TECHNISCHE UNIVERSITÄT MÜNCHEN

TUM School of
Computation, Information and Technology

DOCTORAL THESIS

Persistent Memory in Database Systems

Alexander van Renen

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz:		Prof. Dr. Stefanie Rinderle-Ma
Prüfer der Dissertation:	1.	Prof. Alfons Kemper, Ph.D.
	2.	Prof. Andy Pavlo, Ph.D.
	3.	Prof. Dr. Thomas Neumann

Die Dissertation wurde am 18.01.2022 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 19.05.2022 angenommen.

Abstract

Persistent memory is a promising new hardware that aims to close the gap between traditional storage devices (such as SSD and HDD) and main memory (i.e., DRAM). It inherits the persistency and capacity of SSDs while retaining the low latency and high bandwidth of DRAM at affordable costs. However, there is no free lunch: persistent memory is unlikely to outright eclipse either one and therefore exists as a hybrid device between memory and storage. Nonetheless, its properties are unique and it is up to the systems community to define its role in future software.

In this dissertation, we make two contributions towards this goal. First, we propose an optimized integration of persistent memory into database management systems that makes full use of its persistency, low latency and byte-addressability. Our OLTP-focused system is build on a buffer manager that employs pointer swizzling and different page sizes that can be lazily loaded. It is build to gracefully scale over a three tier storage hierarchy with DRAM, persistent memory and SSDs delivering competitive performance to layer-specialized systems.

Second, we provide one of the first evaluations of actual persistent memory hardware. Our investigation of the bandwidth, latency, and special characteristics is an essential contribution to understanding persistent memory and assists developers in optimizing algorithms and data structures. Using these findings we define a set of guidelines for the efficient use of persistent memory. Adhering to these insights we design highly-tuned algorithms for logging, page propagation, and in-place updates. Further, we suggest a novel technique to interleave multiple persistent writes using user space threads that allows to largely avoid the memory stall associated with a persistent write operation.

Zusammenfassung

Persistenter Speicher ist eine neue vielversprechende Hardware deren Ziel es ist die Lücke zwischen klassischen Plattenspeicher (e.g., SSD oder HDD) und Hauptspeicher (i.e., DRAM) zu schließen. Er versucht die positiven Eigenschaften der beiden Seiten zu kombinieren: Persistenz und hohe Kapazitäten mit niedriger Latenz und hoher Bandbreite zu kompetitiven Preisen. Allerdings, kann man nicht alles haben und somit übertrifft die neue Hardware keine der bestehenden Technologien in Leistung. Deswegen wird vermutet, dass sich persistenter Speicher zunächst als Hybridgerät zwischen Hauptspeicher und Plattenspeicher etabiert. Nichtsdestotrotz sind seine Eigenschaften einzigartig und es ist in den Händen der Gesellschaft für Systementwicklung seine zukünftige Rolle zu definieren.

Diese Dissertation macht zwei Beiträge für in Bezug auf diese Herrausforderung: Als erstes wird eine optimierte Integration von persistentem Speicher in Datenbankmanagementsysteme erarbeitet, welche die Persistenz, niedrigen Latenz, und Byteweise Adressierbarkeit voll ausnutzt. Unser auf OLTP fokussiertes System setzt auf einem Puffermanager mit Zeigerfaltung und variabel großen Seiten, welche dynamisch geladen werden, auf. Es ist entworfen um flüssig über eine Speicherhierarchie mit DRAM, persistenten Speicher und SSD zu skalieren während es kompetitive Performanz zu Systemen liefert die auf eine einzelne Ebene spezialisiert sind.

Als zweitens zeigen wir eine der ersten Auswertungen von echter persistenter Speicher Hardware. Unsere Untersuchung von Bandbreite, Latenz und besonderen Eigenschaften ist ein wesentlicher Beitrag zum Verständnis von dieser Hardware und unterstützt somit Entwickler bei dem Entwurf von Algorithmen und Datenstrukturen. Basierend auf diesen Erkenntnissen werden eine Reihe von Richtlinien für die effiziente Nutzung von persistentem Speicher definiert. Unter deren Berücksichtigung werden dann hochgradig angepasste Algorithmen für Protokollierung, Seitenzurückschreibung und direkten Änderungen entwickelt. Darüber hinaus schlagen wir eine neuartige Technik vor, um mehrere persistente Schreiboperationen unter Verwendung von Nutzer-Ausführungssträngen zu bündeln. Dies ermöglicht es den mit einer synchronen persistenten Schreiboperation verbundenen Systemstillstand weitgehend zu umgehen.b

Contents

Abstract **iii**

1 Introduction **3**
1.1 Persistent Memory . . . 3
1.1.1 Definition . . . 4
1.1.2 Characteristics . . . 6
1.1.3 Challenges . . . 9
1.1.4 Opportunities . . . 13
1.2 Research Methodology . . . 16
1.2.1 Experimental Setup and Methods . . . 16
1.2.2 Evaluation Platforms . . . 17
1.3 Related Work . . . 19
1.3.1 Persistent Memory Development Kit (PMDK) . . . 19
1.3.2 B-Tree-like Index Structures . . . 20
1.3.3 Database Architectures . . . 24

2 Paper 1: Managing Non-Volatile Memory in Database Systems **27**

3 Paper 2: Building Blocks for Persistent Memory **43**

Preface

Our world is powered by (or, more pessimistically: run by) information. Improving the way we store and access information has been a major driving force for scientific, cultural, and social advancements. Arguably, we have come a long way from ancient stone carvings, over the development of ink and paper, to the printing press, and, most recently, semiconductor-based computers. While a printing press could reproduce the same page several times per minute, a modern computer can duplicate this entire dissertation several hunderet times per second and, unlike any preceding device: run complex analytics on it.

This dissertation concerns itself with the storage mediums of modern computers (i.e., the paper of a printing press). Specifically, it investigates the properties, best usage practices, and integration strategies into existing software systems, such as database management systems, for a novel storage technology: *Persistent Memory* promises unprecedented speeds in combination with large capacities and competitive prices. However, there is no free lunch and, thus, the role of Persistent Memory in modern computer architectures is yet to be determined. And while Persistent Memory, obviously, does not introduce the next printing press, it (and, therefore, this dissertation) can hopefully be an improvement on one of its parts.

Chapter 1

Introduction

Database management systems belong to the most widely deployed software. For instance, the SQLite developers estimates that there are around 1 000 000 000 000 (one trillion) actively used SQLite databases:

> *"Billions and billions of copies of SQLite exist in the wild."* – SQLite Website [Webb]

While SQLite might not be seen as the most innovative incarnation of a database management system in existence, the point stands: databases are everywhere and serve as the underpinning of many applications. One of the key reasons for this popularity is the abstraction that database management systems implement: a high level query language (e.g., SQL) on top of a common data layout (e.g., relational) with useful consistency constraints (e.g., ACID). In particular, they hide all details about the used hardware and, in turn, can transparently deliver the performance of any new hardware to applications.

One such new hardware is *persistent memory*: a device that combines properties of volatile main memory (DRAM) and non-volatile storage (e.g., SSD). Persistent memory is a promising technology for improving database systems. In this dissertation, we investigate its properties, develop a set of efficient low-level algorithms, and propose a novel approach to integrate it into database management systems.

In the following sections, we first describe what persistent memory is, how it works, and where it fits into the existing memory hierarchy. We then lay out the main challenges involving the integration of persistent memory into existing software with a strong focus on database management systems. Afterwards, we summarize the contributions made and methodologies used in the publications this dissertation is comprised of. Lastly, we give a focused overview of related works in the field of range index structures and full databases architectures for persistent memory.

1.1 Persistent Memory

As a basis for further discussion, we shall first introduce persistent memory into the existing memory hierarchy, discuss its properties, and explain how it can be used. We, intentionally, shall not dive too deep into the technical details and performance characteristics of persistent memory, as their investigation and description is one of the main contribution of this dissertation (see Chapter 3).

1.1.1 Definition

A traditional computer architecture, based on the von Neumann architecture [vN45], consists of a central processing unit, an input/ouput device, memory and storage[1]. Throughout this dissertation, we differentiate between the terms *memory* and *storage* in the following way:

Memory is volatile and, therefore, loses its state when not constantly supplied with electrical energy. It usually has a smaller capacity than storage devices and offers fine-granular access with low latencies. The most prominent example is dynamic random access memory: DRAM, which is part of almost all modern computing systems (home and server). One exception, which is not the focus of this work, are embedded systems where static random access memory (SRAM) is commonly used.

Storage is non-volatile and, therefore, retains its state over long durations of time (decades) without the need for electrical power. It usually has a larger capacity than memory, no fine-grained access, higher latencies, and a lower bandwidth. Typical examples include solid state drives (SSD) and hard disk drives (HDD). Nowadays, at least one storage device is part of every common home and server computer in order store data persistently.

In the current computer architecture both, memory and storage, have to exist side by side: Due to the volatility of memory, data has to be written to a storage device in order to retain the data for longer durations of time and across power outages. Additionally, the low capacity and high price of memory devices make it economically inefficient to store huge data sets. However, due the high latency of storage devices, data has to be moved to memory in order to perform computations on that data in an efficient manner. In fact, typical modern computers are not capable of processing data directly on a storage device.

Persistent memory (PMem)[2] refers to a device that sits at the intersection between memory and storage, inheriting advantageous properties of both. Ideally, persistent memory would make the separation of memory and storage obsolete and replace both. However, as usual, there is likely no free lunch: persistent memory, as a concept, is unlikely to be superior to both memory and storage at the same time. Throughout this dissertation, we use the following definition of persistent memory, which is in line with a common understanding in research and industry:

Persistent Memory refers to a fine-granular random-access device with low latencies and high bandwidth. It offers large capacities and is capable of retaining its state without electrical power over a long duration of time. More precisely, its access granularity and latency needs to be sufficiently small that it can be utilized directly by a CPU via load and store instructions. Further, its capacity needs to be sufficiently large that it can be used as a permanent data storage location for large amounts of data. And, for practical purposes, the cost of persistent memory needs to be small enough to make it economically feasible.

This abstract idea of persistent memory has been around for several decades (e.g., [AJ89]). However, making the transition from an idea to actual hardware took

[1] In the original von Neumann architecture storage was considered a part of the input/output system. Nowadays, however, it is often seen as a separate entity.

[2] Also referred to as non-volatile memory (NVM), non-volatile random access memory (NVRAM), or storage class memory (SCM).

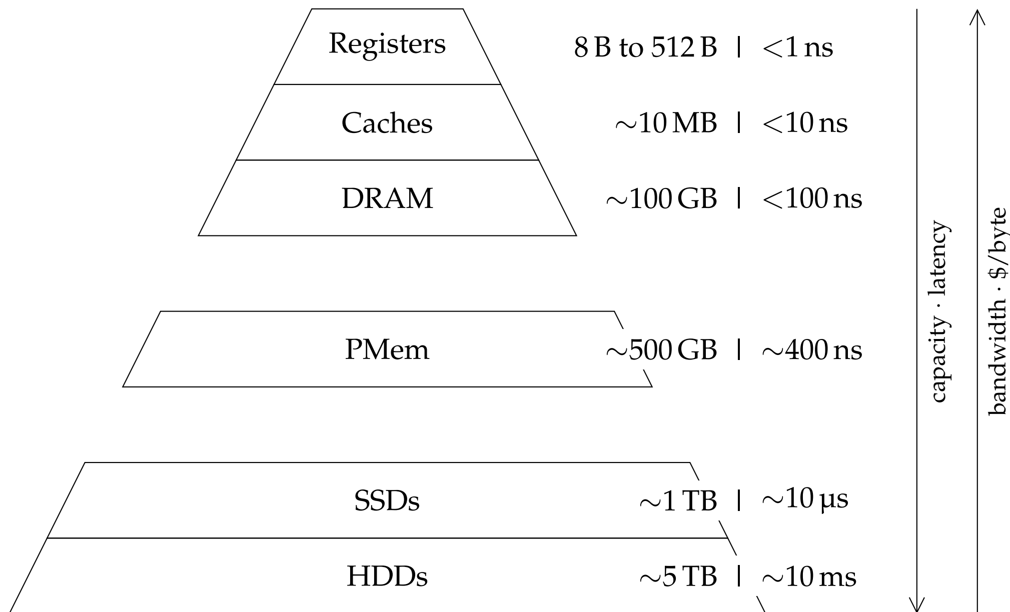


FIGURE 1.1: **Memory Hierarchy** with persistent memory.

until recently: The first and, as of writing (mid 2021), only official release of a practical implementation of persistent memory happened in April of 2019: Intel introduced the so-called Intel® Optane™ DC Persistent Memory [Intb, Intc] device. With this being the only persistent memory hardware available, all non-simulation-based research is conducted on this hardware and we therefore simply refer to it as persistent memory. Note, however, that most ideas in this dissertation and other research on persistent memory extend to the general concept of persistent memory, as previously defined, and is not tied to Intel's incarnation of it.

Figure 1.1 shows the new memory hierarchy with persistent memory added. The individual devices are depicted in a pyramid shape on the left. Typical performance numbers for the individual layers are shown on the right. While the lower section of the pyramid depicts storage devices (SSDs and HDDs) the upper part shows memory devices (DRAM, caches, and registers). There is a clear *gap* in all relevant metrics (bandwidth, latency, capacity, and cost) between the fastest storage device (SSD) and the slowest memory device (DRAM)[3].

This gap constitutes a crux in database design (and, in fact, all storage related systems software) as data needs to be processed in memory, but persistently stored on storage devices [GP87]. Disk-based database systems are designed around this gap: data is bundled together into larger units (pages) which are then transferred between memory and storage via a buffer manager. Logging is then needed to ensure durability guarantees for the data. This constitutes a major performance bottleneck [HAMS08] for database performance (around 45%). Decades of research have produced countless techniques to optimize and avoid this *jump* across the gap. Most recently, throughout the previous decade, in-memory database systems have emerged [BZN05, KN11, FCP+11, IGN+12, PAA+17]. Those argue that DRAM has

[3] Interestingly, this gap is visible in the hardware itself: While storage is attached via SATA or PCIe, memory is physically much closer to the CPU (dedicated memory controller for DRAM) or resides directly on the CPU (cache ad registers). Latency correlates with physical distance to the CPU and thus one might simply ask the question "How far from the CPU is the data" in order to determine its latency and to some extend bandwidth.

evolved to a point where most data sets can be kept completely in DRAM. Therefore, in-memory databases only log and periodically write snapshots to storage for durability while keeping all data in-memory. However, the current DRAM technology is reaching capacity limits on a physical level, has never been cost efficient, and real world data sets continue to grow. Thus, the research focus is starting to shift back to database architectures that incorporate storage devices more actively [Lom19, NF20, HHL20].

With the arrival of persistent memory, systems researches are given a completely new type of device that has the potential of closing or at least bridging the gap between memory and storage (cf. Figure 1.1). This first paper presented in this dissertation aims to examine persistent, measure its performance characteristics, and propose some optimized algorithms that can be used to construct larger systems (cf. Chapter 3). The second paper proposes a database design that incorporates persistent memory into a tradition, disk-based, database system architecture while still utilizing all layers of the storage hierarchy (cf. Chapter 2). In the following, we describe the high level characteristics of persistent memory (Section 1.1.2) and present the challenges (Section 1.1.3) and opportunities (Section 1.1.4) involving persistent memory.

1.1.2 Characteristics

Having introduced the concept of persistent memory and placed it within the existing memory hierarchy, this section describes the history and properties of persistent memory in more detail to set the scene for further discussion.

History of Persistent Memory

Building hardware that implements the idea of persistent memory has been an active area of research among electrical engineers and physicists for decades. On a physical level, persistent memory can, potentially, be based on a number of technologies. Some prominent candidates include: phase-change memory (PCM) [RBB+08, LIMB09, WRK+10], memristors [SSSW08], or spin-transfer torque magnetic RAM (SST-MRAM) [DS10, AKW+13]. While more details about all of these technologies can be found in the respective publications, we give a short primer on PCM as it is the technology that is currently commercially available. The idea behind PCM is to switch a chemical compound between an amorphous and crystalline state by heating it with electrical currents. Depending on the state (phase) of the compound, it has different electrical resistance properties and can thus be used to store information. As neither state deteriorates quickly over time, no energy is required to retain the information over long durations of time.

In 2015, 3D XPoint was announced by Intel and Micron Technology. 3D XPoint is a non-volatile memory technology based on PCM. In 2017, Intel started releasing solid state drives (SSDs) based on 3D XPoint: the Intel® Optane™ SSD. Unlike flash-based SSDs, the Optane SSDs are capable of sustaining a high write throughput [WAA19] for long durations of time. However, Optane SSDs are block-based storage devices attached via M.2 or PCIe and thus do not fall into the definition of persistent memory.

In 2019, Intel released the Intel® Optane™ DC Persistent Memory Modules [Intb, Intc] (Short: Optane DC PMM). Like, the Optane SSDs, the Optane DC PMM is based on the 3D XPoint technology and therefore on phase change memory (PCM). Optane DC PMMs fulfill the definition of true persistent memory: Low latencies and

TABLE 1.1: **PMem Server –** Configuration of the our persistent memory server at TUM.

CPU	**Intel Xeon Gold 6212U**
Frequency	2.40 GHz (3.90 GHz)
# Cores	24
L1 I+D Cache (per core)	64 kB
L2 Cache (per core)	1 MB
L3 Cache	35.8 MB
# AVX-512 Units	2
CPU Supported Memory	1 TB (DRAM + PMem)
DRAM	192 GB (6×32 GB)
PMem	768 GB (6×128 GB)

fine-granular access allow the CPU to interact with the memory directly via load and store instructions. In addition, the bandwidth and capacity is between that of DRAM and SSDs.

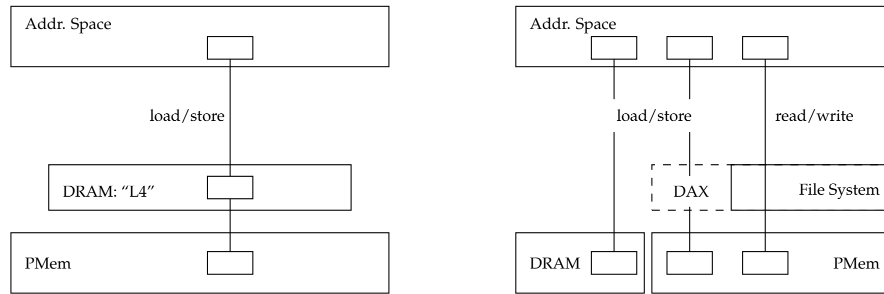
The first version of Optane DC PMM comes in three sizes per module: 128 GB, 256 GB, and 512 GB. Optane DC PMMs live on the memory bus and are managed by the integrated memory controller on the CPU, just like DRAM. Each memory channel needs to be equipped with one Optane DC PMM and one regular DRAM DIMM. Intel's Cascade Lake generation of CPUs supports six memory channels per socket. Hence a fully equipped two socket server machine can host a total of 6 TB of persistent memory. The persistent memory testing server (cf. Table 1.1) at Technical University of Munich (TUM) has one fully equipped CPU with six times 128 GB Optane DC PMM, amounting to 768 GB of persistent memory.

