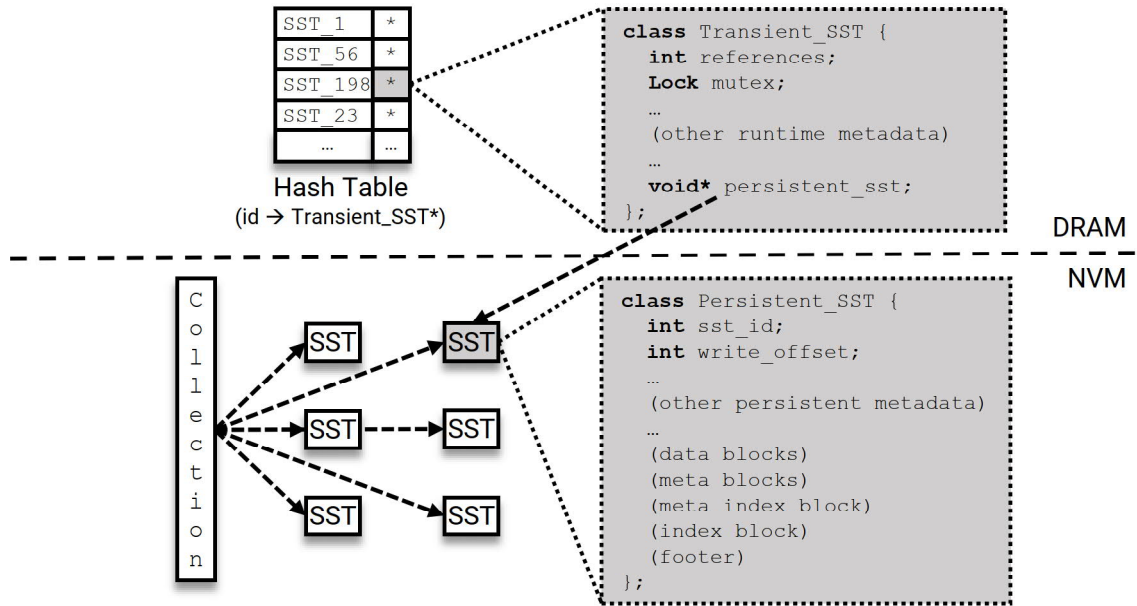


Figure 3.5: *Pmemenv* architecture.

for each allocated object is stored in a **collection** (implemented as a data structure, such as a linked list) located in a fixed memory region of the pool. The user is able to iterate over this collection and retrieve every allocated object in the persistent memory pool, thus preventing persistent memory leaks. PMDK also offers transactional support to enable more complex atomic memory operations than simple allocation and deallocation of blocks of memory. These memory transactions, however, are not used by *Pmemenv*.

Figure 3.5 illustrates the general architecture of *Pmemenv*. It comprises two main parts: an in-memory hash table and a persistent memory pool. Every SST is composed of a transient part and a persistent part. The hash table is used to map the unique identifier of an SST to its transient part. The transient part of an SST includes non-critical metadata that is only required during runtime, such as reference counters, mutexes, and status flags. The transient part also contains a pointer indicating the location of the persistent part in the persistent memory pool¹. The persistent part contains critical data required by basic operations and to rebuild the hash table during restart. In our implementation, the persistent part of an SST is composed of the unique identifier, the current append offset, and the remaining data and index blocks shown in Figure 3.3. The PMDK library already stores the size of allocated persistent memory blocks, therefore, even if this is critical data, we do not store it and rely on the library to provide this information. In case of a system failure, the hash table can be rebuilt by iterating over the collection of pointers pointing to the allocated SSTs and retrieving the SST identifiers. The metadata in the transient SST part is set to default values and the pointer is set to the corresponding address in the persistent memory pool.

The separation of SSTs into transient and persistent parts allows metadata to be moved between them, enabling the system to possibly slide a *persistence bar* to choose which parts to make persistent [OBL⁺14]. In one extreme scenario, all data, metadata, and data structures are allocated in the persistent memory pool. This would reduce the recovery

¹Note that both DRAM and NVM share the same virtual memory space.

time to a minimum at a possible performance cost and additional complexity. While an interesting direction to be explored by future work, we omit more detailed discussions to focus on the challenges discussed in the following sections.

Finally, since all writes to SSTs are log-structured (i.e., append-only), they can use non-temporal stores to bypass the processor cache. Writes of an arbitrary number of bytes to an SST are protected from partial writes by updating the current write offset of the SST (8 B), which is guaranteed to be an atomic operation. The MemTable log and other auxiliary files, such as the catalog of SSTs, are also managed by *Pmemenv* the same way as SSTs.

3.4 2Q CACHE POLICY FOR NVM

LevelDB implements LRU as the default replacement policy for each shard of the *table cache* and *block cache*. In other words, whenever the cache is full and a miss occurs, the least-recently accessed block is evicted to make space for the requested one. However, when the SSTs are in NVM, the processor is able to directly read these blocks without copying them to DRAM. On one hand, DRAM has lower latency and enables faster access. On the other hand, not only the caching components introduce additional complexity, but there is also an overhead for copying data from NVM to DRAM when a miss occurs. This overhead might not be worthwhile when compared to the cost of simply accessing the data directly in NVM. Therefore, the second question we investigate is: *can LSMs still benefit from DRAM caches when byte-addressable NVM is used as persistent storage?*

Ideally, we would like the cache replacement policy to keep track of accesses not only to cached blocks, but also to un-cached ones. This would enable the policy to make a better decision whether it is advantageous to copy a given block to DRAM, avoiding a hotter block to be evicted as a consequence of a miss to a colder one. As an example, this behavior would prevent a table scan from trashing the cache by evicting all of its contents.

The 2Q replacement policy [JS94] considers similar goals. While the original 2Q was proposed in the context of main memory and hard disks, we have adapted the concept to NVM in order to enable zero-copy reads from NVM. In other words, we use 2Q as a cache admission policy, or **placement policy**, rather than as a replacement policy. Similar to the original proposal, our 2Q policy has two components: *AM* and *A1*. *AM* holds cached blocks and is managed by some replacement policy (LRU in our case). *A1* does not hold any blocks, but only keeps track of accesses to un-cached blocks (i.e., blocks read directly from NVM). Since only references to blocks are kept, the space consumption of *A1* is minimal. The size of *A1* is tunable and references are kept in a FIFO queue.

Algorithm 3.1 shows the pseudo-code for the two main functions of the replacement policy. The function **Fix** is called to request access to a block. If the block is already cached, it is directly returned (line 3). Otherwise, if it is not going to cause another block to be evicted or if there was another reference to it in the recent past (line 5), the block is transferred to DRAM (line 6). If the block was accessed recently, it means that it is probably hot and is a good candidate to be copied to DRAM. If both conditions are false, the block is read directly from NVM and a reference to it is added to *A1* (line 9-10). A hash table is used for efficiently looking up blocks on *AM* and *A1*. The function **Victim** is called when the cache

Algorithm 3.1: 2Q policy for NVM pseudo-code.

```
1 function Fix(block_id)
2   if AM.contains(block_id) then
3     return AM.get(block_id)
4   else
5     if !AM.full() OR A1.contains(block_id) then
6       AM.load(block_id)
7       return AM.get(block_id)
8     else
9       A1.add(block_id)
10      return NVM.get(block_id)
11    end if
12  end if
13 end
14
15 function PickVictim()
16   block_id ← AM.remove_lru()
17   A1.add(block_id)
18 end
```

is full and we need to pick a block for eviction. The block picked is the least-recently used one (line 16), but a reference to it is additionally added to A1 (line 17). Since A1 is a FIFO with a limited size, it discards its oldest reference when a new one is inserted.

Figure 3.6 shows the runtime of a binary search over a block of 4 kB of integers in three scenarios: DRAM, NVM (latency approximately $4\times$ that of DRAM), and simulating a miss of LRU in a DRAM cache over NVM storage. In the breakdown of the cost of an LRU miss, it is possible to see the constant overhead introduced by the cache component (fix, unfix, eviction, etc), as well as the huge cost of copying data from NVM to DRAM. The binary search represented in the breakdown is faster than the one in DRAM, because the data was already cached by the CPU caches during the transfer (i.e., the cost is amortized). The cache miss in the LRU policy has a constant cost comprised of lookup, eviction, and copy of a block to DRAM. The additional cost required by the policy exceeds by far the cost of simply doing the binary search in NVM. The 2Q policy introduces two different scenarios for a cache miss: the first scenario has a lower cost as it simply adds a reference to the FIFO and reads directly from NVM, while the second scenario is similar to LRU and has a higher cost. In an NVM context, a well-tuned 2Q policy should prefer to pay the lower miss cost for blocks not frequently accessed and the higher miss cost for blocks expected to be frequently accessed in the near future. While most proposed replacement strategies (LRU, LFU, CLOCK, etc) focus on improving the hit ratio, the idea of being able to choose between two different miss costs adds a new dimension. Assuming that the hit ratio is determined by the replacement strategy and the cache size, a smaller 2Q cache is likely to have more misses than a larger LRU cache. However, if most of the 2Q misses pay the lower cost, similar or even better performance than LRU can be achieved with a lower DRAM consumption. This introduces the non-intuitive idea that a higher hit ratio does not necessarily translate to better performance, as the costs for misses might differ.

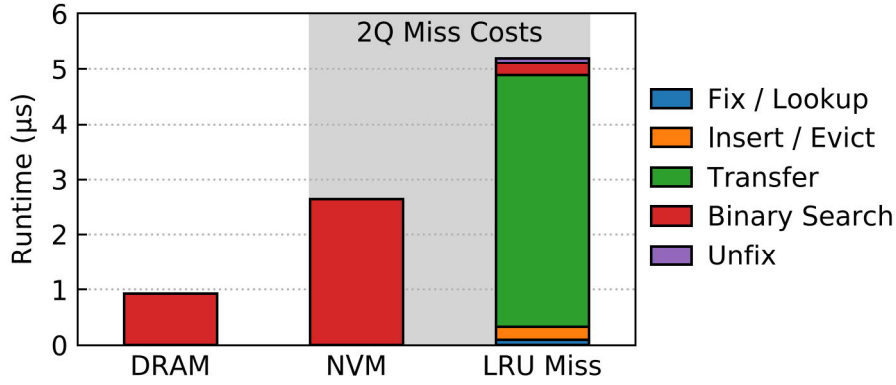


Figure 3.6: Average runtime of binary search over 4 kB of integers.

3.5 EVALUATION

We use the Intel NVM Emulation platform that emulates an NVM device by accessing a dedicated area of DRAM with a higher, tunable latency. The higher access latency to DRAM is achieved through a special BIOS. A full description of this system can be found in [Dul16]. The system is equipped with two Intel Xeon E5 processors. Each one has 8 cores, running at 2.6 GHz, and featuring 32 kB L1 data and 32 kB L1 instruction cache as well as 256 kB L2 cache. The 8 cores of one processor share a 20 MB last-level cache. The system has 64 GB of DRAM and 192 GB of emulated NVM. The emulated NVM device is mounted with the ext4 file system with DAX support. In the experiments, we set the latency of NVM to 360 ns, approximately 4 times the latency of DRAM (90 ns). Considering HDD/SSD is out of the scope of this work, since all experiments are based on the assumption that the storage device can be directly accessed through the CPU caches. The system runs Linux with kernel version 4.4.21. We use PMDK (v1.2)² and LevelDB (v1.19). All the source code was compiled using GCC 4.8.5. We disabled the memory mappings of SSTs and operating system caching in LevelDB. Compression and filtering of SSTs (bloom filters) are also not used. The MemTable is set to its default size of 4 MB. All requests to LevelDB are made from a single thread.

3.5.1 Write Performance

We first analyze runtime and latency of two approaches for writing to NVM: through the ext4 file system with Direct Access (DAX) support (as a drop-in replacement for HDD/SSD) and through *Pmemenv*. As mentioned in Section 3.3, the file system manages these operations at a block granularity (usually 4 kB), while *Pmemenv* allows finer control over written data. This implies that, in a scenario where durability of single operations must be guaranteed, *Pmemenv* is able to write only the changed cache lines. However, most systems implement some sort of group commit to hide write latencies. LevelDB enables this by batching many *Put* operations in a *WriteBatch* that is accumulated in DRAM and is later made durable as a single *Put*. We consider scenarios with different *WriteBatch* sizes. Additionally, we investigate if batching writes in DRAM still offers benefits for *Pmemenv*.

²Originally named NVML, the name change to PMDK was announced on December 11, 2017.

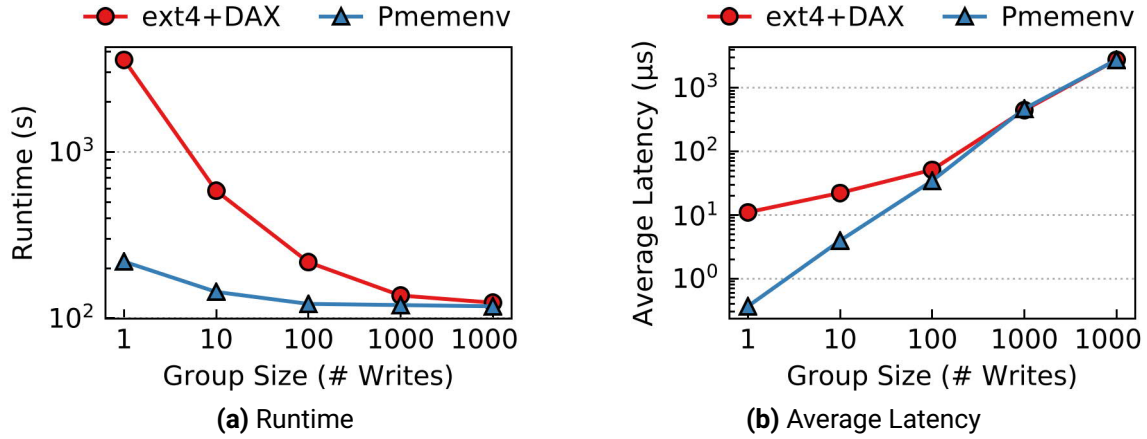


Figure 3.7: Insertion of 100 million key-value records with varying *WriteBatch* size.

File System	1	10	100	1000	10000
ext4 + DAX	23	56	161	586	4750
Pmemenv	12	39	125	517	4749

Table 3.1: Standard deviation of average latency in Figure 3.7b in microseconds.

We run a write-only workload of the *Yahoo! Cloud Serving Benchmark* (YCSB) [CST⁺10] with 100 million key-value records, where each key is 16 B and each value is 112 B, giving a total of 128 B per record. The results are depicted in Figure 3.7.

First, for a group size of one, the *ext4+DAX* configuration has to persist data at the granularity of pages via *fsync*, which incurs a high cost if single operations are to be made durable. *Pmemenv* is able to avoid the kernel path and to persist data at a much smaller granularity, which reduces the overhead related to write operations. Increasing the group size drastically improves the performance of *ext4+DAX* (16 times faster when increasing group size from 1 to 100), as the cost of *fsync* is amortized across many insertions. For *Pmemenv*, an improvement of approximately 50% in runtime is observed when increasing the group size from 1 to 10. For larger group sizes the difference is not significant.

Grouping insert operations in batches introduces a trade-off between throughput (runtime) and latency, as the first requests to arrive in a group are delayed. Figure 3.7b shows the average latency of single insert requests for different group sizes. The standard deviation can be observed in Table 3.1. For smaller group sizes (1 to 100) in *ext4+DAX*, the increased latency is justified by the large gains in runtime, making the batching of operations an obvious choice for most applications. However, for *Pmemenv*, this trade-off is not so clear and the decision of sacrificing latency for better runtime might become a matter of service level agreements, like response time required by applications. Finally, not only *Pmemenv* presents lower runtime and lower average latency than *ext4+DAX*, but also a lower standard deviation for group sizes 1 to 100, which translates to a more predictable performance over time.

3.5.2 Read Performance

We also analyze if better performance can be achieved by dedicating a portion of DRAM for caching hot data. Since writes in LevelDB are made in a separate data structure,

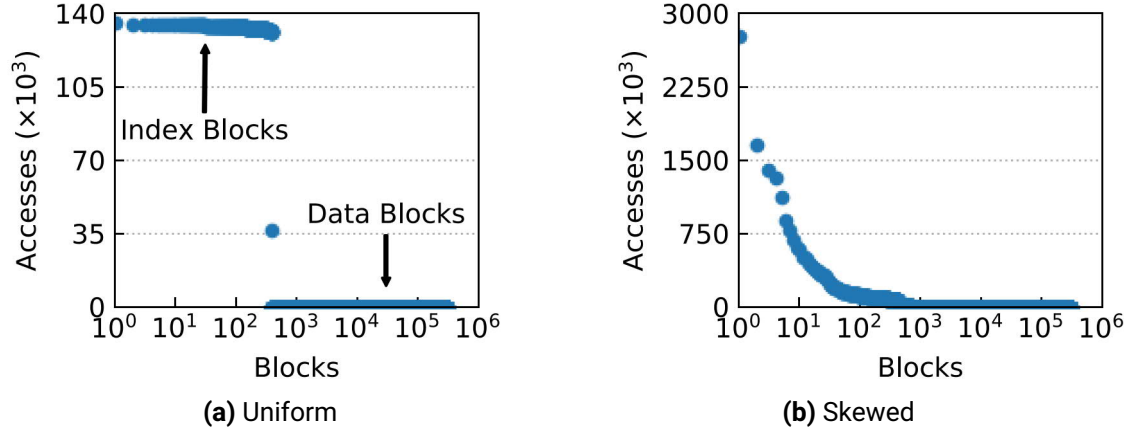


Figure 3.8: Distribution of accesses to blocks.

the remaining caching components benefit mainly read operations. Therefore, we have considered read-only YCSB workloads to better outline the performance impact. Each workload issues 50 million lookups over 10 million key-value pairs. Before each workload is executed, the caches are warmed up by executing read requests until they become completely full. The warmup time is not considered. We analyze two different scenarios: uniform and skewed distribution (80% of requests to 20% of the records) of key requests. It is worth noting that each Get request for a key translates into two block requests, one for the index block and one for the data block. Hence, even a uniform distribution of keys presents a skewed distribution of block accesses. Figure 3.8 illustrates the number of accesses of each block sorted from the most to the last accessed.

Figure 3.9 presents the runtime of both read-only workloads for different system configurations. The X axis represents which portions of SSTs are cached in DRAM. We start with a *No Cache* approach, where the caching components were completely removed and all SSTs are read directly from NVM through *pmemenv*. Later, we gradually increase the DRAM consumption by statically placing portions of every SSTs in DRAM. The *Footers* scenario has the footers of all SSTs in DRAM. The footer of an SST contains pointers to the index blocks, as well as checksums and additional status flags. Footers are frequently accessed (it is where each read in an SST starts) and, since they are relatively small (around 64 B), keeping all of them in DRAM improves performance at a minimal cost of memory consumption. Next, *IndexBlocks* considers holding the index blocks of all SSTs in DRAM. For our workloads, every index block is approximately 18 kB and there are around 500 SSTs, giving a total of about 10 MB additional DRAM consumption (less than 1% of the total size). The observed performance gains are significant and justify the additional memory consumption. At this point, we can conclude that a careful placement of frequently accessed data in DRAM is beneficial despite the low latency of NVM.

However, so far we have only statically placed data in DRAM or NVM, there is no caching component involved. In addition to keeping all index blocks in DRAM, we introduce a caching component for the data blocks, which enables the system to dynamically adapt by keeping frequently accessed data blocks in DRAM. The scenarios *1% Blocks* and *10% Blocks* cache the indicated amount of data blocks. The interesting observation for LRU (default cache policy in LevelDB) is that dedicating additional DRAM harms the system's performance initially. While the performance improves with more DRAM (*10% Blocks*),

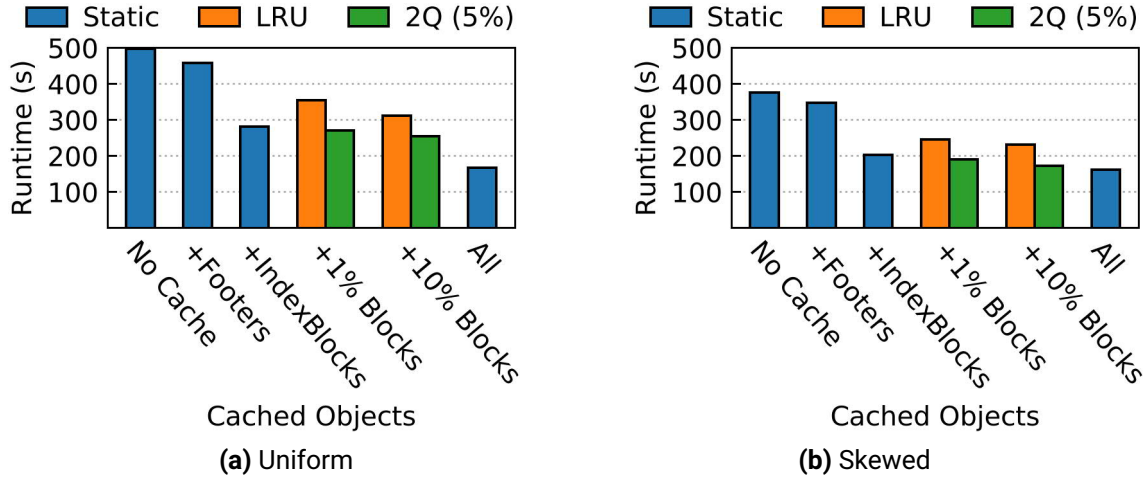


Figure 3.9: Runtime of read-only workload.

larger amounts of DRAM would be required to achieve the same performance of caching only index blocks. This is explained by the cost of cache misses in LRU: lookup, eviction, and transfer of block from NVM to DRAM. If cached data is not accessed enough times, this cost is high compared to the alternative of directly accessing data in NVM and avoiding the overhead caching. As discussed in Section 3.4, the cost of transferring data to DRAM is only worthwhile if the policy can predict that this block will be accessed frequently in the near future.

To alleviate the high cost of misses, we have implemented 2Q to enable a more lightweight policy. We have set the A1 size to 5% of the AM size. Our initial goal with 2Q is to avoid the observed behavior where the system gets slower when more DRAM is dedicated for caching. In contrast to LRU, there is always some performance improvement with larger caches for data blocks. This comes from the fact that 2Q avoids evicting a cached block and moving a new block to DRAM when a miss occurs. Finally, the *All* scenario represents the runtime with the whole dataset cached in DRAM. It is possible to see in Figure 3.9b that the 2Q cache with enough DRAM to hold 10% of the data blocks can achieve similar performance of holding all blocks in DRAM.

3.5.3 Mixed Workloads

Based on the observations from the previous experiments, we analyze the overall behavior of the system in workloads containing both updates and lookups. Two mixed workloads with skewed access are considered: 25% and 50% of updates. We also run the experiments with varying NVM latencies to show that the behavior is the same regardless of the slowdown/speedup incurred by higher/lower latencies.

Similar to previous results, we start with a *NoCache* approach and gradually dedicate more DRAM for caching purposes. The *Group Size 10* has enough DRAM for holding 10 update operations and persist them as a single *WriteBatch*. Later, in addition to that, we reserve enough DRAM to hold *All Indexes*. Finally, we hold up to 10% of the data blocks in a 2Q cache. Figure 3.10 presents the gradual performance gains achieved in each of

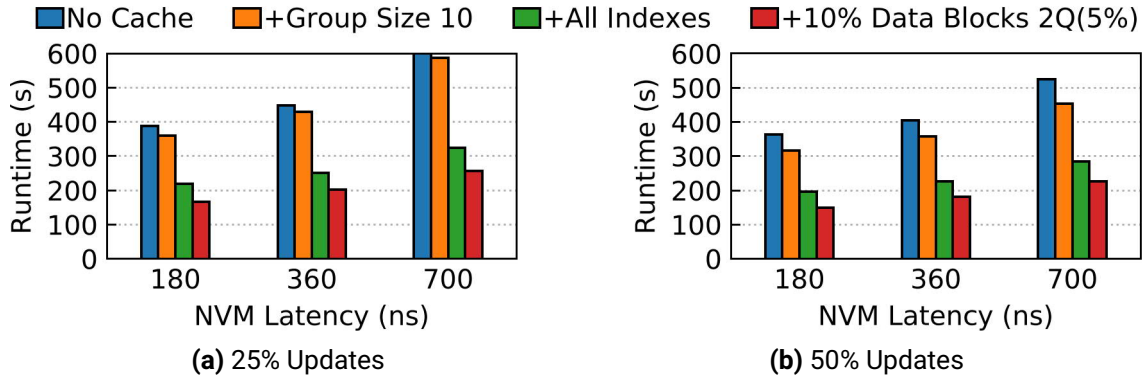


Figure 3.10: Runtime of skewed mixed workload.

these steps. The biggest improvement happens when keeping all index blocks in DRAM. Not only index blocks are frequently accessed, but their additional DRAM consumption is minimal, making it realistic to hold all of them in DRAM and avoiding any replacement policy overhead. Regarding data blocks, while a block cache with 2Q replacement policy offers some benefits in terms of performance, it is up to the user to decide if the cost of additional DRAM justify these gains when compared to eliminating the block cache completely and always accessing data blocks directly in NVM. We consider that enabling the system to manage hot and cold data is important and better caching policies can probably achieve this behavior with even better performance.

3.6 ADDITIONAL CASE STUDY: ROCKSDB

The techniques proposed in the previous sections were evaluated in 2016. By the time, real NVM hardware was not available, and therefore we followed the approach of most works of basing our evaluation in emulation platforms. Since then, much has changed. Not only NVM hardware became available and accessible in the form of Intel Optane DC Persistent Memory Modules, but libraries and kernel modules for managing NVM also evolved and improved significantly. Nevertheless, the assumptions upon which the proposed techniques were based still hold, such as the NVM latency being higher but within the same order of magnitude of DRAM. Therefore, to prove that the concepts introduced are still applicable, we have implemented and evaluated them in a more modern use-case and more recent hardware environment.

Instead of LevelDB, we take RocksDB as a new case study. RocksDB is developed by Facebook and was originally forked from the LevelDB project. In the past few years, RocksDB performance has improved significantly on top of LevelDB, while still following the same LSM architecture. Furthermore, new functionalities and options were added, which enabled RocksDB to be used as the storage engine for more complex SQL systems, such as MySQL (under the name of MyRocks) and MongoDB (under the name of MongoRocks).