Persistent Memory Modes

The persistent memory hardware can be utilized in two distinct modes. In order to switch between modes the system needs to be restarted. A detailed description of how persistent memory can be configured can be found in Chapter 3. In the following, we focus on the characteristics of these two modes and how they can be used in system software (cf. Figure 1.2):

Memory Mode does not utilize the persistency of the Optane DC PMMs. Instead, it exposes the persistent memory to the operating systems as regular, volatile main memory. Therefore, existing software can transparently utilize the large capacity of persistent memory without any changes to the source code. In Memory Mode, the existing DRAM DIMMs on each memory channel operate as a directly mapped cache for the corresponding Optane DC PMM. The data that is written to the inherently non-volatile 3D XPoint medium (PCM) on the Optane DC PMM is, however, still persistent and could be an angle of attack. To circumvent this, the data on the Optane DC PMM is encrypted and the key is kept in volatile memory. Thus making the persistently stored data useless and as safe as data on DRAM DIMMs.

The Memory Mode offers an interesting opportunity to extend the system's main memory and safe operating costs by utilizing cheap Optane DC PMMs. However, it does not expose the hardware's persistency. Hence, in this mode the Optane DC PMMs do not qualify as persistent memory according to our



(A) Memory Mode: Persistent memory becomes the system's (volatile) main memory. DRAM acts as a transparent "L4" cache. From a software point of view the architecture is conceptionally indistinguishable.

(B) App-Direct Mode: Persistent memory can be access directly via a direct access (DAX) enabled file system (middle line) or via the operating systems `read/write` API (right line). In addition, DRAM can be accessed as usual.

FIGURE 1.2: **PMem Architecture:** Comparison of Memory and App-Direct Mode.

definition and are thus irrelevant for persistent memory research. While some performance benchmarks of the Optane DC PMMs in Memory Mode are provided (Chapter 3), the main focus of this dissertation is on the App Direct Mode. Further details on the Memory Mode and how applications are affected by the higher latencies of the Optane DC PMMs and the DRAM cache can be found in this survey [IYZ+19]. In addition, there is active research on how persistent memory can be utilized to supplement DRAM [QSR, LIMB09].

App Direct Mode exposes the Optane DC PMMs as a persistent device to the operating system (e.g., `/dev/pmem0` on Ubuntu). The device can then be partitioned and mounted with a file system like any other device (e.g., SSD or HDD). Similar to other storage devices, the file system layer can be used to allocate memory (i.e., create, delete, and truncate files) and manage access rights. The files on the persistent memory device can be accessed just like regular files or can be mapped directly into the virtual address space of a process. Once mapped, the program can access the persistent memory directly via load and store instructions. A process can map many persistent memory regions and regular volatile memory at the same time into its virtual memory address space. I.e., data can be written to persistent or regular memory depending on the application's needs. Details and requirements of this process are described in Chapter 3 or can be found in many other early persistent memory publication (e.g., [Ouk18, LHO+19, IYZ+19, vRVL+19, YKH+20, Ler21]).

Persistent Programming Model

Even so the CPU can access persistent memory directly via load and store instructions, the data that is being written to a persistent memory address does not immediately become persistent. This is because the data needs to pass through store buffers, caches, and the integrated memory controller before traveling over the memory bus to the persistent memory hardware. On Intel's Cascade Lake architecture, the *asynchronous DRAM refresh domain* (short: ADR domain) is used to refer to the area where data is guaranteed to be durable. It includes the Optane DC PMM and the write pending queue on the integrated memory controller on the CPU. Hence, once the data reaches the write pending queue, it falls withing the ADR domain and,

therefore, can be considered durable. The write pending queue itself does not use 3D XPoint hardware, but rather utilizes the energy stored in capacitors to flush the queue to Optane DC PMMs in case of a power failure [YKH+20].

Due to this architecture, a store instruction to a persistent memory address does not immediately become durable: First, the data needs to pass through the store buffer and the on-CPU caches (L1, L2, and L3) before it reaches the integrated memory controller and becomes durable. This can be achieved by flushing the cache line on which the data resides (`cl_flush`) and then waiting for the flush operation to complete (`sfence`):

```
void PersistentWrite(int* pmem_address, int value) {
        *pmem_address = data;
        cl_flush(pmem_address);
        sfence();
}
```

The first line of this function, simply writes the data (`value`) to the address on persistent memory (`pmem_address`). This store instruction moves the `value` to the CPU's store buffer from which it is then asynchronously written to the cache and then further evicted to persistent memory. While unlikely, at this point in time the process could be yielded or terminated (e.g., crash, shutdown, preemptive scheduling, or a power failure). Hence, the `value` must be considered as *potentially persistent*: the data could still be volatile or already durable; for the developer there is no way to know. Once the data is in this state (i.e., submitted to the store buffer) there is no way to prevent it from making its way to the ADR domain and become durable. However, we can force the data there using a *persistency barrier*: force the CPU to evict the cache line the data resides on (line 2) and using a fence instruction to wait for the eviction instruction to finish (line 3). Doing so forces the data into the ADR domain and, thus, makes it persistent. By using this little building block (store instruction, cache line eviction, and a memory fence) we can achieve *persistent writes*.

In summary, any data written to a persistent memory address can become durable at any point in time after the write, oblivious to the software's control. There is currently no method to prevent data from becoming durable. However, there are programmatic ways (persistency barrier) to force the data into the ADR domain and, thus, make it durable.

1.1.3 Challenges

Novel technologies, such as persistent memory, come with a unique set of characteristics. In this case: byte-addressable and durability with low latency and high bandwidth. These characteristics present many new challenges for system architectures, which are summarized in this section.

Additional Storage Hierarchy Layer

Even before the official release of Intel's Optane DC PMMs, it was widely believed that persistent memory will neither replace memory (DRAM) nor storage (SSDs or HDDs), but rather exist as a new layer in the existing storage hierarchy:

"...not fast enough to replace main memory and they are not cheap enough to replace disks, and they are not cheap enough to replace flash." – Mike Stonebraker 2017 [Sto]

As it turns out, Stonebraker was exactly right and the bandwidth as well as the latency of persistent memory is strictly worse than that of DRAM as the following table from Chapter 3 summarizes:

	peak read BW	required #threads	peak write BW	required #threads	read latency	persistent write lat.
DRAM	113.8 GB/s	15	92.5 GB/s	17	121 ns	`null`
PMem	39.1 GB/s	17	12.5 GB/s	3	403 ns	99 ns

The experiments were conducted on our evaluation machine (as described in Table 1.1) in the context of the second publication of this dissertation (cf. Chapter 3) and are similar in nature and result to those of other publications [LHO+19, YKH+20]. In the experiment, we utilize SIMD instructions and an optimal number of threads (depicted as "required #threads" in the table). The latency numbers are measured with a single thread and no other load on the system. Persistent writes are only reported for persistent memory, as these are not possible on DRAM[4]. Next to the performance gap between DRAM and persistent memory, it can be observed that the read-write asymmetry is much larger on persistent memory, which is owed to the underlying PCM hardware. The lower performance makes it impossible for persistent memory to outright replace DRAM for many performance critical applications.

Compared to SSDs, the price per gigabyte of persistent memory is between 50× to 200× worse depending on the capacity of the used memory module (128 GB, 256 GB, or 512 GB). Further, the persistent memory capacity of a two socket machine is limited to 6 TB (six 512 GB PMMs per CPU). In contrast, there are single SSDs with more than 6 TB of capacity and a two socket machine could host tens of these. For instance, Haas et al. [HHL20] have calculated that a machine can be equipped either with 400 GB of persistent memory or 7.3 GB of flash-based SSDs for a price of $2000. This shows that persistent memory is unlikely to become an economically feasible substitute for current storage hardware in the near future.

Therefore, persistent memory neither replaces traditional storage mediums nor main memory and must therefore be treated as new layer in the memory hierarchy. In addition, its properties differ greatly from DRAM and SSDs: For example, as we show in Chapter 3, unlike DRAM which loads 64 B blocks from the memory module, persistent memory loads blocks of 256 B at a time. Thus, in order to avoid read amplification, algorithms should be designed with this larger block size in mind. In addition, the bandwidth of persistent memory is lower, the latency higher, and it experiences a heavy read/write bandwidth asymmetry (reads are faster). However, persistent memory is a non-volatile memory device and can therefore be used as durable storage, unlike DRAM. Compared to SSDs, persistent memory is not a block-based devices but can be access in byte-granularity and with much lower latencies, similar to DRAM.

Next to this unique set of characteristics, an additional layer in the memory hierarchy provides a challenge on its own: Algorithms, data structures, and system architectures need to be adjusted to utilize this new layer in an efficient way without exceeding the complexity of the application. The difficulty of this problem is evidenced by the number of research papers that are proposing numerous different

[4] Using the same instructions on DRAM that are used for a persistent write on persistent memory (store, flush and fence), we would end up with the same latency: These instructions simply flush the data to the integrated memory controller on the CPU. If we would forego the flush and fence instruction, we would be measuring plain store instructions (`mov`), which have a few cycles of latency [Fog].

approaches (cf. Section 1.3) of how persistent memory should best be integrated into database systems.

Complex Programming Model

As detailed in the previous section (cf. Section 1.1.2), a simple write instruction to persistent memory must be considered *potentially persistent*: The data could still reside in a CPU cache (volatile) or it could be already written to persistent memory (durable)[5]. A *persistent write* (store, flush and fence) can be used to ensure the durability of data. However, evicting a cache line and stalling on it is an expensive operations which idles the CPU for hundreds of cycles (cf. Chapter 3). Thus, for efficiency reasons, not each individual write to persistent memory can be transformed into a persistent write. Systems need to bundle multiple writes together between persistency barriers (flush and fence). There is, however, no guarantee on the order in which these writes will become durable due to the volatile caches. Hence, all write operations must be carefully orchestrated to ensure that potentially persistent data is always in a recoverable state. We call this property *failure atomicity*. Building fast and correct algorithms that are failure atomic, is one of the major challenges when working with persistent memory [GKL20].

```
struct ShadowData {
        int size;
        int valid;
        char* pmem;
};

void Write(ShadowData* dest, char* src) {
        // Write the actual data
        memcpy(dest->pmem, src, dest->size);
        for(int off=0; off<dest->size; off+=64) {
                cl_flush(dest->pmem + off);
        }
        sfence();

        // Set the data valid
        dest->valid = 1;
        cl_flush(&dest);
        sfence();
}
```

The pseudo code above shows a typical pattern on which data structures for persistent memory are build (similar to copy-on-write or shadow paging [APD15]): First the actual data is written to a currently unused location and then a switch is flipped that indicates that the data is now valid.

As illustrated by the example, great care has to be taken to ensure that all data that could have potentially leaked into the persistency domain forms a valid state of the respective data structure from which the application can recover. This poses challenges to the developer and harbors a huge potential for rare and difficult to

[5]Technically, the data is already persistent once it reaches the integrated memory controller. For ease presentation, we use this phrase to mean the data is persistent.

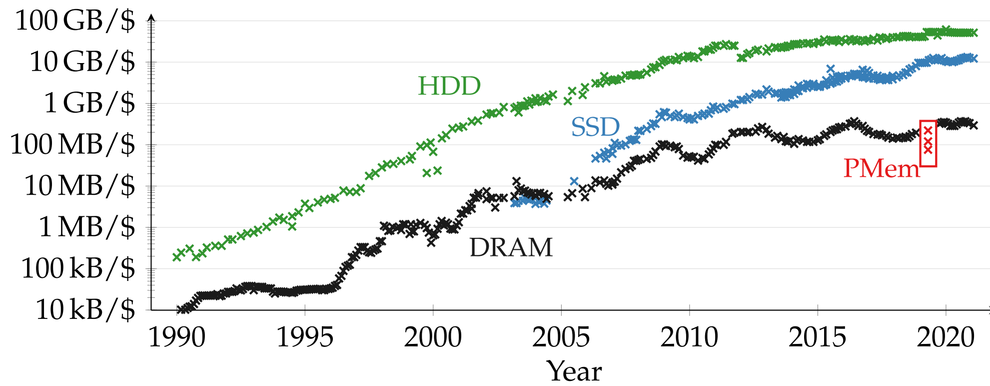


FIGURE 1.3: **Price-performance Ratio of Storage** – Storage and memory prices according to [McC] and PMem prices [Alc].

reproduce bugs. Unlike the very explicit read/write interface of traditional, block-based storage devices, on persistent memory a simple variable assignment is (potentially) a durable operation. Therefore, access to persistent memory has to be carefully scoped and rely on a set of well designed storage primitives that offer efficient operations on persistent memory.

Persistent Memory Cost

Even though persistent memory appears to be much faster than flash storage, it still needs to be economically viable: the benefits for applications have to outweigh its price. The primary focus of systems research is creating software for existing hardware, yet it is still important to consider the cost of that hardware as it determines the composition of the server hardware. In the following, we report the prices as of release and provide some observations and insights.

The Optane DC PMMs were initially available in three sizes: 128 GB, 256 GB, and 512 GB. After the initial release in April of 2019, Intel's pricing guidance for these three memory modules were reported at $577 , $2125 , and $6751 respectively [Alc]. In comparison, the largest commercially available DRAM DIMM had a capacity of 128 GB and was priced around $4500 at that time. Hence, getting large capacities of persistent memory is cheaper compared to DRAM. However, SSDs offer larger capacities (multiple terabytes per SSD) and lower prices.

Figure 1.3 compares the price-performance ratio (ignoring available capacities) of DRAM, SSDs, and HDDs throughout the previous decades. The three available Optance DC PMMs are highlighted in a red rectangle towards the right hand side. Only the smallest Optane DC PMM (uppermost red "x") has a comparable ratio to that of DRAM, because the price per gigabyte for smaller DRAM DIMMs is much better. However, as evidenced by the improving price-performance ratio of all other storage and memory technologies over time, it is likely that persistent memory will improve in that metric over time, as well. For example, around 2007 SSDs had already been on the market a couple of years but still only provided a price-performance ratio of 100 MB/$ which is comparable to that of the initial release of persistent memory hardware in 2019. Since then the price-performance ratio of SSDs has improved one hundred-fold to 10 GB/$ in 2019. While we can not predict the future, it seems likely that there will be some improvement for persistent memory as well. However, much of that is dependent on its initial adoption by the industry and the thus generated revenue that can be utilized to improve the technology.

Endurance

Some research papers [APD15, vRLK+18] that were published before the release of persistent memory analyze the wear (in terms of writes) of the particular data structure or algorithm on the hardware. This is because many of the underlying technologies for persistent memory, experience a low write endurance: around 10^{10} writes for PCM [RBB+08]. Due to this, researchers already suggested wear leveling algorithms for persistent memory [QKF+09, Liu17] before the official release in 2019. Once published, Intel's persistent memory modules were shipped with some kind of wear leveling. No details on the algorithm were disclosed by Intel, but they promise a lifetime of at least 5 years with 24/7 usage (350 PB data written) for a 256 GB module [Weba]. While the decision to include wear leveling was necessary to avoid customers from burning through their persistent memory modules in a manner of minutes to hours, it certainly increases the complexity of the hardware and possibly limits the performance of the hardware. Hence, from a research perspective, it would be interesting to experiment with software-based wear leveling (or algorithms that don't cause a lot of wear). A well engineered system would potentially not need hardware wear leveling and could benefit from a less complex and possibly faster persistent memory hardware.

1.1.4 Opportunities

There exists a large interest in persistent memory in research and industry (cf. Section 1.3). Further, Intel has developed the idea of persistent memory into a commercially available product. This effort, despite the previously stated challenges, suggests that there is a large interest and potential to this new technology. In the following, we will try to mitigate some of the challenges by outlining gradual integration strategies for persistent memory into existing systems to overcome the involved complexity. After that, we describe some of the scenarios in which persistent memory could yield large improvements.

Integration Strategies

While the complexity of persistent memory, both as a new storage layer and with its difficult programming model, pose a challenge, existing applications can adopt persistent memory in a number of ways:

- **Memory Mode:** Applications can benefit from persistent memory without any changes to the source code. In *Memory Mode* (cf. Section 1.1.2), the persistent memory modules of the server are exposed as the systems main memory and the regular DRAM serves as a large ("L4") cache on top of the new persistent-memory-module-backed main memory. In this mode, the non-volatility of persistent memory is not utilized, but nonetheless, the application can benefit from the larger capacity of persistent memory and the better price performance ratio. This, for instance, allows in-memory data warehouses [ALR+17] and other software with high memory demands [LIMB09, QSR, IYZ+19] to transparently scale to even larger sizes. However, with the caveat that the performance might be reduced if the working set exceeds the DRAM cache size.
- **No Persistency:** Even in the *App Direct Mode* persistent memory does not need to be used as a non-volatile device: just because the data is retained does not

mean it has to be reused. Thus, persistent memory can be used as an extension of the applications memory. This can, for instance, be useful for spilling intermediate results during query processing. In that case, only the code locations where the memory for spilling is allocated need to be altered, as writing to the persistent memory is no different than writing to regular DRAM. This approach, however, needs to be used with care if the written data contains sensitive information, as it is retained on the device even if it is not reused.

- **File System:** Persistent memory is exposed to the operating system as a regular device and can be mounted with any compatible file system. For optimal performance, it is then mapped (`mmap`) directly into the programs address space. However, working directly on persistent memory requires special care. Instead, the program can simply use persistent memory via the file system as a block-based device. While standard file systems, like Ext4, work on persistent memory devices, there are already ones that are optimized for persistent memory (e.g., BPFS [CNF+09], SCMFS [WQR13], PMFS [RKK+14], NOVA [XS16, XZM+17], Strata [KFH+17], or Kuco [CLZ+21]). In this scenario, persistent memory is essentially used as a fast SSD. Further, there it requires no changes to the source code (only the path needs to be pointed to a persistent memory device). However, it does not make use of the byte addressability of persistent memory [IYZ+19].

The listed integration strategies each only utilize a subset of the advantages of persistent memory. However, depending on the use case, this (e.g., spilling intermediate results during query processing) might be enough.

Future Opportunities

Many of the areas where persistent memory could have a large impact (e.g., logging, page flushing and caching, spilling, transactions, and persistent data structures) have, naturally, already been studied and are not re-iterated here. References to these areas of works can be found in Section 1.3. Instead, we want to highlight some less commonly focused areas for which persistent memory could be promising. Some of these scenarios might currently be out of reach due to economical reasons. Other could help persistent memory to overcome its current cost-drawback and fund further development.

- **Larger Main Memory:** The growth of DRAM capacity has slowed in the previous decade and it is widely believed that it is getting close to its physical capacity limits [dra07]. Therefore, in the long run, a new technology is needed to continue growing the size of big data systems. The prominent candidates are persistent memory (PCM) and flash storage. While flash storage is more complicated to access due the block-based interface it is already cheap, in contrast to persistent memory, and very affordable [LHKN18, NF20].

- **Random Access Stores Engines:** Graph and object oriented database systems are, compared to relational ones, less popular. Broadly speaking, both data models (graph and object) organize their data less sequential than the relation schema and are therefore more prone to random accesses [JBGS21, BGJS21]. Persistent memory, unlike traditional storage mediums, like SSDs and HDDs, offers fast random accesses and byte-granularity which might be a good fit for these kind of access patterns.