Like LevelDB, RocksDB assumes that the main storage media is SSD (potentially HDD) and does not leverage NVM. Rather than leveraging NVM for the main storage, as in our initial approach with LevelDB, we opted for a more moderate approach on RocksDB by

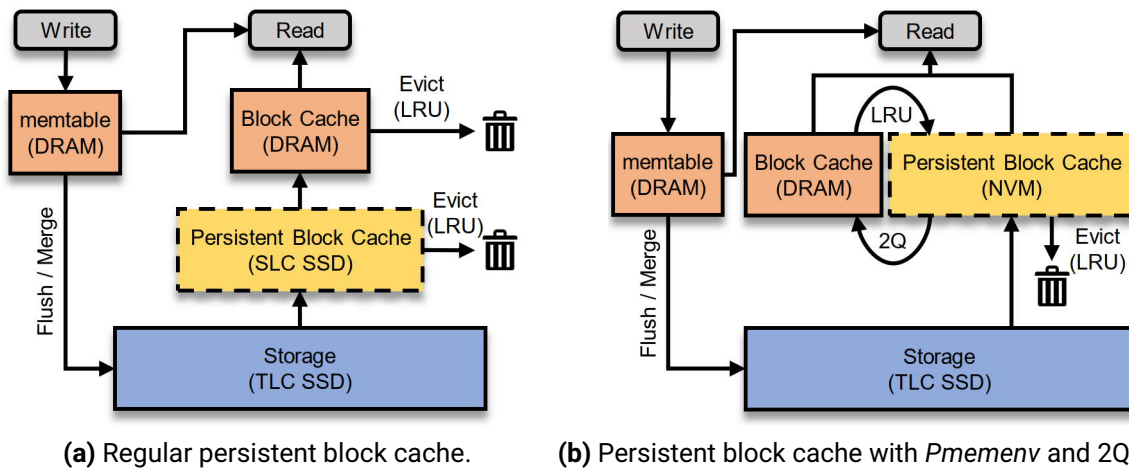


Figure 3.11: Overview of the main components of the RocksDB architecture.

leveraging NVM in the *persistent block cache* component. The *persistent block cache* is an optional component of RocksDB and, like the *block cache*, it holds frequently accessed blocks from the main storage and serves as an intermediate caching layer³, as shown in Figure 3.11a. While the *block cache* is implemented in DRAM, one of the intended uses of the *persistent block cache* is to leverage SLC SSDs, which are faster and more expensive, to cache frequently accessed blocks from the main storage, which is usually stored in regular TLC SSDs. Therefore, blocks in the *persistent block cache* are stored in files and managed through the regular file system interface.

In order to leverage NVM instead of SLC SSD, we implemented an NVM-aware *persistent block cache* by employing both the *Pmemenv* and the 2Q placement policy introduced in the previous sections, as seen in Figure 3.11b. This not only avoids the overhead of file system interfaces and allows blocks in the *persistent block cache* to be accessed directly, but it also guarantees that only hot blocks are migrated to the DRAM *block cache* through 2Q, while blocks accessed less frequently are accessed directly in NVM. Differently than the *Pmemenv* on LevelDB, we re-implemented the *Pmemenv* in the *persistent block cache* of RocksDB without the aid of PMDK, since cache operations are not complex enough to justify a more sophisticated library. Still, we access NVM through the same way: mapping a file on a DAX file system to the virtual memory space of our application (RocksDB).

3.6.1 Evaluation

We use the environment shown in Table 3.2 to evaluate the impact of *Pmemenv* and the 2Q placement policy in the *persistent block cache* of RocksDB. We initially load RocksDB with 1 billion records with keys of 8 B and values of 100 B. The total size of the dataset is 110 GB and stored on SSD. Since both the *block cache* and the *persistent block cache* serve only read requests, we run a workload consisting only of *Get* operations to stress these components. We focus on two cache configurations. In the first scenario, no *persistent*

³Unlike LevelDB, we configure RocksDB to make no distinction between *index blocks* and *data blocks*. Therefore *index blocks* are also cached by the same *block cache* and *persistent block cache*, rather than by a dedicated *table cache*.

Processor	Intel Xeon Platinum 8260L CPU (35.75 MB Cache, 2.40 GHz)
Main Memory	96 GiB DDR4 2666 MHz (6 × 16 GiB modules)
NVM	Intel Optane DCPMM 1.5 TiB (6 × 256 GiB modules)
SSD	Intel SSD DC P3700 Series 1.6 TiB (PCIe 3.0 x4, NVMe)
RocksDB	6.2.2
Operating System	Linux 5.3.4-3
Compiler	gcc-8.2.1

Table 3.2: Server used for RocksDB benchmarks.

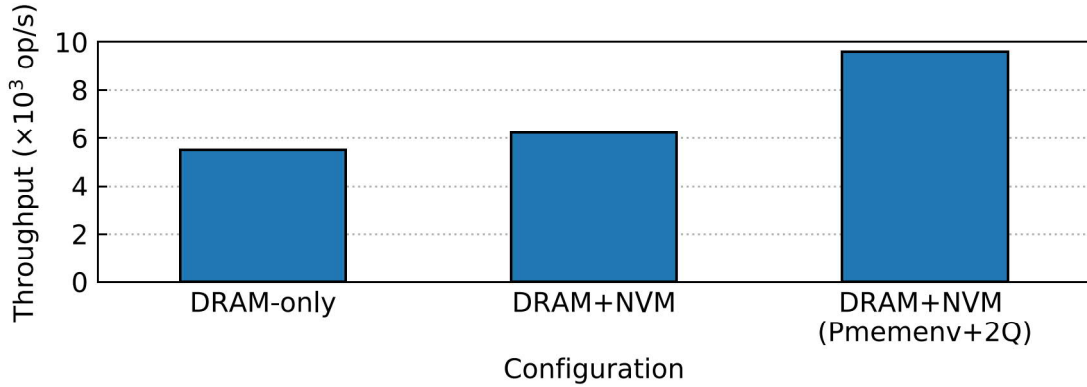


Figure 3.12: Throughput achieved by a single thread in a read-only workload.

block cache is used, only the regular DRAM *block cache* with a size of 10 GB. In the second scenario, we reduce the size of *block cache* from 10 GB to 2 GB, and redirect the costs to a *persistent block cache* in NVM. For that, we assume a DRAM price of 11,70 US\$/GB and an NVM price of 5,40 US\$/GB⁴, which leads to a *persistent block cache* with capacity of 17 GB. The goal is to have the same cost in both scenarios, so we can focus on comparing the performance achieved in each case.

Figure 3.12 shows the throughput achieved by a single thread in three scenarios. The *DRAM-only* represents the scenario previously described with a *block cache* of 10 GB and no *persistent block cache*. *DRAM+NVM* represents the scenario in which a *block cache* of 2 GB is used together with a *persistent block cache* of 17 GB, but the *persistent block cache* is the one provided by RocksDB, i.e., it is not NVM-aware and accesses NVM through the regular file system interfaces, just like a regular SSD. Finally, the *DRAM+NVM (Pmemenv+2Q)* represents the same memory distribution, but with our custom implementation of the *persistent block cache*, which uses *Pmemenv* and the 2Q placement policy. We see in the *DRAM+NVM* case that, while simply employing NVM in the regular *persistent blocks cache* already yields improvements, those are rather small. On the other hand, making the *persistent block cache* NVM-aware through the *Pmemenv* and 2Q significantly improves the throughput in comparison to the *DRAM-only* approach. Figure 3.13 extends this evaluation to a scenario with multiple threads, showing that the throughput improvements still hold.

Finally, we evaluate the benefits of NVM in terms of a *warm* startup. Whenever RocksDB is started (either after a normal shutdown or a failure), requests are issued and the *block*

⁴A single DRAM module of 128 GB costs approximately 1500 US\$, while an Intel Optane DCPMM with the same capacity is estimated to cost 695 US\$ [Aco19].

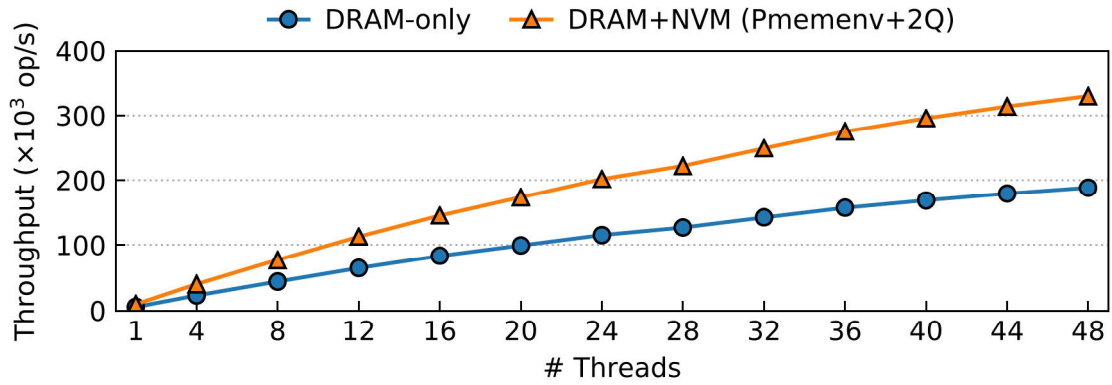


Figure 3.13: Throughput achieved by multiple threads in a read-only workload.

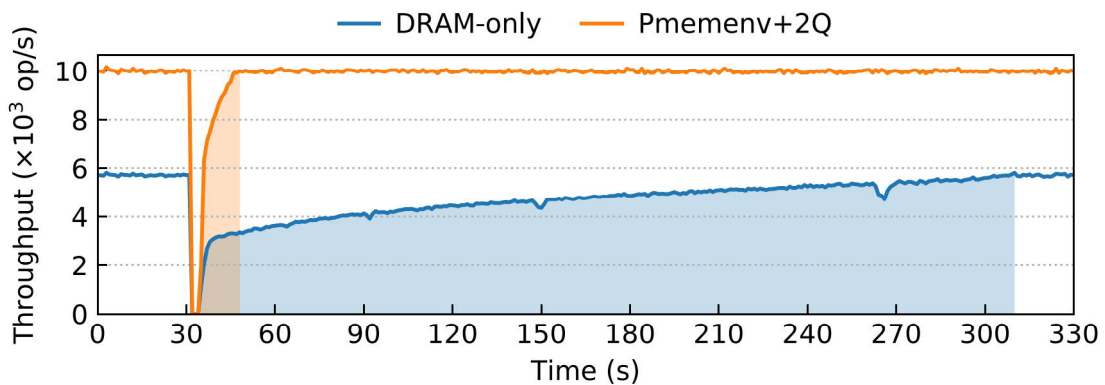


Figure 3.14: Time elapsed to warm up the block cache and reach the peak performance.

cache is slowly warmed up until it is completely populated, at which point the system achieves its peak performance. However, the blocks in the *persistent block cache* survive across shutdowns and can be accessed immediately, enabling a faster peak-performance startup. Figure 3.14 shows this scenario, with the throughput in the Y axis and the elapsed time in the X axis. The two scenarios depicted are the *DRAM-only* with a *block cache* of 10 GB of DRAM and the *Pmemenv+2Q* with a *block cache* of 2 GB of DRAM and *persistent block cache* of 17 GB of NVM. Initially we see the peak performance of both systems for the read-only workload with a single thread. We then shutdown and restart both systems at the 30 s mark. The filled areas below the curves indicate the time elapsed from the startup until the peak performance is reached. The *Pmemenv+2Q* takes only 15 s, since only the 2 GB of the *block cache* have to be populated and most requests can be served immediately from the *persistent block cache*. Meanwhile, even if the throughput reaches acceptable levels in the first few seconds after startup, the *DRAM-only* requires 280 s to finally reach its peak throughput. Therefore, the conclusion is that the concepts introduced by *Pmemenv* and *2Q* can be directly applied to the *persistent block cache* of RocksDB, allowing not only a higher throughput for the same cost, but also enabling the system to achieve its peak throughput much sooner after a restart.

4

B+TREES

The B+Tree is one of the most traditional storage management architectures. In comparison to an LSM, a B+Tree does not require regular garbage collection or other sorts of internal reorganization, since records are commonly update in place. This leads not only to better a performance of point lookup and range scan operations, but also to a more robust behavior, as the system does not slowdown because of merge processes lacking behind. These are some of the characteristics that make B+Tree architectures attractive in the context of relational databases. However, the update-in-place nature of B+Trees makes it impractical to leverage NVM as the main persistent storage, since data can easily be corrupted. Previous attempts to leverage NVM on B+Trees give up at least one of three aspects: *NVM byte-addressability*, *NVM persistency*, or *B+Tree update-in-place strategy*. Therefore, in this chapter we investigate how all three of them can be achieved by proposing a buffer pool architecture, discussing implementation details, empirically evaluating the overhead, and elaborating on the performance expectations of the proposed approach. The goal of this chapter is to answer the following questions:

- How can B+Trees leverage both persistency and byte-addressability of NVM?
- How to handle corruptions of update-in-place strategies?

4.1 B+TREE AND NVM

Due to its ubiquity in database systems [Com79], previous works have investigated opportunities to leverage NVM in the context of B+Trees. The main challenge stems from two B+Tree properties: records are updated in-place and records are stored in sorted order within each node (or “page”). Updating records in B+Tree nodes in NVM while respecting these properties can lead to data being corrupted and lost. This is a similar scenario to the example in Figure 1.1 in Section 1.2, in which the sorted array can be seen as a B+Tree node. We classify the works that address this challenge in three categories illustrated in Figure 4.1 and discuss them in the following.

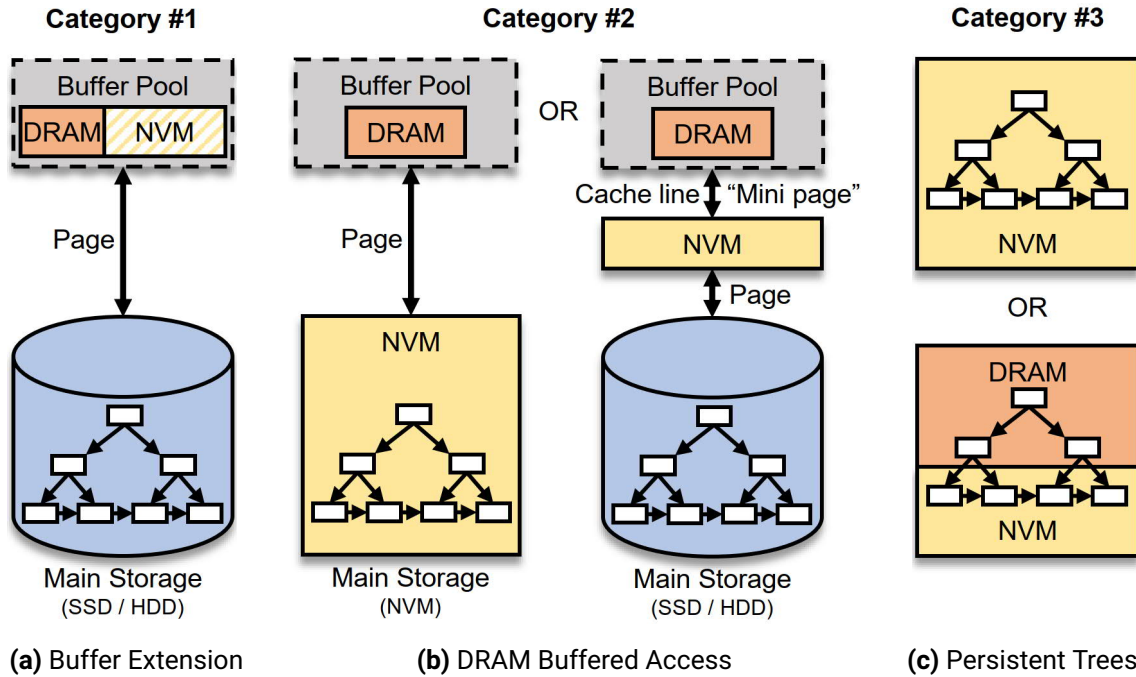


Figure 4.1: The three categories of B+Tree architectures proposed by previous works.

4.1.1 Category #1: Buffer Extension

Systems organize their records in a B+Tree format on persistent media (traditionally HDD and SSD), with each node being represented by a page. Since pages need to be read from storage to memory to be accessed, a portion of DRAM is used as a buffer pool to hold these pages. In this scenario, previous works propose to leverage the lower costs and higher density of NVM to extend the buffer pool capacity [CJY15, OCXH14, LJW⁺19], as shown in Figure 4.1a. The observation is that a larger buffer pool enables more pages to be on byte-addressable media and, therefore, avoid expensive I/O to traditional storage devices. While not often discussed, this scenario can also benefit from NVM-aware buffer pool policies to properly place pages either in DRAM or NVM, such as the 2Q policy presented in Section 3.4. Furthermore, some of the works in this category also focus on *wear leveling* and reducing *write amplification*, which are concerns similar to the ones in the early days of flash memory, but not often considered in the context of NVM [OCXH14, LJW⁺19].

Since the buffer pool (or a part of it) resides in NVM, buffered pages are preserved across system restarts and, therefore, could be accessed instantly. However, as previously discussed, the pages in NVM can be corrupted if a failure occurs during an update. Therefore, a pessimistic assumption is made by discarding all contents in NVM during restart and treating it as a fresh buffer pool by reading the original pages from the main storage device after recovery procedures. In other words, the persistency of NVM is not leveraged (represented by the striped lines of NVM in Figure 4.1a). This is the main weakness of these approaches. The main advantages are the reduced costs (by employing cheaper NVM instead of DRAM) and low implementation effort (since NVM is treated like DRAM).

Finally, we note that a similar behavior to the works in this category can be achieved at the hardware level, by configuring the DCPMM in *memory mode*, as discussed in Section 2.1.2. However, in such case, NVM is not exposed to the application, thus preventing the buffer manager from applying custom replacement and migration policies.

4.1.2 Category #2: DRAM Buffered Access

As shown in Figure 4.1b, rather than extending the buffer pool, works in this category assume that the buffer pool resides solely in DRAM and NVM is used either as the main storage device (left-hand side of Figure 4.1b) [Kim15, PWGB13] or as a caching layer between DRAM and the main storage on HDD or SSD (right-hand side of Figure 4.1b) [vRLK⁺18]. The common point between these two approaches is that the system never accesses NVM directly, but it copies pages to the DRAM buffer pool before operating on them. As a result, the system has full control over page propagation to NVM, since they are only done explicitly by copying a page from DRAM to NVM, similar to the regular interaction between memory and storage. This control over writes to NVM enables the system to take measures, such as *copy-on-write*, to prevent pages from being corrupted by partial writes from DRAM to NVM, as in the case where not all the affected cache lines are correctly persisted. If NVM is used as a caching layer (right-hand side of Figure 4.1b), keeping the pages consistent enables the system to directly read them from NVM after a restart, thus avoiding the I/O to the main storage and enabling a warmer restart.

Leveraging the persistency of NVM is the main advantage of the approaches in this category. On the other hand, the main disadvantage is the unnecessary data movement between DRAM and NVM, since pages in NVM could be directly accessed by the CPU. An alternative employed by van Renen et al. [vRLK⁺18] is to leverage the fine-granularity access of NVM rather than page-based access. These fine-grained accesses can be either the size of a single cache-line (typically 64 B) or multiple cache-lines (also referred to as “mini-pages”). While this reduces the unnecessary movement of data, it does not eliminate it completely and further complicates memory management in the buffer pool, since it must manage pages of multiple sizes. In other words, the byte-addressability of NVM is not fully leveraged, according to our definition of byte-addressability discussed in Section 1.1.

4.1.3 Category #3: Persistent Trees

We have previously discussed works in this category in Section 2.3. As shown in Figure 4.1c, some variants of persistent B+Trees place all nodes in NVM [CJ15, YWC⁺15, ALML18], while hybrid variants place only leaf nodes in NVM and inner nodes in DRAM, under the observation that inner nodes can be rebuilt from leaf nodes during restart [OLN⁺16]. Nevertheless, in both cases nodes in NVM are updated directly, since these B+Trees are plain data structures with no buffer management or more traditional recovery procedures. Consequently, simple operations, such as the insertion or update of a record, might cause data corruption. To prevent that, the typical approach taken is to ensure that the data structure is consistent at all times by carefully enforcing the order of writes through hardware instructions such as SFENCE, CLFLUSHOPT and CLWB. These hardware instructions introduce not only overhead, but also additional complexity on the implementation, compared to classical textbook implementations of data structures. Furthermore, persistent B+Trees still have to prevent records from being updated in-place, as there is no way to protect against partial writes. Therefore, *copy-on-write* strategies, such as *shadow paging* [Lor77], are used when updating the B+Tree nodes. An alternative is to leverage the fine-granular access of NVM and keep records within nodes unsorted, in such a way that they are always written to a new position within the same node. A validity bitmap in the header of the node

Category	Persistency	Byte-Addressability	Update-In-Place
Buffer Extension	No	Yes	Yes
DRAM Buffered Access	Yes	No	Yes
Persistent Trees	Yes	Yes	No
Persistent Buffer Pool with Optimistic Consistency	Yes	Yes	Yes

Table 4.1: Categories of B+Tree architectures that leverage NVM.

can then be atomically updated to indicate which records are valid. Even if this avoids the overhead of making a complete copy of the page (as it is the case in *shadow paging*), it still breaks the update-in-place property, even if in a smaller granularity. One could argue that, as contradicting as it may sound, these alternatives employ *shadown “paging”* on the record-level, rather than on the page-level. Therefore, while persistent B+Trees leverage both the persistency and byte-addressability of NVM, they introduce additional complexity and negatively impact operations that rely on the sorted order of records, such as tree traversals and sorted range-scans.

4.2 PERSISTENT BUFFER POOL WITH OPTIMISTIC CONSISTENCY

While the categories previously discussed have advantages, they do not provide three important properties at once: *persistency*, *byte-addressability*, and *update-in-place*. Table 4.1 summarizes the approaches and their characteristics. We propose a new approach to achieve all these properties: *Persistent Buffer Pool with Optimistic Consistency*.

Our approach is motivated by the following observations. First, like *Category #1* and *Category #2*, we consider that a buffer pool is the perfect abstraction for managing the interaction between all devices in the storage hierarchy. *Category #3* does not consider other devices (such as SSD and HDD) and does not enable dynamic placement of data (such as migrating hot nodes to DRAM when the workload changes). Second, like *Category #1* and *Category #3*, directly accessing NVM is required to fully leverage its byte-addressability and treat it as more than faster storage. Therefore, the unnecessary data movement introduced by *Category #2* is not optimal. Third, like *Category #2* and *Category #3*, data persisted on NVM should be leveraged during system restart. Ignoring persisted data and starting from a clean state, as in *Category #1*, is simply using NVM as cheap memory.

We consider that NVM is not only cheap memory and not only faster storage, it is actually both and should be treated accordingly. On a high level, we propose to use NVM to extend the buffer pool (similar to *Category #1*) but still directly accessing it (like *Category #3*) and leveraging persisted pages upon restart (like *Category #2*). Rather than preventing data corruption by obsessively enforcing a fine-grained consistency for every write to NVM, we just “hope” that corruption will not happen¹, but still put in place mechanisms to detect and recover them during restart. As a result, NVM is treated as both memory (during runtime) and storage (during recovery).

¹Hence “*optimistic consistency*”.

4.2.1 Architecture and Assumptions

Figure 4.2 shows an overview of the architecture we propose for implementing the *Persistent Buffer Pool with Optimistic Consistency*. The main assumption is that the B+Tree is used in a transactional environment and accessed through a buffer pool. We also assume a **no-force/no-steal** strategy [HR83]. The *no-force* implies that pages in the buffer pool are not flushed to the main storage at commit time. Therefore, durability is guaranteed by physiological *write-ahead logging* (WAL), as defined by the ARIES protocol [MHL⁺92]. The *no-steal* entails that pages in the main storage will not contain updates made by loser transactions. This is possible under the observation that the buffer pool capacity can be significantly increased with NVM, up to the point where the complete working set of pages fits in the buffer pool and no pages need to be evicted to the main storage mid-transaction. This simplifies recovery, as undo log records are only required during normal processing (in case of transaction abort) and can be discarded afterwards, without the need of being written to the WAL. Many systems implement the *no-force/no-steal* strategy, usually by employing transaction-private redo/undo log buffers [DKO⁺84, MWMS14, TZK⁺13, SGH18]. Finally, we assume modern transactional recovery techniques based on WAL [GGS16, Sau17], such as *instant recovery* and *instant restart*, with **single-page recovery** being the main requirement [GK12]. Further buffer management techniques, such as *pointer swizzling* [GVK⁺14] and low overhead replacement policies [LHKN18] are desired, but not required. The focus of our approach is handling corruption that might occur when a failure happens while updating pages directly in the NVM portion of the buffer pool.

As Figure 4.2 shows, the buffer pool is extended with NVM. The ratio between DRAM and NVM can be configured by the user, with one extreme being *DRAM-only* (faster runtime, higher price, slower recovery) and the other extreme being *NVM-only* (slower runtime, lower price, faster recovery). The WAL is optionally stored in NVM for better performance, following approaches proposed by related work [CKKS89, AJ89, FHH⁺11, GXH⁺11, HSQ14, WJ14]. For simplicity, we do not elaborate on which alternative is used, as this is orthogonal to the rest of the architecture. We also note that for reducing costs, the system should periodically migrate log records from the WAL in NVM to a *log archive* on cheaper devices, such as SSD, HDD, and tape. For better performance, SSD is also used for the *main storage*.

During normal processing, a page to be updated will be either on the main storage (SSD) or in the buffer pool (NVM or DRAM). Figure 4.3 shows the life cycle of a page. In case the page is on SSD, it is loaded to the NVM portion of the buffer pool. In case the page is in DRAM, there is a *hit* and the page is updated normally. If the page is in NVM, one of two actions can occur. First, the page can be identified as “hot” by a placement policy, such as the 2Q discussed in Section 3.4, in which case it is copied to DRAM and updated there. Second, the page can be simply “warm”, in which case the update is done directly to NVM. More elaborated policies can be explored to further improve page placement, such as deciding if a page on SSD should be read to NVM or directly to DRAM, or if a page evicted from DRAM should be discarded or copied to NVM. To focus on a simple presentation, we do not discuss these policies in detail, however they have been discussed by related work [APM19, vRLK⁺18] and are an interesting direction for future work.

A solution to avoid corruption when updating the page directly in NVM is to employ similar techniques to *Persistent Trees* by combining *copy-on-write* techniques with the hardware instructions *SFENCE*, *CLFLUSHOPT* and *CLWB*, as discussed in Section 4.1.3. This has two

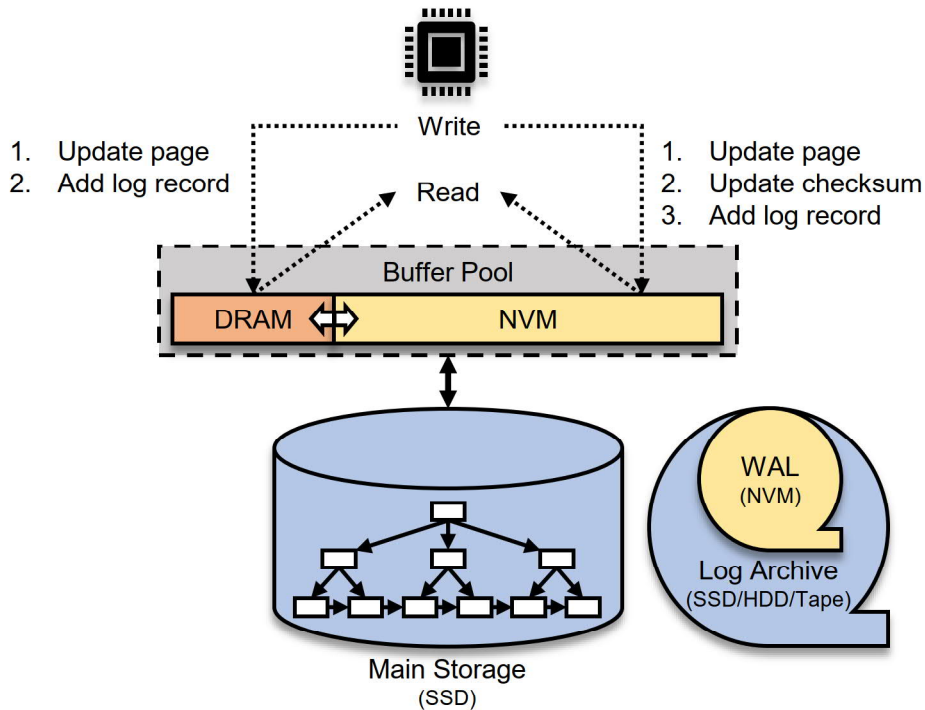


Figure 4.2: Architecture overview of our proposed persistent buffer pool. The user can tune the ratio between DRAM and NVM used by the buffer pool. Reads and writes from/to the buffer pool are done practically in the same way, not mattering if the page resides on the DRAM portion or the NVM portion. Furthermore, pages persisted in the NVM portion are leveraged for a warm restart, thus enabling a faster peak-performance recovery.

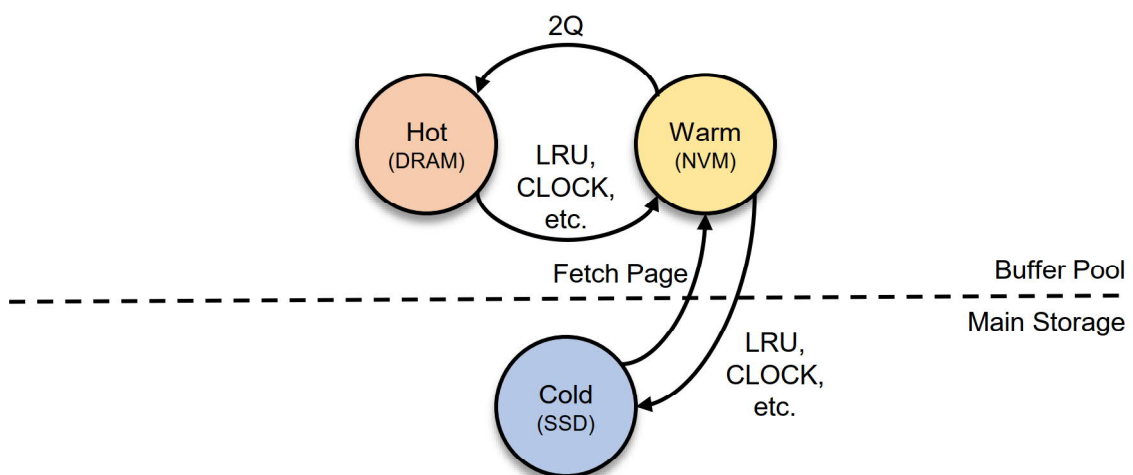


Figure 4.3: Typical life cycle of a page that migrates through different devices.

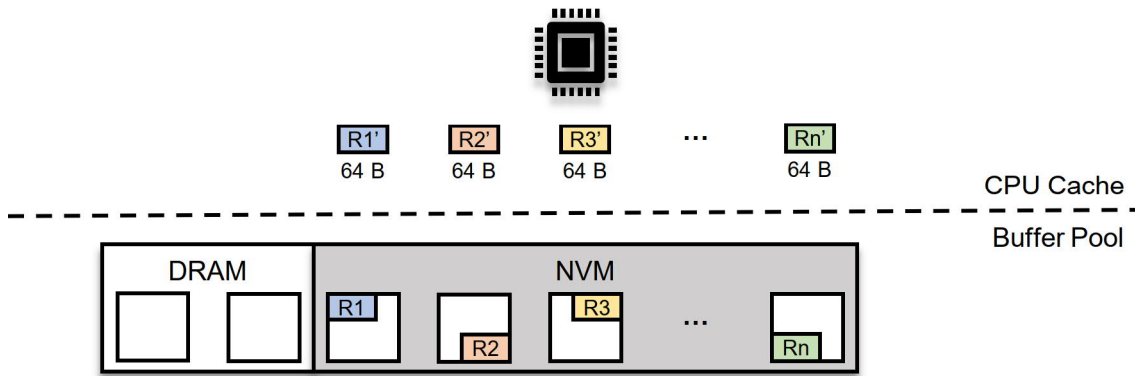


Figure 4.4: A single record the size of a cache line (64 B) of each page is updated. The updated records are lost if there is a failure. Pages in the NVM portion of the buffer pool are corrupted even if the transactions committed.

major drawbacks. First, it does not only break the update-in-place property of pages in NVM (as previously discussed), it also potentially requires pages in NVM to have a different format and be treated differently than pages in DRAM. For example, regular pages in DRAM can keep records sorted, while pages in NVM are log-structured and have an additional atomic bitmap to validate updates. This complicates buffer management and introduces additional overhead, as pages must be converted from one format to another whenever they are moved between devices. While the overhead of converting between page formats might be negligible when compared to the cost of SSD I/O, this overhead becomes relatively higher when compared to the cost of copying a page between NVM and DRAM. The second drawback is that writes to NVM cannot be amortized by the CPU caches, since updated cache lines must be eagerly flushed with `CLFLUSHOPT` and `CLWB`. This exposes writes to the higher latency of NVM and incurs an even higher runtime overhead. These drawbacks arrive as a consequence of protecting the system from corruption at all costs. We propose to go in an opposite direction by acknowledging that corruptions are inevitable and embrace them as part of our buffer pool design.

4.2.2 Embracing Corruption

Updating pages directly in NVM without enforcing any kind of consistency will lead to pages being corrupted eventually. Allowing these corruptions to happen might sound unfeasible in the real world. However, this approach is motivated by three observations.

First, **failures are rare**. This is important, considering that corruptions can only occur if a *partial-write* happens as a consequence of a failure, such as a power outage or an abrupt system crash due to software errors. If systems fail too often, the issue must be addressed at a different level, such as replacing the hardware. Therefore, the overhead of neurotically enforcing the consistency of each small update to a page in NVM is undesirable.

Second, **the number of pages that can be corrupted is bound**. This claim is backed by a thought experiment. A corruption happens if we update a record in a page in NVM and this update is lost after a failure because it was not evicted (partially or completely) from the

volatile CPU cache. In other words, the update was never persisted to NVM. Assume the B+Tree nodes are 4 kB and that a single record the size of a cache line (64 B and also the transfer unit between CPU cache and NVM) is updated on each node in NVM. Figure 4.4 illustrates this scenario. Considering the cache size of a modern CPU is approximately 40 MB², the maximum number of updated records that the cache can hold is given by:

$$\frac{40 \text{ MB}}{64 \text{ B/record}} = 655\,360 \text{ records (pages)}$$

Since we assumed that each updated record belongs to a different page, that is also the maximum number of pages that were corrupted by lost updates after the failure. Note that CPU caches are *multi-way set associative* and holding a single cache line for each page is unlikely. In reality, cache lines are evicted much sooner by the CPU and persisted back to NVM due to associativity conflicts, leading to fewer pages being corrupted. Nevertheless, we make a pessimistic and conservative assumption to strengthen our argument. Considering that the smallest DCPMM commercialized by Intel is 128 GB, if it is completely used by the buffer pool, the total number of 4 kB pages it can hold is given by:

$$\frac{128 \text{ GB}}{4 \text{ kB/page}} = 33\,554\,432 \text{ pages}$$

In such a scenario, a maximum of 2% of the pages in the buffer pool can be corrupted in the worst case (655360 of 33554432). This is generalized by Equation (4.1), which states that the percentage of corrupted pages in the worst case is proportional to the ratio between the size of the CPU cache and the size of NVM used by the buffer pool. Since the size of CPU caches is much smaller and more constant³ than the size of NVM, in most scenarios the maximum number of corrupted pages will be a small fraction of all the pages in the buffer pool. Figure 4.5 further aids in visualizing this relation by showing the percentage of pages that can be corrupted in the worst case (Y axis) when increasing the size of NVM dedicated to the buffer pool (X axis). As in the previous thought experiment, we assume a CPU cache of 40 MB and pages of 4 kB.

$$\# \text{ Corrupted Pages} \approx \frac{\text{Size of CPU Cache}}{\text{Size of NVM Buffer Pool}} \quad (4.1)$$

As seen, the relative number of pages that can be corrupted in the worst case drastically drops for large NVM capacities. Such large buffer pools are common, even in the context of DRAM, given the ever-increasing demands of data processing. However, it is worth noting that, even if the number of pages that can be corrupted is low, it can still be much higher than acceptable industry standards⁴. Finally, this leads to our third and most important motivation: **corrupted pages can still be detected and recovered**. The main insight is that the corrupted pages are in the buffer pool, not in the main database storage. In a transactional environment, the atomicity and durability are already guaranteed by WAL, therefore issuing CLFLUSHOPT or CLWB after each update to a page in the buffer pool is not necessary, as the log serves as the single source of truth. In the following we elaborate on the techniques used for detecting and recovering corruptions.

²For example, the Intel Xeon Platinum 8260L CPU has 35.75 MB of last-level cache (L3).

³The capacity of DRAM and NVM can be increased by acquiring more and/or larger modules. Meanwhile, expanding the capacity of the CPU caches requires acquiring a completely new CPU.

⁴As an example, Amazon S3 is designed for 99.99999999% (11 9's) of data durability.

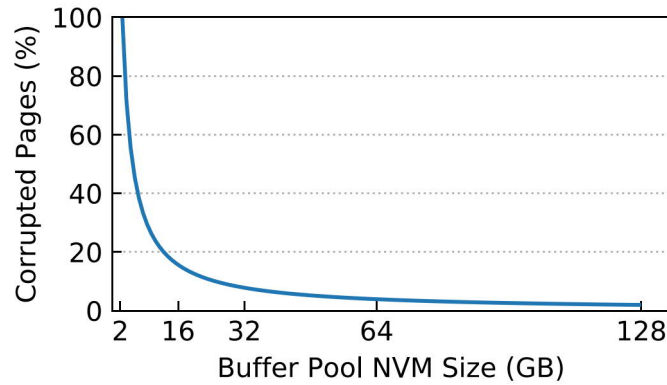


Figure 4.5: Percentage of pages in the buffer pool that can be corrupted in the worst case.

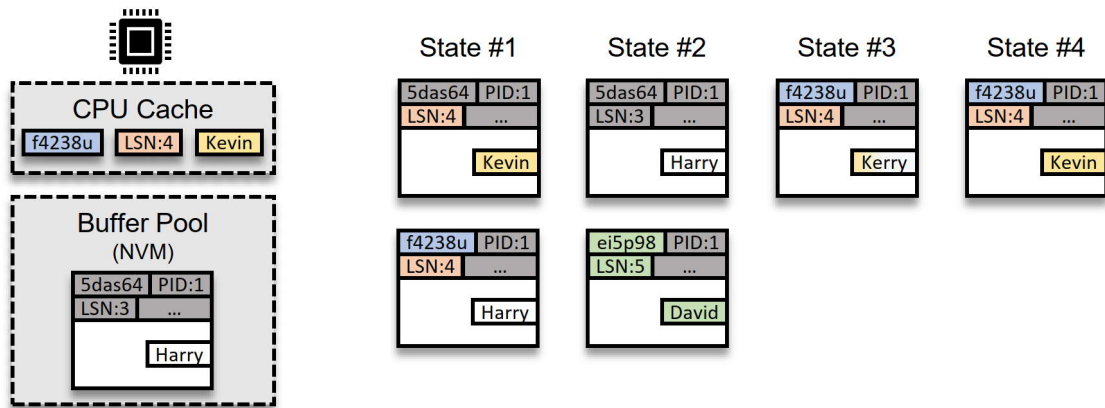
4.3 DETECTING CORRUPTIONS

We saw that pages in the NVM portion of the buffer pool can be corrupted because updates might not have been properly persisted. Consequently, during restart after a failure, these pages will still be in the buffer pool, but their state is unknown. They can either be consistent and ready to be accessed, or corrupted. Since we want to leverage the persistency of NVM, we must identify the state of a page prior to accessing it. Fortunately, this can be done by leveraging the existing *checksum* that many database systems store in the header of each page, thus not requiring radical changes to the system architecture.

Traditionally, the checksum of a page is calculated and updated when the page is written from volatile memory (DRAM) to persistent storage (SSD/HDD). When a page is fetched from persistent storage to the buffer pool, its checksum is calculated again and compared to the checksum stored within the page. The goal is to detect page corruptions caused by media failures, such as a bad HDD sector. In the case of NVM, the insight is that the corruption that may happen due to non-persisted cache lines can be generalized to a media failure, just like in the case of SSD and HDD. Therefore, we rely on the checksum already present in the page headers when updating pages in NVM.

During runtime, whenever a page in the NVM portion of the buffer pool is updated, the system calculates the new checksum for the whole page and updates it. This is the only difference to pages in the DRAM portion, which have their checksum calculated and updated only when they are flushed to persistent media, rather than for every update. This difference is also shown in Figure 4.2. In addition to the checksum, other fields in the page header are the unique page identifier (PID) and the page log sequence number (pLSN), as in the ARIES algorithm [MHL⁺92]. For every update to the page, the LSN is updated to correlate the state of the page with respect to the logged update. It is worth noting that neither the updated record, the new LSN, or the new checksum are eagerly flushed from the CPU cache back to the page in NVM. Eventually the CPU will naturally evict the affected cache lines and they will be persisted. This is an important advantage, as it allows these updates to be amortized by the CPU cache and not be exposed to the higher NVM latency.

During restart after a failure, recovery routines take place, starting by *log analysis*. During the *log analysis* phase, the WAL is scanned starting from the last checkpoint towards its



(a) Update to a page in NVM

(b) Possible states after a failure

Figure 4.6: The states a page may be found in the NVM portion of the buffer pool during restart. The dark gray area indicates the header of the page, which contains the checksum, PID, and pLSN, among other fields.

end. At the end of *log analysis* we know what was the state of the database right before the crash, such as dirty pages and active transactions. In the context of our proposed buffer pool architecture, the most important information retrieved during log analysis is the eLSN (expected LSN) of each page, which is the LSN of the last committed update. After *log analysis*, we assume modern *instant recovery* takes place [GG16]. The system starts to accept requests right after *log analysis* and pages are recovered on-demand the first time they are accessed. If a page being accessed was on the DRAM portion of the buffer pool, it was completely lost and it must be recovered by retrieving an older version of the page from the main storage and replaying the relevant log records to it. However, in the case a page accessed is discovered in the NVM portion of the buffer pool, it might be in a consistent state and it could be accessed right away, without requiring any further recovery. In order to decide the state of such page, the checksum of the page is calculated and compared to the checksum stored in the page header. The result of this comparison, together with the LSN stored in the page header, defines the state of the page.

4.3.1 Possible States

Figure 4.6 shows a simple scenario with all the possible states a page in NVM may be found during restart. Figure 4.6a shows a single record of a page in NVM being updated from "Harry" to "Kevin". As previously mentioned, the system increments the pLSN and calculates and updates the new checksum. Since there are no guarantees of which cache lines were persisted, a failure might happen at any point and the page can be found in the states shown in Figure 4.6b during restart. Figure 4.7 shows the decision tree to determine the state of the page. In the following, we elaborate on each one of these states

State #1: Corrupted Page

In this case, not all updated cache lines were evicted from the CPU cache. Figure 4.6b shows two possible scenarios. First (top), only the updated record and the new pLSN were

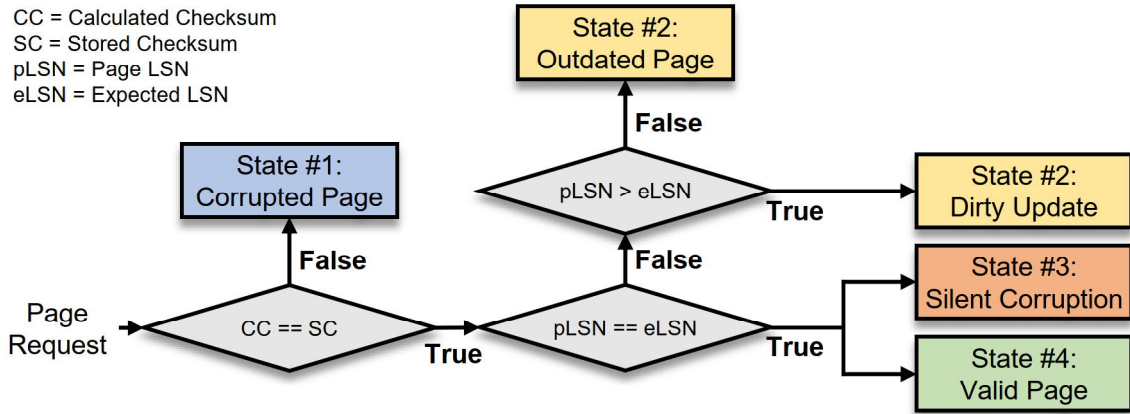


Figure 4.7: Decision tree to determine the state of a page in NVM during restart.

evicted. Second (bottom), only the new checksum and the pLSN were evicted. While other situations are possible, the relevant part is that the calculated checksum of the page will not match with the checksum stored within the page, therefore the page is corrupted.

State #2: Outdated Page & Dirty Updates

In this case, the calculated checksum and stored checksum match. However, while this guarantees that the page is not corrupted, it does not guarantee the correct version of the page. Therefore, the pLSN stored in the page must be compared to the eLSN retrieved during *log analysis*. They will not match in two situations. First (top), the pLSN is lower than the eLSN, since none of the updated cache lines were evicted from the CPU cache, meaning that the page is in a physically consistent, but **outdated** state. Second (bottom), the pLSN is higher than the eLSN, since all updated portions of the page were evicted from the CPU cache, but the transaction that made these updates did not commit before the failure, therefore the page is physically consistent but contains **dirty updates**. This distinction is relevant because, while in both cases the page cannot be accessed and must be recovered, their recovery procedure is slightly different, as discussed later in Section 4.4.

State #3: Silent Corruption

In this case, not all updates were evicted from the CPU cache, represented by only the first two characters of the updated record being evicted, leading to “Kerry”, which is a corrupted state. However, it could be that the calculated checksum of this corrupted state matches the checksum stored in the page due to hash collisions in the checksum algorithm. This is known as a *silent corruption*, as the checksum cannot detect it. This is a worst-case scenario, as data loss will be unnoticed. However, this is a common trade-off for checksum-based approaches that are utilized at many levels of the hardware and software stack. The critical point is the probability of the worst-case occurring. As we will show in the following, the probability is lower than the probability for silent data loss in other scenarios that are accepted in practice. Even considering the low probability, one could, in pedantic scenarios, analyze the whole WAL to detect silent corruptions. However,

the probability that reading the WAL leads to an error might be higher than having a silent corruption in the NVM portion of the buffer pool.

We note that the *calculated checksum* of the page is only compared to the checksum currently stored in the page header (*stored checksum*). This means that if the *calculated checksum* collides with any checksum that the page had in the past, this will not be a problem, as long as it is different from the current *stored checksum*. A checksum of 64 bit has 2^{64} distinct values. Assuming a hash function that uniformly distributes the results across the domain, the probability of the *calculated checksum* being the same as the *stored checksum* in a single page is given by:

$$P(\text{collision for a single page}) = \frac{1}{2^{64}} \approx 5.4 \times 10^{-20}$$

Therefore, the inverse of that is the probability of having no collision in a single page, which is given by:

$$P(\text{no collision for a single page}) = 1 - \frac{1}{2^{64}}$$

As previously mentioned in Section 4.2.2, the number of pages that can be corrupted is bound and a CPU cache of 40 MB can corrupt up to 655360 pages per failure. Therefore, the probability of having no collision in any of these pages is given by:

$$P(\text{no collision in any page}) = \left(1 - \frac{1}{2^{64}}\right)^{655360}$$

This leads to the inverse probability, which is the probability of having a collision in at least one page:

$$P(\text{collision in at least one page}) = 1 - \left(1 - \frac{1}{2^{64}}\right)^{655360} \approx 3.5 \times 10^{-14}$$

Considering industry standards that typically require 11 9's of durability (i.e., 10^{-11} probability of data loss) [Ama06], the probability of a silent corruption for each crash is acceptable. To further strengthen the argument, the probability of at least one page being corrupted in n pages is given by:

$$P(\text{collision in at least one page in } n \text{ pages}) = 1 - \left(1 - \frac{1}{2^{64}}\right)^n$$

We can now calculate how many pages need to be considered as events to have a 50% chance of a corruption. The following equation shows how many pages have to be corrupted:

$$0.5 = 1 - \left(1 - \frac{1}{2^{64}}\right)^n, n = 1.28 \times 10^{19} \text{ pages}$$

Based on the assumption that 655360 is the number of pages that can be corrupted per failure in the worst case, the number of failures that need to happen to reach a 0.5 probability of at least one page being corrupted is:

$$\frac{1.28 \times 10^{19}}{655360} \approx 1.95 \times 10^{13} \text{ failures}$$

It is worth noting that hash collisions are only a problem if they happen exactly before a failure, not during normal runtime. If we assume that 10 failures happen in a year, it takes 10^{12} years for a single server to be exposed to a 50% chance of a silent corruption. Considering estimations that modern cloud providers have fleets of approximately 1 million servers, it would still take 10^6 years. Finally, these probabilities are a pessimistic and hypothetical worst-case scenario. In practice, the assumed events will not happen, such as the CPU cache holding a single cache line of each page. Therefore, since the probability of silent corruption is smaller than in other cases that are accepted in practice, we transitively conclude that our case is acceptable.

State #4: Valid Page

Finally, in this case all the updated cache lines were evicted from the CPU cache and persisted to NVM. When the page is requested for the first time and discovered in the NVM portion of the buffer pool, the *calculated checksum* will match the *stored checksum* and the *pLSN* will match the *eLSN* retrieved during *log analysis*. Therefore, the page can be accessed without further recovery procedures. For buffer pools with large NVM capacities, most pages are expected to be in this state.

4.4 REPAIRING CORRUPTIONS

This section describes the recovery algorithm of the *Persistent Buffer Pool with Optimistic Consistency*. As previously mentioned, after a failure the *log analysis* phase takes place and once it completes, pages can be requested and are recovered on-demand. Algorithm 4.1 shows the pseudo-code of the two main functions: *Fix* and *RecoverPage*. The *Fix* function is called whenever a page is requested by its *page_id*. In case this is the first access to the page after the failure, the recovery procedure is triggered (line 2-4). Otherwise, the *Get* function (line 7) executes regular buffer management procedures, such as fetching the page from main storage and *pinning* it to memory to prevent early eviction.

The *RecoverPage* function starts by identifying if the page being requested was in the NVM portion of the buffer pool at the time of the failure. This is done by querying an *allocation table* which is stored in a fixed position in the beginning of the NVM space (line

11). The *allocation table* maps page identifiers to the location offset of the respective pages within the pool. In the case the page is not found in NVM, it was potentially in DRAM, thus requiring regular recovery procedures by fetching it from main storage and replaying the relevant log records (line 28-30). Otherwise, a pointer to the page is retrieved from the *allocation table* (line 12) and the page checksum is calculated (line 13) and compared to the checksum stored within the page in order to identify its state (line 14).

As previously discussed, *State #3* is unlikely and therefore ignored. In case the checksums do not match, the page is corrupted and must be recovered (*State #1*). Since we cannot make further assumptions about its state, the algorithm discards the page, fetches an older version from main storage, replays the relevant redo log records, and finally returns the page (line 15-16). If the checksums match, the *pLSN* is compared to the *eLSN*. If they are equal, the page is in the correct version (*State #4*) and can be returned immediately (line 25). If the *pLSN* is higher than the *eLSN*, then the page contains updates made by uncommitted transactions. Given the *no-steal/no-force* strategy of the buffer pool, rolling back changes is not possible since UNDO information is never persisted to the WAL, therefore the page must be discarded, an older version is fetched from the main storage, and the log records are replayed to “roll forward” the page to its most recent and consistent state (line 15-23), like in *State #1*. If the *pLSN* is lower than the *eLSN*, the page is in an old state and missing updates made by committed transactions, thus only requiring the log records generated by those transactions to be replayed directly on it (line 23). This case has the advantage of saving an additional I/O operation by leveraging the page found in NVM rather than fetching an older version from the main storage, thus being potentially faster. Finally, the recovered page is returned (line 25).

The original *single-page recovery* [GK12] was proposed in the context of traditional databases with volatile memory buffer pool (DRAM), persistent main storage (HDD/SSD), and regular database backups. The original proposal consists of repairing corrupted pages on the main storage by fetching an older version of the page from a backup location and replaying the log records. The advantage of the proposed *Persistent Buffer Pool with Optimistic Consistency* is that it relies on an existing and effective recovery technique rather than re-inventing the wheel. *Single-page recovery* is generalized by applying the same concept in a different context: the main storage is to the persistent buffer pool the same thing that the backup is to the main storage.

4.5 PERFORMANCE EVALUATION AND EXPECTATIONS

This section empirically evaluates the main overhead introduced by the *Persistent Buffer Pool with Optimistic Consistency*: calculating checksums. We also discuss and propose alternatives to reduce this overhead. Finally, we elaborate on the end-to-end performance expectations of the the overall system.

Algorithm 4.1: Pseudo-code for the Fix and RecoverPage functions.

```
1 function Fix(page_id)
2   if IsFirstAccess (page_id) then
3     return RecoverPage (page_id)
4   end if
5   // Return a pointer to the page in the buffer pool, potentially
6   // fetching it from the main storage if a page miss occurs
7   return Get (page_id)
8 end
9
10 function RecoverPage(page_id)
11   if IsOnNVM (page_id) then
12     page ← Get (page_id)
13     checksum ← CRC64 (page)
14     if checksum ≠ page.getChecksum() then // State #1
15       page ← Fetch (page_id) // Fetch page from main storage
16       ReplayLog (page) // Apply log records
17       return page
18     else
19       if page.LSN ≠ GetExpectedLSN (page_id) then // State #2
20         if page.LSN > GetExpectedLSN (page_id) then // Dirty update
21           page ← Fetch (page_id)
22         end if
23         ReplayLog (page) // Dirty update or outdated
24       end if
25       return page // State #2 or State #4
26     end if
27   else // Page was in DRAM and was lost
28     page ← Fetch (page_id) // Fetch page from main storage
29     ReplayLog (page) // Apply log records
30     return page
31   end if
32 end
```

Processor	Intel Xeon Platinum 8260L CPU (35.75 MB Cache, 2.40 GHz)
Main Memory	96 GiB DDR4 2666 MHz (6× 16 GiB modules)
NVM	Intel Optane DCPMM 1.5 TiB (6× 256 GiB modules)
Operating System	Linux 5.3.4-3
Compiler	gcc-8.2.1

Table 4.2: Server used for micro benchmark.

4.5.1 Checksum Overhead

To achieve the three characteristics discussed in the beginning of this chapter (*persistence*, *byte-addressability*, and *update-in-place*), we propose calculating the checksum for the whole page for each record inserted, updated, or deleted. This creates an overhead which might not be negligible and has a direct impact in the final performance of the system. The overhead depends on the size of the record being updated. If the record is large and spans a significant portion of the page, the overhead is proportionally lower. In the case of insertions, the sorted order of records in a B+Tree node must be kept, and therefore, in average, half of the page will be accessed anyway for moving other records to accommodate the new one. A similar situation happens with deletions. On the other hand, updating a record does not require other records to be moved, and, therefore, the overhead of calculating the checksum becomes relatively higher. To provide a strong argument, we look at the worst-case scenario of updating a small record. We evaluate and compare different strategies through a micro-benchmark. The micro benchmark consists of measuring the time required for updating a random record in a random page stored in a persistent buffer pool of 10 GB of NVM using a single thread. We calculate the CRC checksum through fast vectorized hardware instructions provided by Intel SSE 4.2 [Int20a]. Table 4.2 shows the configuration of the server used for the micro benchmark.