- **Whole System Persistency:** With the introduction of persistent memory, the gap between memory and storage has narrowed. The persistency domain is a term used to describe the locations in which data is persistent. The border of this domain (assuming Intel Optane DC PMMs) is the integrated memory controller (iMC) on the CPU: Each byte that reaches the iMC can be considered persistent from a software point of view. However, the buffer on the iMC is actually comprised of faster volatile memory. This can be observed in the measured read/write latency of persistent memory [LC19, IYZ+19, vRVL+19, vRVL+20]: When there is no memory pressure, a persistent write takes around 100 ns, while a read take 400 ns. However, the write latency of phase change memory, the underlying technology behind Intel's persistent memory, is higher than its read latency. This discrepancy is due to the fact that a write operation only has to reach the volatile memory in the iMC, while a read has to fetch the data from PCM. To still guarantee the durability of every byte that reaches the iMC, the size-capped buffers on the iMC are flushed on a power failure using the energy from capacitors.

  While this is a great performance improvement, CPU caches and registers are still volatile and pose great programming challenges for the development of persistent memory systems as outlined in Section 1.1.3. An almost logical next step is to further extend the persistency domain to include everything on the CPU. This idea was already published in 2012 under the name of while-system persistence [NH12]. Having access to large and cheap persistent memory, an interesting step could be to start designing programming models and systems for hardware that is completely persistent. Such a system would eliminate volatile memory and therefore greatly simplify the von Neumann architecture, improve performance (no more I/O), and simplify the programming model. Note that the last aspect, would be a game changer for systems hardware considering the complexity involved in managing the transition from memory to storage in, for example, database systems.

- **Mobile Computing:** A large issue with persistent memory is its price: As previously shown, it is currently not the most cost efficient option compared to the faster DRAM and cheaper SSDs. In order to succeed there needs to be an application that leads to initial revenue and, therefore, further development. One such area could be mobile computing: the ability to instantly switch a device on and off can increase the battery power. In addition, as mentioned in the previous point, the reduced complexity could be beneficial for the device's complexity, cost, and size. However Intel's Optane PPM is currently only available to server CPUs and motherboards.

- **Instant Cloud:** Software as a Service (SaaS) and, especially, lambda functions allow for great elasticity for customers. A service can be spun up for short duration of time and only be used as long as actually required. Using persistent memory, much of the state of the customer's software could be kept alive and therefore allow for much faster start up times. In fact, early persistent memory storage engines like SOFORT [OBL+14] are designed to provide almost instantaneous start up times. This could be developed into a "query as a service"-architecture, where shot tasks can be run on a large, instantly available state.

1.2 Research Methodology

At the start of this dissertation (2016), we sat out to explore how persistent memory can be integrated into database management systems. However, due to persistent memory becoming commercially available in 2019, the research question changed and we explored what the characteristics of persistent memory are.

In the first publication [vRLK+18] (cf. Chapter 2) of this dissertation, we proposed a novel idea that integrates persistent memory into the buffer manager while utilizing its advantageous properties (e.g., byte-addressable). However, the system was designed and developed on a simulation platform that aims to emulate the behavior of persistent memory and not on actual hardware. Using simulation was a common approach, because real persistent memory hardware was not yet available in 2018. During this work, we realized that simulation-based research needs to make many assumptions on the characteristics of the hardware. Therefore, a detailed study of the persistent memory hardware would be necessary to aid future researchers and systems designers, once the actual hardware became available. In a cooperation with Intel, we were able to utilize an early prototype to design a set of benchmarks to measure persistent memory performance and its special characteristics. Based on those results, we developed a number of algorithms for common tasks on persistent memory (e.g., efficient logging). Once persistent memory became publicly available in 2019, we published these results in a short paper [vRVL+19] based on the prototype hardware. Later, this short paper was invited to the "Best of DaMoN'19" special issue of the VLDB Journal. This extended version [vRVL+20] includes several additional algorithms and was evaluated on actual persistent memory hardware (no prototype). In the following, we will first describe our experimental methods used through the whole dissertation and then detail the evaluation systems.

1.2.1 Experimental Setup and Methods

All proposed algorithms throughout the two publications this dissertation is comprised of were implemented from scratch. Similar to most systems and database management software, the low-level programming languages C and, mostly, C++ were used. This allowed to embed assembly instructions directly into the source code, which was required for some of the new instructions added specifically for persistent memory (e.g., the cache line write back "`clwb`" instruction). In addition, these languages gave us direct control of the applications memory management and provided easy access to Intel's intrinsics library [Inta], which was used for SIMD-based algorithms. Lastly, both languages are considered to have very little overhead which is essential for the development of highly efficient low-level algorithms.

Common strategies for ensuring code quality, such as unit and integration tests as well as code reviews were used to ensure the correctness of the implemented algorithms. Particularly useful was the method of A/B testing where a complex implementation of an algorithms is validated by comparing its results with those of a simple implementation over many randomly generated inputs. For tracking down errors in our code, we used the `gdb` debugger [gdb] and tracing tools such as `valgrind` [val] and the well known method of `printf()`-debugging. To test for failure atomicity on persistent memory (i.e., can a data structure resident on persistent memory be recovered in case of a crash), we carefully inserted intentional crash points into our source code at critical location to test the recovery policy. Similar techniques for testing crash consistency have been proposed since then [OBLL16,

DLCL21] and are, for example, easily available in the form of the `pmreorder` tool within the persistent memory development kit [PMD].

Obtaining statistically significant results can be difficult in many branches of science where population numbers are often small. However, in computer science experiments are mostly evaluated on automated machines and often run for very short times. In our case, the experiments were conducted on a single machine and usually only took microseconds or milliseconds to run. Therefore, it was feasible to run a single experiment thousands or even millions of times. This allowed us to obtain stable results and, as common in computer science, no explicit tests for statistical significance were conducted due to the extremely high population counts. If not otherwise mentioned, the number of iterations for an experiment was adjusted so that it ran for roughly one minute. When reporting the time of a single experiment run, the average time of these iterations were used. We did not report any latency distributions or tail latencies for most experiments, because the time of a single experiment was, as mentioned, very short (microseconds). For instance, many of our experiments essentially measure the latency of a single store instruction (roughly one hundred nanoseconds). Any high level, application facing algorithm were tail latencies are important is comprised of several of those building blocks and thus averages them as well. In addition, there is no easy way to keep track of the latencies of single store instructions without significantly impacting the experiment (an additional store instruction would be required which impacts the timing, memory pressure and the caches).

1.2.2 Evaluation Platforms

Due to persistent memory hardware becoming available during the writing of this dissertation, a number of different platforms were used for our evaluations. Due to a lack of commercially available persistent memory hardware, we had to rely on simulators in the beginning. In particular, the SCM[6] emulation platform (SEP, described in the following paragraphs). Later we were able to use Intel's Early Prototype (AEP) and then, once persistent memory became available, the actual Intel Optane PMMs. In the following we describe the various platforms.

Simulators: In 2015, at the start of this dissertation, persistent memory was not yet commercially available. However, the large interest in the technology had already lead to a number of simulation techniques: A common technique (e.g., used by Mnemosyne [VTS11], FOEDUS [Kim15], and WORT [LLS+17]) was to manually insert delays whenever persistent memory was accessed. While this allowed to model read/write asymmetry and gave fine control over the exact latency, it is not possible to control bandwidth and the impact on CPU internals such as pipelining and prefetching. In contrast, x86-64 simulators, such as PTLsim [You07], can in principle model the entire CPU accurately and, with the use of extensions, persistent memory. However, many details of modern CPUs are not public and the execution is very slow making larger experimental evaluations difficult. Another approach suggests to utilize the different NUMA nodes of a multi-socket machine to emulate persistent memory latencies and bandwidth. While this does not introduce artificial delays on the CPU that might impact the execution, the latency and bandwidth can only be configured on a few discrete predefined levels (depending on the number of sockets and their distance) and there is no read/write asymmetry. Lastly, the Quartz [VMCL15] system tracks read accesses to memory over a duration of time

[6] SCM := storage class memory, a different name for persistent memory.

(epoch) and then delays the program at the end of the epoch depending on the number of accesses. In addition, it allows to configure the bandwidth on a few discrete levels by changing the DRAM thermal control settings [HR10]. However, writes were not delayed and the read/write asymmetry was not implemented in the public version. Next to these software-based simulation techniques, Intel had released the SCM Emulation Platform (SEP) [Dul16], which provided a hardware-based simulation of persistent memory. The system added additional cycles when accessing persistent memory to emulate a higher latency and, similar to Quartz, used DRAM thermal control to configure the bandwidth. This allowed for an accurate simulation (except for the read/write bandwidth asymmetry) and was therefore widely used (e.g., [OBL+14, APD15, APP16, OLN+16]). However, it required special hardware that was provided by Intel to selected industry and research partners and, thus, not publicly available. The authors of this dissertation were graciously granted access to one of those machines by Fujitsu Laboratories, to whom we are very grateful to. We used the SEP machine in our first paper of this dissertation (Chapter 2).

However, independent of the simulator, in the absence of actual hardware all characteristics of persistent memory, despite the multitude of simulators and many configuration options, had to be estimated. To accommodate for this uncertainty, we, as many others, varied certain key characteristics (such as latency) in our evaluation (cf. Chapter 2). However, many details (such as the 256 byte blocks) were unknown at that time and could not be accounted for.

Actual Hardware: Once Intel's Optane DC Persistent Memory Modules became commercially available in April of 2019 our university was able to obtain an evaluation platform (cf. Table 1.1) and all future research was conducted on it. The actual hardware gave us direct insight into the characteristics of Intel's instantiation of persistent memory and allowed us to work out all unknowns that were missing on the simulation platforms (Chapter 3). We decided to obtain a two socket machine equipped with only a single CPU in order to reduce cost. While the study of NUMA effects in conjunction with persistent memory is promising, the topic of persistent memory is grand enough and the machine can always be upgraded in the future.

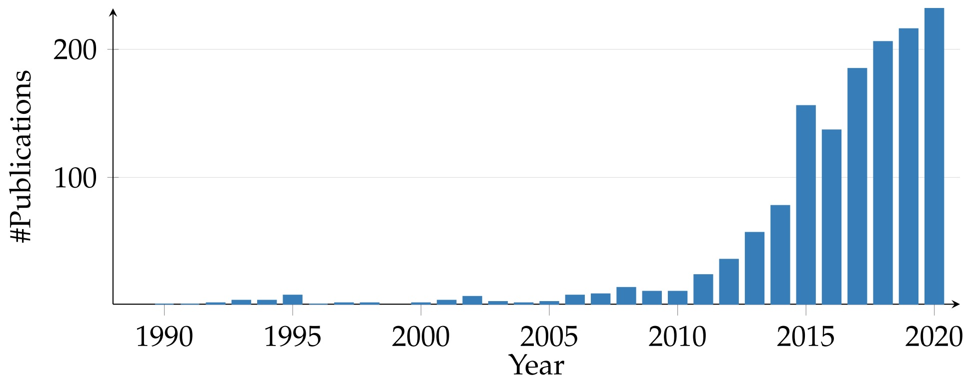


FIGURE 1.4: **PMem Publications** – Number of publications relating to persistent memory on dblp [TD].

1.3 Related Work

Persistent memory, as a novel storage medium, has already impacted many areas in database systems in particular and systems software in general. Even so many of the techniques proposed in this dissertation are generally applicable, their application is focused on database systems. Therefore, the related work section also focuses on the use of persistent memory in database systems.

Persistent memory has been the subject of much attention from research as well as industry throughout the previous decade. With Intel's release of Optane DC Persistent Memory Modules in 2019, the number of yearly publications has climbed well into the hundreds (cf. Figure 1.4). Therefore, this related work focuses on range index structures and database system architectures for persistent memory. However, many references regarding the history, hardware, transactions, and logging can be found in earlier sections of this dissertation. In addition, we refer to the related work sections of the publications of this dissertation (cf. Chapter 2 and Chapter 3).

1.3.1 Persistent Memory Development Kit (PMDK)

There are a multitude of logging algorithms and transaction libraries for persistent memory. Most prominent is the persistent memory development kit (PMDK) [PMD], a widely supported framework for the development of persistent memory applications. The PMDK provide several abstraction layers, starting with simple wrapper functions around flush and fence operations (`libpmem`). Building on this, it offers abstractions for logging (`libpmemlog`) and atomic block-based access to persistent memory (`libpmemblk`). While the PMDK algorithms are often more generally applicable, they suffer in performance compared to specialized solutions [FHH+11, CCA+, IKK16, KTB+19]. Using the logging functionality, the PMDK implements a transaction library (`libpmemobj`). While this library provides a good baseline, many optimizations have been suggested for more specialized transactions [LC97, VTS11, GDV13, CCV15, KPS+16] including full integrations into programming languages [KTB+19]. Götze et al. [GTS20] evaluate these basic building blocks. Besides, there are various algorithms to semi-automatically transform regular data structures and algorithms for persistent memory [BC16, DDGZ18, LMK+19, FBW+20].

1.3.2 B-Tree-like Index Structures

B+-Trees [BM70] are the de-facto standard [Com79] range search data structure in the context of database systems due to their high fan-out, page granularity, and balanced structure. Consequently many research papers on persistent memory indexes have focused on tree-like data structures that are often similar to B-Trees. In 2018, Götze et al. [GvRL+18] have summarized these by categorizing them by their interaction pattern with persistent memory. In the following, we extend this list into the present and order it chronologically. The list is focused on tree-like index structures, as those are more widely applicable due to their support for range queries. However, there has also been work on hashing-based structures [BHC+13, DHK+15, SDUP15, NIK+17, ZH18, ZHW18, NCC+19, LHWL20] and log structured merge trees [LOSL17, LOLS17, KBG+18, KLN+19, LCK+20, YCJ+21].

Table 1.2 gives an overview of all index structures with a brief description, which types of memory and storage it uses, and on which platform it was originally deployed and tested.

PCM-Tree (2011) Chen et al. investigate efficient algorithms and data structures for persistent memory [CGN11]. As one of the key optimization goals, they identify the reduction of writes to persistent memory. Due to the read-write skew and the lower latency of persistent memory, write operations are especially expensive and should therefore be minimized. Based on this analysis, they propose to use non-dense and unsorted BTree nodes on the leaf layer as this is updated most frequently. This avoids many writes in case of inserts or deletions, as the key/value arrays does not have to be compacted or expanded. Due to the lack of a name, we simply refer to their idea as the *PCM-Tree*.

CDDS-Tree (2011) One of the first index structures proposed for persistent memory is the CDDSTree [VTRC11]. In the paper, Venkataraman et al. define the term *Consistent and Durable Data Structures* (CDDS), which is a programming model for data structures for persistent memory that ensures consistency and durability. This is achieved by using versioning: every update or insert creates a new version of an object (e.g, a tuple in a BTree). Once the object is created, fully persisted and linked to the data structure, a global version number is atomically updated. Thus the data structure is moved from one consistent state to another. Based on the CDDS model, they design a B-Tree (coined the "CDDSTree"), that implements this model.

NV-Tree (2015) The NV-Tree [YWC+15] is a B+-Trees that is optimize to reduce the number of writes to persistent memory. Based on the PCM-opt BTree, the NV-Tree also employs non-dense and unordered nodes (slightly different implementation, however). In addition, they introduce the idea of *selective consistency*: The data in inner nodes of a B+-Tree is redundant and can be reconstructed from the data in the leaf nodes. Therefore, it is not necessary to ensure its consistency and durability during operation. Instead, it can simply be reconstructed after a crash during the recovery phase. The selective data consistency idea makes it possible to implement all operations, including splits and merges, of leaf nodes using atomic operations without the need for redo-logging. However, due to the recovery of the inner nodes, the NV-Tree requires a recovery phase.

wB+Tree (2015) The wB+Tree [CJ15] is a follow up work on the PCM-opt BTree. In order to reduce the number of writes to persistent memory, it also uses non-dense and unordered leaf nodes. To speed up key search operations within a node, a small indirection array is used which contains slot indexes to the keys in sorted order. In contrast to the NV-Tree, they ensure consistency for all nodes (inner and leaf) and

TABLE 1.2: **PMem Range Indexes:** A cronological list of range indexes for persistent memory. The evaluation column shows on which platform the system has been evaluated in the original publication: PTLsim [You07], Quartz [VMCL15], SCM evaluation platform (SEP) [Dul16], MultCallFlushLRU [WC08], or NVDIMM [JED]

Year	Name	Idea	DRAM	Evaluation
2011	PCM-Tree	non-dense +unordered leafs		PTLsim
2011	CDDS-Tree	BTree with versioning		MultCallFlushLRU
2015	NV-Tree	selective consistency +write optimized leafs		NVDIMM
2015	wB+Tree	non-dense, indirectly ordered nodes, redo-logging		DRAM or PTLsim
2016	FPTree	fingerprint selective persistency	X	SEP
2017	WORT	three radix trees atomic operations		Quartz
2017	HiKv	hybrid BTree + hash table on DRAM and PMem	X	DRAM + manual delays
2018	FAST+FAIR	dense sorted arrays special shift functions		Quartz
2018	Bz-Tree	Bw-Tree [LLS13] persistent MwCAS		flash-backed NVDIMMs
2019	DPTree	indexed (DRAM) log PMEM with delta tree (DRAM)	X	Optane
2019	RECIPE	generic transformation rules for non-blocking atomic indexes	X	Optane
2020	HART	top: Hybrid ART (DRAM) bottom: Bz-Tree (PMem)	X	Optane
2020	RStore	Log-structured Index	X	Optane
2020	LB+-Trees	B+-Tree optimized for 256 B	X	Optane
2021	WOBTree	sorted tree nodes by replacing array with in-place list		Quartz
2021	UPSkipList	atomic skip list for PMem		Optane

thus retain a constant start up time. Node split and merge operations utilize a small size-capped redo-log. All other operations are performed atomically.

FPTree (2016) The Fingerprint Tree (FPTree) [OLN+16] is a BTree-like index structure with several unique features: (1) Selective Persistency: The authors take the idea of selective consistency, introduced in the NV-Tree, even further and place inner nodes on DRAM instead of persistent memory. Inner nodes in a BTree are secondary date, i.e., they can be reconstructed using the primary data in the leaf layer. In addition to having the benefit that crash consistency can be avoided, the inner nodes also benefit from the lower access latencies of DRAM. In case of a crash, they can simply be reconstructed. (2) Fingerprinting: Due to the higher access latency of persistent memory, the FPTree leaf nodes are designed in a way to reduce both reads and writes. This is achieved by filling the first cache line of a node with one byte finger prints of each resident key. In case of a lookup, this array is probed and only potential matches need to be retrieved. This lowers the number of required cache lines to be loaded to two on average for a positive lookup.

WORT (2017) Lee et al. [LLS+17] investigate how radix trees can be adopted for persistent memory and present three approaches: (1) The Write Optimal Radix Tree (WORT) is a simple radix tree adopted for persistent memory. Compared to a BTree, radix trees do not require sorted nodes and thus are well suited for persistent memory. In fact, a new subtree can simply be added with a single 8 B atomic write operation. However, radix trees often experience a high space consumption due to underutilized nodes. (2) To overcome this, they make use of the adaptive radix tree [LKN13] and propose the Write Optimal Adaptive Radix Tree (WOART). Using several different and more intricate node types the under utilization can be efficiently reduced. However, the additional complexity requires several writes per node in order to add a new subtree, which in turn requires several synchronization barriers to ensure failure atomicity on persistent memory. (3) To overcome this, they propose a version that uses copy-on-write (ART+CoW): Instead of using a complex algorithm to update a node in place, the copy and replace the node in its parent. In there evaluation, the three variants show similar performance characteristics.