Three strategies are initially considered. First, “*No Checksum*” simply updates the record in the NVM page. Since consistency is not enforced and there is no way to detect corrupted pages, this implies that all pages in NVM must be discarded after a failure, thus not leveraging the NVM persistence at all. This serves as our baseline, as pages are updated like in DRAM, not requiring any additional work, thus being the fastest way to update a record. Second, “*Copy To DRAM*” consists of copying the whole page from NVM to a DRAM buffer prior to the update. The record is then updated in DRAM, thus avoiding corruption in NVM. We note that “*No Checksum*” and “*Copy To DRAM*” are equivalent to the way pages in NVM are updated in the *Category #1* and *Category #2* discussed in Section 4.1. The *Category #3* is not considered, as the page format and update method are significantly different, requiring out-of-place updates and explicit cache-line flushes. Finally, “*Checksum*” represents our proposed technique and consists of updating the record in NVM, calculating the checksum for the whole page, and writing the new checksum to the header of the page. Figure 4.8 shows the runtime of each approach (Y axis) while varying the size of the record updated (X axis). Different page sizes are also considered (Figures 4.8a to 4.8d). For further clarity, the slowdown of each approach relative to “*No Cache*” is also calculated and shown in Table 4.3.

We make three observations. First, as expected, the “*No Cache*” approach has the same performance in all scenarios, since larger records are likely to still be buffered by the

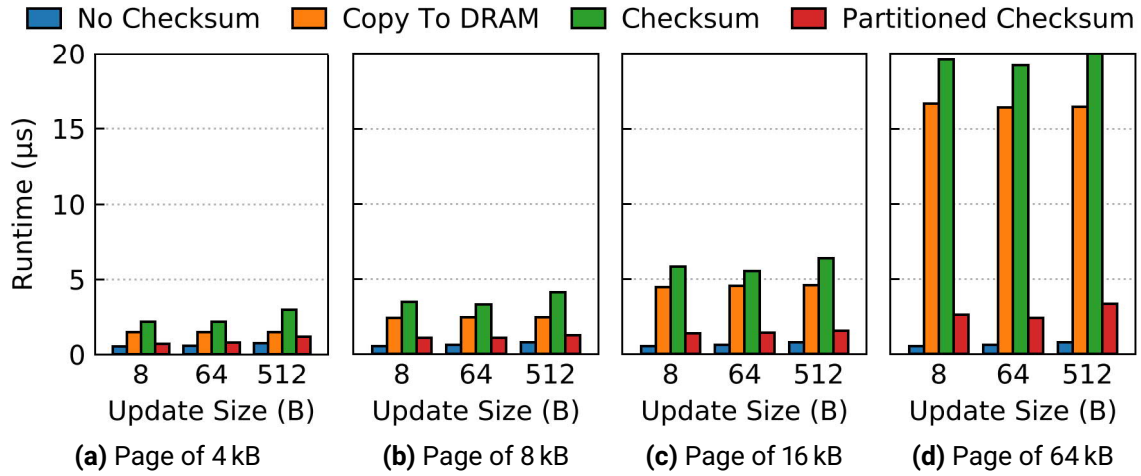


Figure 4.8: Runtime of updating a record in NVM for each approach considered.

Page Size	Update Size	Slowdown Factor		
		Copy To DRAM	Checksum	Partitioned Checksum
4 kB	8 B	2.7	4.0	1.3
	64 B	2.5	3.7	1.3
	512 B	2.0	4.0	1.6
8 kB	8 B	4.5	6.5	2.0
	64 B	4.2	5.7	1.9
	512 B	3.3	5.5	1.7
16 kB	8 B	8.3	10.9	2.6
	64 B	7.8	9.5	2.4
	512 B	6.0	8.4	2.0
64 kB	8 B	31.3	36.8	4.9
	64 B	28.4	33.4	4.1
	512 B	21.8	26.6	4.4

Table 4.3: Slowdown of each approach calculated by dividing their respective runtime in Figure 4.8 by the runtime of updating the record in NVM (“No Checksum”). Lower is better.

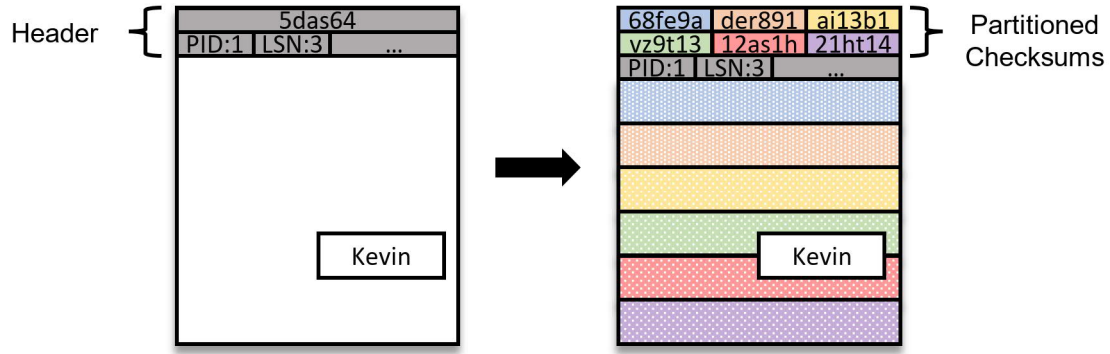


Figure 4.9: Partitioned checksum.

DCPMM. This aligns with previous works that reported that DCPMM employ its own on-chip caches to buffer accesses to the underlying media in units of 256 B [vRVL⁺19, IYZ⁺19]. Furthermore, the size of the page in this case plays no role. Second, both “Copy To DRAM” and “Checksum” become slower for large page sizes, since they require accessing the whole page (either while copying to DRAM or while calculating the checksum). Third and most important, “Checksum” performs worse than “Copy To DRAM” in all cases. This shows that, computing the checksum for the whole page is more expensive than simply copying the page to DRAM. Even if the difference is small, one could argue that the high runtime overhead defeats the purpose of the *Persistent Buffer Pool with Optimistic Consistency*. Therefore, we propose a simple alternative: *partitioned checksums*.

Calculating the checksum for a whole 4 kB page is too expensive if only 8 B are updated. Therefore, the motivation of *partitioned checksums* is reducing the amount of data that has to be read for calculating the checksum. As Figure 4.9 shows, instead of a single checksum for the whole page (left-hand side), the page is logically divided in smaller partitions and each partition has its own checksum (right-hand side). In the scenario in fig. 4.9, if the record “Kevin” is updated, only two checksums (green and red) need to be calculated. This significantly reduces the overhead, as shown by the “*Partitioned Checksum*” results in Figure 4.8 and Table 4.3, in which 8 checksums per page are used. It is worth noting that, while multiple checksums are used, the unit of recovery, repair, and corruption detection is still a page. Therefore, if one of the checksums of a page does not match the checksum calculated for its respective partition, the whole page is considered corrupted.

The number of checksums used for each page can be set as a system parameter. The trade-off introduced in this case is between additional space requirement (more checksums) and higher overhead for updating pages directly in the NVM portion of the buffer pool (less checksums and larger partitions). Table 4.4 shows the space overhead introduced by having 32 checksums of 8 B each ⁵ in the different page sizes previously discussed.

Finally, it is worth remembering that the overhead shown in our evaluation is the worst-case scenario of small updates to a page. Furthermore, this overhead only refers to updating a single record in a page in NVM. This situation is expected to happen sporadically, since a good placement policy, such as the 2Q discussed in Section 3.4, should migrate frequently modified pages to DRAM. Such pages not only avoid the checksum overhead, but also benefit from the lower latency of DRAM. In addition to *partitioned checksums*, future work

⁵This would fit exactly in the reported 256 B buffer unit of DCPMM.

Page Size	Space Overhead	Partition Size
4 kB	6.2%	128 B
8 kB	3.1%	256 B
16 kB	1.5%	512 B
64 kB	0.3%	2048 B

Table 4.4: Trade-off between space overhead and partition size (higher runtime overhead) for multiple page sizes with 8 B and 32 checksums per page.

might as well explore alternatives to reduce the checksum overhead, such as increasing the optimism by calculating the checksum of a page after a certain number of changes. This would introduce a new trade-off between a higher number of corrupted pages (checksums calculated less often) and higher runtime overhead (checksums calculated more often).

4.5.2 Runtime and Recovery

This section discusses the expected end-to-end behavior of the *Persistent Buffer Pool with Optimistic Consistency*. We compare this to traditional ARIES recovery [MHL⁺92], as well as to modern *Instant Recovery* [GGS16]. We note that *Instant Recovery* is not orthogonal to ARIES, but rather it builds on top of it to improve the recovery time by enabling on-demand recovery. Similarly, the *Persistent Buffer Pool with Optimistic Consistency* also extends the techniques introduced by ARIES and *Instant Recovery*. Therefore, the approaches discussed here are not competitors per se, but improvements on top of each other.

Figure 4.10 shows the expected behavior when recovering from a failure. The ARIES system starts by executing transactions until a failure happens, at which point the system is offline and the throughput drops to zero. The three recovery phases take place, indicated by the gray areas. *Log analysis* starts by scanning the recovery log from the last checkpoint until its end. It collects information about the state of the system right before the failure, such as dirty pages and active transactions, which is used as input for the next phases. The *redo* phase replays log records of updates to pages that might have not been properly flushed from the buffer pool. Finally, the *undo* phase rolls back changes made by loser transactions. Once the *undo* phase is completed, the system starts to accept requests and the buffer pool is slowly warmed up until the throughput reaches its original state. If the buffer pool employs a *no-steal* strategy, *undo* is not necessary during recovery.

In the case of *Instant Recovery*, the main difference of the *log analysis* phase is that exclusive locks of loser transactions are reacquired. After *log analysis*, the system starts to accept requests immediately. The *redo* and *undo* phases are triggered on-demand, and therefore they do not necessarily correspond to the gray areas in Figure 4.10. The on-demand *redo* is triggered when a page is requested by a post-failure transaction. The page-by-page *redo* is enabled by keeping log records linked through a per-page logical chain. Similarly, *undo* is triggered when a post-failure transaction tries to acquire a lock previously held by a loser transaction. In this case, the loser transactions is rolled back. *Instant Recovery* increases the availability of the system by starting to process transactions much sooner, reaching its peak performance when the buffer pool is warmed up, the hot pages are recovered, and transactions holding high-contented locks are rolled back.

Finally, we discuss the proposed *Persistent Buffer Pool with Optimistic Consistency* (shown simply as *Persistent Buffer* in Figure 4.10). We assume the same memory budget for all systems. In other words, if ARIES and *Instant Recovery* have a buffer pool of 100 GB of DRAM, we assume that the sum of the DRAM and NVM portions in the *Persistent Buffer* is also 100 GB, with the NVM portion being larger (e.g., 20 GB of DRAM and 80 GB of NVM).

The first expectation is that the peak performance will be reached much sooner after *log analysis*. This is a consequence of pages lingering in the NVM portion of the buffer pool across failures. All pages in DRAM and a few of the pages in NVM will require recovery (as previously discussed). However, since the DRAM portion is smaller, less pages are lost. Furthermore, most of the pages in NVM will be accessible immediately. Therefore, faster “*peak-performance*” recovery is enabled by two factors: **less work to be done during recovery** and **the buffer pool is kept warm across failures**.

The second expectation is that the initial performance of *Persistent Buffer* will be lower than that of the two other approaches. Assuming the same memory budget, the lower performance is a result of the higher NVM latency and the overhead introduced by calculating the checksums. In this case, the performance can be improved by increasing the amount of DRAM and decreasing the amount of NVM in the buffer pool. This ratio between DRAM and NVM dedicated to the buffer pool should be implemented as a system parameter to enable a tunable behavior. Figure 4.11 shows the effects of varying this proportion. Increasing the amount of DRAM will lead to a higher throughput during normal processing, but also will increase the recovery time. On the other hand, dedicating more NVM to the buffer pool will lead to lower throughput but faster recovery. One might argue that failures are rare, and therefore lowering the throughput to the detriment of faster recovery is not a good trade-off. However, there is an additional hidden factor: cost. Since NVM is cheaper than DRAM, one could keep the same buffer pool capacity for lower costs, or increase the buffer capacity for the same price (thus reducing “*page misses*” and expensive I/O to the main storage). Enabling such a customizable behavior to trade between performance and costs is attractive in cloud scenarios, in which customers require higher performance during peak hours, but also want to reduce costs at low-demand periods⁶.

4.6 DISCUSSION

In this chapter, we explored how to leverage NVM in the context of B+Trees. The first contribution is the classification of existing approaches into 3 categories. We observed that all the approaches fail at achieving three characteristics: *persistence*, *byte-addressability*, and *update-in-place*. The second contribution is the proposal of our approach, a *Persistent Buffer Pool with Optimistic Consistency*, to achieve all these goals, thus answering the questions raised in the beginning of the chapter. Like *Category #1* and *Category #2*, we extend the traditional buffer pool infrastructure to manage NVM. Like *Category #3*, we write directly to pages in NVM in a consistent, but optimistic, manner.

The persistent B+Trees in *Category #3* are stand-alone data structures and rely on a “*force*” strategy by eagerly flushing cache lines from the CPU cache to NVM, thus pessimistically

⁶Anecdotally, industry professionals have communicated in private conversations that certain customers are happy to accept trade-offs such as “80% of the performance for 50% of the cost”.

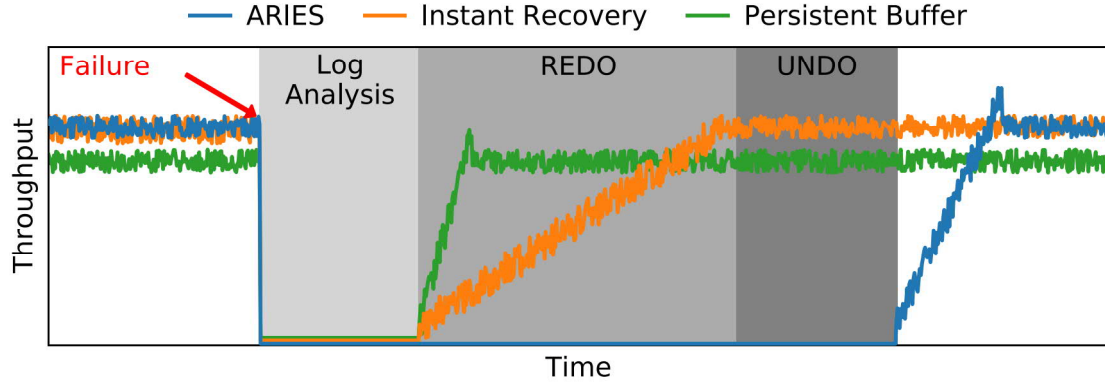


Figure 4.10: Expected behavior during system restart and recovery after a failure.

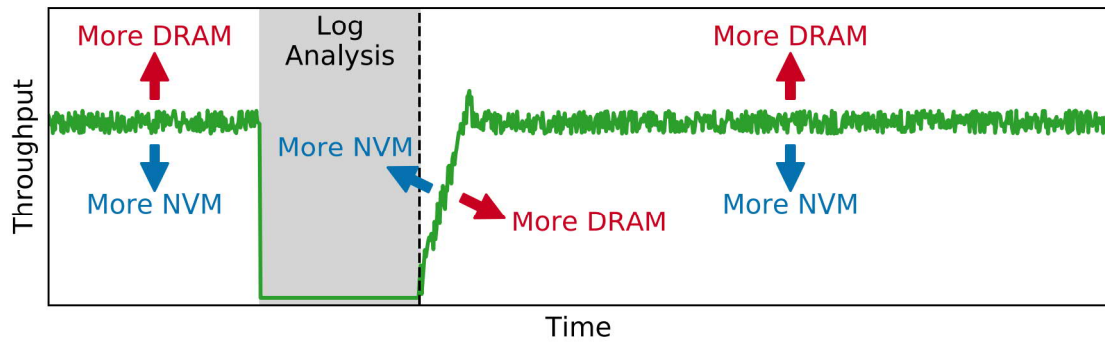


Figure 4.11: Effects of varying the ratio between DRAM and NVM in the proposed *Persistent Buffer Pool with Optimistic Consistency*.

protecting the data from corruption at all times and guaranteeing the consistency of single operations. Our main insight is that B+Trees are more commonly used in a transactional environment in which a group of operations must be atomically executed, rather than only single operations. The usual technique for achieving this transactional semantics is through *logging*, or more precisely *write-ahead logging* (WAL). Therefore, we argue that leveraging WAL makes more sense than simply getting rid of it. We make the observation that WAL is enough to guarantee the consistency of the system, and, therefore, a “force” strategy is not required neither between the buffer pool and the main database storage, nor between the CPU cache and the NVM portion of the buffer pool. In other words, as long as WAL is intact, the system can always be recovered.

Alternative approaches propose to leverage NVM to get rid of WAL, based on a common misconception that logging is the bottleneck of database systems. However, we argue that, in addition to B+Trees, WAL also became ubiquitous in the context of modern database systems. Not only WAL guarantees atomicity and durability, it also enables important business features such as *partial rollbacks* and *database auditing*. In cloud environments, WAL is also used for replication and consensus between distributed servers. Therefore, all these issues would have to be properly addressed in order to completely get rid of the log. This would require a significant effort and changes in modern architectures, going against the general proposal of this work to gradually evolve modern systems to leverage NVM rather than completely rewrite them.

5

INDEX+LOG KEY-VALUE STORES

Many applications employ key-value stores (KVS) in at least some point of their software stack. These KVS are typically non-transactional and one of the most common use-cases is as intermediate caching service in web environments, so clients can avoid expensive network round-trips to a distant data center or server. This is seen in many scenarios, such as caching query results of a database, or caching popular web contents in a *content distribution network* (CDN). Therefore, the data cached is often unstructured and heterogeneous, possibly comprising objects such as relational tuples, a YouTube video, or a web page. Since no assumption about the data can be made, these KVS often employ what is known as an *index+log* architecture. In this architecture, records are organized in a log-structured area to facilitate space management of arbitrarily large records and of structural record changes, such as adding a new attribute to a tuple. A separate index data structure is used to efficiently locate records in the log-structured area. Furthermore, since caching systems require short response times, they are often a single-level system completely in-memory (DRAM). This leads to increased costs and limited capacity. While employing storage devices such as HDD and SSD would address these issues, they would not only require software changes to the single-level architecture employed by these systems, but also incur high performance penalties. The high latencies of HDD and SSD become particularly prohibitive, as enabling short and predictable response times is a main requirement. In this chapter, we present the design of *RStore*, an *index+log* KVS designed to address these issues. *RStore* employs NVM to enable large capacity and lower costs, when compared to DRAM-only KVSs. *RStore* also focuses on achieving short and predictable response times, which are measured in terms of low tail latency. Therefore, in addition to leveraging the lower latency of NVM, we also discuss all of the design decisions that enable *RStore* to achieve its goals. The two high-level questions being investigated in this chapter are:

- How to extend the capacity and lower the costs of *index+log* KVSs?
- How to achieve short and predictable response times in the form of low tail latency?

In comparison to the proposals in Chapters 3 and 4, implementing a new system from scratch, as is the case of *RStore*, might seem to go against the overall principles of this thesis of transforming existing systems to leverage NVM without requiring disruptive changes. However, we note that none of the techniques employed by *RStore* are novel or disruptive per se. Message-passing communication, cooperative multitasking, and log-structured storage are all well-known and established concepts, employed to some extent by other systems. The contribution of *RStore* lies in the combination of these techniques.

5.1 THE CASE FOR TAIL LATENCY

Key-value stores (KVS) comprise a class of systems that cover a wide range of use-cases. They are more often used for caching and storage management in applications like websites, mobile apps, real-time systems, distributed trust, etc. Many of these applications share characteristics that differ from those of more traditional OLTP and OLAP systems:

- Unstructured and heterogeneous records.
- Many small requests issued by a large number of clients.
- Write requests constitute most of the overall workloads.
- Low and predictable latency requirements for single-record requests.
- High load changes over time requiring scalable behavior and elasticity.

In particular, latency becomes critical in many of these scenarios. As an example, search engines require extremely low latency to interactively predict results while the user is still typing a search term. Other examples include real-time communication between devices in the context of IoT and a fluid interaction with the user in the context of augmented reality. For such cases, having a low average latency is often not enough and therefore tail latency plays a major role in the performance analysis of a system. To make the case for tail latency, assume that 100 HTTP requests are required to load a website and there is service-level agreement (SLA) for the website to be loaded in less than 1 s in 99% of the cases. However, even if the server has a 99%-ile latency of 100 ms but at least 10 of these requests must happen sequentially, the amount of times the 1 s SLA is achieved drops from 99% to 90% (0.99^{10}). The importance of tail latency has already been discussed in previous works and acknowledged multiple times in industry [DB13, DHJ⁺07, Gre13].

Unfortunately, most modern KVS are throughput-oriented, in the sense that their design decisions mainly focus on increasing the amount of requests processed over time, at times by sacrificing the latency, as is the usual case of techniques like batching and group-commit. Furthermore, many other components of a system have a negative impact on the tail latency. The operating system scheduler might arbitrarily preempt threads at undesirable points, introducing additional overhead for context switches. The traditional network stack often implies unnecessary movement of data and coarse-grained locks. Storage devices, such as SSDs, require periodical internal reorganization to enable wear-leveling. Garbage collection and defragmentation is also employed in memory allocators and compaction and merging in log-structured systems like LSMs. As a consequence, it is challenging to adapt traditional KVS to become latency-oriented, since the overall latency is affected by the latency of each individual component and usually there is not a single culprit for being the bottleneck. Therefore, to design a latency-oriented system from the ground up, it is required to reduce the latency at each individual component by employing design decisions different than most traditional systems.

5.2 GOALS AND OVERVIEW

The design decisions of *RStore* are guided by two main goals. First, enable low and **predictable latency**. In this context, the focus is on low tail latency of single requests, as these become critical for many use-cases. Second, *RStore* should enable efficient use of hardware resources such as CPU, memory, and storage. An efficient use of CPU requires not only achieving a high throughput, but also a **scalable throughput** to the number of cores in the system. In terms of memory and storage, the goal is to achieve a good ratio that enables **lower costs** while not harming the tail latency. The main design points that enable *RStore* to achieve these goals can be summarized as:

- **Asynchronous execution** enables cores to be always doing “useful” work, leading to efficient usage of CPU resources. This is achieved through asynchronous message-passing communication and cooperative multitasking, avoiding preemptive scheduling and enabling *RStore* to scale with an increasing number of cores.
- **Hybrid DRAM+NVM architecture** allows a good balance between cost and performance. Most of the primary data is stored in NVM, while a small portion of DRAM is used to hide the higher latency of NVM.
- **Log-structured storage** enables efficient space utilization for arbitrary large records and robust performance even under high memory utilization.
- **User-space networking** eliminates the typical bottlenecks of the operating systems network stack and allows zero-copy semantics by directly copying data between network card buffers and non-volatile memory.

5.3 EXECUTION MODEL

The execution model of *RStore* is motivated by the principles of reactive systems and the actor model. To that aim, two main execution techniques are employed: message-passing communication and cooperative multitasking.

5.3.1 Reactive Systems and Actor Model

RStore aims at the principles of reactive systems [BFKT14]: message-driven, resilient, responsive, elastic. These principles were already identified by early work of Joel Barlett, Jim Gray, and Bob Horst at Tandem Computers [BGH87] and Joe Armstrong on the Erlang programming language [Arm03], but it was not until the recent need for large-scale systems that they gained more attention, also in database systems [BBG⁺14, BDKM17].

One of the ways to achieve such characteristics is enabling concurrency through the actor model [HBS73]. An actor (or a “partition” in *RStore*) is an independent and isolated logical entity treated as the universal primitive for concurrent computation. Since actors are

isolated, the only way of communication is by **message passing** (see more in Section 5.3.2). This level of isolation also provides **resilience** by means of fault containment and localized repair, i.e., the failure of an actor is not propagated through the whole system. When combined with replication techniques, the system can achieve higher availability and **responsiveness**. Furthermore, the isolation and independence provide a higher degree of system-wide **elasticity** by allowing actors to be easily distributed and relocated across cores, NUMA nodes, or potentially different machines through the network. Finally, an important consequence of all these aspects is the simplification of development and maintenance of large and complex systems. As an example, one of the best practices of good programming is eliminating special cases. The actor model achieves that by eliminating any differences between local and remote communication (or between scale-up and scale-out) at the programming level, i.e., no matter where actors reside, they communicate in the exact same way (message passing).

5.3.2 Message-Passing Communication

While many fundamental concepts of concurrent and parallel programming were introduced in the 1960's in the context of time sharing in multi-user single-core environments, it was not until early 2000's that *true parallelism* became widespread thanks to the advent of multicore CPUs. As a consequence, system architectures and algorithms had to be revisited to fully exploit the potential of multiple cores.

The many programming models that emerged from this time were classified by Silberschatz et. al. [SGG14] into two dimensions: **interprocess communication** (message passing vs. shared memory) and **problem decomposition** (task parallelism vs. data parallelism). Most modern systems employ shared memory for communication between processes (which we will refer from now on as threads) and task parallelism, i.e., distinguished tasks are executed on the same data.

In shared-memory communication, threads must carefully coordinate by means of mutual exclusion implemented through mechanisms such as locks (a.k.a. latches), semaphores, and lock-free algorithms. To enable algorithms to scale with many cores, the mutually-exclusive critical sections must be as small as possible to avoid contention. To achieve that, these algorithms have to be re-architected, which often leads to more complex and less general approaches. On top of that, the ever increasing number of cores and degrees of parallelism of modern hardware requires these algorithms to be constantly revised and optimized. As a consequence, the programming of large systems becomes significantly more complex and expensive if one is to leverage this increasing level of parallelism. In terms of performance, previous work has shown poor scalability [HAMS08] and a high number of wasted CPU cycles [STPA16] in the context of database systems. Finally, the additional complexity may introduce subtle bugs that are difficult to find as it is harder to reason about execution order in the presence of arbitrarily interwoven threads.

In contrast to shared memory, *RStore* employs **asynchronous message-passing** for inter-process communication and **data parallelism** for problem decomposition. In other words, similar to systems like *HStore* [SMA⁺07], each core runs a single worker thread that only accesses a partition of the complete dataset. Each worker thread has an associated

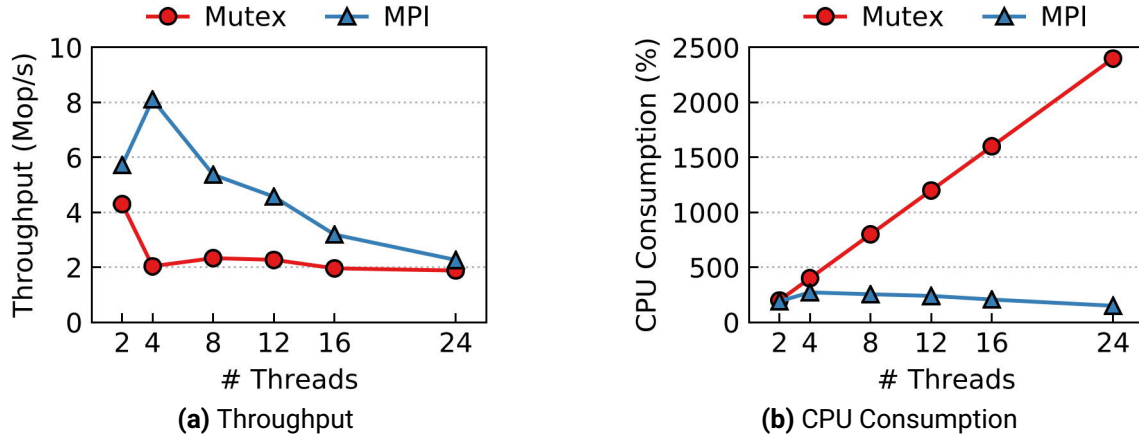


Figure 5.1: Comparison of shared-memory (mutex) and message-passing synchronization over multiple cores.

message queue, to which it can receive requests sent by other threads. If a given thread requires data residing on another partition, it must send a message to request the data from the thread owning that partition. Since communication is asynchronous, the first thread is free to execute further requests while it waits for the response of the message.

While message passing may sound heavyweight compared to shared memory, it allows for a more efficient usage of CPU resources. To prove this point empirically, Figure 5.1 compares both shared memory and message passing approaches when incrementing a single counter for an increasing number of cores. For the shared memory scenario, each thread acquires a mutex, increments the counter and then releases the mutex. For the message passing case, a single thread owns the counter and is responsible for incrementing it, while the other threads send messages to it with a request to increment the counter on their behalf. It is worth noting that, while incrementing a counter can be done more efficiently than with a mutex, we use this example to simulate a high contention scenario. This scenario is realistic as avoiding contention becomes harder considering an ever growing number of cores on future CPUs.

In Figure 5.1a, although message passing presents a higher throughput for a small number of cores (due to reduced cache-coherency events), the throughput drops significantly with more cores, while the shared memory scenario remains constant. However, in Figure 5.1b, shared memory consumes an increasing amount of cycles while providing no additional performance benefits, i.e., these cycles are wasted while message passing maintains a constant CPU consumption.

5.3.3 Cooperative Multitasking

Section 5.3.2 stated that *RStore* relies on data parallelism on a system-wide level. In other words, data is partitioned (by hash or range) and each core runs a single thread, acting as an independent KVS instance. However, it is inevitable that a task (such as processing a client request) will be hindered of making progress when waiting for blocking events such as a message reply, storage I/O, or a network request. Therefore, task parallelism

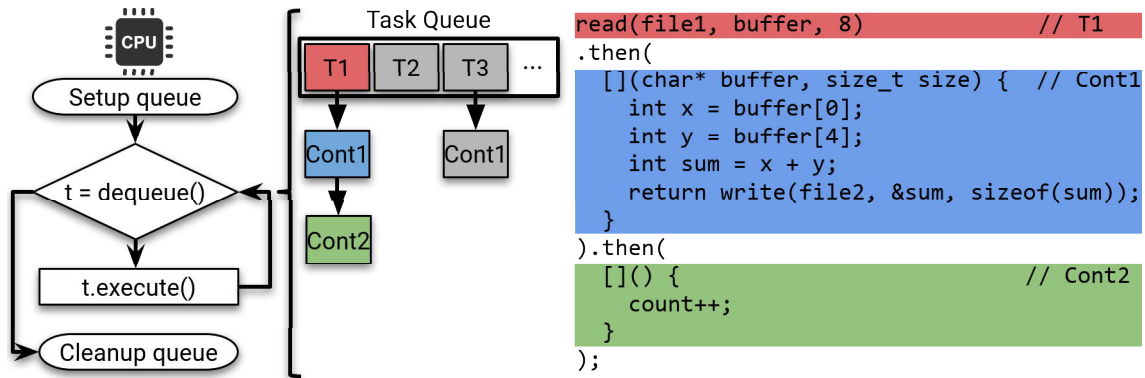


Figure 5.2: A single thread executes multiple tasks through cooperative multitasking in an event-loop. Each task is executed to completion and dependencies between tasks are expressed by a chain of *future* and *continuation* objects.

within a single-threaded partition is also desired to make efficient use of CPU resources. Nonetheless, allowing many threads on the same core does not only lead to expensive context switches, but also to unpredictable behavior, as we have little control over the preemptive task scheduling employed by the kernel. Alternatively, *RStore* employs task parallelism within a partition through a light weight user-space cooperative multitasking.

The left-hand side of Figure 5.2 shows the single-threaded multitasking model of *RStore*. Each thread has a scheduler and a task queue. The scheduler runs an event loop that picks a task from the queue and then executes it to completion (i.e., non-preemptive). Tasks are expressed by means of *futures*, *promises*, and *continuations*. These concepts are common to programming languages, such as Scala [HPM⁺13], and to frameworks for building distributed systems, such as Seastar [Scy15b]. A more detailed definition can be found in the work by Miller et al. [PPM16]. We quote the Scala manual which describes:

“A future is a placeholder object for a result that does not yet exist. A promise is a writable, single-assignment container, which completes a future. Promises can complete the future with a result to indicate success, or with an exception to indicate failure.” [HPM⁺13]

The right-hand side of Figure 5.2 shows a small example: read two integers from `file1` (4 B each), sum their values, write the sum to `file2`, and finally increment a counter. Since file operations are blocking events, the `read()` and `write()` functions delegate the I/O to a helper thread and immediately return a *future object* as a result. A continuation `Cont1` can be chained to this object through the `then()` method and the code passed as argument will only be executed once the I/O result becomes available. Meanwhile, the single-thread is free to execute the next task in the queue (T2). Once the read result is available, the scheduler can execute `Cont1`. Writing to the file will return another future object to which the `Cont2` is chained. Therefore, continuations are used to express dependency between tasks that are executed asynchronously. This model enables full control of the execution flow, as context switches can only happen at well-defined parts of the code (i.e., between tasks and continuations), resulting in a more predictable behavior of the overall system.

5.4 LOG-STRUCTURED STORAGE

The concept of log-structuring was first proposed by Mendel Rosenblum and John Ousterhout in the context of file systems [RO92]. The original motivation was to mitigate the bottleneck of HDDs by exploiting their faster sequential write bandwidth, while serving most reads from main memory, based on the increasing memory capacities by the time.

Flash SSDs reduced the performance gap between sequential and random I/O, albeit sequential I/O requests may still be faster as they better exploit the SSD's inner parallelism. Nevertheless, writing a block to flash requires erasing it first, which can only be done at a larger granularity than writing. This created a new motivation for log-structuring, as new writes can be directed to fresh blocks and space can be reclaimed at a later point in time, thereby reducing the amount of erase cycles required. Most *flash translation layers* (FTL) of modern SSDs rely on some sort of log-structuring. At a system scale, the log-structured design offers additional benefits that are exploited by a wide range of modern systems, as discussed in the following.

First, systems like *RocksDB* [Fac12] and *SILT* [LFAK11] use a log-structured merge-tree (LSM) [OCGO96] to reduce the write amplification by batching writes in memory and writing them in a log-structured manner to persistent storage. The reduced write amplification leads to a longer life-time of SSDs, as these can endure a limited amount of erase cycles. Other systems like *LogBase* [VWA⁺12], *Hyder* [BRD11], and *LLAMA* [LLS13] also use log-structured storage in the context of SSDs.

Second, in the context of DRAM and NVM, gains in write performance might be relatively smaller and one might be tempted to employ update-in-place strategies. However, log-structuring enables better memory management in terms of lower fragmentation and predictable performance in high utilization scenarios. *RAMCloud* [OGG⁺15] adopted a log-structured memory allocator [RK014] to leverage these benefits and allow robust performance even in face of application changes (e.g., expand records of a table from 100 B to 130 B). A similar concept was applied in the context of NVM [HRB⁺17].

Third, log-structuring makes it trivial to perform atomic writes, as only the head of the log must be updated to reflect an arbitrarily large group of operations. This becomes even more convenient in an NVM scenario, as the programmer has little control over CPU caches, which makes it cumbersome to efficiently implement update-in-place strategies while keeping data consistent at all times.

Benefits also entail drawbacks following the *no-free-lunch* conjecture. Unlike update-in-place, a log-structured strategy organizes records by creation time and allows multiple versions of a record to co-exist. This causes three general problems. First, since records are appended to the end of the log, there is low locality for operations such as a sorted range queries, requiring multiple random accesses. Second, point lookup operations become more expensive as they may inspect multiple locations until the most recent version of a record is found, as it is the case in LSMs. Third, garbage collection is needed to delete older entries and reclaim space. However, these problems are less critical in the context of NVM. The low latency reduces the cost of many random accesses required by read operations, while the high bandwidth allows efficient garbage collection, as large portions of live data must be moved to a new location.

To summarize, not only there is goodness in log-structured designs, but as already noted by David Lomet [Lom93]:

“Log structured file system has wonderful potential as the underpinning of a database system, solving a number of problems that are known to be quite vexing, and providing some additional important benefits.”

5.5 NETWORKING

In many KVS scenarios, multiple parallel requests are received from clients through the network and many messages are exchanged between remote machines. Therefore, the network plays a major role and is a critical point of optimization. Saturating the network bandwidth becomes challenging while offering low and predictable latencies. While better bandwidth usage could be achieved by classical techniques for trading-off latency for higher throughput, they should be avoided at the network level if one is to offer a system with robust performance. In common scenarios where the vast majority of requests have less than 320 B [AXF⁺12], the processing overhead per package becomes relatively higher.

Operating system kernels offer applications a general-purpose networking stack. While convenient, kernel networking has issues such as expensive context switches, unnecessary copy of data between NIC, system cache, and application buffers, and poor scalability due to large lock granularities. To circumvent this issues, libraries such as DPDK [Lin10] enable access to the NIC in the user-space. As a consequence, systems are able to tailor the network stack to their use-cases such as zero-copy usage and avoid context switches.

Figure 5.3 compares the performance between kernel networking and DPDK in a microbenchmark. To isolate the impact of DPDK we implemented an HTTP echo-server within our system. The server receives parallel HTTP packages of 100 B from multiple remote clients and send them back, without further complex processing. Figure 5.3a shows how DPDK enables the throughput to scale with an increasing number of cores. Figure 5.3b shows the tail latency of a server with 4 threads using both kernel and DPDK networking while increasing the number of packages sent by clients. At 400 thousand packages per second, the 99%-ile of kernel networking increases abruptly, which reflects the throughput achieved with 4 threads in Figure 5.3a. Meanwhile, DPDK not only enables a predictable latency behavior (no abrupt spikes) but even the worst latency (99.99%-ile) is lower than the 99%-ile of kernel networking. Additional percentiles of kernel networking are much higher and are omitted to enable a better visualization of absolute numbers.

For the reasons mentioned above, we opted for using DPDK on *RStore* for the client and server communication. While a simple client-server communication does not leverage DPDK to its full potential, it is an important building block for extending the communication to many servers in a future distributed context. In such scenario, multiple messages are exchanged between servers and efficient networking becomes even more critical.

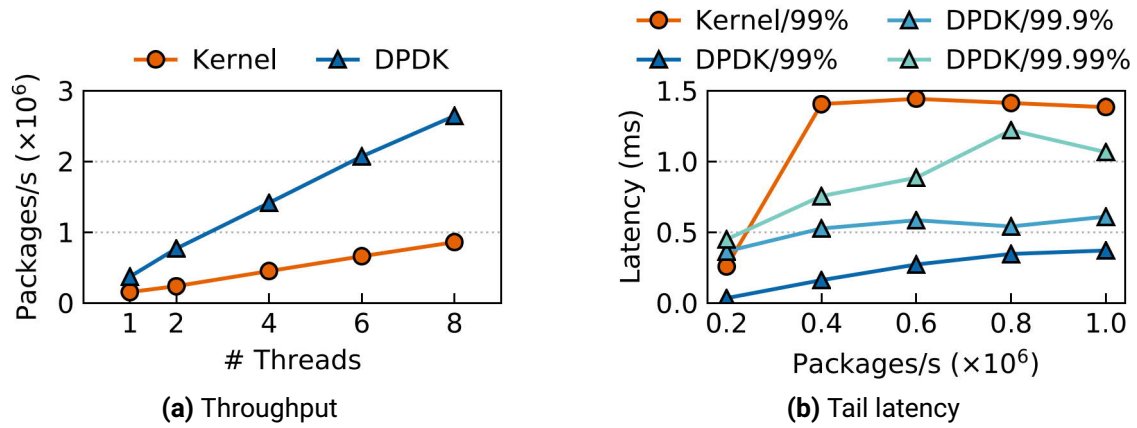


Figure 5.3: HTTP echo server processing 100 B packages using kernel networking and DPDK. Not only DPDK enables a more scalable throughput, but also lower tail latency.

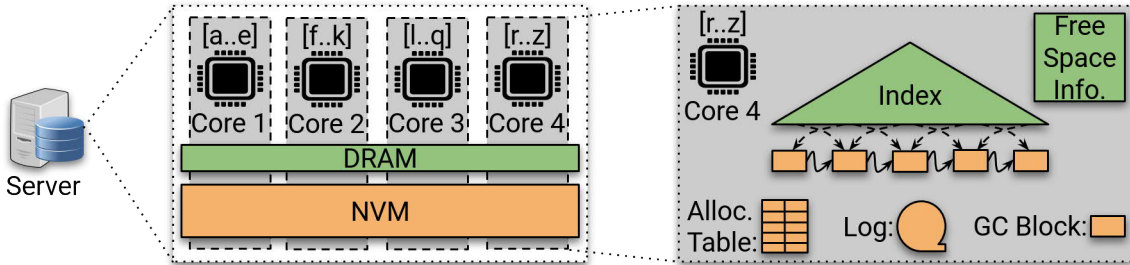


Figure 5.4: RStore architectural overview. The system is partitioned on a per-core basis and each partition runs its own instance of a single-threaded KVS. Each instance follows an *index+log* architecture and the communication between partitions happens through message passing.

5.6 IMPLEMENTATION DETAILS

In this section, we describe details of the overall *RStore* architecture. Figure 5.4 gives a complete overview of the whole system. In this case, the server spans the whole key range $[a..z]$ of a dataset. Further to the center of the figure we have the internal organization of the server. The server is a 4-core system equipped with DRAM and NVM. The whole system is internally partitioned on a per-core basis and communication between cores is done by message-passing, as previously explained in Section 5.3.2.

5.6.1 NVM Allocation on RStore

The log-structured approach significantly facilitates space management, as arbitrarily sized records are accommodated naturally by appending to the end of a block without the need of moving other records for creating space. NVM physical devices are segmented into 2 MB chunks, which is the unity of physical allocation. *RStore* implements an allocation table that maps each physical segment to a logical segment within a logical device. Figure 5.5 illustrates such organization.

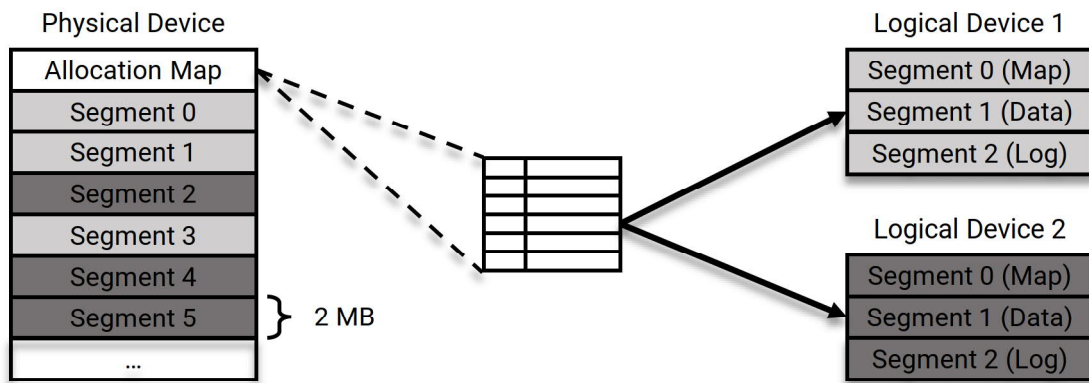


Figure 5.5: NVM device allocation.

Information of which segments are free and used are stored in the first physical segment of the device in the form of an allocation table. Physical segments currently used are mapped to a single logical segment in a logical device. Since keeping the consistency of allocation mapping is critical, the allocation table is implemented as a persistent data structure in which updates are guaranteed to be atomically persisted.

While the overhead of an additional indirection from logical to physical segments can be avoided, it allows each partition of *RStore* to be fully independent by accessing an isolated logical device. It also eases load balancing and reorganization across physical devices, as segments can be moved simply by updating the mapping in the allocation table.

Finally, *RStore* handles logical segments of three different types: log, data, map. Log segments are used for storing log records, which are used for durability and recovery. Data segments can be further divided into smaller blocks of pre-defined sizes (2 kB, 16 kB, 64 kB, 2 MB). These blocks are used for the log-structured storage of records and for overflow blocks in case of large values (more in Section 5.6.4). Map segments contain entries that are used to track information of blocks currently allocated in data segments.

5.6.2 Log-Structured Storage and Indexing

The architecture of *RStore* is commonly referred as **index+log** architecture, and employed by many systems both in academia [RK014, HRB⁺17, VWA⁺12] as well as in industry, such as Bitcast at Riak [SS10], Sparkey at Spotify [Spo14], and FASTER at Microsoft [CPK⁺18]. The main idea is to separate the concerns of data access and space management by decoupling them. This contrasts with approaches such as clustered B+Trees and LSMs, in which primary and indexing data are managed within the same data structure.

RStore employs a log-structured NVM area comprised of fixed-size blocks (64 kB). Even if NVM is byte-addressable and differs from traditional block devices, it is still desirable to organize data in blocks (or “pages”), as it represents a unit of space allocation, garbage collection (see Section 5.6.3), fault containment/detection, and possibly localized repair [GK12]. Records are appended to a block until the block becomes full and is then marked as immutable. Once a record is appended to this log-structured area, a pointer

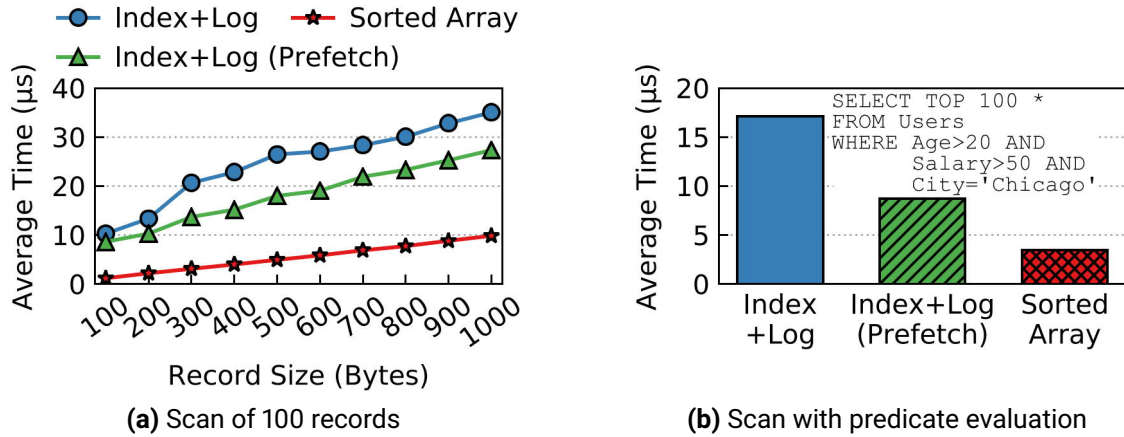


Figure 5.6: Average time for scanning records in a sorted array (sequential memory access) vs. in index+log (random memory access).

to it is inserted into a tree index residing completely in DRAM, enabling a more efficient access than simply scanning the existing records.

This separation of concerns offers important advantages. First, there is more flexibility regarding the representation of persistent and runtime data. For example, any data structure can easily be integrated to index the records in NVM. Second, the index structure contains only fixed-size entries with pointers to the actual data, which simplifies memory management within the data structure. Third, the index structure consumes only a small amount of additional memory. A workload analysis at Facebook [AXF⁺12] shows that the vast majority of keys are no larger than 20 B, while the majority of values are at least 300 B. If the index structure stores whole keys and a pointer to the record in NVM, its space consumption is less than 10%. Fourth, handling records in a log-structured manner enables efficient usage of NVM space by reducing fragmentation and avoiding large over provisioning (B+Tree nodes are usually kept 75% full to accommodate future records). This becomes important since *RStore* does not make any assumptions about data format, origin, or schema. While a well-defined schema enables performance optimizations by the underlying system, *RStore* trades these gains for enough flexibility to be used either as a NoSQL KVS and cache, or as the storage engine for a more complex relational engine.

Despite its advantages, the mentioned architecture introduces drawbacks that must be properly addressed. First, the separation between primary and indexing data is not optimal in terms of spatial locality. Systems designed for HDDs had to exploit at maximum the spatial locality since sequential accesses were significantly faster than random accesses. This assumption still holds for modern SSDs and DRAM and exploiting cache-oblivious data structures are relevant [BDF05, GL01], but the performance gap between random and sequential access is much smaller. This gap can be further reduced by exploiting DRAM prefetching techniques [CAGM04, KFG15, PLMA17] that can be directly applied to NVM.

Figure 5.6 shows the scan performance of a log-structured storage and of a sorted array through a microbenchmark. We isolated other components to better understand the trade-offs. Figure 5.6a shows the average time (Y axis) for scanning 100 records with varying size (X axis) and copying them to the network buffer with a single thread. As previously mentioned, we show how we can reduce the gap between *Index+Log* and *Sorted Array* by

firing an asynchronous memory prefetch for the next record while the current record is being copied to the network buffer. Figure 5.6b shows another case in which a scan of 100 records has a predicate to evaluate, which introduces additional CPU time, allowing a better overlapping between execution and prefetching. We argue that, while we trade-off scan performance for other benefits (such as easy memory management and garbage collection), we can still improve the worst case, albeit still being slower than a scan in sorted storage. Nevertheless, in the context of a system accessed through a modern network, the performance benefits are mostly blurred by higher network latency.

The second disadvantage is that, while the space management of primary data is made in a log-structured manner, the space management of indexing data still has to be handled. Fortunately, index space management is significantly simplified by using fixed-size index entries, as previously mentioned. At one extreme, only 8 B pointers to the actual records can be used as index entries. In such case, index operations are more costly, as every key comparison must go out-of-node to fetch the actual key in NVM. In the context of a tree index, we employ techniques such as prefix truncation and *poor man's normalized keys* [GL01] to keep a copy of a small portion of the key within the node. In most cases, this small portion of the key is enough to resolve comparisons without accessing out-of-node data. By changing the amount of bytes dedicated to store the in-node portion of keys, we can trade between memory consumption and access performance, while keeping fixed-size index entries.

Third, while the index can exploit the lower latency of DRAM, it must be rebuilt in case of failures. We rebuild the index during startup from the key-value records in the log-structured storage. Since *RStore* is composed of independent partitions, the index for each one of these partitions can be recovered on-demand, i.e., accessing data during restart does not require a complete rebuild of all indexes. A similar approach is used by hybrid NVM-DRAM data structures [OLN⁺16, XJXS17]. To limit speedup recovery, regular snapshots of the index can be taken by flushing the whole data structure to NVM.

5.6.3 Garbage Collection

Traditional LSM implementations employ a merge operation to reclaim space of obsolete records and consequently reduce the number of persistent components (also called SSTs) that must be inspected during reads. *RStore* relies on a global index to access records, therefore, a read operation does not have to consider multiple copies of a record, as only the most recent one is indexed. Nevertheless, the cost of index operations increases with the size of the index.

LSMs can reduce the cost of inspecting multiple SSTs by employing Bloom filters. However, in the context of NVM, two points must be considered. First, memory consumption of Bloom filters is not negligible for large data, even if the memory budget is optimally distributed across levels [DAI17]. Second, Bloom filters are used to avoid expensive disk I/O which incurs high latency. On NVM, accesses incur a much lower latency and therefore the performance gains of avoiding these accesses relative to the additional overhead introduced by Bloom filters are smaller. In other words, probing the Bloom filter already incurs a memory access, which is a similar cost to directly searching the key in NVM.

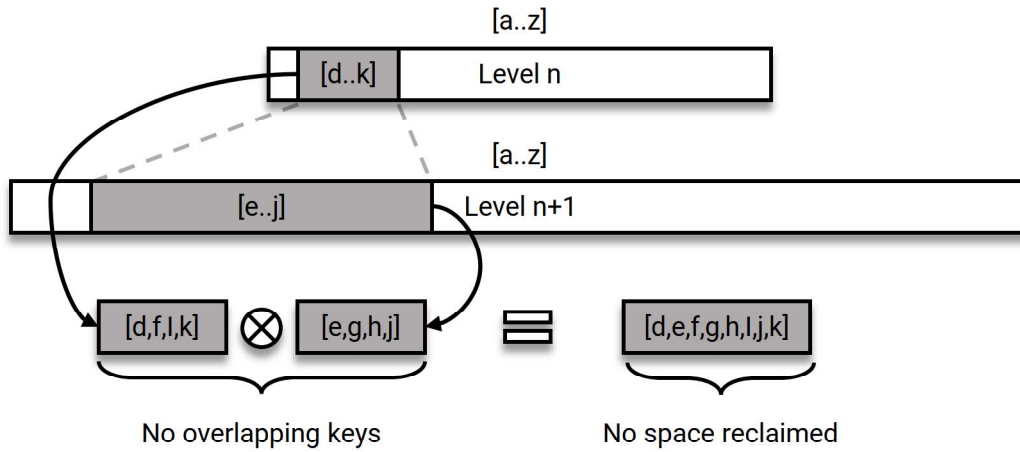


Figure 5.7: False overlap leads to inefficient space reclamation and unnecessary write amplification in merges of LSMs.

Figure 5.7 shows the merge process of an LSM. The merge starts by selecting a range of records from *Level n* for merging with records from *Level n+1* that have an overlapping key range. The problem of relying on overlapping key ranges for garbage collection is that there is no guarantee of how much space will be reclaimed. In other words, the key ranges defined by min and max keys may overlap but the records themselves might not. As shown in Figure 5.7, in the worst case there is no overlap of records and the merge process is superfluous, thus increasing the write amplification. The phenomenon is referred to as **false overlap** and has been discussed in previous work [LAK16]. Alternatives such as *logical merging* through pointer manipulation may help in reducing the amount of duplicated records and consequently in improving lookup performance, but it does not help with reclaiming space, which is critical when a device is mostly full. Furthermore, the merge operation is hard to parallelize, as it depends on the key distribution of the workload. A uniform distribution allows an easier parallelization of the merge operation, as disjunct ranges can be merged, while a skewed distribution causes only a subset of the whole key range to be merged frequently.

The garbage collection of *RStore* was designed to be oblivious to the aforementioned effects caused by using key ranges as victim-picking strategy. The core idea is to keep live information about free space and valid records in each NVM block. In a way, it resembles the *trim* command in early SSDs, in which the user actively provide information about unused space to facilitate garbage collection by the *flash translation layer* (FTL). Tracking this information on a record granularity introduces overhead during runtime, as blind inserts/updates/deletes are not possible anymore. Nevertheless, it facilitates garbage collection, which is the main source of unpredictable performance in many systems. In other words, *RStore* takes a small, but predictable, performance penalty during normal processing in order to reduce the unpredictability of garbage collection.