HiKv (2017) The hybrid index key value store (HiKv) [XJXS17] utilizes both DRAM and PMem. The authors build a hash table on persistent memory to support point queries and to ensure persistency. In order to support scan operations, they also maintain a BTree in DRAM which completely mirrors the data. Placing the BTree in DRAM avoids the many costly writes to persistent memory required by sorted BTree nodes as well as the complexity of other proposed node designs. New tuples are synchronously inserted into the hash table (thus ensuring durability) and placed into a update queue for the BTree in order to reduce the latency. The update queue is processed by a background worker thread. Whenever an operation requires the BTree (i.e., a scan), HiKv blocks all updates to the hash table until the update queue is empty. With an empty update queue, a consistent state of the BTree is reached and the scan can start executing on the BTree. During the scan the hash table can resume updates, but the update queue must not be applied in the BTree in order to keep the consistent state. While the design uses both DRAM and persistent memory in an efficient way it does require roughly twice the amount of memory and the queuing is questionable when faced with many scans.

FAST+FAIR (2018) Previous work has moved away from ordered B+-Tree nodes by using additional indirections in the meta data, due to the high write amplification. In contrast, Hwang et al.[HKWN18] propose the concept of endurable transient inconsistent states (ETIS), which is an intermediate state of an update that can be tolerated and repaired by other operations. Their failure-atomic shift (FAST) algorithm

allows inserting into an ordered dense array. The key idea is that dependent CPU instructions are not reordered and node pointers in B+-Trees are unique: Each element is shifted by one position individually, which is a dependent operation and therefore not reordered. In addition, child pointers are carefully duplicated to make the inconsistent state detectable and therefore endurable. Based on the FAST and their failure-atomic in-place rebalancing (FAIR), they augment an in-memory B+-Tree (FAST+FAIR) to be crash consistent on persistent memory.

Bz-Tree (2018) Constructing failure atomic algorithms for complex data structures such as B+-Tree is difficult and incurs performance overheads, as evidenced by HiKv moving the BTree to DRAM or FPTree implementing selective persistency. The Bz-Tree [ALML18] tackles this challenge by using an indirection that is capable of updating several bytes in a failure atomic and thread safe way. This helper function is called persistent multi-word compare and swap (`PMwCAS`) [WLL18] and based on the (non-persistent) multi-world compare and swap (`MwCAS`) [HFP02]. They integrate `PMwCAS` into the Bw-Tree [LLS13], a lock free B+-Tree. They thus drastically reduce the complexity while maintaining a instantly recoverable high performance tree structure.

DPTree (2019) Because of the high write latency to of persistent memory, the differential persistent tree (DPTree) [ZSC+19] is optimize to reduce writes by reducing the structural maintenance overhead (i.e., updates to meta information). This is achieved by batching modifications: All updates are initially inserted into a small DRAM-resident B+-Tree and, for durability, into a log on persistent memory. The small buffer is periodically merged into a larger tree that contains all key value pairs. This tree employs selective persistency: a linked list on persistent memory that is index with a DRAM-resident radix tree. The linked list nodes contain multiple key value pairs. The meta information of these nodes (e.g., entry count, order, fingerprints) is versioned and lazily reconstructed, thus allowing the merge procedure to skip additional work to keep them crash consistent.

RECIPE (2019) Lee et al. [LMK+19] propose a generic set of algorithm to transform non-blocking concurrent data structures for DRAM to a failure atomic persistent memory version. One of the key insights is that these data structures already allow endurable transient inconsistent states (as proposed by FAST+FAIR) when, for instance, a reader can see intermediate updates made by a writer. Lee et al. provide a list of conditions and transformation rules which they demonstrate by applying it to B+-trees, tries, radix trees, and hash tables with limited modifications to the source code.

HART (2020) Based on the adaptive radix tree (ART) [LKN13], Zhang et al. propose the Hybrid ART (HART) [ZLJW20]. Compared to WOART and ART+CoW, the HART is a hybrid index that utilizes both DRAM and persistent memory in order to make use of the advantages performance characteristics of DRAM. The upper part of the index structure is an ART-like structure that captures a certain number of bits from the keys and kept in DRAM. The lower part is organized as a BTree (wB+Tree-like) and stored on persistent memory. It stores the entire key and is used to recover the volatile ART after a restart. The boundary between the two structures can adjusted depending on memory budgets and performance requirements.

RStore (2020) Lersch et al. [LSOL20] propose a key value store, called RStore, that is optimized for predictable low latencies. This is in contrast to many other key value stores which are often optimized for a high throughput but utilizing techniques such as group commits and other batching techniques. However, low latencies are often a requirement and persistent memory is well suited for it due to its non spinning nature (compared to disk) and lack of garbage collection (compared to flash). RStore

achieves this by using selective persistency in form of a log-structured index: the data is written to a log file on persistent memory, while an index on top of the log file is constructed in DRAM. To remain competitive with throughout and scalability they implement user space networking and a shared-nothing partitioning of the data with message passing between the worker threads.

LB+-Trees (2020) Early performance studies [LC19, IYZ+19, vRVL+19, vRVL+20] of persistent memory not only showed that write operations should be avoided but also that the write granularity of persistent memory is 256 B. Similar to in memory structures that are optimized for the DRAM cache line size, they propose the LB+-Tree [LCW20], which is optimized for the 256 B persistent memory block size. The LB+-Tree is a B+-Tree with a node size of a multiple of 256 B. In addition, they deploy several techniques to reduce the block writes, most prominent: they insert entries into the first 256 B block of a node, which requires only a single block write as the header of the node is also located in the first block. Once this block is full, they move multiple entries to other blocks in bulk in order to retain the single block write per insert property.

WOBTree (2021) WOBTree [WLZ+21] is a write optimized B+-Tree for persistent memory. They propose to optimize a B-Tree for PMem by reducing the write amplification based on cache line granularity (64 B). Note that the granularity can be adjusted in their design and could therefore also be tuned to the 256 B persistent memory blocks. This is achieved by logically splitting a B+-Tree node into several subnodes that are aligned to cache lines boundaries. These are still stored in the respective node (locality), but act as a ordered doubly linked list. Compared to a sorted array-based node layout, the number of cache lines that have to be modified is greatly reduced in this list-based structure. Modifications are implemented using failure atomic updates. To speed up search operations on the list, bloom filters and a small single layer index on top of the list is used.

UPSkipList (2021) While previous work has often tried to overcome the block-based pages with sorted keys in B+-Trees, Chowdhury et al. [CG21] optimize a the fine-granular skip list data structure for persistent memory. They utilize an existing lock-free skip list [HS08] and transform it into a failure atomic skip list for persistent memory using the RECIPE. The RECIPE is extended to work on non-blocking atomic data structures.

In science, the ability to repeat an experiment is crucial: It allows researchers to compare results of other groups to their own. One big issue with many of the persistent memory data structures is that they are not easy to compare: In the original publications, they are tested on specific hardware (simulation, emulation, early prototypes, and actual hardware), they are largely closed source, and they use different benchmarks. Therefore, it is difficult to compare the performance and thus empirically determine the optimal index structure for a given problem. Recently, a benchmark for persistent range indexes [LHO+19, HLWO20] and an evaluation of various hash-based approaches [HCW+21] has tried mitigating this problem.

1.3.3 Database Architectures

A big question concerning persistent memory is how it can be integrated into a database management system [DAP+, Ouk19]. As a first possible option, a database (or key-value store) can simply be run on a persistent memory backed file system [IYZ+19]. This approach requires little adjustments to the implementation, but has a limited performance benefit as persistent memory is not fully utilized. As a

next step, the storage sub-system (e.g., logging and buffer management) can be optimized for persistent memory. However, considering that persistent memory is a radical new storage device with unique properties, this can still not fully utilize it. The most disruptive approach is to optimize the database architecture for persistent memory. In the following, we list several impactful approaches.

Arulraj et al. [APD15] have compared three database architectures for persistent memory. They propose to use persistent memory as the primary location for table data as well as index structures. In their work they compare engines based on in-place update, log-structured storage, and copy-on-write algorithms. Each approach is optimized for persistent memory. Their experimental results suggest that in-place update-based engines are a promising candidate due to good performance and low write amplification.

SOFORT [OBL+14] is a storage engine for systems with DRAM and persistent memory. They, as the name suggests ("sofort" is German for "immediately"), optimize for a near instantaneous restart time. Similar to SAP HANA [FCP+11, FML+12], they use a columnar storage format and a delta-merge approach. Using multi-version concurrency control, they organize their tables as append only structures on persistent memory. This simplifies the code (only appends) and tuples of in-flight transactions do not have to be deleted after a crash, as the version number automatically marks them as invalid. Dictionary encoded columns keep their dictionary in DRAM to speed up lookups and are recoved in parallel to speed up restart. For a database with 10 M rows, they achieve recovery times below two seconds.

While SOFORT statically assigns tables to persistent memory and dictionary indexes to DRAM, FOEDUS [Kim15] implements a more dynamic approach to make better use of the fast DRAM: The database is organized as one large B+-Tree which is based on Masstree [MKM12] and Foster B-Trees [GKK12] and called master tree. The pages of the master tree are fixed-size and initially reside on persistent memory. Whenever a page is accessed, it is copied to DRAM using a buffer pool and read or written their. To ensure durability, they use redo logging. In contrast to traditional buffer pools, a dirty page is never directly written back to persistent memory. Instead, the WAL is asynchronously and periodically merged into the master tree nodes on persistent memory. Once merged, the dirty pages can be dropped from the cache. They use pointer swizzling [GVK+14] to speed up tree traversal. Each page reference contains a pointer to the copy of the page on persistent memory and on to the one on DRAM (`null` if nonexistent).

Traditional database systems deploy fixed-size pages and write-ahead logging, because SSDs and HDDs only have page-granular read/write capabilities. Given that this assumption no longer holds true for persistent memory based storage, Arulraj et al. [APP16] propose a new recovery method: write-behind logging, which has been integrated into Peloton [PAA+17]. Transactions make changes to a DRAM-resident copy of the database. At group commit time, all changes are flushed to persistent memory. This can be done without a single memory fence operations, as the ordering of these writes is not important. In order to detect uncommitted tuples after a crash, they use a multi-versioned storage layout (which is also used to implement multi-version concurrency control). Once all changes are flushed to persistent memory, a group commit record is written that logs the new version ids and thus marks the modifications as valid. To handle long running transactions, they allow gaps in the ranges of valid version ids. This approach greatly reducing the write amplification of the system, as each tuple is only written once compared to WAL-based systems. However, the current design requires two copies of the database (DRAM and persistent memory) and lacks a WAL, which is often used for replicas.

SAP HANA [FCP+11, FML+12], as a large commercial in-memory database for OLAP and OLTP, has been adopted for persistent memory [ALR+17]. They use a three tier architecture where the primary location of durable data is on disk. Persistent memory serves as an extension of DRAM and can be immediately reused after a crash (essentially a persistent cache over SSD/HDD). HANA keeps the write-optimized delta store on DRAM and places the large read-only portion of the tables in persistent memory. This avoids the difficulties of in-place updates on persistent memory, as it is only updated by the delta merge process. In addition, keeping the delta store on DRAM retains the high performance of updates due to DRAM's lower latencies. Contrary to SOFORT, they place the string dictionaries on persistent memory to enable faster reload times. Queries read data directly from persistent memory. They report only minor slow downs when running on persistent memory and can benefit greatly from the faster restart time.

Much work on persistent memory was focused on transaction processing, the high bandwidth and large capacity can also be beneficial for analytical processing as shown by SAP HANA. However, the characteristics of persistent memory are not quiet like DRAM. Therefore, Götze et al. [GBS18] prose an analytical system for persistent memory. The key idea is to adopt BDCC+ [BBS16] for persistent memory. BDCC+ is a multidimensional clustering approach that maps multiple attributes to a single artificial clustering key by which the table is ordered. In their work, they design a novel storage format optimized for persistent memory and demonstrate significant performance gains.

There have been multiple systems that aim to adopt persistent memory by integrating it via the buffer manager [LLO19, Kim15, vRLK+18, APM19]. A recent incarnation of this approach is Spitfire [ZAPC21]: a three tier buffer manager for DRAM, persistent memory, and SSD. They improve upon the HyMem [vRLK+18] storage engine, which is one of the publications of this dissertation (cf. Chapter 2). First, they implement a multi-threaded buffer manager, which makes the system more realistic. In addition, it increases the memory pressure which can significantly influence the replacement strategy. Using only a single thread in HyMem, we had essentially unlimited DRAM and persistent memory bandwidth. Second, they conduct their research on actual persistent memory hardware. Having been published before the release of Intel Optane DC Persistent Memory, HyMem was evaluated on a simulation platform. Third, they process transactions directly on persistent memory resident pages, while HyMem always needs to copy the page to DRAM. Using multi-threading, actual hardware, and processing in persistent memory, Spitfire constitutes an advanced storage engine for the implementation and evaluation of three tier buffer placement strategies and makes a number of significant advancements: (1) Having the ability to process transactions directly on persistent memory increases the number of decisions the data placement algorithm has to take. For example: When a page is loaded from SSD, should it be placed in PMem or in DRAM? Spitfire analyzes all possible data paths and breaks them down into four decisions. They proposes a probabilistic approach where each data migration is controlled by a tunable probability variable. (2) Having four probability variables, they use machine learning to tune these during execution to the specific workload and the characteristics of the underlying machine. Thus, Spitfire is a dynamic system that can still adopt to likely changes of the still young persistent memory hardware.

Chapter 2

Paper 1: Managing Non-Volatile Memory in Database Systems

This paper tries answering the question of how persistent memory (PMem) can be integrated into an OLTP database management system. The proposed solution (called: *HyMem*) is optimized for the special characteristics of PMem (e.g., random access and byte-addressability) while still utilizing other layers in the memory hierarchy (i.e., DRAM and SSD). I follow the design of a tradition storage engine comprised of a B+-Tree [BM70], ARIES-style write-ahead logging [MHL+92], and a buffer manager. The main contribution of this paper is the optimization and combination of the individual components for PMem: The buffer manager lazily loads individual cache lines from PMem into DRAM pages. Compared to a standard buffer manager, which eagerly loads the entire page, this technique vastly reduces the amount of I/O, because often only a portion of each page is needed, especially in OLTP. Given that HyMem is only loading parts of a page, compacted DRAM pages ("mini pages") are used to reduces the memory footprint in the buffer cache for sparsely populated pages.

Another contribution is the evaluation, which compares HyMem with four exiting systems: an in-memory engine, a B+-Tree placed directly on PMem, a buffer manager for DRAM and NVM, and a buffer manager for DRAM and SSD. HyMem gracefully scales across all layers of the new memory hierarchy (i.e., with PMem) and thus constitutes a cost efficient option [Lom18]. It offers, depending on the data set size, competitive performance to systems dedicated to the particular layer:

DRAM: Leis et al. [LHKN18] have shown that a well designed buffer manager can be competitive with the performance of in-memory systems. Building on these results (mainly by using pointer swizzling) HyMem has competitive performance with an in-memory DBMS. Unlike in-memory systems, HyMem is not limited to DRAM, which is both capped in capacity and expensive.

PMem: Many PMem-based storage engines are not using DRAM, which is still present in machines today. Utilizing the low latency and high bandwidth of DRAM gives HyMem an edge when the data set fits into DRAM and its PMem-optimized data migration policy allows it to stay competitive for larger data sets.

SSD/HDD: Compared to data structures that work directly (in-place) on PMem, HyMem can easily support a third layer (i.e.: SSD or HDD) due to the use of fix-sized pages. Therefore, HyMem can scale to larger-than-PMem data sets.

Contributions: The system implementation and evaluation as well as the writing of the paper itself was done by the first author. Co-authors helped with the system design, prove reading, and identifying related work. In particular, Fujitsu granted us access and provided help with using a persistent memory system.

Managing Non-Volatile Memory in Database Systems

Alexander van Renen
Technische Universität München
renen@in.tum.de

Viktor Leis
Technische Universität München
leis@in.tum.de

Alfons Kemper
Technische Universität München
kemper@in.tum.de

Thomas Neumann
Technische Universität München
neumann@in.tum.de

Takushi Hashida
Fujitsu Laboratories
hashida.takushi@jp.fujitsu.com

Kazuichi Oe
Fujitsu Laboratories
ooe.kazuichi@jp.fujitsu.com

Yoshiyasu Doi
Fujitsu Laboratories
yosh-d@jp.fujitsu.com

Lilian Harada
Fujitsu Laboratories
harada.lilian@jp.fujitsu.com

Mitsuru Sato
Fujitsu Laboratories
msato@jp.fujitsu.com

ABSTRACT

Non-volatile memory (NVM) is a new storage technology that combines the performance and byte addressability of DRAM with the persistence of traditional storage devices like flash (SSD). While these properties make NVM highly promising, it is not yet clear how to best integrate NVM into the storage layer of modern database systems. Two system designs have been proposed. The first is to use NVM exclusively, i.e., to store all data and index structures on it. However, because NVM has a higher latency than DRAM, this design can be less efficient than main-memory database systems. For this reason, the second approach uses a page-based DRAM cache in front of NVM. This approach, however, does not utilize the byte addressability of NVM and, as a result, accessing an uncached tuple on NVM requires retrieving an entire page.

In this work, we evaluate these two approaches and compare them with in-memory databases as well as more traditional buffer managers that use main memory as a cache in front of SSDs. This allows us to determine how much performance gain can be expected from NVM. We also propose a lightweight storage manager that simultaneously supports DRAM, NVM, and flash. Our design utilizes the byte addressability of NVM and uses it as an additional caching layer that improves performance without losing the benefits from the even faster DRAM and the large capacities of SSDs.

ACM Reference Format:

Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing Non-Volatile Memory in Database Systems. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3183713.3196897

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.

ACM ISBN 978-1-4503-4703-7/18/06...$15.00
https://doi.org/10.1145/3183713.3196897

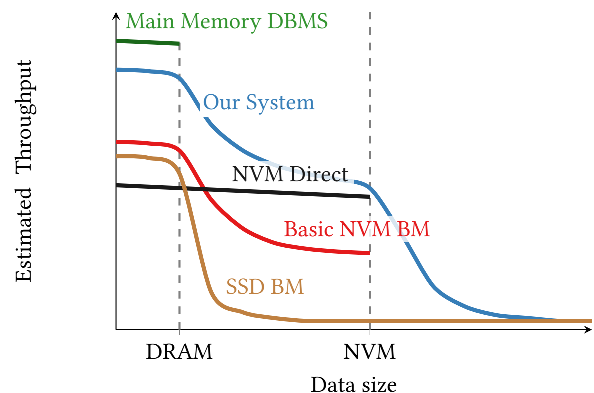


Figure 1: System designs under varying data sizes.