Figure 5.8 gives an overview of the algorithm. A block initially has 100% of free space, which is reduced as the block is filled. When the block is full, it becomes immutable. Whenever a record is deleted or a new version is inserted, the free space information of the corresponding block is updated. The *free space heap* tracks the free space of each block, which allows identifying the block that will yield the largest amount of space when

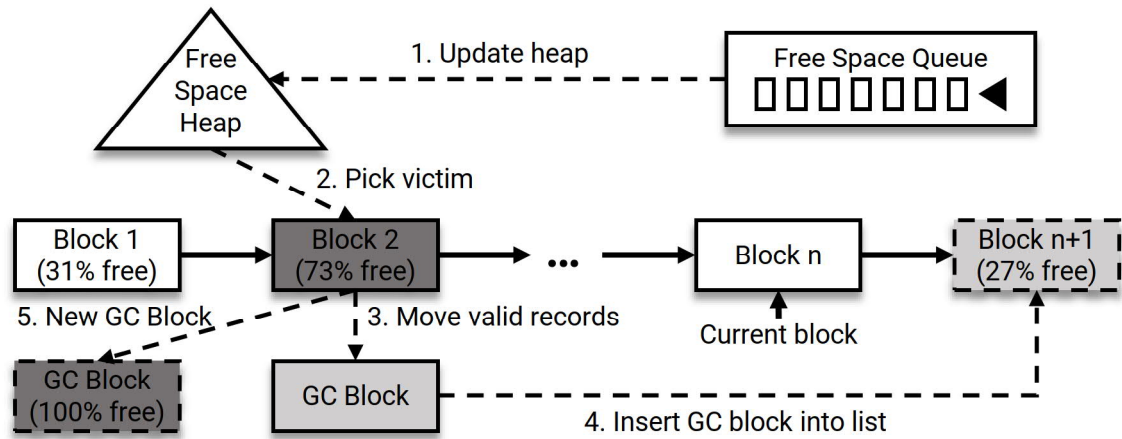


Figure 5.8: Garbage collection algorithm of *RStore*.

reclaimed. Since free space of blocks changes frequently, maintaining the heap structure is expensive. Therefore, whenever the free space of a block is changed for the first time, a reference to this block is added to the *free space queue*. By doing so, the heap is updated only when garbage collection is required, thereby alleviating the heap overhead during runtime. When garbage collection is triggered, **step 1** is to update the *free space heap* with blocks in the *free space queue*, i.e., blocks in which the free space changed since last garbage collection. With the *free space heap* updated, **step 2** is to pick block with largest amount of free space (in this case Block 2), referred as *victim*. Next, valid records from the victim block are moved to a dedicated *garbage collection block* in **step 3**. Finally, in **step 4** and **step 5**, the garbage collection block becomes a new block at the end of the list and the victim block becomes the new dedicated block to be used by the next iteration of garbage collection, respectively.

Figure 5.9 compares our algorithm and traditional LSM merge (*RocksDB*). We limit the available space to 16 GB and load it until little space is left in order to force garbage collection. Both systems run on top of NVM described in Section 5.8 and we use *levelled compaction* in *RocksDB*. To isolate the algorithm impact, in Figure 5.9a we run an update-only workload for 5 minutes with a single-thread serving requests sent at a rate of 25000 requests per second (a rate that both systems can easily sustain). Not only *RStore* has lower tail latency, but it is constant and unaffected by skew. Figure 5.9b shows the throughput over time including the load phase (gray area) using 16 threads. In addition to *levelled compaction*, we run *RocksDB* with *universal compaction*. Universal compaction trades higher read and space amplification for lower write amplification and is more cumbersome (as noted in the first drop during the load phase). It also requires double the amount of space, which explains the throughput drop to zero after the load phase: the system becomes unresponsive since not enough space is available for compaction. Finally, the absolute throughput number is not important, instead the focus is on the drop when the device is full and garbage collection becomes critical. The average throughput of *RocksDB* drops by 38% and *RStore* by only 7%.

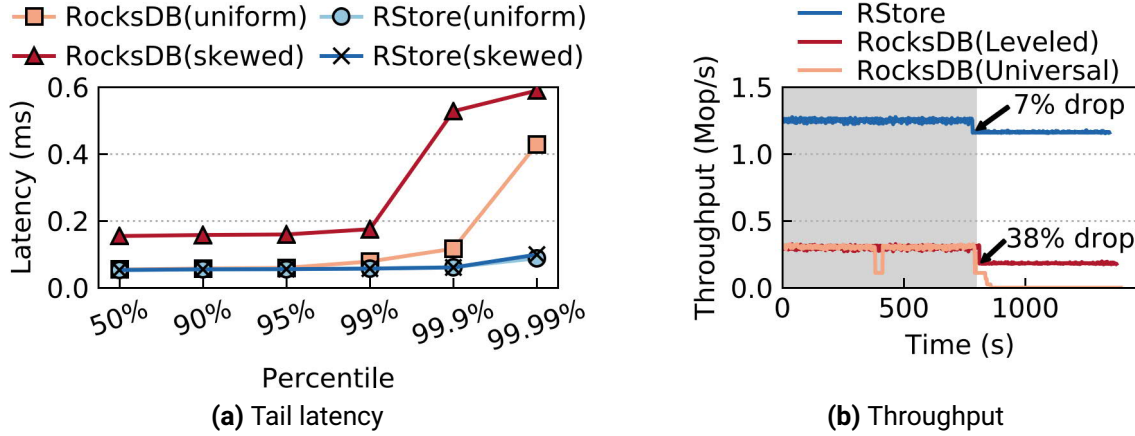


Figure 5.9: Impact of garbage collection in the performance when storage device is full.

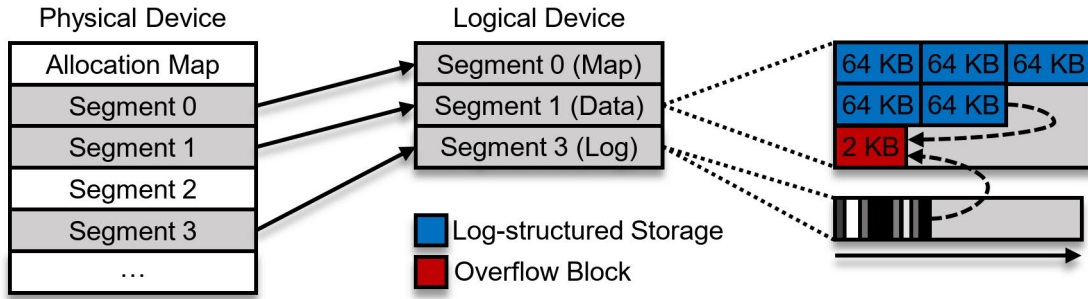


Figure 5.10: Large values are stored only once in an overflow block, reducing write amplification. Records and log records point to the single copy of a large value.

5.6.4 Logging and Recovery

In addition to the log-structured storage, each partition of *RStore* has a local recovery log. Since operations to the log-structured storage are easily made atomic, one may consider that it obviates the need for separated logging. However, the log acts as a central component which can be used by third-party systems for state machine replication through protocols such as RAFT [OO14]. In this case, log records are sent to remote replicas and the network bandwidth becomes the bottleneck. Therefore, we use redo-only logical logging, which has smaller log records compared to traditional physiological logging, thus better leveraging network bandwidth.

An initial concern is that the recovery log doubles the write amplification. However, decoupling logging from storage facilitates replication of higher-level operations. As an example, while the log-structured storage only operates through basic single record operations such as *insert*, *delete*, and *update*, the recovery log allows multi-record operations, such as deletion of multiple keys based on a given prefix, to be transmitted as a single log record. Furthermore, to alleviate the write-amplification introduced by logging, large keys and values are stored only once in an *overflow block* which is then referred by both log record and key-value record. Figure 5.10 illustrates this case in which a value larger than 2 kB is inserted. The larger part of the value is stored in the overflow block which is referred by both the respective log record and key-value record.

Another concern is that latency of writes is doubled, since every write must be flushed twice to NVM: one to the log, another one to storage. However, only writes to the log must be eagerly persisted. Writes to storage do not have to be explicitly flushed and can be amortized by CPU caches. Since storage is log-structured, data is not overwritten. In case of a crash before a record is evicted from the CPU cache, it can be recovered by replaying the recovery log.

After a system failure, recovery starts by rebuilding the in-memory index and free space information from the records present in the log-structured storage. The recovery log is then analyzed and any missed operations are replayed. Since each partition of *RStore* is independent, this process is completely parallelizable and a partition can start to serve client requests without waiting for a complete system recovery.

5.7 SYSTEM OPERATIONS

In this section, we describe the steps of basic operations. Unlike other systems, *RStore* does not support *blind operations*, since free space information must be tracked for garbage collection. Each operation is initially assigned to the partition spanning the range that covers the given key.

An **insertion** of a record starts by searching the index for the given key. If the key already exists the operation fails. Otherwise a log record corresponding to the operation is written to the log, and the record itself is written to the log-structured storage. An entry containing the key and pointer to the record is then inserted in the index structure.

The **update** of a record is similar to an insertion, with two main differences. First, the operation fails if the key does not exist. Second, the update is done by inserting a new version of the record which invalidates the old one. Therefore, the corresponding pointer in the index must be updated to point to the new record and the old record must be invalidated by resetting a validity bit. Validity bits of records are kept in-memory and must be rebuilt during restart, since immutable blocks of the log-structured storage cannot be updated in-place. Additionally, the size of the old record is added to the free space information of the block containing it. A **deletion** works like an update in which a special tombstone record with no value is inserted and the corresponding entry in the index is deleted rather than updated.

The **point lookup** of a record traverses the indexing structure to find the record for the given key. If the full key is stored in the index, the lookup makes a single access to NVM to retrieve the full record (if it exists). If fixed-size partial keys are used for indexing, the lookup might require additional accesses to records in NVM to compare the full keys in case the partial key is not enough to resolve a comparison when traversing inner nodes.

Range lookups may span multiple partitions. Therefore, a partition is chosen to coordinate the operation. It then forwards the range lookup operation by sending a message to all other partitions that span key ranges overlapping with the one specified by the operation. Each partition then independently executes the range lookup locally by traversing the index data structure. Even if the records are not sorted on the log-structured storage, their

corresponding index entries are, enabling the records to be retrieved in sorted order. Once a local range lookup is completed, the partition replies the results to the coordinating partition, which is responsible for collecting the multiple results and issuing the final reply of the operation.

5.8 EVALUATION

In this section, we present performance results of an end-to-end evaluation of systems. The metrics we are most interested in are throughput scalability and low tail latency.

5.8.1 Methodology

We run all systems on a single machine and use a second client machine sending a high number of parallel requests. The indicated number of threads is the same for both server and client. To overload the server, each client thread opens 8 connections to the server and issues asynchronous requests (at any point in time a client thread has 8 in-flight requests).

We measure throughput and latency on the client side. For throughput, we collect the amount of operations completed every 1 second. At the end of the execution, we use the list of operations completed per second for calculating the average throughput as well as the standard deviation. For tail latency, measuring each individual request would introduce too much compute and memory overhead, therefore we randomly sample up to 500 thousand requests every 1 second and use the total amount of samples to plot the latency percentiles.

5.8.2 Environment

The server has an Intel Xeon Platinum 8260L CPU, 96 GB of DRAM (6×16 GB DIMMS), and 1.5 TiB of Intel Optane DC Persistent Memory (6×256 GB modules). The client has an Intel Xeon CPU E5-2699 v4 and 128 GB of DRAM (8×16 GB DIMMS). Both client and server use a 10 GbE Intel Ethernet Controller X540-AT2. The network cards are accessed through DPDK(v17.02). The Linux version is 5.3 on both machines. The NVM modules are combined into a single namespace in *fsdax* mode and accessed through an *ext4* file system with the DAX option enabled. All the systems benchmarked rely on Intel Optane DC Persistent memory for storage. It is either accessed as an SSD replacement through the regular file system API, or accessed directly as persistent memory (in the case of NVM-aware systems, like *RStore*).

5.8.3 Other Systems

In addition to *RStore*, we also benchmark three popular KVS systems: ***memcached*** (v1.5.16), ***Redis*** (v5.0.5), and ***RocksDB*** (v6.2.2). We also compare to ***FASTER*** (v2019.11.18.1) [CPK⁺18], a more recent system which employs modern techniques. Both *memcached* and *Redis* are often used as a web cache. While they enable flushing memory contents to persistent media as a background task, the default scenario is purely in-memory. We disable their caching behavior to guarantee that records loaded by the client are not arbitrarily discarded by the LRU policy. The available memory is set to 32 GB. Finally, it is worth noting that *Redis* is a single-thread system.

RocksDB is an LSM persistent KVS to make efficient use of SSDs. To enable a fairer comparison, we run *RocksDB* on top of NVM as well. We disable Bloom filters and compression of values, since other systems do not employ them. Furthermore, since I/O to NVM is faster than to SSD, the overhead of compressing data prior to writing to persistent storage is relatively higher and the gains of avoiding I/O with Bloom filters are relatively lower. The compression of keys is kept. We use a block cache of 6 GB. Additional parameters were changed according to the tuning guide available at the official repository¹. We make the complete settings available². *RocksDB* does not have networking, so we adapted the same network layer of *memcached*. Finally, it is worth noting that *RocksDB* does not explore the byte-addressability of NVM, using it as a faster SSD. Previous work improved *RocksDB* to better leverage NVM [EGA⁺18], but these changes are not available in the main repository. Finally, while comparing absolute throughput numbers may not be completely fair, the comparison of overall system behavior is still relevant.

FASTER also has a log+index architecture, using a lock-free hash table for indexing and epochs for concurrency control. Unlike *RStore*, in *FASTER* keys are not part of the index, which reduces its memory footprint. Furthermore, it requires that the amount of hash buckets is a power of 2. For 100 B and 1000 B values we set the amount of hash buckets to 2^{26} and 2^{23} which gives an average of 2.3 and 1.9 records per bucket and 4 GB and 0.5 GB memory consumption, respectively. We limit the log size to 32 GB. *FASTER* does not offer networking, therefore we adapted it to work with *memcached* network stack. We show results for the **in-memory** version of *FASTER*. We omit the persistent version as it showed lower performance and does not access NVM directly, therefore the results could be unfair and misleading. Consequently, we have also disabled checkpointing.

RStore keeps the index completely in DRAM, which contains keys (approximately 25 B) and pointers to the complete records in NVM (8 B). We apply hash partitioning to *RStore* and set the amount of available memory to 32 GB. In addition to the persistent version, we also benchmark a fully in-memory variant of *RStore* (tagged with *IM*).

¹<https://github.com/facebook/rocksdb/wiki/Setup-Options-and-Basic-Tuning>

²<https://gist.github.com/llersch/6a6fd515b9db8a87ed860573e3417961>

5.8.4 Throughput Scalability

In this section, we measure the throughput when increasing the number of threads at the server side. It is worth noting that each client thread sends up to 8 parallel requests to the server at any point in time. We run the Yahoo! Cloud Serving Benchmark [CST⁺10], issuing *Put* and *Get* requests. Our *Put* operations are done on existing records (updates), therefore the dataset size does not increase. We vary the ratio between these requests to simulate different workload scenarios: read-heavy (90% *Get*, 10% *Put*), balanced (50% *Get*, 50% *Put*), and write-heavy (10% *Get*, 90% *Put*). Following workload trends [AXF⁺12], we set the key size to approximately 20 B with an additional prefix of 4 B while having large-value (1000 B) and small-value (100 B) scenarios.

We analyze two load scenarios that achieve 16 GB of payload data. In other words, for 1000 B values 16 million records are inserted, while for 100 B values 160 million records are inserted. After the load phase, we run each workload for 5 minutes. Figure 5.11 and Figure 5.12 show the results for 1000 B and 100 B payloads, respectively. The shaded part indicates the hyper-threading zone. We make three observations.

First, both *Redis* and *RocksDB* perform worse than the others. This is expected, since, as previously mentioned, *Redis* is a single-thread system and therefore the X-axis represents only the amount of clients sending requests. The main reason *RocksDB* presents a lower performance is the fact that it does not fully leverages the byte-addressability of NVM, simply accessing it like a faster SSD. Nevertheless, it is worth noting how *RocksDB* performs better than *Redis* for the read-heavy scenarios since it is able to leverage multiple threads. As soon as the amount of write operations increase, *RocksDB* is exposed to the higher write latency of NVM during flushing and compaction.

Second, the performance of *RStore* and *memcached* degrades when the amount of write operations increases. For *RStore*, write operations expose the higher NVM write latency, as well as it triggers garbage collection due to the log-structured organization. The in-memory variant, *RStore(IM)*, is able to scale better and saturate the network limit with fewer cores since it is not affected by NVM. For *memcached*, no additional allocation is required because only existing records are updated. However, it introduces additional overhead when acquiring a coarse-grained mutex every time a record is updated. *FASTER* and *RStore(IM)* scale well and have a similar behavior, as both saturate the network in Figure 5.11 and scale almost linearly in Figure 5.12.

The throughput of *RStore* is slightly higher than *memcached* with 1000 B and slightly lower with 100 B, but both scale similarly across many threads. Furthermore, it is worth noting that *RStore* stores most of its data in NVM, which introduces a higher latency as a trade-off for lower costs. Nevertheless, *RStore* still achieves a good performance due to the combination of the techniques mentioned previously. Finally, it is possible to see the impact NVM has on *RStore*, since *RStore(IM)* saturates the network with fewer cores in Figure 5.11 while offering higher throughput in Figure 5.12.

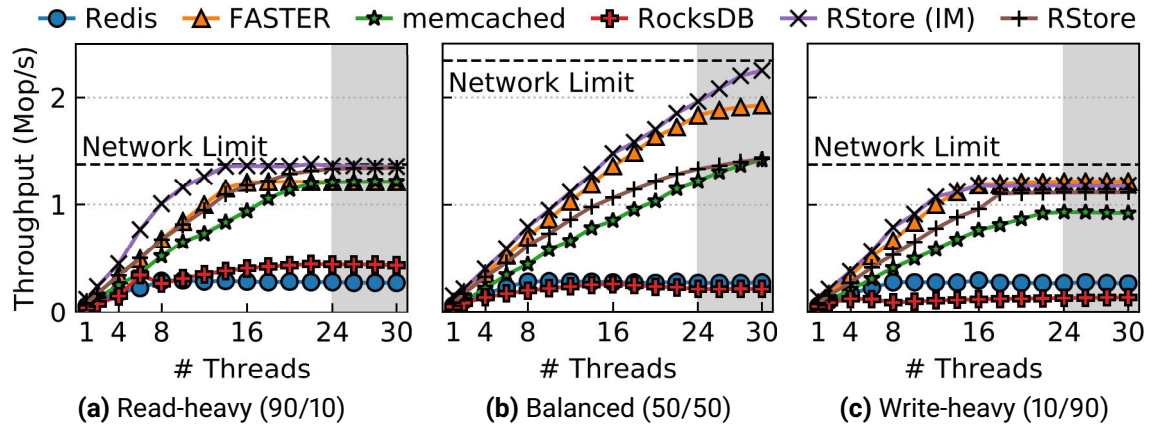


Figure 5.11: Throughput of YCSB workloads with values of 1000 B over multiple threads.

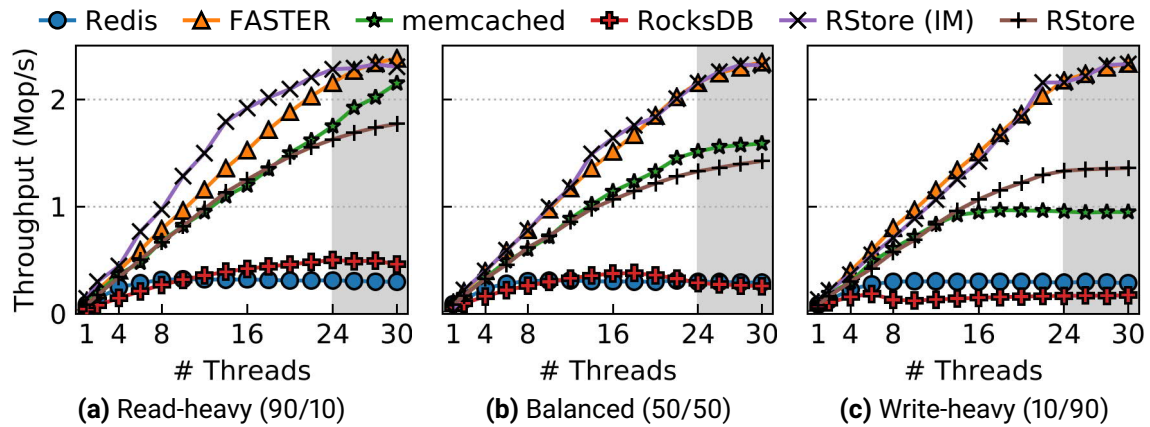


Figure 5.12: Throughput of YCSB workloads with values of 100 B over multiple threads.

5.8.5 Tail Latency

We analyze the tail latency in form of latency percentiles to evaluate the predictability of the systems. However, tail latency is a metric that does not live on its own. It must be considered in the context of the pressure being put on the server by the clients. Even if the throughput of the system scales linearly, the more overloaded the system is, the higher the tail latency percentiles are. In other words, one should ask the question: “How fast can I go before the tail latency is affected?”. Therefore, we set a fixed number of 16 threads on the client and throttle the rate at which requests are sent to control the pressure we put on the server side. The server runs with 4 threads, since the throughput of the systems is not too different at this point, as seen in Figures 5.11 and 5.12.

Figure 5.13 and Figure 5.14 show the tail latency for the **read-heavy**, **balanced**, and **write-heavy** workloads (rows) and the rate of requests being sent by the client (columns) which increases across plots from left to right. We omit rates higher than 500 thousand op/s because none of the systems can sustain higher throughput at 4 threads, as seen before.

At the end of the run phase, we have a list of all observed requests and sort them by latency. This sorted list is used to plot the minimum, maximum, and percentiles of latency for these requests. We set a high-level goal of achieving sub-millisecond tail latency, marked by the gray area in each plot. Therefore, systems with good tail latency must have a curve as straight and low as possible inside the gray area.

As previously mentioned, the higher the pressure being put by the client, the higher the tail latency is. That means that not only the curves become steeper but also higher overall, as can be seen in the behavior of *Redis* in Figure 5.13 for the read-heavy workload when comparing 80k op/s and 160k op/s, for example. Another observation is that the behavior of systems do not change after a certain point, as is the case of *RocksDB* for all workloads in Figure 5.14 after 160k op/s. The reason is that at this point the pressure being put on the server is higher than the throughput it can deliver, causing the client to reach its maximum amount of outstanding requests while waiting for the server. In other words, at this point we consider that the tail latency of the server has already reached an undesirable behavior.

For most scenarios, *RocksDB* has the worst tail latency, since it accesses NVM through the regular file system interface. *Redis*, *memcached* and *FASTER* have a good behavior for low pressure scenarios such as 20k op/s for all workloads in Figure 5.13. After this point, *Redis* becomes more unpredictable. The exception is the write-heavy scenario at 320k op/s and 500k op/s, in which the behavior of all systems except *RStore* and *RStore(IM)* become worse. Overall *RStore* has a higher tail latency, since it requires at least one access to NVM per operation. However, *RStore* also has a straighter line at high load scenarios (500k op/s) with write operations, this being a consequence of log-structuring and asynchronous message-passing communication. *RStore(IM)* has a similar behavior, but is able to keep a lower tail latency.

Figure 5.14 shows the same scenarios for smaller requests (100 B value). The main observation is that *RStore(IM)* behaves better than other systems in more cases. The overhead of package processing is relative to the size of the package, therefore, *RStore* in general has an additional benefit in these cases by being the only system using DPDK. This is seen more notably for most workloads at 320k op/s and 500k op/s.

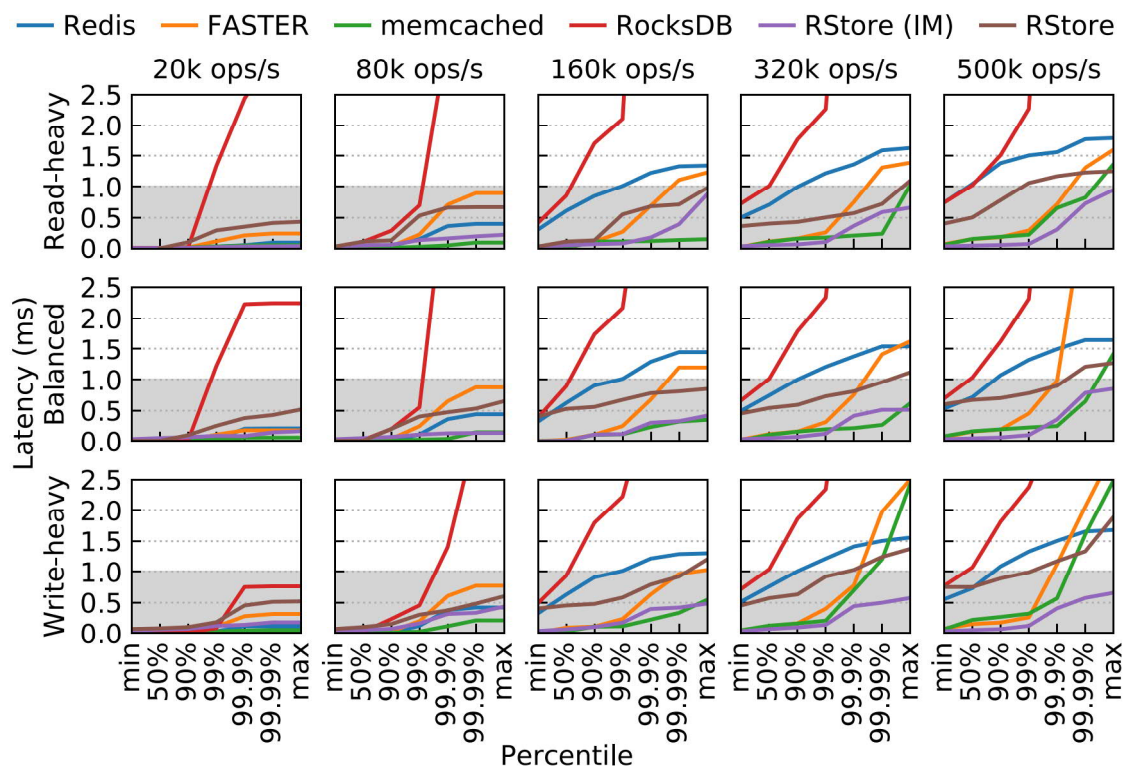


Figure 5.13: Tail latency of YCSB workloads with values of 1000 bytes and 4 threads. Each column indicates the rate at which clients send operations to the server (label at the top). Each row indicates the workload (label at the left).

Figures 5.15 and 5.16 present the experiments with 16 threads on the server side and values of 1000 B and 100 B, respectively. We compare *RStore* only to *memcached*, since it is the system with better tail latency behavior among all the other systems. We consider two pressure scenarios: 1 million (light color) and 2 million (dark color) operations per second. In all cases, while *memcached* has a better behavior at 1M op/s, this behavior is not sustained when the pressure is increased to 2M op/s. On the other hand, *RStore* has a slightly worse behavior at 1M op/s but is able to keep it more stable for the 2M op/s case, having both of its curves between the *memcached* curves.

5.8.6 Scans

The operations that suffer the most from the *index+log* architecture are sorted range scans. Even if scans are less common for the use-cases that we target with *RStore* (like web-caching), we still consider important to support it to some extent. We discussed in Section 5.6.2 the limitations and possible improvements. Here we show an end-to-end evaluation of ranged scans on *RStore*.

Small scans are more common for our use-cases and they are unlikely to span more than one or two partitions. Figure 5.17 shows the throughput (X axis) for different scan sizes (number of records) for records of 100 B over multiple threads (Y axis). It is worth noting that scans are bandwidth intensive and therefore network quickly becomes the bottleneck.

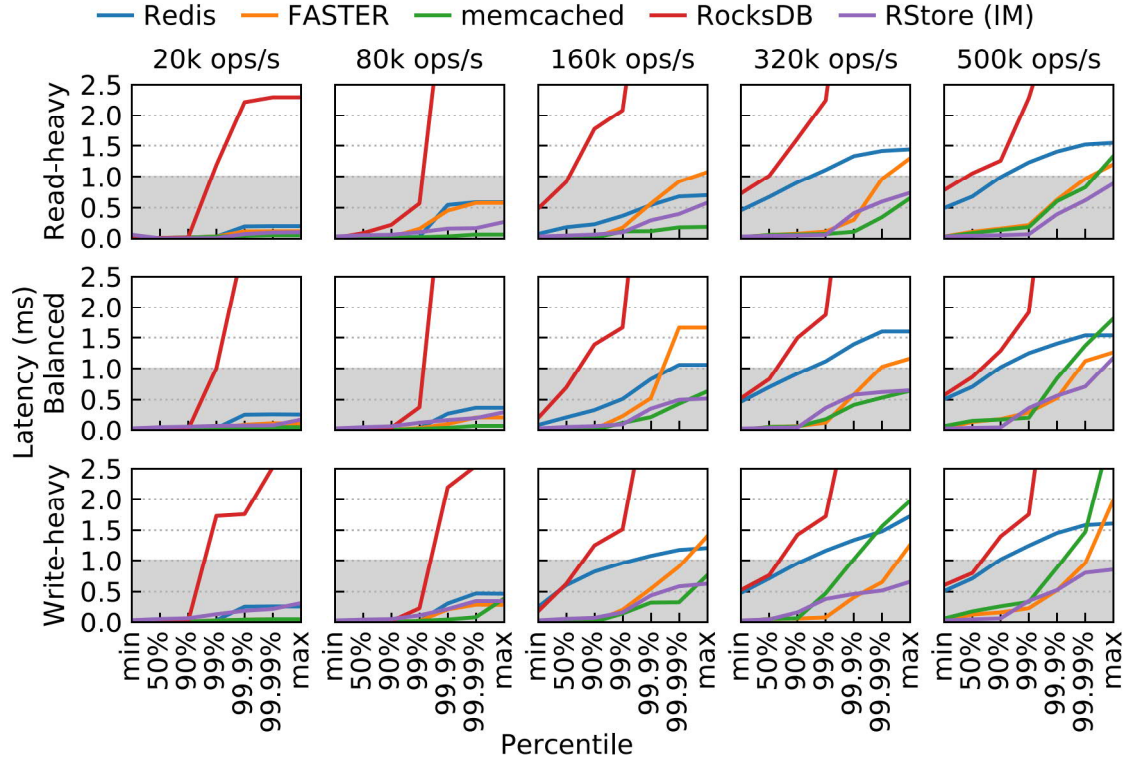


Figure 5.14: Tail latency for YCSB workload with values of 100 bytes values and 4 threads. Same organization as Figure 5.13.

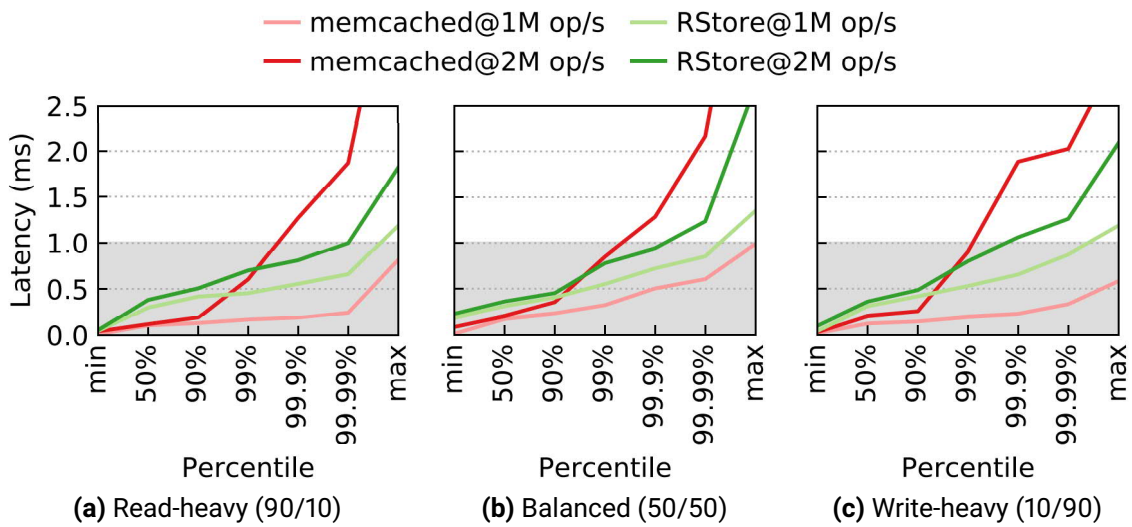


Figure 5.15: Tail latency for YCSB workloads with values of 1000 B and 16 threads.

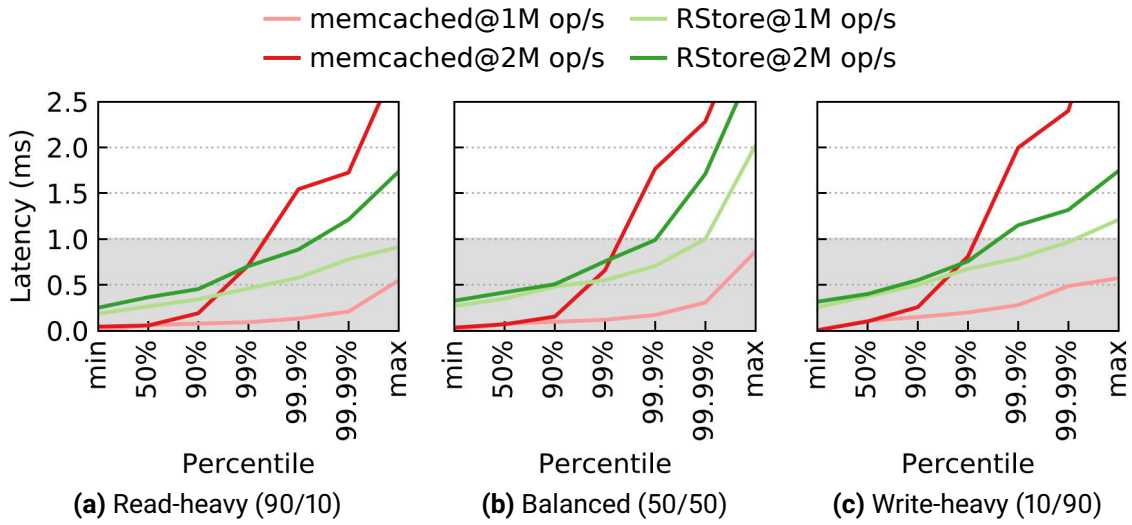


Figure 5.16: Tail latency for YCSB workloads with values of 100 B and 16 threads.

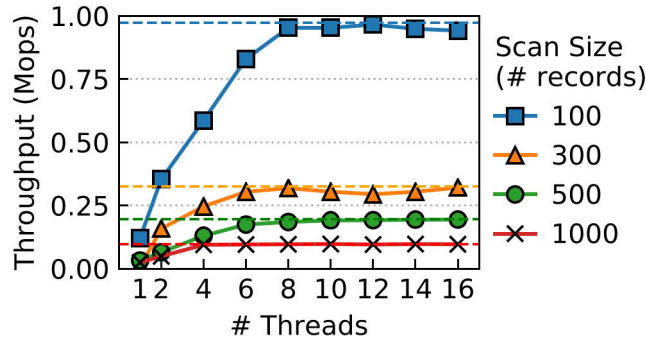


Figure 5.17: Scan performance over multiple threads.

The colored dashed-lines show the point at which the network bandwidth is saturated for each scan size. We can see that for 100 records, we are able to saturate the bandwidth with only 8 cores. For scan sizes of 300, 500, and 1000 records, we saturate the bandwidth with 6, 6, and 4 cores, respectively.

5.8.7 Memory Consumption

One of the goals of *RStore* is to have reduced costs by using cheaper NVM for storage, in contrast to completely in-memory systems. We collected the amount of memory consumed (DRAM and NVM) by each system after loading them with 16 million records with 1000 B payload and 160 million records with 100 B bytes payload. Figure 5.18 shows these numbers with the raw size indicated by the gray area. Since each record has a key and additional overhead space introduced by each system, the scenario with 100 B payload requires more space. With 1000 B payload, all systems have a similar memory consumption (DRAM for *memcached*, *Redis* and *FASTER*; NVM for *RStore* and *RocksDB*). The additional DRAM consumption of *RStore* is due to the index, while in *RocksDB* it is caused by the 6 GB block cache and memtable. With 100 B payload, *Redis* requires more DRAM than *memcached* and *RStore* requires more NVM than *RocksDB*. As mentioned previously, *RocksDB* compresses

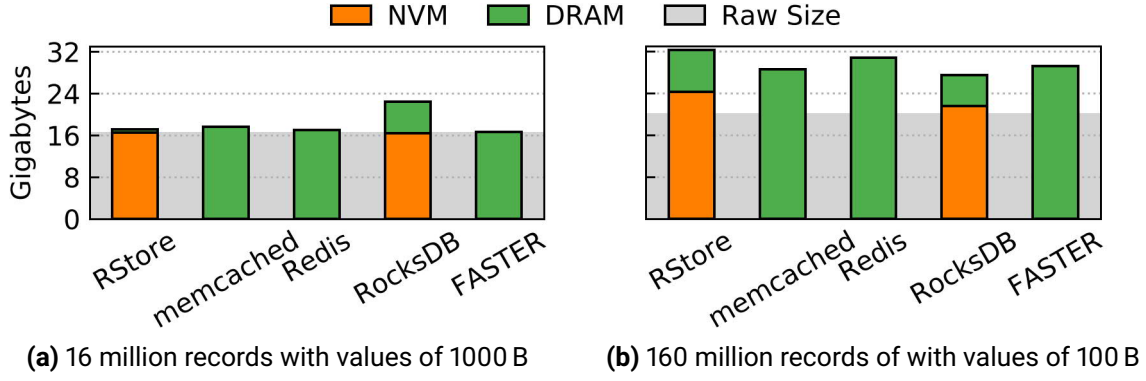


Figure 5.18: Memory consumption of each system.

Value Size	RStore	memcached	Redis	RocksDB	FASTER
1000 B	98\$	207\$	200\$	160\$	195\$
100 B	227\$	335\$	361\$	187\$	342\$

Table 5.1: Approximate cost (in US\$) of each system based solely on their memory consumption depicted in Figure 5.18.

keys, which allows a lower space consumption on NVM. It is also worth noting that with a larger amount of records, *RStore* requires more DRAM for the index than *RocksDB*. Moreover, since *RStore* requires the index to be completely in DRAM, it is less flexible in tuning the memory budget.

Finally, considering the memory consumption and current prices of DRAM (1500\$ for 128 GB) and NVM (695\$ for 128 GB) modules [Aco19], we have anecdotally calculated the memory and storage price of each system in Section 5.8.7. Figures 5.19a and 5.19b show the throughputs of the *Balanced* workload, shown before in Figures 5.11b and 5.12b, divided by the respective costs of Section 5.8.7. These values serve as an initial expectation of the rate of costs between the systems considering their performance. While *RStore* has a throughput similar to *memcached* and lower than *FASTER* in Figure 5.11b, it has a much better performance when we compare the throughput relative to the cost of storage, as shown in fig. 5.19a. In Figure 5.19b, the throughput relative to cost of *RStore* is much closer to the other systems, showing no significant advantage for the scenario with 100 B values.

5.9 RELATED WORK

The concepts presented in this chapter were already explored to some extent in other systems. Related work was partially covered for each aspect of *RStore* in their respective sections, therefore here we elaborate on more recent complete systems that share some of the same design decisions.

RAMCloud [OGG⁺15] is a KVS implemented as a distributed hash table that relies on large amounts of DRAM to store all of its data with the goal of achieving extremely low latency.

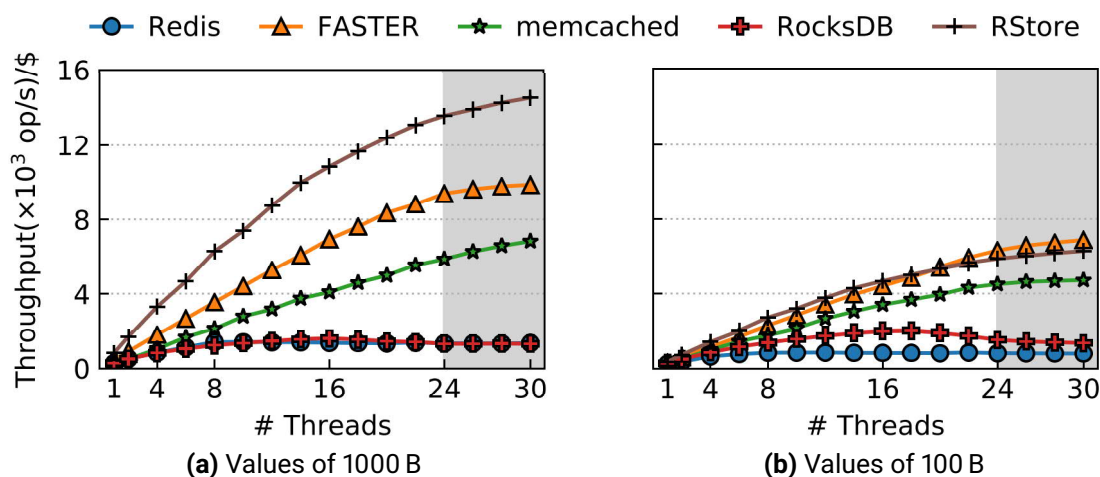


Figure 5.19: Rate of throughput for workload *Balanced* divided by cost (higher is better).

RStore target similar goals of not only low latency, but predictable latency. It also explores modern storage hardware (NVM) for reduced storage costs.

Anna [WFLH18] is a distributed KVS that also relies on a thread-per-core model and message passing rather than shared memory communication. While this work focus more on the internal organization and storage aspects of *RStore* in a single node context, *Anna* uses a standard C++ hash table for storing records and focus more on the distributed aspects.

FASTER [CPK⁺18] is a persistent KVS that also relies on a log-structured organization of records while enabling update-in-place for in-memory regions. *RStore* does not allow updates-in-place, since records are written directly to persistent storage and updating data in-place could lead to corruption and inconsistencies between replicas that could not be undone by our roll-forward recovery method. *FASTER* uses a lock-free hash table for indexing and epochs for concurrency control of operations such as garbage collection, index resizing, page flushing, and checkpointing. The index in *RStore* can be either a hash table or a search tree, in which case it also supports range scans. On one hand, *FASTER* does not keep the keys in the index, which reduces its DRAM footprint, on the other hand *RStore* saves memory by storing records in NVM. *FASTER* is able to leverage SSDs through its hybrid log, while *RStore* does not support SSD but supports NVM.

ScyllaDB [Scy15a] is a distributed database compatible with *Apache Cassandra*. It also focuses on low and predictable latency and implements an asynchronous execution model through *future-promise-continuation* concepts offered by the *Seastar Framework* [Scy15b]. *ScyllaDB* also uses user-space networking through DPDK for efficient package processing.

The project *Orleans* at Microsoft Research [BBG⁺14] offers a toolset for building cloud-native systems. It shares some of the high-level goals of *RStore*, such as following an actor-based model to enable easier development and scaling of largely distributed systems.

Different than *RStore*, that still implements logical logging separated from log-structured storage, *LogBase* [VWA⁺12] also implements a log-structured storage but relies on the atomicity of writes to completely get rid of write-ahead log. Nevertheless, *LogBase* manages the log-structured storage through files on SSD and delegates replication to HDFS.

6

CONCLUSION

This thesis discussed use-cases of non-volatile memory (NVM) in modern storage management architectures. We proposed new techniques to leverage NVM and enable systems to achieve higher performance and lower costs. Rather than radical changes, these techniques focused on gradual and incremental improvements over established architectures. Chapter 2 covered the relevant background and related work. Over the course of this thesis, each of the three main chapters investigated a different architecture. In the beginning of each of these chapters we raised two questions that guided the work described in the chapter. In the following we summarize each chapter and answer these questions.

Chapter 3 investigated opportunities to leverage NVM in the context of **log-structured merge-trees** (LSM). The first question posed was: *“what is the impact on LSMs if we replace all persistent storage by NVM?”*. To answer that, we initially relied solely on NVM as persistent storage of LevelDB, a popular LSM storage manager. We proposed *Pmemenv*, a persistent memory environment to enable direct and fine-grained management of LevelDB files (SSTs). Considering writing to NVM, we compared the performance between the regular file system interface and *Pmemenv*, for different *WriteBatch* (group commit) sizes. We showed that the impact of NVM and *Pmemenv* on LSMs is that smaller *WriteBatch* sizes can be used to achieve a lower average latency without major negative impacts on the throughput. In other words, the benefit of batching multiple writes to amortize the access to persistent media is lower when NVM is accessed directly in LSMs. The possibility of directly accessing NVM through *Pmemenv* then raised a second question: *“do LSMs still benefit from DRAM caches when NVM is used as persistent storage?”*. We observed that statically placing hot data, such as index blocks, in DRAM improves the performance. However, we have also noticed that a dynamic data placement in DRAM through a traditional cache component and replacement policy is not always beneficial. The main insight is that the overhead of copying a block from NVM to DRAM is only worth if this block will be read frequently enough in the future. Otherwise the block should be read directly in NVM to avoid this overhead. We enabled this behavior by implementing an NVM-aware cache component for LevelDB that uses **2Q as a placement policy to DRAM**. Our evaluation showed that 2Q never harms the performance and enables the system to make better decisions regarding which blocks should be moved to DRAM and which blocks should be read in NVM. We concluded Chapter 3 by applying the same concepts to build an NVM-aware *persistent block cache* for RocksDB, a more modern version of LevelDB, and showed that the benefits still hold when evaluated on real NVM hardware.

Chapter 4 discussed existing approaches to leverage NVM in the context of **B+Trees**. These approaches were classified in three categories. We observed that none of the categories can benefit from three characteristics at once: the persistency of NVM, the byte-addressability of NVM, and the update in-place strategy of B+Trees. The first question investigated was: “*how can B+Trees leverage both persistency and byte-addressability of NVM?*”. Like other works, we considered that integrating NVM in the existing buffer manager is a simple, yet powerful, approach. Not only NVM can be used to extend the capacity of the buffer pool, but the buffer manager enables B+Trees to use NVM in the context of a more complete storage hierarchy comprised of other devices, such as DRAM, SSD, and HDD. We proposed placing NVM side-by-side with DRAM, meaning that pages in NVM can be directly accessed without being copied to DRAM, i.e., pages are byte-addressable. In order to leverage the persistency and access pages on the NVM portion of the buffer pool after a restart, these pages must be consistent. We discussed how this consistency is hard to guarantee if records are updated in-place directly in NVM, which led us to our second question: “*how to handle corruptions of update-in-place strategies?*”. Rather than enforcing that pages are consistent at all times, we introduced the concept of **optimistic consistency**. We designed the *Persistent Buffer Pool with Optimistic Consistency* that does not enforce the consistency of pages in NVM and optimistically expect them not to be corrupted, but is still able to detect and repair corruptions. The main insight is that B+Trees are commonly used in a transactional environment and that durability is guaranteed through *write-ahead logging* (WAL), as proposed by the ARIES algorithm. Corrupted pages can be detected through checksums, and the WAL suffices to recover corrupted pages. In other words, corruptions of pages in NVM are generalized to regular media failures. This is achieved by applying the well-known concepts of checksums and single-page recovery to a new context: NVM. We evaluated the runtime overhead of calculating checksums through microbenchmarks and elaborated on the overall behavior of a complete system. We argued that the benefit of our *Persistent Buffer Pool with Optimistic Consistency* is enabling the system to adapt to different demands by explicitly trading high throughput for faster peak-performance recovery and lower costs.

Chapter 5 described the design of *RStore*, an **index+log** key-value store (KVS). *RStore* is targeted at use-cases such as web caching, IoT, and augmented reality. The typical workloads of these use-cases require low and predictable response times and are commonly addressed by single-level DRAM-only systems, such as *memcached* and *Redis*. The DRAM-only design implies that these systems have limited capacity and high costs, which raised our first question: “*how to extend the capacity and lower the costs of index+log KVSs?*”. While employing storage devices such as SSDs would address these issues, changes to adapt the single-level architecture to manage files would be required and the high latency of these devices would increase the response times. *RStore* employs NVM to store records in the log-structured area, while having a DRAM index to efficiently access these records. Since both NVM and DRAM share the same virtual memory address space, it maintains the single-level architecture of DRAM-only systems. NVM is also denser and cheaper than DRAM, which enable *RStore* to achieve larger capacity and lower costs. While the latency of NVM is higher than DRAM, it is still lower than flash-based SSDs and introduces a smaller response time penalty. However, looking only at the average latency of requests is not enough to ensure predictable response times in the targeted workloads, tail latency must also be considered. Therefore, the second question raised was: “*how to achieve short and predictable response times in the form of low tail latency?*”. In addition to NVM and the *index+log* architecture, *RStore* combines techniques such as message-passing communication, cooperative multitasking, and user-space networking. Our evaluation showed

that the combination of these design decisions enable *RStore* to achieve a competitive throughput and lower costs. The evaluation also shows that, even if NVM increases the response time of *RStore* in comparison to a DRAM-only variant, the tail latency does not drastically increase under high loads, which leads to a more robust behavior.

Other works explored NVM in the context of storage management and database systems by proposing novel architectures. Some of these works are very NVM-centric and disruptive, in the sense that they do not take into consideration current hardware and traditional software architectures. While novel architectures and a complete software rewrite might be required in a future where NVM will replace all storage devices, and possibly even DRAM, currently this is not the case. To enable modern systems to immediately leverage NVM, we argue for an incremental evolution rather than a revolution. This same approach was taken by works that improved database logging protocols by using NVM, however, other system components had not been explored enough yet. Furthermore, some of the techniques proposed in this thesis might sound too obvious once they have been presented, due to their simplicity. We argue that simplicity is major advantage. In other words, the novelty comes from applying well-known concepts, such as caching policies, buffer management, checksums, recovery, and log-structuring, to a new context: NVM.

Many opportunities exist for future work. In the context of LSMs, the merge process is critical component for achieving high performance, and therefore NVM-aware merge policies can lead to significant improvements. As for B+Trees, the complete implementation and evaluation of our *Persistent Buffer Pool with Optimistic Consistency* would constitute a major contribution. Unfortunately, even if modern transaction-oriented recovery techniques, such as *single-page recovery*, are well-understood and accepted, they have not been adopted by many systems yet. Since these techniques are a prerequisite for our proposed buffer pool, implementing them would require significant engineering effort that could not be completely tackled in this thesis, specially considering the extent of the effort required for integrating NVM in the code base of two complete systems used in production (LevelDB and RocksDB) and for implementing *RStore*. Nevertheless, we presented the initial idea and discussed its implementation and trade-offs. The *RStore* system was developed in the context of much larger industry project which aims at building a complete distributed system. We believe that the concepts of *RStore* provide a solid foundation that future work can extend to a distributed environment. Finally, we conclude by stating that, what is paramount for future work, and what we tried to achieve in this thesis, is that proposed techniques should be incremental rather than disruptive, in order to keep them simple and easy to be adopted by modern systems, or as once written by Jim Gray [GR93]:

"Don't be fooled by the many books on complexity or by the many complex and arcane algorithms you find in this book or elsewhere. Although there are no textbooks on simplicity, simple systems work and complex don't."

BIBLIOGRAPHY

- [ABGA17] Raja Appuswamy, Renata Borovica-Gajic, Goetz Graefe, and Anastasia Ailamaki. The Five-Minute Rule Thirty Years Later and its Impact on the Storage Hierarchy. In *Proc. ADMS (VLDB Workshop)*, pages 1–8, 2017.
- [Aco19] Paul Acorn. Intel Optane DIMM Pricing: \$695 for 128GB, \$2595 for 256GB, \$7816 for 512GB. <https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html>, 2019. Accessed: 2020-09-01.
- [AJ89] Rakesh Agrawal and HV Jagadish. Recovery Algorithms for Database Machines with Non-Volatile Main Memory. In *Proc. IWDW*, pages 269–285, 1989.
- [ALML18] Joy Arulraj, Justin J. Levandoski, Umar Farooq Minhas, and Per-Åke Larson. BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory. *PVLDB*, 11(5):553–565, 2018.
- [ALR⁺17] Mihnea Andrei, Christian Lemke, Günter Radestock, Robert Schulze, Carsten Thiel, Rolando Blanco, Akanksha Meghlan, Muhammad Sharique, Sebastian Seifert, Surendra Vishnoi, Daniel Booss, Thomas Peh, Ivan Schreter, Werner Thesing, Mehul Wagle, and Thomas Willhalm. SAP HANA Adoption of Non-Volatile Memory. *PVLDB*, 10(12):1754–1765, 2017.
- [Ama06] Amazon. Simple Storage Service (S3). <https://aws.amazon.com/s3/>, 2006. Accessed: 2020-09-01.
- [APM19] Joy Arulraj, Andy Pavlo, and Krishna Teja Malladi. Multi-Tier Buffer Management and Storage System Design for Non-Volatile Memory. *CoRR*, abs/1901.10938, 2019.
- [APP16] Joy Arulraj, Matthew Perron, and Andrew Pavlo. Write-Behind Logging. *PVLDB*, 10(4):337–348, 2016.
- [Arm03] Joe Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2003.
- [AXF⁺12] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proc. ACM SIGMETRICS/PERFORMANCE*, pages 53–64, 2012.

- [BBG⁺14] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed Virtual Actors for Programmability and Scalability. Technical Report MSR-TR-2014-41, Microsoft Research, 2014.
- [BCB16] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-Juergen Boehm. Makalu: Fast Recoverable Allocation of Non-Volatile Memory. In *Proc. OOPSLA*, pages 677–694, 2016.
- [BDF05] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-Oblivious B-Trees. *Proc. SIAM J. Comput.*, 35(2):341–358, 2005.
- [BDKM17] Philip A. Bernstein, Mohammad Dashti, Tim Kiefer, and David Maier. Indexing in an Actor-Oriented Database. In *Proc. CIDR*, 2017.
- [BFKT14] Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson. The Reactive Manifesto. <https://www.reactivemanifesto.org/>, 2014. Accessed: 2020-09-01.
- [BGH87] Joel Bartlett, Jim Gray, and Bob Horst. Fault Tolerance in Tandem Computer Systems. In *The Evolution of Fault-Tolerant Computing*, pages 55–76, 1987.
- [BRD11] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. Hyder - A Transactional Record Manager for Shared Flash. In *Proc. CIDR*, pages 9–20, 2011.
- [CAGM04] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. Improving Hash Join Performance through Prefetching. In *Proc. ICDE*, pages 116–127, 2004.
- [CCA⁺11] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories. In *Proc. ASPLOS*, pages 105–118, 2011.
- [CGN11] Shimin Chen, Phillip B. Gibbons, and Suman Nath. Rethinking Database Algorithms for Phase Change Memory. In *Proc. CIDR*, pages 21–31, 2011.
- [Cha78] H. Chang. On Bubble Memories and Relational Data Base. In *VLDB*, pages 207–229, 1978.
- [Cho17] Jeongdong Choe. Intel 3D XPoint Memory Die Removed from Intel Optane™ PCM (Phase Change Memory). <https://www.techinsights.com/blog/intel-3d-xpoint-memory-die-removed-intel-optanetm-pcm-phase-change-memory>, 2017. Accessed: 2020-09-01.
- [Chu71] Leon Chua. Memristor - The Missing Circuit Element. *IEEE Transactions on Circuit Theory*, 18(5):507–519, 1971.
- [Chu11] Leon Chua. Resistance Switching Memories are Memristors. *Applied Physics A*, 102(4):765–783, 2011.
- [Chu19] Leon O. Chua. Resistance Switching Memories are Memristors. In *Handbook of Memristor Networks*, pages 197–230. Springer, 2019.
- [CJ15] Shimin Chen and Qin Jin. Persistent B+-Trees in Non-Volatile Main Memory. *PVLDB*, 8(7):786–797, 2015.