1 INTRODUCTION

Non-volatile memory (NVM), also known as Storage Class Memory (SCM) and NVRAM, is a radically new and highly promising storage device. Technologies like PCM, STT-RAM, and ReRAM have slightly different features [35], but generally combine the byte addressability of DRAM with the persistence of storage technologies like SSD (flash). Because commercial products are not yet available, the precise characteristics, price, and capacity features of NVM have not been publicly disclosed (and like all prior NVM research, we have to resort to simulation for experiments). What is known, however, is that for the foreseeable future, NVM will be slower (and larger) than DRAM and, at the same time, much faster (but smaller) than SSD [13]. Furthermore, NVM has an asymmetric read/write latency—making writes much more expensive than reads. Given these characteristics, we consider it unlikely that NVM can replace DRAM or SSD outright.

While the novel properties of NVM make it particularly relevant for database systems, they also present new architectural challenges. Neither the traditional disk-based architecture nor modern main-memory systems can fully utilize NVM without major changes to their designs. The two components most affected by NVM are logging/recovery and storage. Much of the recent research on NVM has optimized logging and recovery [5, 16, 22, 36, 45]. In this work, we instead focus on the storage/caching aspect, i.e., on dynamically deciding where data should reside (DRAM, NVM, or SSD).

Two main approaches for integrating NVM into the storage layer of a database system have been proposed. The first, suggested by Arulraj et al. [4], is to use NVM as the primary storage for relations as well as index structures and perform updates directly on NVM. This way, the byte addressability of NVM can be fully leveraged. A disadvantage is that this design can be slower than main-memory database systems, which store relations and indexes in main memory and thereby benefit from the lower latency of DRAM. To hide the higher NVM latency, Kimura [25] proposed using a database-managed DRAM cache in front of NVM. Similar to a disk-based buffer pool, accesses are always performed on in-memory copies of fixed-size pages. However, accessing an uncached page becomes more expensive than directly accessing NVM, as an entire page must be loaded even if only a single tuple is accessed. Furthermore, neither of the two approaches supports very large data sets, as the capacity of NVM is limited compared to SSDs.

In this work, we take a less disruptive approach and implement NVM as an additional caching layer. We thus follow Michael Stonebraker, who argued that NVM-DIMMs are …

> *"… not fast enough to replace main memory and they are not cheap enough to replace disks, and they are not cheap enough to replace flash."* [41]

Figure 1 sketches the performance characteristics and capacity restrictions of different system designs (*Buffer Manager* is abbreviated as *BM*). Besides the two NVM approaches ("Basic NVM BM", "NVM Direct"), we also show main-memory systems ("Main Memory"), and traditional SSD buffer managers ("SSD BM"). Each of these designs offers a different tradeoff in terms of performance and/or storage capacity. As indicated in the figure, all existing approaches exhibit steep performance cliffs ("SSD BM" at DRAM size and "Basic NVM BM" at NVM size) or even hard limitations ("Main Memory" at DRAM size, "NVM Direct" at NVM size).

In this work, we propose a novel storage engine that simultaneously supports DRAM, NVM, and flash while utilizing the byte addressability of NVM. As the "3 Tier BM" line indicates, our approach avoids performance cliffs and performs better than or close to that of specialized systems. NVM is used as an additional layer in the storage hierarchy supplementing DRAM and SSD [7, 13]. Furthermore, by supporting SSDs, it can manage very large data sets and is more economical [18] than the other approaches. These robust results are achieved using a combination of techniques:

- To leverage the byte-addressability of NVM, we cache NVM accesses in DRAM at cache-line granularity, which allows for the selective loading of individual hot cache lines instead of entire pages (which might contain mostly cold data).
- To more efficiently use the limited DRAM cache, our buffer pool transparently and adaptively uses small page sizes.
- At the same time, our design also uses large page sizes for staging data to SSD—thus enabling very large data sets.
- We use lightweight buffer management techniques to reduce the overhead of in-memory accesses.
- Updates are performed in main memory rather than directly on NVM, which increases endurance and hides write latency.

The rest of this paper is organized as follows. We first discuss existing approaches for integrating NVM into database systems in Section 2. We then introduce some key techniques of our storage engine in Section 3 before describing how our approach supports DRAM, NVM, and SSDs in Section 4. Section 5 evaluates our storage engine by comparing it with the other designs. Related work is discussed in Section 6, and we conclude the paper in Section 7.

2 BACKGROUND: NVM STORAGE

Several architectures for database systems optimized for NVM have been proposed in the literature. In this section, we revisit the two most promising of these designs and abstract their general concepts into two representative system designs. They are illustrated in Figure 2 alongside the approach that we propose in this paper.

2.1 NVM Direct

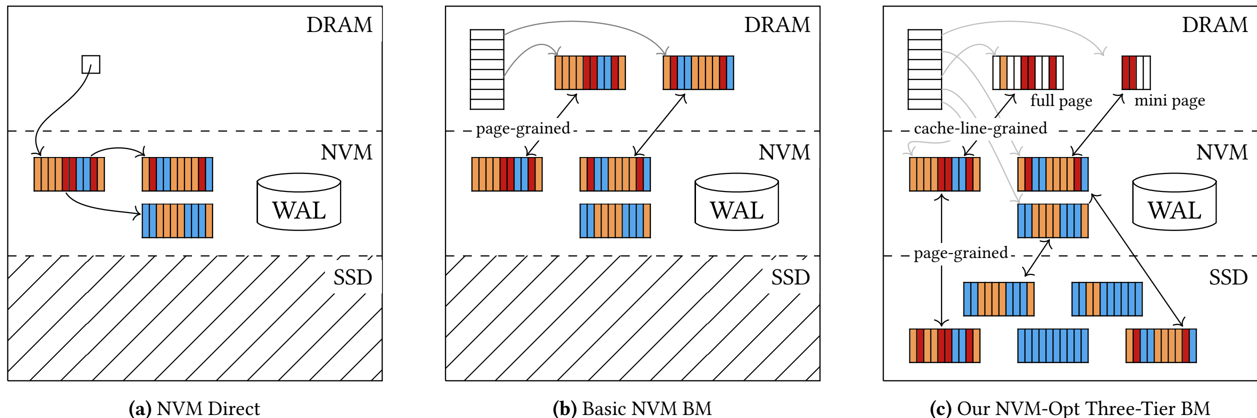
NVM, which offers latencies close to DRAM and byte addressability, can be used as the primary storage layer. A thorough investigation of different architectures for NVM Direct systems has been conducted by Arulraj et al. [4]. Their work categorizes database systems into in-place update, log-structured, and copy-on-write engines, before adapting and optimizing each one for NVM. Experimental results suggest that in most cases an in-place update engine achieves the highest performance as well as the lowest wear on the NVM hardware. Therefore, we chose this in-place update engine as a reference system for working directly on NVM (Figure 2a).

One challenge of using NVM is that writes are not immediately persistent because NVM is behind the same CPU cache hierarchy as DRAM and changes are initially written to the volatile CPU cache. It is only when the corresponding cache line is evicted from the CPU cache that the update becomes persistent (i.e., written to NVM). Therefore, it is not possible to prevent a cache line from being evicted and written to NVM, and each update might be persisted at any time. It is, however, possible to force a write to NVM by flushing the corresponding cache lines. These flushes are a building block for a durable and recoverable system.

Logging is implemented as follows. A tuple is updated by first writing a write-ahead log (WAL) entry that logs the tuple id and the changes (before and after image). Then, the log entry needs to be persisted to NVM by evicting the corresponding cache lines. Intel CPUs with support for NVM like the Crystal Ridge Software Emulation Platform [14], which is also used in our evaluation, provide a special instruction for that: `clwb` allows one to write a cache line back to NVM without invalidating it (like a normal `clflush` instruction would). In addition, to ensure that neither the compiler, nor the out-of-order execution of the CPU reorders subsequent stores, a memory fence (`sfence`) has to be used. Thereafter, the log entry is persistent and the recovery component can use it to redo or undo the changes to the actual tuple. At this point, the transaction can update and then persist the tuple itself. After the transaction is completed, the entire log written by the transaction can be truncated because all changes are already persisted to NVM.

As illustrated in Figure 2a, the design keeps all data in NVM. DRAM is only used for temporary data and to keep a reference to the NVM data. The log is written to NVM as well.

The NVM direct design has several advantages. By keeping the log minimal (it contains only in-flight transactions), recovery is very efficient. In addition, read operations are very simple because the system can simple read the requested tuple directly from NVM.



(a) NVM Direct **(b)** Basic NVM BM **(c)** Our NVM-Opt Three-Tier BM

Figure 2: NVM-Based Storage Engine Designs – NVM Direct (a) stores all data on NVM, which allows for cache-line-grained accesses. Basic buffer managers (b) fixed-size pages in DRAM, but require page-grained accesses to NVM. Our design (c) uses fixed-size pages to enable support for SSD and, in addition, supports cache-line-grained loading for NVM-resident data to DRAM. (hot, warm, cold cache lines)

However, there are also downsides. First, due to the higher latency of NVM compared to DRAM, it becomes more difficult to achieve a very high transaction throughput. Second, working directly on NVM without a buffer wears out the limited NVM endurance, thus potentially causing hardware failures. Third, an engine that works directly on NVM is difficult to program, because there is no way to prevent eviction and any modification is potentially persisted. Therefore, any in-place write to NVM must leave the data structure in a correct state (similar to lock-free data structures, which are notoriously difficult).

2.2 Basic NVM Buffer Manager

Given the downsides of the NVM direct approach, using DRAM as a cache in front of NVM may seem like a promising alternative. A well-known technique for adaptive memory management between a volatile and persistent layer is a buffer manager. It is used by most traditional disk-based database systems and can easily be extended to use NVM instead of SSD. We illustrate this idea in Figure 2b.

In buffer-managed systems, all pages are stored on the larger, persistent layer (NVM). The smaller, volatile layer (DRAM) acts as a software-managed cache called a buffer pool. Transactions operate only in DRAM and use the `fix` and `unfix` functions to lock pages into the buffer pool while they are accessed. In the traditional buffer manager (DRAM + SSD/HDD), this was necessary because it is not possible to make modifications directly on a block-oriented device. In the case of NVM, we argue that it is still beneficial because of the higher latency and the limited endurance of NVM.

An NVM-optimized variant of this approach has been introduced in the context of the research prototype FOEDUS [25]. The memory is divided into fixed-size pages, and transactions only operate in DRAM. Instead of storing page identifiers, like a traditional buffer manager, FOEDUS stores two pointers. One identifies the NVM-resident copy of the page, the other one (if not `null`) the DRAM one. When a page is not found in DRAM, it is loaded into a buffer pool. FOEDUS uses an asynchronous process to combine WAL log entries and merge them into the NVM-resident pages to achieve durability. Thus, a page is never directly written back but only indirectly via the log. The system is optimized for workloads fitting into DRAM. NVM is mostly used for durability and cold data.

Our goal, in contrast, is to support workloads that also access NVM-resident data frequently. Therefore, we extend the idea of a buffer manager and optimize it to make NVM access cheap. A non-optimized version is used as a baseline to represent layered systems.

2.3 Recovery

Besides the storage layout, the logging and recovery components of database systems are also greatly impacted by the upcoming NVM hardware. Log entries can be written to NVM much faster than to SSD. Therefore, from a performance perspective, it is always beneficial to replace SSD storage with NVM as the logging device.

In this work, we focus on the storage layout and therefore implement the same logging approach in each evaluated system. This makes it possible to compare only the advantages and disadvantages of the storage engine itself with less interference of other database components.

We use write ahead logging with redo and undo information. The undo entries enable one to perform rollback and to undo the effect of loser transactions during recovery. The redo entries are used to repeat the effects of committed transactions during recovery if it was not yet persistent. The buffer-manager-based systems keep a log sequence number per page to identify the state of the page during recovery. In the NVM direct approach, this is not necessary, as changes are always immediately written to NVM. Therefore, only in-flight transactions need to be undone and every committed transaction is durable already.

Logging for NVM-based systems can be and has been optimized in prior work [5, 36]. While each of the described storage architectures can benefit from more advanced logging techniques, we believe that the impact on the storage engine is largely orthogonal and the two problems can be treated independently.

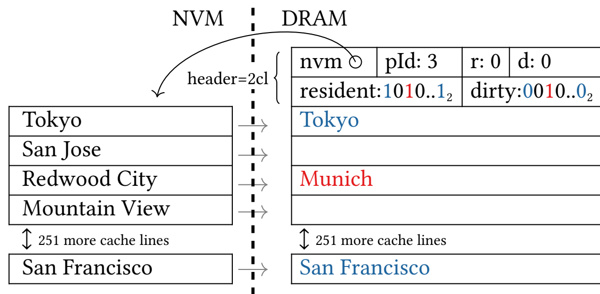


Figure 3: Cache-Line-Grained Pages – The bit masks indicate which cache lines are resident and which are dirty.

3 NVM BUFFER MANAGEMENT

The goal of our architectural blueprint is a system that performs almost as well as a main-memory database system on smaller data sets but scales across the NVM and SSD storage hierarchy while gracefully degrading in performance (cf. Figure 1). For this purpose, we design a novel DRAM-resident buffer manager that swaps cache-line-grained data objects between DRAM and NVM—thereby optimizing the bandwidth utilization by exploiting the byte addressability of NVM. As illustrated in Figure 2c, scaling beyond DRAM to SSD sizes led us to rely on traditional page-grained swapping between NVM and SSD. Between DRAM and NVM, we adaptively differentiate between full-page memory allocation and mini-page allocation to further optimize the DRAM utilization. This way, individual "hot" data objects resident on mostly "cold" pages are extracted via the cache-line-grained swapping into smaller memory frames. Only if the mini page overflows, is it transparently promoted to a full page—but it is still populated one cache-line at a time. We also devise a pointer swizzling scheme that optimizes the necessary page table indirection in order to achieve nearly the same performance as pure main-memory systems, which obviate any indirection but incur the memory wall problem once the database size exceeds DRAM capacity.

3.1 Cache-Line-Grained Pages

Compared to flash, the low NVM latency (hundreds of nanoseconds) makes it feasible to transfer individual cache lines instead of entire pages. Using this, our so-called *cache-line-grained pages* are able to extract only the hot data objects from an otherwise cold page. Thus, we preserve bandwidth and thereby increase performance. In the following, we discuss the details of this idea.

A single SSD access takes hundreds of micro seconds. It is therefore important to transfer large chunks (e.g., 16 kB) at a time in order to amortize the high latency. Therefore, a traditional buffer manager has to work in a page-grained fashion: When a transaction fixes a page, the entire page is loaded into DRAM and the data stored on it can be processed. Our buffer manager, in contrast, initially only allocates a page in DRAM without fully loading it from NVM. Upon a transaction's request for a certain memory region, the buffer manager retrieves the corresponding cache lines of the page (if not already loaded).

The page layout is illustrated in Figure 3. The cache-line-grained pages maintain a bit mask (labeled `resident`) to track which cache lines are already loaded. In the example, the first, third, and last cache line is loaded, as indicated by a bit set to 1 at the corresponding position in the bit mask. Similar to the `resident` bit mask, the `dirty` bit mask is used to track and write back dirty cache lines. The `r` and `d` bits indicate whether the entire page is resident and dirty, respectively. To allow for the loading of cache lines on demand, pages additionally store a pointer (`nvm`) to the underlying NVM page. With 16 kB pages, there are 256 cache lines and therefore the two bit masks require 32 byte each. Together with the remaining fields ($|\texttt{nvm}| + |\texttt{pId}| + |\texttt{r}| + |\texttt{d}| = (8 + 8 + 1 + 1)$ byte = 18 byte), the entire header ($(2 * 32 + 18)$byte = 82 byte) fits into 2 cache lines (128 byte) and thus incurs only a negligible space overhead of less than 0.8 %.

While this cache-line-grained design includes an extra branch on every access (to check the bit mask), it often reduces the amount of memory loaded from NVM into DRAM drastically: As an example, consider the leaf of a B-tree [6] where pairs of keys and values are stored in sorted order. Assuming a page size of 16 kB and a key and value size of 8 byte each, there are at most $16\,\text{kB} \div (8\,\text{byte} + 8\,\text{byte}) = 1024$ entries on a single page. A lookup operation only requires a binary search, which uses $log_2(1024) = 10$ cache lines at most. Therefore, our design only needs to access $64\,\text{byte} * 10 = 640\,\text{byte}$, instead of 16 kB. While this is already a huge difference, it can be even greater. In the case of the YCSB and TPC-C benchmarks, which we use in our evaluation (Section 5), we measured an average of 6.5 accessed cache lines per lookup.

A system allowing for cache-line-grained accesses is more difficult to program than a conventional page-based approach. The reason for this is that all data needs to be made resident explicitly before accessing it and marked dirty after modifying it. But working with a cache-line-granularity is optional and it is also possible to load and write back entire pages. Therefore, we only implement the operations, that provide the most benefit, in a cache-line-grained fashion: like lookup, insert, and delete. Other infrequent or complicated operations (like restructuring the B-tree) are implemented by loading and processing the full page (avoiding the residency checks). The overhead of checking the residency of every single cache line only pays off if we access a small number of cache lines. During scan operations or the traversal of inner nodes in a B-tree, many cache lines are accessed and cache-line-grained loading should therefore not be used.

3.2 Mini Pages

Cache-line-grained pages reduce the consumed bandwidth to a minimum by only loading those cache lines that are actually needed. However, the page layout described previously still consumes much more DRAM than necessary. In this section, we introduce a second, smaller page type called *mini page*, which reduces the wasted memory. Consider the B-tree leaf example from above: Merely 640 byte out of 16 kB are accessed, but the system still allocates the 16 kB (excluding the header) in DRAM for the page. This problem is known from traditional disk-based systems: An entire page is loaded and stored in DRAM even if only a single tuple is required, wasting valuable NVM bandwidth and DRAM capacity. In the following, we will use the term *full page* to refer to a traditional page, as it was introduced before. Note that both pages (mini and full) are able to use

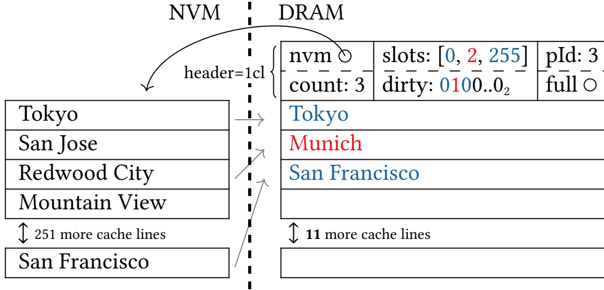


Figure 4: Mini Pages – The `slots` array indicates which cache lines are loaded (max 16). If promoted, `full` points to the full page.

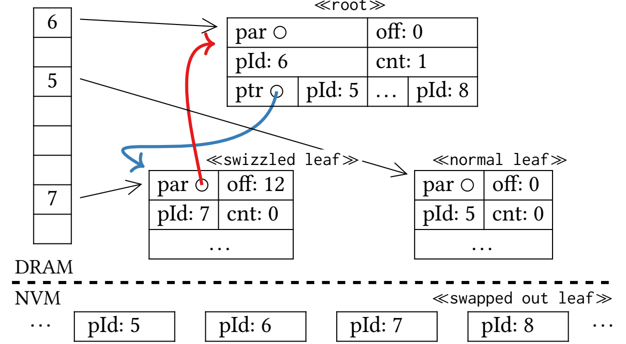


Figure 5: Pointer Swizzling – A B-tree with a root (pId: 6) and three child pages: A swizzled page (pId: 7), a normal DRAM page (pId: 5) and a page currently not in DRAM (pId: 8).

cache-line-grained loading to optimize bandwidth utilization—but not DRAM utilization. Hence, the term *cache-line-grained page* can refer either to a *mini page* or a *full page*.

The implementation of mini pages is illustrated in Figure 4. It consumes only 1088 byte of memory and is able to store up to 16 cache lines. The `slots` array implements the indirection. It stores the physical cache line id for each of the 16 cache line slots. For example, the cache line with the content "San Francisco" is located at the physical index 255 on the page but loaded into slot 2 on the mini page. Therefore, the `slots` array stores the index 255 at position 2. The array only requires 16 byte because each physical cache line id fits into one byte. In total, the mini page header fits into a single cache line: $|nvm| + |slots| + |pId| + |count| + |dirty| + |full| = (8 + 16 + 8 + 1 + 2 + 8)$ byte = 43 byte. This is a very low overhead (0.3 %) when compared to the size of a full page, which would be used in a system without mini pages. The `count` field indicates how many cache lines are loaded, e.g., a value of three means that the first three cache line slots of the mini page are occupied. The additional `dirty` bit mask indicates which cache lines must be written back to NVM when the page is evicted. In our example, the cache line "Redwood City" changed to "Munich" and needs to be written back.

Accessing memory on a mini page is more complicated than on a full page. Due to the mapping of cache lines, data members on the page can-not be directly accessed. Therefore, we use an abstract interface that enables transparent page access:

```
void* MakeResident(Page* p, int offset, int n)
```

The function takes a page `p` as an input and returns a pointer to the memory at the specified `offset` with a length of `n` bytes. In case `p` is a full page, the specified cache lines are loaded (if not yet resident) and a pointer to it is returned. Otherwise, in case of a mini page, the function searches the `slots` array for the requested cache lines. If these are not yet resident, they are loaded and added to the slots array. Afterwards, a pointer to the offset in the (now) resident cache line is returned. Thus, this basic interface transparently resolves the indirection within the mini pages. Compared to a traditional page, the only difference is that memory on mini pages can no longer be accessed directly but only via this function.

Mini pages need to guarantee that the memory returned by these functions is always contiguous, i.e., if more than one cache line is requested (e.g., cache lines with id 3 and 4), they need to be stored consecutively (i.e., 4 needs to be stored directly after 3) in the mini page's data array. Otherwise, the returned pointer would only be valid for the first cache line. To guarantee this, our implementation maintains the cache lines in sorted order (w.r.t. their memory locations). The overhead of maintaining order is small because, there are at most 16 elements and it is not on the critical path (only after loading from NVM). The benefit of this approach is that it simplifies implementation, as it avoids complicated reordering logic.

When a mini page does not have enough memory left to serve a request, it is promoted to a full page. To do this, an empty full page is allocated from the buffer manager and stored in the mini page's `full` member. Next, the current state of the mini page is copied into the newly allocated full page: all resident cache lines, the residency and dirty information. If the example mini page in Figure 4 was promoted, the newly initialized full page would look like the one in Figure 3. Finally, the page mapping table in the buffer manager is updated to point to the full page. From now on, the mini page is called *partially promoted* and all requests to the mini page are forwarded to the full page. It is only safely garbage collected, once the last transaction unfixes it. This is guaranteed to happen, as the page mapping table points to the full page and therefore no new references to the mini page are created. This feature is convenient for the data structures using mini page because this way its reference to the mini page is not invalidated when a promotion happens. Thus, a promotion is hidden from data structures and does not incur additional complexity.

3.3 Pointer Swizzling

While the buffer pool, allows the system to benefit from the low latency DRAM cache, it also introduces a non-neglectable overhead. In this section, we introduce pointer swizzling, a technique that reduces this overhead (mainly the page table lookup) by dynamically replacing page ids with physical pointers. In a traditional buffer manager (DRAM+SSD/HDD), this overhead is only noticeable if most of the working set fits into DRAM. Otherwise, the page has to be loaded from SSD/HDD anyway, which is orders of magnitude slower compared to the hash table lookup. In contrast to traditional buffer managers, in our proposed system, this overhead is also relevant for larger workloads. We cannot hide the overhead

behind even slower loads because (a) these loads are not that much slower (NVM latency is a lot lower than that of flash) and (b) the amount of data read is much less due to the cache-line-grained loading. Therefore, it is important for our system to minimize these management overheads as much as possible.

Pointer swizzling has recently been proposed in the context of traditional buffer managers (DRAM + SSD/HDD) [17]. The idea is to encode the address of the page directly in the page identifier for DRAM-resident pages. The most significant bit of the page identifier determines whether the remaining bits constitute a page identifier or a pointer. Thus, when accessing a page, the system first checks whether the most significant bit is set. If so, the remaining bits encode a pointer that can be dereferenced directly. Otherwise, the remaining bits are a page identifier and the system has to check the hash table in the buffer manager to find the page or load it from NVM if not present. This way, the hash table lookup is avoided for DRAM-resident pages and thereby the overhead is reduced to a single branch.

Figure 5 illustrates our implementation of pointer swizzling. On the left-hand side, the buffer manager's page mapping table is shown. It maps page identifiers (numbers in the table) to pages (represented by arrows). The example shows a B-tree with one `root` node (`pId` 6) and three leaf nodes: The first one (pId 7) is a `swizzled leaf`. The `root` can use a pointer (blue arrow) to access it instead of the page id. The second one (`pId` 5) is a `normal leaf` (not swizzled) and the third one (`pId` 8) is a `swapped out leaf`—currently not located in DRAM.

In the example, the root page has one swizzled child (as indicated by the `cnt`) field. A page with swizzled children can never be swapped out because the pointer to the swizzled child would be persisted. When a swizzled page (the left child with page identifier 7 in the example) is swapped out, it needs to update its parent (located via the `par` pointer): First, it decreases the child count, which is located at a fixed offset. Second, it converts the pointer pointing to itself back into a normal page identifier. The location of this pointer can be found using the offset field (`off`). The parent pointer (8 byte) and the offset (2 byte) require an additional 10 byte of space in the page header and therefore still fit into the mini page (1 cache line) and full page header (2 cache lines). Pointer swizzling is compatible with various data structures (trees, heaps, hashing); it only requires fix-sized pages and these additional header fields.

Consider a swizzled mini page. When a mini page is promoted to a full page, the swizzling information needs to be updated as well. This happens when the partially promoted mini page is unfixed. Until then it acts as a wrapper around the full page. When it is unfixed, the pointer in the parent page needs to be redirected to the full page. In addition, the pointer to the parent (`par`) and offset (`off`) in the mini page need to be copied to the full page.

4 THREE-TIER BUFFER MANAGEMENT

So far we have presented cache-line-grained loading, mini pages, and pointer swizzling as building blocks for an efficient DRAM buffer pool over an NVM storage layer. The next step towards our goal of building a storage engine that scales across the NVM and SSD hierarchy (cf. Figure 1) is to add flash (SSD) as a third layer. Such a three-tier design drastically increases the maximum workload size in comparison to that of an in-memory or NVM-only system. This section describes the involved buffer replacement strategies and a low overhead way of adding this third layer.

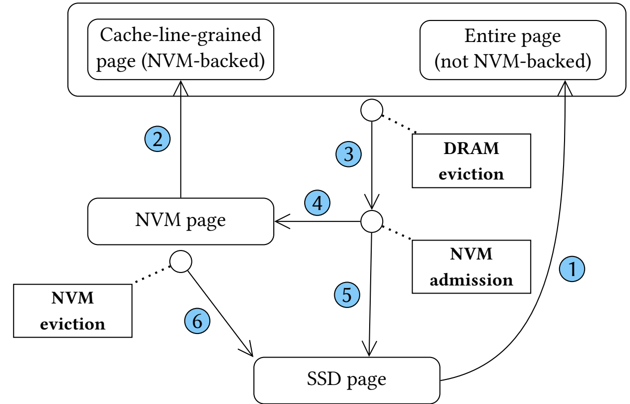


Figure 6: Page Life Cycle – There are five possible page transitions and the three critical decisions (DRAM eviction, NVM admission, and NVM eviction).

Although adding support for SSDs does not improve performance, it is still important, as it allows for the management of larger data sets and can also be more economical: Real-world data is often hot/cold clustered (e.g., older data is accessed less frequently). To process the hot data as fast as possible, it should reside in main memory. But it is neither a good practice to keep the cold data in a separate system nor is it cheap to buy huge amounts of DRAM to obtain enough storage to keep the cold data in DRAM as well. Our layered approach solves this problem by providing close to main memory speed for the hot data (provided it fits into DRAM) while also supporting cheap SSD storage for cold data in a single system. Beyond other systems, it even allows one to compactify the individual working sets of an application via mini pages.

4.1 Design Outline

In our three-tier architecture, buffer management is done by using both NVM and DRAM as selective caches over the SSD storage layer (Section 4.2). Pages are only accessed (read and written) in DRAM, and write ahead logging (WAL) is used to ensure durability. When a page is evicted from DRAM, it is either admitted to NVM or written back to SSD, depending on how hot the page is.

To locate pages on NVM, a page table (similar to the one in DRAM) is required. To avoid overheads, this can be implemented by using a combined page table for both DRAM and NVM—reducing the number of table lookups from two to one (Section 4.3).

For recovery (Section 4.4), we propose using textbook-style write ahead logging and an ARIES-based restart procedure. In a three-tier architecture, it becomes necessary to reuse the content of NVM-resident pages to allow for faster restarts. Therefore, the content of the combined mapping table is reconstructed after a restart.

4.2 Replacement Strategies

A three-tier architecture needs to manage two buffer pools (DRAM and NVM) instead of one. In this section, we detail the page transitions that can occur and describe the three necessary replacement decisions: DRAM eviction, NVM eviction and NVM admission. The process is illustrated in Figure 6 and can be used as an overview.

Initially, all newly-allocated pages start out on SSD. When a transaction requests a page, it is loaded directly and completely into DRAM (①). The pages are loaded completely because the block-based device only allows for a block-based access. We do not put pages into NVM when they are loaded from SSD because accesses are served directly from DRAM and putting them into NVM as well would only waste NVM memory. Pages are only admitted to NVM when they are swapped out of DRAM. Pages loaded directly from SSD are not NVM-backed and therefore cannot operate in a cache-line-grained fashion. This is only possible when the page is loaded from NVM (②) and therefore NVM-backed.

When there are no more free slots available in DRAM, the buffer manager needs to evict any of the DRAM-resident pages (③) in order to make room for a new one. **DRAM eviction** is the first out of three decisions our buffer manager has to perform. The goal is simply to keep the hottest pages in DRAM. We use the well-known clock (or second chance) algorithm, which performs reasonably well in both overhead and quality. It continuously loops through all pages in the buffer pool and swaps those pages out that have not been touched since the previous iteration.

Once a page is chosen to be evicted from DRAM (③) and is not already resident in NVM, it is considered for **NVM admission**, which is the second out of three decisions. This decision is a more difficult one because the goal is to identify warm pages instead of hot pages. It has been studied in the context of the ARC replacement strategy [34], where two queues are used to identify warm pages in order to optimize the replacement of hot pages in a two-layer system. Building on this idea, we use one set, which we call the *admission set*, to identify recently accessed pages. The idea is to admit pages to NVM that were recently denied admission. Each time a page is considered for admission, the system checks whether the page is in the admission queue. If so, it is removed from the set and admitted into NVM (④). Otherwise, it is added to the set and remains only on SSD (⑤). By limiting the size of the admission set, we make sure that it only contains pages that were recently considered for admission. This way, pages that are frequently swapped out of DRAM are admitted into NVM, but those that are only loaded once do not pollute NVM.

The third replacement decision is **NVM eviction**, i.e., choosing a page to be swapped out of NVM (⑥) when a new page is admitted. To keep our implementation simple, we also use the clock algorithm for this decision; however, as this is a rather expensive operation (writing a page (16 kB) to NVM and, if it was dirty, to SSD), one could opt for a slower algorithm that in turn yields better quality.

4.3 Combined Page Table

For workloads fitting into NVM, the third layer (SSD) is not used and should therefore not cause an overhead. We achieve this by the use of a *combined page table*, which stores both mappings (page identifier → DRAM location and page identifier → NVM location) in one hash table. The resulting structure is shown in Figure 7. When retrieving a page, the memory address of the page can be used to determine whether it is located in DRAM or NVM. If a page is not found in DRAM, its NVM-resident copy is still found and can be directly used without an additional hash table lookup. Therefore, there are no extra steps involved compared to a two-layer system. However, the size of the hash table differs because of the additionally stored mapping for NVM pages. According to our experimental results, the introduced overhead is less than 5 %.

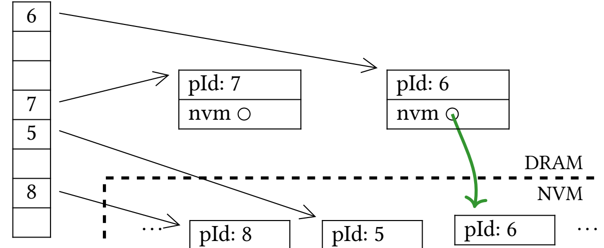


Figure 7: Single-Table Mapping – Using one hash table for DRAM and NVM-resident pages eliminates most overhead for managing the SSD layer. The hash table entries are identified by their location in memory (DRAM or NVM).

4.4 System Restart

The page mapping table is performance critical and is therefore stored in DRAM. After a restart, it needs to be rebuilt instead of recovering solely from SSD, which would have two major drawbacks: First, the time until the system can process the first transaction would be higher because more log entries have to be replayed, as the SSD version of the pages are older than the NVM version. Second, once the system is restarted, it takes a longer time to reach the pre-crash throughput again because not only the DRAM cache but also the NVM cache is empty. Therefore, our system reconstructs the page mapping table after a restart. This requires scanning over all NVM pages, reading their page identifiers and adding them to the DRAM-resident page table. This technique was not feasible for slow non random access mediums (like flash or HDD) but performs reasonably well for NVM. Reading the page identifiers for 100 GB of NVM takes slightly less than 1 second, but allows for a faster restart.

5 EVALUATION

In this section, we present an experimental analysis of our proposed architecture and compare it with other NVM management techniques. To provide a fair comparison, all evaluated architectures are implemented within the same storage engine. Consequently, all systems use the same logging scheme, B+-tree, and test driver. The only difference is the way DRAM, NVM, or/and SSD are accessed. This allows us to measure the difference between the architectures mostly independent of individual implementation choices.

5.1 Experimental Setup

We conduct our experiments on the Crystal Ridge Software Emulation Platform (SEP) provided by Intel [14]. It is a dual-socket

system equipped with Intel(R) Xeon(R) CPU E5-4620 v2 processors (2.6 GHz, 8 cores, and 20 MB L3 cache). This processor is extended to include the `clwb` instruction and is able to configure the NVM latency between 165 ns to 1800 ns (unless otherwise noted, 500 ns are used). The `clwb` instruction in combination with memory fences (`sfence`) is used to persist a certain cache line. Unlike the `clflush` instruction, it does not invalidate the cache line, thus triggering a reload on the next access, but only writes it back to the underlying memory and marks it as unmodified. We use the libpmem [1] library from the libpmem.io stack as a platform-independent wrapper around these instructions. The machine is equipped with 48 GB of DRAM, out of which 32 GB are connected to the first socket. To avoid NUMA effects, we conduct our experiments exclusively on this socket. The simulated NVM-DIMMs are exposed as a block device, which is formatted as an *ext4* file system and mounted with DAX (direct access) support enabled. This file system is then mapped into the address space of our process and can be directly accessed without file system overheads due to DAX. Note that buffer-managed systems do not require a special NVM allocator [38] because pointers into or the dynamic allocation of NVM are not used.

We implement each table as a B+-tree using C++ templates and 16 kB pages (page size is not restricted to the OS's virtual memory page size). The B-tree uses binary search and stores key and payloads in separate arrays (sorted by keys).

When ingesting data in preparation for a benchmark, the load factor of the B-tree is configured to 0.66. The term data size is used to describe the total memory consumption of the B-tree after loading the data. Therefore, if the flat data is 5 GB in size, the resulting data size would be around 7.5 GB due to the tree structure and the load factor. Transactions are executed on a single thread. By using no-steal and no-force in the buffer manager in combination with traditional write ahead logging (WAL), we ensure durability. In all experiments, the transactions and the code executing them are implemented in C++ and compiled together with our storage engine into one binary.

5.2 Workloads

We used YCSB and TPC-C in our experiments, which mostly use OLTP-style transactions. These are better suited to evaluate a storage engine, as they pose a greater challenge than OLAP-style full table scans.

YCSB is a popular key-value store benchmark framework [11]. It consists of a single table with a 4 byte primary key and 10 string fields of 100 byte each. YCSB defines simple "CRUD"-style operations on this table, which can be combined into workloads. In the YCSB experiments, we focus on point lookups, updates, and range scans (inserts and deletes are evaluated using TPC-C). We use Zipf-distributed ($z = 1$, non clustered popular keys) keys to model real-world data skew [20]. The corresponding row is located in the table and a (uniformly) randomly chosen field is read (lookup, range scan) or updated (updates). We define three workloads, which are generalizations of the five pre-defined example workloads (A-E) in YCSB.

YCSB-RO uses 100 % point lookup operations (same as YCSB "Workload C").

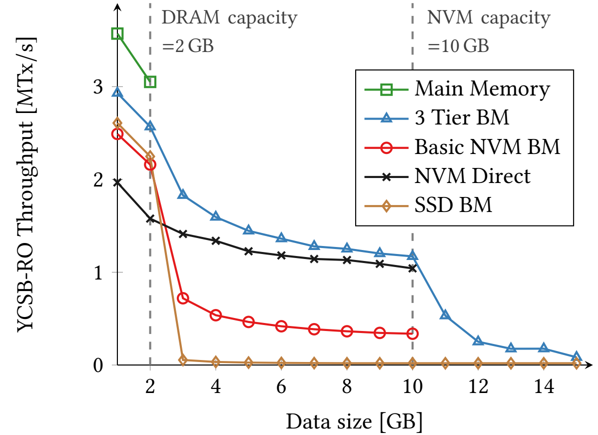


Figure 8: YCSB-RO – Performance for varying data sizes on read-only YCSB workload. The capacity of DRAM, NVM, and SSD is set to 2 GB, 10 GB, and 50 GB, respectively.

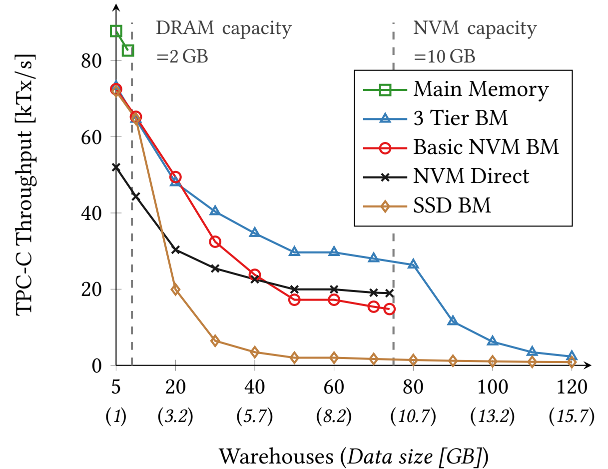


Figure 9: TPC-C – Performance in TPC-C for an increasing number of warehouses. The capacity of DRAM, NVM, and SSD is set to 2 GB, 10 GB, and 50 GB, respectively.