- [CJY15] Kaimeng Chen, Peiquan Jin, and Lihua Yue. Efficient Buffer Management for PCM-Enhanced Hybrid Memory Architecture. In *Proc. APWeb*, volume 9313, pages 29–40, 2015.
- [CKKS89] George P. Copeland, Tom W. Keller, Ravi Krishnamurthy, and Marc G. Smith. The Case For Safe RAM. In *VLDB*, pages 327–335, 1989.
- [Com79] Douglas Comer. The Ubiquitous B-Tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.
- [CPK⁺18] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin J. Levandoski, James Hunter, and Mike Barnett. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proc. ACM SIGMOD*, pages 275–290, 2018.
- [CST⁺10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proc. SoCC*, pages 143–154, 2010.
- [DAI17] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal Navigable Key-Value Store. In *Proc. ACM SIGMOD*, pages 79–94, 2017.
- [DB13] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Commun. ACM*, 56(2):74–80, 2013.
- [DGS80] Keith L. Doty, Joel D. Greenblatt, and Stanley Y. W. Su. Magnetic Bubble Memory Architectures for Supporting Associative Searching of Relational Databases. *IEEE Transactions on Computers*, 29(11):957–970, 1980.
- [DHJ⁺07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon’s Highly Available Key-value Store. In *Proc. SOSP*, pages 205–220, 2007.
- [DHK⁺15] Biplob Debnath, Alireza Haghdooost, Asim Kadav, Mohammed G. Khatib, and Cristian Ungureanu. Revisiting Hash Table Design for Phase Change Memory. *SIGOPS Oper. Syst. Rev.*, 49(2):18–26, 2015.
- [DKO⁺84] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and David A. Wood. Implementation Techniques for Main Memory Database Systems. In *Proc. ACM SIGMOD*, pages 1–8, 1984.
- [Dul16] Subramanya R. Dulloor. *Systems and Applications for Persistent Memory*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 2016.
- [DVTBH13] Michael FL De Volder, Sameh H Tawfick, Ray H Baughman, and A John Hart. Carbon Nanotubes: Present and Future Commercial Applications. *Science*, 339(6119):535–539, 2013.
- [DWS⁺08] Xiangyu Dong, Xiaoxia Wu, Guangyu Sun, Yuan Xie, Hai Li, and Yiran Chen. Circuit and Microarchitecture Evaluation of 3D Stacking Magnetic RAM (MRAM) as a Universal Memory Replacement. In *Proc. ACM DAC*, pages 554–559, 2008.

- [EGA⁺18] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim M. Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing DRAM Footprint with NVM in Facebook. In *Proc. EuroSys*, pages 42:1–42:13, 2018.
- [Eva15] Jason Evangelho. Intel and Micron Jointly Unveil Disruptive, Game-Changing 3D XPoint Memory, 1000x Faster Than NAND. <https://hothardware.com/news/intel-and-micron-jointly-drop-disruptive-game-changing-3d-xpoint-cross-point-memory-1000x-faster-than-nand>, 2015. Accessed: 2020-09-01.
- [Fac12] Facebook. RocksDB. <https://rocksdb.org/>, 2012. Accessed: 2020-09-01.
- [FHH⁺11] Ru Fang, Hui-I Hsiao, Bin He, C. Mohan, and Yun Wang. High Performance Database Logging using Storage Class Memory. In *Proc. ICDE*, pages 1221–1231, 2011.
- [FML⁺12] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.
- [GG97] Jim Gray and Goetz Graefe. The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb. *ACM SIGMOD Record*, 26(4):63–68, 1997.
- [GGS16] Goetz Graefe, Wey Guy, and Caetano Sauer. *Instant Recovery with Write-Ahead Logging: Page Repair, System Restart, Media Restore, and System Failover, Second Edition*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2016.
- [GK12] Goetz Graefe and Harumi A. Kuno. Definition, Detection, and Recovery of Single-Page Failures, a Fourth Class of Database Failures. *PVLDB*, 5(7):646–655, 2012.
- [GKC⁺11] B Govoreanu, GS Kar, YY Chen, V Paraschiv, S Kubicek, A Fantini, IP Radu, L Goux, S Clima, R Degraeve, et al. 10× 10nm² Hf/HfO_x Crossbar Resistive RAM with Excellent Performance, Reliability and Low-Energy Operation. In *IEEE IEDM Technical Digest*, pages 31–6, 2011.
- [GKP⁺10] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudré-Mauroux, and Samuel Madden. HYRISE - A Main Memory Hybrid Storage Engine. *PVLDB*, 4(2):105–116, 2010.
- [GL01] Goetz Graefe and Per-Åke Larson. B-Tree Indexes and CPU Caches. In *Proc. ICDE*, pages 349–358, 2001.
- [GP87] Jim Gray and Gianfranco R. Putzolu. The 5 Minute Rule for Trading Memory for Disk Accesses and The 10 Byte Rule for Trading Memory for CPU Time. In *Proc. ACM SIGMOD*, pages 395–398, 1987.
- [GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

- [Gra07] Goetz Graefe. The Five-Minute Rule Twenty Years Later, and how Flash Memory Changes the Rules. In *Proc. DaMoN (ACM SIGMOD Workshop)*, page 6, 2007.
- [Gre13] Brendan Gregg. *Systems Performance: Enterprise and the Cloud*. Prentice Hall Press, USA, 1st edition, 2013.
- [GSE⁺94] Jim Gray, Prakash Sundaresan, Susanne Englert, Kenneth Baclawski, and Peter J. Weinberger. Quickly Generating Billion-Record Synthetic Databases. In *Proc. ACM SIGMOD*, pages 243–252, 1994.
- [GVK⁺14] Goetz Graefe, Haris Volos, Hideaki Kimura, Harumi A. Kuno, Joseph Tucek, Mark Lillibridge, and Alistair C. Veitch. In-Memory Performance for Big Data. *PVLDB*, 8(1):37–48, 2014.
- [GXH⁺11] Shen Gao, Jianliang Xu, Bingsheng He, Byron Choi, and Haibo Hu. PCMLogging: Reducing Transaction Logging Overhead with PCM. In *Proceedings of the 20th ACM Conference on Information and Knowledge Management (CIKM)*, pages 2401–2404. ACM, 2011.
- [H⁺08] Yiming Huai et al. Spin-Transfer Torque MRAM (STT-MRAM): Challenges and Prospects. *AAPPS Bulletin*, 18(6):33–40, 2008.
- [HAMS08] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP Through the Looking Glass, and What We Found There. In *Proc. ACM SIGMOD*, pages 981–992, 2008.
- [HBS73] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proc. IJCAI*, pages 235–245, 1973.
- [HKWN18] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *Proc. USENIX FAST*, pages 187–200, 2018.
- [HPM⁺13] Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. Scala Documentation: Futures and Promises. <https://docs.scala-lang.org/overviews/core/futures.html>, 2013. Accessed: 2020-09-01.
- [HR83] Theo Härder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.
- [HRB⁺17] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. Log-Structured Non-Volatile Main Memory. In *Proc. USENIX ATC*, pages 703–717, 2017.
- [HSQ14] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. NVRAM-Aware Logging in Transaction Systems. *PVLDB*, 8(4):389–400, 2014.
- [HYY⁺05] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, H. Nagao, and H. Kano. A Novel Nonvolatile Memory with Spin Torque Transfer Magnetization Switching: SpinRAM. In *IEEE IEDM Technical Digest*, pages 459–462, 2005.

- [IC20] Stratos Idreos and Mark Callaghan. Key-Value Storage Engines. In *Proc. ACM SIGMOD*, pages 2667–2672, 2020.
- [IKK16] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proc. ASPLOS*, pages 427–442. ACM, 2016.
- [Int14] Intel Corporation. Persistent Memory Development Kit (PMDK). <https://github.com/pmem/pmdk/>, 2014. Accessed: 2020-09-01.
- [Int19] Intel Corporation. Processor Counter Monitor. <https://github.com/opcm/pcm>, 2019. Accessed: 2020-09-01.
- [Int20a] Intel Corporation. Instruction Set Extensions Technology. https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=SSE4_2, 2020. Accessed: 2020-09-01.
- [Int20b] Intel Corporation. Intel Optane DC SSD Series. <https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/optane-dc-ssd-series.html>, 2020. Accessed: 2020-09-01.
- [Int20c] Intel Corporation. Intel Optane Persistent Memory. <https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html>, 2020. Accessed: 2020-09-01.
- [Int20d] Intel Corporation. Intel® 64 and IA-32 Architectures Optimization Reference Manual. <https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html>, 2020. Accessed: 2020-09-01.
- [IYZ⁺19] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR*, abs/1903.05714, 2019.
- [JS94] Theodore Johnson and Dennis E. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *VLDB*, pages 439–450, 1994.
- [KELS62] Tom Kilburn, David B. G. Edwards, Michael J. Lanigan, and Frank H. Sumner. One-Level Storage System. *IRE Trans. Electron. Comput.*, 11(2):223–235, 1962.
- [KFG15] Yusuf Onur Koçberber, Babak Falsafi, and Boris Grot. Asynchronous Memory Access Chaining. *PVLDB*, 9(4):252–263, 2015.
- [Kim15] Hideaki Kimura. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *Proc. ACM SIGMOD*, pages 691–706, 2015.
- [Knu98] Donald Ervin Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching, 2nd Edition*. Addison-Wesley, 1998.
- [KR79] H. T. Kung and John T. Robinson. On Optimistic Methods for Concurrency Control. In *VLDB*, page 351, 1979.

- [LAK16] Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Towards Accurate and Fast Evaluation of Multi-Stage Log-structured Designs. In *Proc. USENIX FAST*, pages 149–166, 2016.
- [Lam78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.
- [LBD⁺11] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwillig. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *PVLDB*, 5(4):298–309, 2011.
- [LFAK11] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *Proc. SOSP*, pages 1–13, 2011.
- [LHKN18] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. LeanStore: In-Memory Data Management beyond Main Memory. In *Proc. ICDE*, pages 185–196, 2018.
- [LHO⁺19] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. Evaluating Persistent Memory Range Indexes. *PVLDB*, 13(4):574–587, 2019.
- [LHWL20] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable Hashing on Persistent Memory. *PVLDB*, 13(8):1147–1161, 2020.
- [Lin10] Linux Foundation. DPDK. <https://www.dpdk.org/>, 2010. Accessed: 2020-09-01.
- [Lin15] Linux Foundation. LIBNVDIMM: Non-Volatile Devices. <https://www.kernel.org/doc/Documentation/nvdim/nvdim.txt>, 2015. Accessed: 2020-09-01.
- [LJW⁺19] Ruicheng Liu, Peiquan Jin, Zhangling Wu, Xiaoliang Wang, Shouhong Wan, and Bei Hua. Efficient Wear Leveling for PCM/DRAM-Based Hybrid Memory. In *IEEE HPCC/SmartCity/DSS*, pages 1979–1986, 2019.
- [LLO19] Lucas Lersch, Wolfgang Lehner, and Ismail Oukid. Persistent Buffer Management with Optimistic Consistency. In *Proc. DaMoN (ACM SIGMOD Workshop)*, pages 14:1–14:3, 2019.
- [LLS13] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. LLAMA: A Cache/Storage Subsystem for Modern Hardware. *PVLDB*, 6(10):877–888, 2013.
- [LLS⁺17] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *Proc. USENIX FAST*, pages 257–270, 2017.
- [LOLS17] Lucas Lersch, Ismail Oukid, Wolfgang Lehner, and Ivan Schreter. An analysis of LSM caching in NVRAM. In *Proc. DaMoN (ACM SIGMOD Workshop)*, pages 9:1–9:5, 2017.
- [Lom93] David B. Lomet. The Case for Log Structuring in Database Systems. In *Proc. HPTS*, pages 136–140, 1993.

- [Lor77] Raymond A. Lorie. Physical Integrity in a Large Segmented Database. *ACM TODS*, 2(1):91–104, 1977.
- [LOSL17] Lucas Lersch, Ismail Oukid, Ivan Schreter, and Wolfgang Lehner. Rethinking DRAM Caching for LSMs in an NVRAM Environment. In *ADBIS*, volume 10509, pages 326–340, 2017.
- [LSOL20] Lucas Lersch, Ivan Schreter, Ismail Oukid, and Wolfgang Lehner. Enabling Low Tail Latency on Multicore Key-Value Stores. *PVLDB*, 13(7):1091–1104, 2020.
- [LXCW19] Mengxing Liu, Jiankai Xing, Kang Chen, and Yongwei Wu. Building Scalable NVM-based B+tree with HTM. In *Proc. ICPP*, pages 101:1–101:10, 2019.
- [LZY⁺10] Benjamin C. Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. Phase-Change Technology and the Future of Main Memory. *IEEE Micro*, 30(1):143, 2010.
- [MHL⁺92] C. Mohan, Don Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM TODS*, 17(1):94–162, 1992.
- [Mic17] Micron Technology, Inc. Micron Advances Persistent Memory with 32GB NVDIMM. <https://investors.micron.com/news-releases/news-release-details/micron-advances-persistent-memory-32gb-nvdimm>, 2017. Accessed: 2020-09-01.
- [Mic18] Micron Technology, Inc. NVDIMM. <https://www.micron.com/products/dram-modules/nvdimm>, 2018. Accessed: 2020-09-01.
- [Mic20] Micron Technology, Inc. 3D XPoint Technology. <https://www.micron.com/products/advanced-solutions/3d-xpoint-technology>, 2020. Accessed: 2020-09-01.
- [MWMS14] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. Rethinking Main Memory OLTP Recovery. In *Proc. ICDE*, pages 604–615, 2014.
- [NCC⁺19] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H. Noh, and Beomseok Nam. Write-Optimized Dynamic Hashing for Persistent Memory. In *Proc. USENIX FAST*, pages 31–44, 2019.
- [NIK⁺17] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. Dalí: A Periodically Persistent Hash Map. In *Proc. DISC*, volume 91 of *LIPICs*, pages 37:1–37:16, 2017.
- [OBL⁺14] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. SOFORT: A Hybrid SCM-DRAM Storage Engine for Fast Data Recovery. In *Proc. DaMoN (ACM SIGMOD Workshop)*, pages 8:1–8:7, 2014.
- [OBL⁺17] Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes. Memory Management Techniques for Large-Scale Persistent-Main-Memory Systems. *PVLDB*, 10(11):1166–1177, 2017.
- [OCG096] Patrick E. O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O’Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.*, 33(4):351–385, 1996.

- [OCXH14] Yi Ou, Lei Chen, Jianliang Xu, and Theo Härder. Wear-Aware Algorithms for PCM-Based Database Buffer Pools. In *Web-Age Information Management - WAIM 2014 International Workshops: BigEM, HardBD, DaNoS, HRSUNE, BIDASYS*, volume 8597 of *Lecture Notes in Computer Science*, pages 165–176. Springer, 2014.
- [OGG⁺15] John K. Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen M. Rumble, Ryan Stutsman, and Stephen Yang. The RAMCloud Storage System. *ACM TOCS*, 33(3):7:1–7:55, 2015.
- [OLN⁺16] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proc. ACM SIGMOD*, pages 371–386, 2016.
- [OO14] Diego Ongaro and John K. Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proc. USENIX ATC*, pages 305–319, 2014.
- [Ouk18] Ismail Oukid. *Architectural Principles for Database Systems on Storage-Class Memory*. PhD thesis, Dresden University of Technology, Germany, 2018.
- [Ovs68] Stanford R. Ovshinsky. Reversible Electrical Switching Phenomena in Disordered Structures. *Phys. Rev. Lett.*, 21:1450–1453, Nov 1968.
- [PAA⁺17] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C. Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. Self-Driving Database Management Systems. In *Proc. CIDR*, 2017.
- [PLMA17] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. Interleaving with Coroutines: A Practical Approach for Robust Index Joins. *PVLDB*, 11(2):230–242, 2017.
- [PPM16] Kisalaya Prasad, Avanti Patil, and Heather Miller. Programming Models for Distributed Computing: Futures and Promises. <http://dist-prog-book.com/chapter/2/futures.html>, 2016. Accessed: 2020-09-01.
- [PSU⁺70] A Pohm, C Sie, R Uttecht, V Kao, and O Agrawal. Chalcogenide Glass Bistable Resistivity (Ovonic) Memories. *IEEE Transactions on Magnetics*, 6(3):592–592, 1970.
- [PWGB13] Steven Pelley, Thomas F. Wenisch, Brian T. Gold, and Bill Bridge. Storage Management in the NVRAM Era. *PVLDB*, 7(2):121–132, 2013.
- [RK014] Stephen M. Rumble, Ankita Kejriwal, and John K. Ousterhout. Log-structured Memory for DRAM-based Storage. In *Proc. USENIX FAST*, pages 1–16, 2014.
- [RO92] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM TOCS*, 10(1):26–52, 1992.
- [Sau17] Caetano Sauer. *Modern Techniques for Transaction-Oriented Database Recovery*. PhD thesis, Kaiserslautern University of Technology, Germany, 2017.

- [Scy15a] ScyllaDB Inc. ScyllaDB. <https://www.scylladb.com/>, 2015. Accessed: 2020-09-01.
- [Scy15b] ScyllaDB Inc. Seastar. <http://seastar.io/>, 2015. Accessed: 2020-09-01.
- [SDUP15] David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. NVC-Hashmap: A Persistent and Concurrent Hashmap For Non-Volatile Memories. In *Proc. IMDM (VLDB Workshop)*, pages 4:1–4:8, 2015.
- [SGG14] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts Essentials, 2nd Edition*. Wiley, 2014.
- [SGH18] Caetano Sauer, Goetz Graefe, and Theo Härder. FineLine: Log-Structured Transactional Storage and Recovery. *PVLDB*, 11(13):2249–2262, 2018.
- [SKD⁺16] David Schwalb, Girish Kumar, Markus Dreseler, Anusha S., Martin Faust, Adolf Hohl, Tim Berning, Gaurav Makkar, Hasso Plattner, and Parag Deshmukh. Hyrise-NV: Instant Recovery for In-Memory Databases Using Non-Volatile Memory. In *Proc. DASFAA*, volume 9643, pages 267–282, 2016.
- [SMA⁺07] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era (It’s Time for a Complete Rewrite). In *VLDB*, pages 1150–1160, 2007.
- [Spo14] Spotify. Sparkley: Simple Constant Key/Value Storage Library, for Read-heavy Systems with Infrequent Large Bulk Inserts. <https://github.com/spotify/sparkey>, 2014. Accessed: 2020-09-01.
- [SS10] Justin Sheehy and David Smith. Bitcask: A Log-Structured Hash Table for Fast Key/Value Data. <https://riak.com/assets/bitcask-intro.pdf>, 2010. Accessed: 2020-09-01.
- [SSSW08] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. The Missing Memristor Found. *Nature*, 453(7191):80–83, 2008.
- [STPA16] Utku Sirin, Pinar Tözün, Danica Porobic, and Anastasia Ailamaki. Micro-architectural Analysis of In-memory OLTP. In *Proc. ACM SIGMOD*, pages 387–402, 2016.
- [Tec18] Viking Technology. DDR4 NVDIMM. <https://www.vikingtechnology.com/products/nvdim/DDR4-nvdim/>, 2018. Accessed: 2020-09-01.
- [Tob16] Tobias Klima. Using Non-volatile Memory (NVDIMM-N) as Byte-Addressable Storage in Windows Server 2016. <https://channel9.msdn.com/events/build/2016/p470>, 2016. Accessed: 2020-09-01.
- [TZK⁺13] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy Transactions in Multicore In-Memory Databases. In Michael Kaminsky and Mike Dahlin, editors, *Proc. SOSP*, pages 18–32, 2013.
- [VKW⁺13] Vish Viswanathan, Karthik Kumar, Thomas Willhalm, Patrick Lu, Blazej Filipiak, and Sri Sakthivelu. Intel Memory Latency Checker. <https://software.intel.com/content/www/us/en/develop/articles/intelr-memory-latency-checker.html>, 2013. Accessed: 2020-09-01.

- [vRLK⁺18] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. Managing Non-Volatile Memory in Database Systems. In *Proc. ACM SIGMOD*, pages 1541–1555, 2018.
- [vRVL⁺19] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. Persistent Memory I/O Primitives. In *Proc. DaMoN (ACM SIGMOD Workshop)*, pages 12:1–12:7, 2019.
- [VTRC11] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *Proc. USENIX FAST*, pages 61–75, 2011.
- [VWA⁺12] Hoang Tam Vo, Sheng Wang, Divyakant Agrawal, Gang Chen, and Beng Chin Ooi. LogBase: A Scalable Log-structured Database System in the Cloud. *PVLDB*, 5(10):1004–1015, 2012.
- [WFLH18] Chenggang Wu, Jose M. Faleiro, Yihan Lin, and Joseph M. Hellerstein. Anna: A KVS for Any Scale. In *Proc. ICDE*, pages 401–412, 2018.
- [WJ14] Tianzheng Wang and Ryan Johnson. Scalable Logging through Emerging Non-Volatile Memory. *PVLDB*, 7(10):865–876, 2014.
- [WK16] Tianzheng Wang and Hideaki Kimura. Mostly-Optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores. *PVLDB*, 10(2):49–60, 2016.
- [WLL18] Tianzheng Wang, Justin J. Levandoski, and Per-Åke Larson. Easy Lock-Free Indexing in Non-Volatile Memory. In *Proc. ICDE*, pages 461–472, 2018.
- [Wri83] William E. Wright. Some File Structure Considerations Pertaining to Magnetic Bubble Memory. *Computer Journal*, 26(1):43–51, 1983.
- [XJXS17] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *Proc. USENIX ATC*, pages 349–362, 2017.
- [XS16] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proc. USENIX FAST*, pages 323–338, 2016.
- [YKH⁺20] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proc. USENIX FAST*, pages 169–182, 2020.
- [YWC⁺15] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proc. USENIX FAST*, pages 167–181, 2015.
- [Zak81] M. Zaki. Magnetic Bubble Memory Structures for Relational Database Management Systems. *International Journal of Computer & Information Sciences*, 10(5):341–358, 1981.

- [ZAP⁺16] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *Proc. ACM SIGMOD*, pages 1567–1581, 2016.
- [ZH18] Pengfei Zuo and Yu Hua. A Write-Friendly and Cache-Optimized Hashing Scheme for Non-Volatile Memory Systems. *IEEE TPDS*, 29(5):985–998, 2018.
- [ZHW18] Pengfei Zuo, Yu Hua, and Jie Wu. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *Proc. USENIX OSDI*, pages 461–476, 2018.
- [ZTKL14] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *Proc. USENIX OSDI*, pages 465–477, 2014.



PIBENCH

We designed **PiBench** to allow an unified and fair comparison of different persistent data structures, and easy adoption by future work. As Figure A.1 shows, the data strucure being tested must be compiled into a shared library and linked to PiBench following a defined API, or through a wrapper that translates requests from PiBench's API. The API consists of a pure abstract class that encapsulates common operations (insert, lookup, delete, scan, update) and a `create_index` function for instantiating the benchmarked data structure. To use PiBench, the user only needs to derive a class that implements the API. PiBench then issues requests against the instantiated object.

PiBench executes a `load` phase and a `run` phase, like YCSB [CST⁺10]. It provides various options for customization, such as key/value sizes, the number of records to be loaded, the numbers and types of operations to be executed, and ratio of each type of operation. Keys and values are generated randomly following a chosen distribution and seed to allow reproducible executions. PiBench supports three random distributions as defined by Gray et al. [GSE⁺94]: `uniform`, `self similar`, and `zipfian`. Since the random distributions generate integers in a contiguous range, with the skewed distributions favoring smaller values, we apply a multiplicative hashing function [Knu98] to each generated integer to scatter the keys across the complete integer domain, thus avoiding frequently accessed keys to be clustered together. A prefix can be prepended to keys to analyze the impact of key compression and comparison methods. PiBench uses multiple threads to issue requests and relies on the data structure under evaluation to handle concurrent accesses.

PiBench dedicates a thread to periodically collect statistics, such as the number of completed operations within a specified time window. This allows a better understanding of performance over time by enabling standard deviation to be easily calculated in addition to the average throughput. Finally, we use the Processor Counter Monitor (PCM) library [Int19] and `ipmwatch`¹ to collect hardware counter metrics (such as memory accesses and cache misses). As shown in Figure A.2, PCM measures memory traffic between CPU caches and both DRAM and DCPMMs at 64 B granularity. The DCPMMs rely on a buffer layer to hold hot data [Int20d, YKH⁺20]. We use `ipmwatch` to measure the traffic between the buffer and the media itself, which happens at 256 B granularity. Finally, PiBench is open-source² and available at a web application³.

¹Available as part of Intel VTune Amplifier 2019 since Update 5.

²<https://github.com/sfu-dis/pibench>

³<https://pibench.org/>

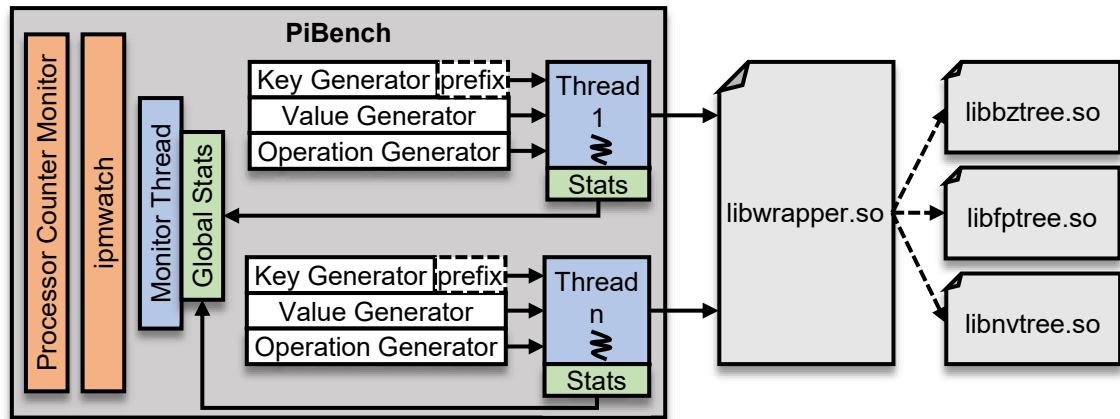


Figure A.1: Overview of the PiBench architecture. Independent worker threads issue requests to data structures and a monitor thread aggregates them in specified time intervals.

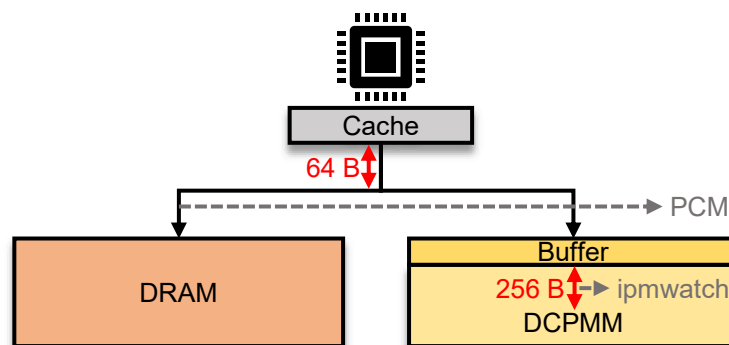
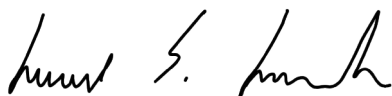


Figure A.2: Tools used by PiBench to measure memory traffic. PCM measures traffic on the memory bus, which happens at a cache line granularity (64 B). The *ipmwatch* tool is DCPMM specific, and it measures the traffic between the on-chip buffer and the underlying persistent media, which happens at a granularity of 256 B.

CONFIRMATION

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

A handwritten signature in black ink, appearing to be 'Hendrik S. ...'.

Dresden, den 30. Oktober 2020