YCSB-R/W uses x% update and $(100 - x)$% point lookup operations, where x can be configured between 0 to 100 (configurable mix of "Workload A" and "Workload C").

YCSB-SCAN uses 100 % range scan operations. Each one has a random length between 1 and 100 (like "Workload E", but without the 5% inserts).

TPC-C is considered the industry standard for benchmarking transactional database systems. It is an insert-heavy workload that emulates a wholesale supplier. Like most research TPC-C implementations (e.g., [25, 42]), we do not implement think times.

5.3 Architecture Comparison

In this paper, we set out to design a system that performs well in all three layers (DRAM, NVM, and SSD) of next generation servers. This experiment compares the performance of different storage

engines with two workloads: YCSB-RO (in Figure 8) and TPC-C (in Figure 9). In both scenarios, we use 2 GB of DRAM, 10 GB of NVM and 50 GB of SSD. The horizontal axis increases the workload size by 1 GB at a time starting at 1 GB up to 15 GB.

The figures are divided into three areas by two dashed lines that show the capacity of DRAM and NVM. Therefore, the area on the left shows the performance for workloads fitting completely into DRAM; the one in the middle covers workloads fitting into NVM and the one on the right is for workloads exceeding NVM capacity. In the following, we describe the behavior of the different storage engines in these areas.

DRAM Area: For both workloads the main-memory variant (-⊟-) performs best (YCSB-RO with 3.6 MTx/s and TPC-C with 88 kTx/s). It is clear that it is impossible for any buffer-managed architecture (-▲-, -⊖-, -◆-) to outperform a main memory system in this area. The overheads of fixing and unfixing pages and, in the case of our proposed system, checking the residency of individual cache lines can be minimized but never completely avoided. The basic buffer manager for DRAM and NVM (-⊖-) and the traditional buffer manager for DRAM and SSD (-◆-) have roughly the same throughput as our approach (-▲-) in the DRAM area. The key differences related to performance are pointer swizzling and the cache-line-grained access. Mini pages do not play a role here, as they are rarely used (without the need to swap pages out, every page becomes a full page eventually). Our system needs to check whether a cache line is resident before accessing it. But, as the figure shows, the pointer swizzling speedup is higher than the cache-line-grained access slowdown. If the Basic NVM BM or the traditional one was extended to use pointer swizzling as well, they would come out slightly ahead. The NVM Direct system (-✶-) performs worst in the first area because it does not use the fast DRAM but only the slower NVM.

NVM Area: In the NVM area, the line for the main memory system (-⊟-) vanishes, because such a system cannot support workloads exceeding the size of DRAM. The NVM Direct system (-✶-) is not impacted by the fact that the workload no longer fits into DRAM, as it is not using DRAM. Its performance is decreasing because of the growing workload size and the fewer L3 cache hits. The two buffer management systems suffer a lot, as they have to start swapping pages in and out of SSD. The throughput of Basic NVM BM (-⊖-) and the SSD BM (-◆-) drop below that of NVM Direct because page misses are more likely (due to the lack of mini pages) and each page miss needs to retrieve an entire page (due to the lack of cache-line-grained pages). Our three-tier system (-▲-) also drops in performance, but is still able to outperform the NVM Direct system due to the benefits of caching data in DRAM. The performance decrease is less severe in TPC-C than in YCSB-RO. This can be explained by the fact that the working set (the hot data) in TPC-C is only a portion of the entire data and can therefore be better cached in the buffer-managed systems.

SSD Area: In the last area, the NVM Direct system (-✶-) and the Basic NVM BM (-⊖-) can no longer handle the large amounts of data and therefore no longer show up in the figure. Our system (-▲-) has another performance drop, as it needs to load more and more data from SSD now. In the TPC-C experiment, this drop is delayed and does not occur right after the dashed line, as the hot data still fits into NVM at this point. Only when scaling the data size up to around 90 warehouses does it start accessing SSD and drops in performance. This drop is unavoidable because even a single SSD access (around 1 millisecond) every 26 transactions (number of transaction executed per 1 millisecond at 90 warehouses) will cut the transaction rate in half. On the very right of both figures (15 GB of data), the performance advantage of having NVM is still present. The SSD BM (-◆-) is a factor of 4.3 slower in YCSB-RO and 2.8 in TPC-C.

These results show that a carefully engineered buffer manager can outperform an NVM direct system, be competitive with an in-memory system and greatly speed up workloads exceeding NVM capacity.

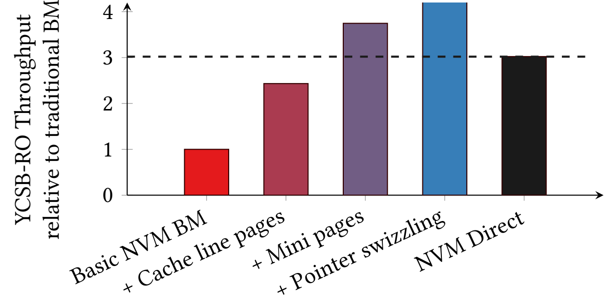


Figure 10: Performance Drill Down – Effect of proposed optimizations relative to a traditional buffer manager on NVM (YCSB-RO with 10 GB of data, read only, 2 GB DRAM, and 10 GB NVM).

5.4 Performance Drill Down

Traditionally, buffer management is viewed as a technique with high overhead [21]. Therefore, working directly on NVM is, initially, a good idea. In this section, we show that by leveraging the novel properties of NVM (mainly byte addressability), it becomes possible to outperform an NVM in-place engine. We first break down the performance gains of the individual techniques proposed in our architecture and then analyze their overheads. In both experiments, we use 2 GB of DRAM, 10 GB of NVM, and around 6.5 million tuples, which amount to roughly 10 GB.

5.4.1 Performance Gains Breakdown. The benefits of the proposed techniques are shown using YCSB-RO in Figure 10. We first measure the performance of our system configured as a Basic NVM BM and then cumulatively enable the optimizations proposed in this paper (cache-line-grained pages, mini pages and pointer swizzling). The performance of the improvements is given relative to that of the Basic NVM BM. For comparison, we also added a line showing the performance of the NVM Direct engine.

As the first improvement, we add *cache-line-grained accesses*, as described in Section 3.1. It allows the buffer manager to load individual cache lines from NVM instead of having to load the entire page, which dramatically reduces the number of loaded cache lines by a factor of 55 from around 652.5 M to 11.8 M.

The next improvement are *mini pages*, which are detailed in Section 3.2. These pages can store less cache lines compared to a full page, but in turn also require less storage. This allows them to use the available DRAM more efficiently because, on many pages,

only a few cache lines are touched and thus more hot tuples are kept in DRAM. Using mini pages, the number of loaded cache lines is reduced by a factor of 2 and ends up at 5.6 M.

The last improvement, introduced in Section 3.3, is *pointer swizzling*. It essentially avoids the costly hash table lookup to map the page identifier to a page and replaces it with a single branch and a pointer chase for hot pages. Thus, lowering the overhead of the buffer manager indirection and making it more competitive with the architectures that do not require this indirection.

Overall, the experiment shows that it is possible to leverage the system's DRAM to increase the throughput by deploying a buffer manager. It also shows that it is necessary to specifically optimize buffer managers for NVM to achieve good performance on these new systems.

5.4.2 Overhead Analysis. On the other side, the proposed optimizations also have overheads associated with them. To show these CPU overheads, we measure YCSB-SCAN using a fill factor of 100%. We start with our base line, the "Basic NVM Buffer Manager" in the first row, and cumulatively add our optimizations showing their throughput relative to the baseline:

	Small Scan (range = 100)	Full Scan (range = |table|)
Basic NVM BM (100%)	50 000 scans/s	0.34 scans/s
+ Cache-Line-Grained	104.2 %	91.3 %
+ Mini Pages	93.1 %	90.8 %
+ Pointer Swizzling	93.8 %	90.9 %

While the base line implementation loads each page completely during the scan, the *cache-line-grained* one loads each tuple individually. Due to the perfect fill rate of leaf pages, almost no loads are avoided when all tuples on a page are needed. But, for small scans, cache-line-grained loading still benefits, as only around 3 pages (50 tuples) are touched and the pages on the edges of the range are not read completely. In the case of the full table scan, each page is loaded completely and the tracking of individual cache lines has no benefit and thus reduces the throughput.

The use of *mini pages* is almost never beneficial for scans because due to the large YCSB tuples, the mini pages are promoted as soon as more than 1 tuple is accessed. This happens frequently in both cases, therefore, the system suffers in throughput.

Lastly, the use of *pointer swizzling* has little effect on the performance of scans, as it is only used for finding the starting point of the scan. The only difference is an additional branch when fixing and unfixing a page during the scan and the increased memory consumption (16 byte of additional data in the buffer frames header).

This experiment shows that the proposed techniques incur a maximum overhead of around 10 %. Note, however, that this overhead can trivially be avoided using a hinting mechanism that selectively disables cache-line-grained and mini pages for full table scans.

5.5 Hybrid Structures

Using only NVM can result in sub-optimal performance due to its fairly high latency. Therefore, some NVM-optimized data structures incorporate DRAM into their design. One recent proposal of a hybrid data structure is the FPTree [39]. It is a B+-Tree that places its leaf nodes in NVM and its inner nodes in DRAM, thus gaining fast lookups (as inner node traversal is fast due to low DRAM latencies) and still being durable (as leaf nodes are in NVM and can be used to reconstruct inner nodes upon a restart).

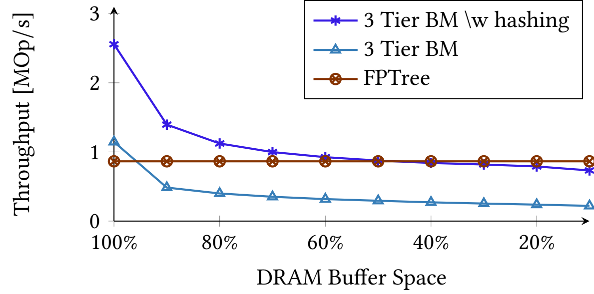


Figure 11: Hybrid DRAM-NVM Systems – Uniformly distributed lookup keys in tree with 100 M 8 byte keys values pairs.

In the experiment shown in Figure 11, we compare our approach with a reimplementation of the FPTree. For a fair comparison, we use the same experimental setup as the original FPTree paper: Uniformly-distributed point lookups in a single tree with 8 byte integer keys and values. As in the paper, the FPTree is configured to use 4096 entries in inner pages (64 kB) and 56 entries in leaf pages (960 B). Our tree uses a page size of 16 kB for all nodes (around 1000 entires). We use 100 M tuples, resulting in a tree size of roughly 2.5 GB in both systems. The horizontal axis depicts the percentage of pages that fit into DRAM for the buffer-managed systems.

The results show that our system (–▲–) can only outperform the FPTree (–⊗–) when 100 % of the data fits into the DRAM cache. This is caused by a different leaf node layout in the trees: While our leaves use a sorted array of keys and binary search, the FPTree uses a hash-based leaf node layout ("fingerprints"). For point lookups, this layout allows the FPTree to reduce the number of NVM accesses in leaf nodes (from around 8 down to 2).

However, our proposed three-tier architecture is agnostic to the leaf node design. Therefore, we can optimize our leaves for point lookups by implementing a hashing structure (based on open addressing). The resulting system (–✱–) remains competitive with the FPTree, even with lower DRAM cache sizes. While a hashing layout performs well for point lookups, it introduces overheads for scans (the nodes need to be sorted just in time) and lower bound queries (nodes need to be scanned completely).

The most important advantage of a buffer-managed approach regarding performance is that it is able to adapt to skewed workloads. In this experiment (Figure 11), we measured a uniformly-distributed access pattern, which is the worst case for caching. When using a Zipf distribution ($z = 1$), the buffer-managed system is able to cache larger portions of the data with limited DRAM. This achieves a 30 % higher throughput at 50 % DRAM and only slightly drops beneath the FPTree line at 10 % DRAM.

5.6 Impact of NVM Characteristics

As of today, NVM is not commercially available as a byte addressable storage device. Therefore, there is still uncertainty about the

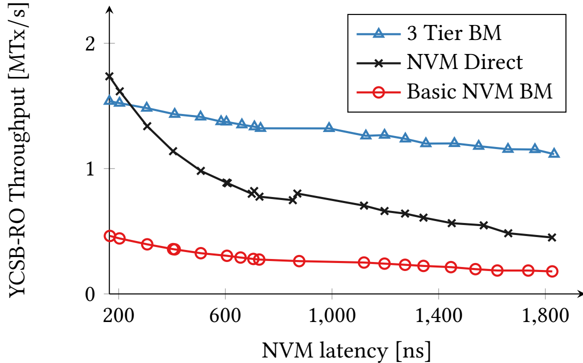


Figure 12: NVM Latency – The impact of varying NVM latencies on the YCSB-RO performance (YCSB with 10 GB of data, read only, 2 GB DRAM, and 10 GB NVM).

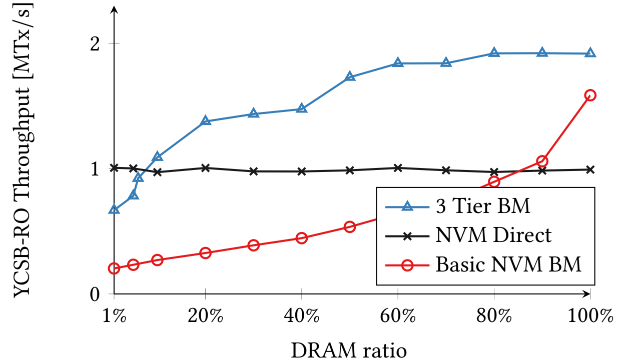


Figure 13: DRAM Buffer Size – YCSB-RO performance for varying amounts of DRAM and a fixed NVM capacity (YCSB with 10 GB of data, read only and 10 GB NVM).

exact characteristics of the hardware. In this section, we evaluate how the NVM latency and capacity impact the individual engines.

5.6.1 NMV Latency. We first investigate how the three systems perform under various NVM latencies. Our test platform allows us to vary the latency between 165 ns to 1800 ns. We use the YCSB-RO workload to determine in which areas the benefit of a buffer manager outweigh the incurred overheads. The data size is 10 GB, the NVM capacity is 10 GB, and the DRAM capacity is 2 GB. The results are shown in Figure 12. The vertical axis shows the throughput, while the horizontal one depicts the various NVM latencies.

At the lowest possible latency (around 165 ns), the NVM Direct system (-×-) is slightly faster than our NVM optimized buffer manager (-△-). At this point, the DRAM latency advantage is too marginal for a buffer manager to be beneficial. With increasing NVM latency, all systems become slower, but the buffer-managed ones are not as much impacted by the latency increase because they use the constantly fast DRAM as a cache. The NVM Direct system, on the other hand, loads all its data from the increasingly slow NVM. Starting at a latency of around 300 ns, the buffer manager outperforms the NVM Direct system.

The Basic NVM BM (-○-) also decreases in performance with an increasing NVM latency. The slope of the curve suggests that for even higher latencies, this non-optimized approach would also outperform the NVM Direct system. This is because the fast DRAM cache becomes more valuable as the latency difference between NVM and DRAM increases.

5.6.2 NVM Capacity. Besides latency, the exact capacity of NVM is another unknown parameter. In this experiment, we look at the ratio between the capacities of DRAM and NVM. The results are shown in Figure 13. The NVM capacity is fixed at 10 GB. The DRAM capacity is increased from 100 MB up to 10 GB (horizontal axis). Consequently, a value of 20 % implies a DRAM capacity of 2 GB.

The NVM Direct engine (-×-) is not impacted by the changing ratio between NVM and DRAM, as it is not using DRAM. The other two engines benefit from more DRAM. The Basic NVM BM (-○-), however, requires a lot more DRAM (around 80 % of NVM) to outperform the NVM Direct engine. The 3 Tier BM (-△-), on the other hand, outperforms the NVM Direct engine very early on (around 7 %). In addition, it is worth noting that starting at around 85 %, the 3 Tier BM does not require any NVM accesses anymore due to the mini pages. Thus, from this point on, the performance remains constant.

6 RELATED WORK

NVM will likely trigger a drastic redesign of existing database systems. In this section, we first analyze the state of the art for integrating NVM into database systems at the storage layer before discussing other aspects.

The SOFORT database engine [36, 40] proposes a copy-on-write architecture for NVM. All primary data is placed and modified directly in NVM, thus eliminating the need to load it into main memory after a restart. This way, SOFORT is able to achieve an almost instantaneous restart and can resume working at a pre-shutdown throughput instantly. The placement of secondary data, like indexes, can be considered a tuning parameter: Placing secondary data in main memory allows the user to increase system performance due to the lower latencies, but it also increases the restart time, as the data needs to be reconstructed.

For systems that modify primary data directly in NVM, one interesting question is the endurance of this storage technology. NVM has a limited endurance [35], i.e., a given NVM cell will fail after a certain number of write operations. Therefore, Arulraj et al. [4] compare three generic approaches for data management on an NVM-only architecture: in-place updates, copy-on-write and log-structured (LSM) system. Their experiments suggest that an in-place update architecture is usually best, as it delivers good throughput while minimizing the wear on the NVM hardware. This is not a coincidence, as these two goals (maximizing endurance and throughput) are in support of each other: By minimizing the number of accesses to NVM, one also improves performance, due to the, compared to DRAM, relatively high NVM latency.

FOEDUS [25] tries to take advantage of this finding: Modifying data directly on NVM has the advantage of fast restart times, but it suffers from a higher access latency, leaves the available DRAM unused, and wears out the NVM. FOEDUS therefore uses a

two-layered approach: data can reside either on NVM or DRAM. The idea is similar to classic disk-based systems: The memory is divided into fixed-size pages, which are loaded from NVM into a DRAM-resident buffer pool for read and write operations. In order to update the persistent state on NVM, FOEDUS periodically runs an asynchronous process that updates the NVM-resident snapshot using the databases log.

The SAP HANA in-memory database system integrates NVM by utilizing its "delta" and "main" storage separation [2]. The immutable and compressed bulk of the data ("main") is stored on NVM, while the updatable part ("delta"), which contains recent changes, remains in main memory. This simple approach nicely fits HANA's architecture, but is not applicable to most database systems. Another specific way of exploiting NVM is to use it as a cache for LSM-based storage [30].

Microsoft Siberia [15] is an approach for extending the capacity of main-memory database systems. By logging tuple accesses [31], infrequently-accessed tuples are identified and eventually migrated to "cold" storage [15] (e.g., SSD). This high-level concept could also be used to add support for NVM-based cold storage for main-memory systems.

There are also proposals to integrate persistency into NVM-resident data structures. However, performing in-place updates on NVM requires customized data structures to avoid data corruption [9]. Multiple NVM-specialized data structures, including CDDS Tree [43], HiKv [46], NV-Tree [47], FPTree [39], WORT [26], wBTree [10], and BzTree [3], have been proposed. These data structures optimize the node layout for NVM and explicitly manage persistency using appropriate cache write back instructions. In our design, in contrast, the data structure design is largely transparent to the storage engine (except for the requirement of fixed-size pages). Furthermore, since the storage engine takes care of persistency, write back instructions are inserted automatically.

In contrast to the approaches discussed above, we propose transparently integrating NVM into the memory hierarchy. While some systems use NVM mostly as a means to achieve durability or to extend the main memory capacity, in our approach, NVM is an integral part: We leverage not only the persistency but also the byte addressability by loading individual cache lines from NVM into DRAM. This way, we can deploy variable-size pages, which allow us to keep hot tuples in DRAM instead of hot pages.

Our three-layer architecture unites DRAM, NVM, and SSD into one transparent memory, thus enabling workloads that far exceed the capabilities of main-memory databases. We are, to the best of our knowledge, the first to study memory management in a database context for DRAM, NVM, and SSDs. Three-layer architectures have already been investigated [8, 12, 23, 32, 33] for different storage layers (namely: DRAM, SSD, HDD). However, the vastly different properties of NVM (low latency and byte addressability) compared to traditional durable storage technologies (SSD and HDD) requires a drastically different architecture. For instance, when loading data from SSD, it has to be done in a page-grained fashion. This is neither required nor the most efficient way of dealing with NVM.

While this paper focuses on storage, NVM also poses challenges and opportunities for other aspects, for example, for testing [37], memory allocation [38], and query processing [44]. Another component affected by NVM is logging/recovery [5, 16, 22, 36, 45]. Write-behind logging [5], for example, is a recovery protocol specifically designed for multi-version databases that use NVM as primary storage. On commit, all newly-created changes of a transaction (versions) are persisted—instead of only persisting the traditional write-ahead log. While our current implementation uses write-ahead-logging and single-version storage, it would also be possible to combine our storage engine with write-behind logging. We defer evaluating different logging and recovery schemes to future work.

The pmem.io library [1] is the de facto standard for managing NVM. It offers various abstraction levels, from low-level synchronization utilities (libpmem) to full-fledged transactional support (libpmemobj). Like all NVM-optimized database systems, we use the low-level primitives in order to have full control over persistency.

7 CONCLUSIONS

NVM will have a major impact on current hardware and software systems. We evaluated three approaches for integrating NVM into the storage layer of a database system: One that works directly on NVM, a FOEDUS-style buffer manager based on fixed-size pages, and our novel cache-line optimized storage engine. We found that by taking the byte addressability into account, it becomes possible to outperform the other two approaches while supporting large data sets on SSD as well.

This result is achieved using a number of techniques. While a traditional buffer manager loads entire pages, we use NVM's byte-addressability to load individual cache lines instead, reducing the transferred memory between DRAM and NVM enormously. Enabled by the cache-line-grained pages, we introduce mini pages, which store only a few cache lines but also use less memory. These pages use the limited DRAM capacity more efficiently. We also optimized our buffer manager for in-memory and in-non-volatile-memory workloads by using pointer swizzling. This enables us to be competitive with in-memory DBMSs and systems working directly on NVM. In summary, we ended up with a system that achieved a performance close to that of main-memory database systems if the workload fits into DRAM. At the same time, our system performs better than NVM-only systems if the workload fits into NVM and similar to disk-based performance for even larger workloads.

In isolation, each of these techniques is either well-known or fairly simple. The novelty comes from combining these ideas into a coherent and effective system design. We argue that conceptual simplicity is a major advantage, or in the words of Jim Gray:

> *"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."* [19]

ACKNOWLEDGMENTS

We would like to thank Dieter Kasper and Andreas Blümle for helping with the SEP system.

A APPENDIX

A.1 Multi Threading

In this paper, we compared 5 radically different storage system designs. To keep the implementation effort reasonable and the comparison fair, all implementations and experiments are single-threaded. However, given current hardware trends, any modern storage engine should efficiently support multi-threading. The FOEDUS [25] and LeanStore [27] projects have shown that page-based storage engines can efficiently be synchronized for modern multi-core CPUs. The two key techniques for achieving this are version-based latches that allow readers to proceed optimistically without physically acquiring latches [29], and epoch-based memory reclamation [42]. Another alternative for implementing synchronization is hardware transactional memory, which has been shown to be effective at synchronizing B-trees [24, 28, 39]. In the following, we discuss some additional synchronization aspects of our design. We assume the use of per-page, version-based latches (and do not rely on hardware transactional memory).

Cache-line-grained accesses may cause a read to physically change a page if the requested page is not yet resident. Therefore, such "cache line faults" make it necessary to upgrade the page latch to exclusive mode. Note that this only affects a single (leaf) page and is therefore unlikely to cause contention. Furthermore, frequently-accessed (hot) pages will generally be fully resident and will therefore not cause latch upgrades for read accesses. Another aspect that requires some care is the promotion of mini pages to a full pages, which becomes necessary once more than 16 cache lines have been accessed. Promotion is done by exclusively latching the mini page (source) and the full page (destination) before copying the data.

In traditional (textbook-style) buffer managers, the mapping table itself often becomes a synchronization bottleneck because all page accesses have to touch this data structure. In our design, in contrast, all frequently accessed pages will be swizzled. Accessing a hot page therefore does not require accessing the mapping table. Finally, regarding the replacement strategies, to achieve good scalability, decentralized algorithms like Second Chance should be preferred over centralized ones like LRU.

A.2 Scaling the Data Size

In this section, we evaluate the performance of the proposed system under larger workloads. The results of this experiment are shown in Figure 14. The vertical axis shows the throughput in million transactions per second. On the horizontal axis, we scale up the number of tuples and the capacity of the database proportionally. The axis shows the data size, which is the required space for the tuples once loaded into a B-tree. The DRAM capacity is set to a fifth of that of NVM.

Starting at the bottom, the Basic NVM BM (-o-) is dominated by the cost of reading entire pages from NVM. Therefore, the throughput is barely impacted by the increasing number of tuples or the larger control structures in the database.

The NVM Direct system (-×-) and the 3 Tier BM (-△-) both drop in performance as the workload size is increased. But this decrease is a lot more severe for the NVM Direct system (almost a factor of two). This can be explained by the decreasing ratio of hot tuples fitting into the L3 cache. Therefore, it is important to measure larger workloads when working with NVM, as the L3 cache can speed up small ones and hide the difference between DRAM and NVM.

Finally, there is a bump in the performance of the 3 Tier BM. Before it starts to drop at around 30 GB, the performance increases. This is an artifact of the Zipf distribution generator we used. The ratio of accessed pages that fit into the buffer pool increases up to 30 GB.

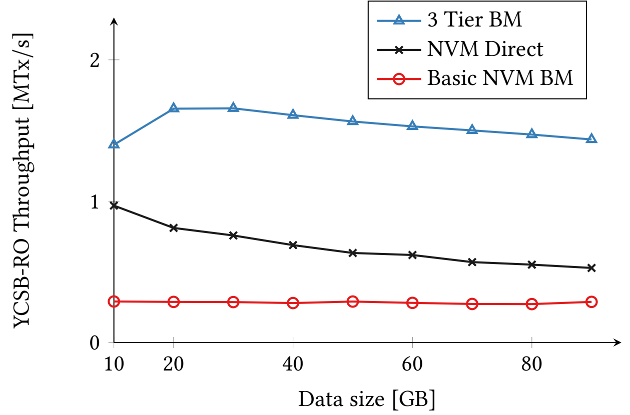


Figure 14: Large Workloads – YCSB-RO point lookup performance for large workload sizes. The NVM capacity is configured to match the data size, and the DRAM capacity is set to a fifth of that of NVM.

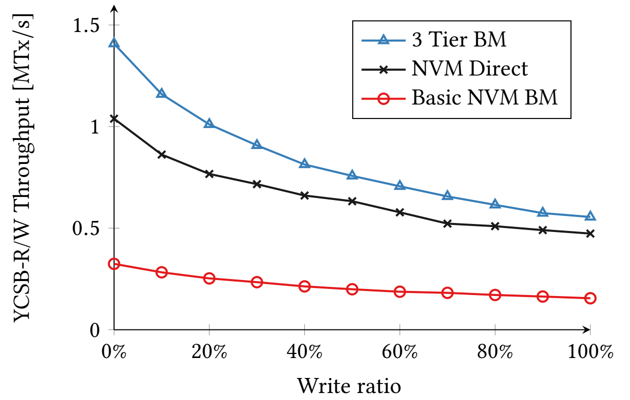


Figure 15: Update Performance – YCSB-R/W performance with an increasing amount of write transactions (YCSB with 10 GB of data, 2 GB DRAM, and 10 GB NVM).

A.3 Updates

We now analyze the impact of writes. To do this, we set up an experiment where we run YCSB with an increasing amount of update transactions. The results are shown in Figure 15. The horizontal axis shows the percentage of update transactions, while the vertical one depicts the throughput. We, again, measured the three competing systems: Our NVM-optimized buffer manager (-△-), the system working directly on NVM (-×-) and a basic buffer manager

for NVM (-⊖-). The systems are configured to use 2 GB of DRAM and 10 GB of NVM. The size of the workload is 10 GB, just fitting into NVM.

With an increasing amount of write transactions, all systems degrade in performance, because more log entries and more tuple data needs to be written back to NVM. Compared to the read-only setting, the throughput of the Basic NVM BM is only half as high in the write-only case. The other two systems, NVM Direct and our NVM-optimized buffer manager, still outperform the Basic NVM BM in every setting, but also drop by a similar factor in performance. The experiment shows that our system is as stable as the other systems under a write-heavy workload. In addition, consider that the figure shows a rather unfavorable configuration. Our system would benefit from a change of the workload size in either direction: On the one hand, with a smaller workload, the ratio of tuples fitting into DRAM becomes higher and therefore, the buffer manager faster. On the other hand, with a larger workload, the NVM direct system could not run at all because it would not fit into NVM any more.

A.4 Endurance and Wear

In the previous experiments, our system has demonstrated a large performance benefit for workloads fitting in DRAM and has allowed for workloads larger than the capacity of NVM. We also showed that with write-heavy workloads, the performance remains competitive. In this section, we show another important advantage of using the buffer manager instead of working directly on NVM.

NVM has a limited endurance and therefore wears out and eventually fails. By using a buffer manager, the life-time of NVM can be greatly increased. To back up this hypothesis, we added counters that measure the number of writes to each individual NVM cache line. The results are shown in Figure 16.

The vertical axis shows the number of writes for each cache line. The cache lines are ordered by the number of writes and displayed on the horizontal axis. Both axes are logarithmic to better visualize the data. In the experiment, we compare a write only run of YCSB with 10 million transactions. As in the previous experiments, the data size is 10 GB, the NVM capacity 10 GB and the capacity of DRAM 2 GB.

The figure shows two advantages of our proposed system (-▲-) compared to the NVM Direct system (-×-). First, the total number of writes to NVM is reduced down to 4.7 M from 25 M. Second, and even more importantly, these write operations are spread out a lot more evenly. While the NVM Direct system has several cache lines that are written to 60 K times, the cache lines written to most with the buffer-managed approach are written to 3 times. The reason for this can easily be explained: The buffer manager caches pages that are frequently accessed in DRAM to increase performance. As a nice side effect, this also prevents many writes to these cache lines in NVM.

A.5 Restart Time

One major advantage of NVM is fast recovery and restart time. In our implementation, a write ahead log (WAL) is written to NVM. The creation of a WAL has two advantages: First, high availability is extremely important in a production environment and usually

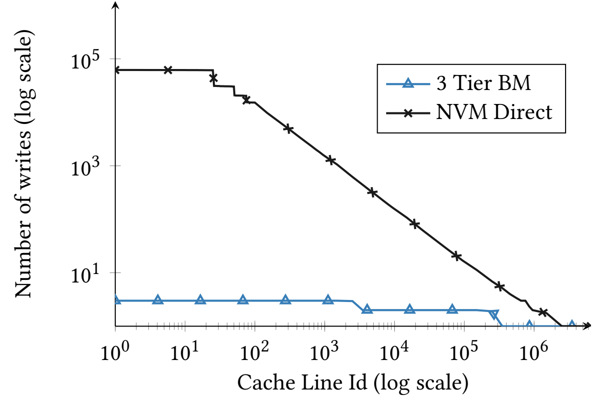


Figure 16: NVM Wear – The sorted number of writes for each cache line (YCSB-R/W ($x = 100$) with 10 GB of data, 2 GB DRAM, and 10 GB NVM).

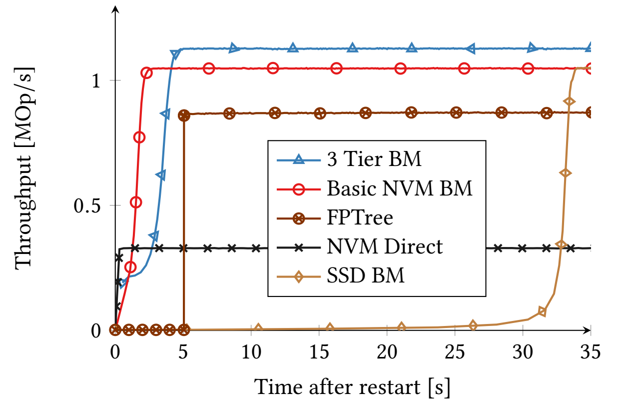


Figure 17: Restart Time – Ramp-up phase for uniformly distributed lookups with 100 M 8 byte key/8 byte value pairs. The entire workload fits into the buffer pool.

achieved with hot standbys in the event the primary system goes down. To keep the standby system up to date, it is necessary to supply it with a stream of changes from the primary. Second, (in comparison with hybrid and NVM-only systems) a write ahead log bundles many small writes into one large sequential write at the end of the transaction. This is beneficial considering the limited endurance and asymmetric write latency of NVM.

In the experiment shown in Figure 17, we compare the throughput immediately after a clean restart (until peak throughput is reached). In the "NVM direct" system (-×-), the durable storage and the working memory are the same (both NVM). Therefore, these systems achieve a very fast, almost instantaneous, restart, because nothing has to be loaded explicitly (except for warming the CPU caches). FPTree (-⊗-), in contrast, must reconstruct its inner pages by scanning all leaf nodes, which takes about 5 seconds in our experiment. Once the reconstruction is completed, FPTree reaches pre-restart throughput almost instantaneously. Traditional buffer managed systems (-◇-) can begin processing transactions immediately after a restart, but they suffer in performance due to

a cold buffer cache and high SSD latencies. The basic NVM buffer manager (-⊖-) is similar, but recovers much faster, as it can fill its buffer cache from NVM instead of SSD. Our three-tier architecture (-△-) needs to perform the reconstruction of the combined page table (cf. Section 4.3), which is quite fast, however (around 200 ms). It reaches peak performance slightly more slowly than the basic NVM BM due to mini pages, which only get lazily promoted.

A.6 Debugging Cache-Line-Grained Access

To easily detect usage failures in mini pages, we developed a debugging mode that checks both reads and writes. For reads, all cache lines within the page are marked as uninitialized memory when it is fixed. A memory checking tool (e.g., valgrind) can then be used to detect invalid reads. To detect invalid writes, the page initializes all cache lines to a specific ("magic") sequence. When it is unfixed, it validates that only those cache lines that are marked as dirty have been changed (false positives are possible but very unlikely).

REFERENCES

[1] Persistent memory programming. http://www.pmem.io. Accessed: 2018-02-13.
[2] M. Andrei, C. Lemke, G. Radestock, R. Schulze, C. Thiel, R. Blanco, A. Meghlan, M. Sharique, S. Seifert, S. Vishnoi, D. Booss, T. Peh, I. Schreter, W. Thesing, M. Wagle, and T. Willhalm. SAP HANA adoption of non-volatile memory. *PVLDB*, 10(12):1754–1765, 2017.
[3] J. Arulraj, J. J. Levandoski, U. F. Minhas, and P. Larson. Bztree: A high-performance latch-free range index for non-volatile memory. *PVLDB*, 11(5):553–565, 2018.
[4] J. Arulraj, A. Pavlo, and S. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *SIGMOD*, pages 707–722, 2015.
[5] J. Arulraj, M. Perron, and A. Pavlo. Write-behind logging. *PVLDB*, 10(4):337–348, 2016.
[6] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *SIGFIDET*, pages 107–141, 1970.
[7] P. Bonnet. What's up with the storage hierarchy? In *CIDR*, 2017.
[8] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang. SSD bufferpool extensions for database systems. *PVLDB*, 3(2):1435–1446, 2010.
[9] A. Chatzistergiou, M. Cintra, and S. Viglas. REWIND: recovery write-ahead system for in-memory non-volatile data-structures. *PVLDB*, 8(5):497–508, 2015.
[10] S. Chen and Q. Jin. Persistent B+-trees in non-volatile main memory. *PVLDB*, 8(7):786–797, 2015.
[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154, 2010.
[12] J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson. Turbocharging DBMS buffer pool using SSDs. In *SIGMOD*, pages 1113–1124, 2011.
[13] S. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan. Data tiering in heterogeneous memory systems. In *EuroSys*, 2016.
[14] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *EuroSys*, 2014.
[15] A. Eldawy, J. J. Levandoski, and P. Larson. Trekking through siberia: Managing cold data in a memory-optimized database. *PVLDB*, 7(11):931–942, 2014.
[16] R. Fang, H. Hsiao, B. He, C. Mohan, and Y. Wang. High performance database logging using storage class memory. In *ICDE*, pages 1221–1231, 2011.
[17] G. Graefe, H. Volos, H. Kimura, H. A. Kuno, J. Tucek, M. Lillibridge, and A. C. Veitch. In-memory performance for Big Data. *PVLDB*, 8(1):37–48, 2014.
[18] J. Gray and G. R. Putzolu. The 5 minute rule for trading memory for disk accesses and the 10 byte rule for trading memory for CPU time. In *SIGMOD*, pages 395–398, 1987.
[19] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.
[20] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *SIGMOD*, pages 243–252, 1994.
[21] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, pages 981–992, 2008.
[22] J. Huang, K. Schwan, and M. K. Qureshi. NVRAM-aware logging in transaction systems. *PVLDB*, 8(4):389–400, 2014.
[23] W. Kang, S. Lee, and B. Moon. Flash as cache extension for online transactional workloads. *VLDB Journal*, 25(5):673–694, 2016.
[24] T. Karnagel, R. Dementiev, R. Rajwar, K. Lai, T. Legler, B. Schlegel, and W. Lehner. Improving in-memory database index performance with intel transactional synchronization extensions. In *HPCA*, 2014.
[25] H. Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *SIGMOD*, pages 691–706, 2015.
[26] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh. WORT: write optimal radix tree for persistent memory storage systems. In *FAST*, pages 257–270, 2017.
[27] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann. LeanStore: In-memory data management beyond main memory. In *ICDE*, 2018.
[28] V. Leis, A. Kemper, and T. Neumann. Scaling HTM-supported database transactions to many cores. *IEEE Trans. Knowl. Data Eng.*, 28(2):297–310, 2016.
[29] V. Leis, F. Scheibner, A. Kemper, and T. Neumann. The ART of practical synchronization. In *DaMoN*, 2016.
[30] L. Lersch, I. Oukid, W. Lehner, and I. Schreter. An analysis of LSM caching in NVRAM. In *DaMoN*, 2017.
[31] J. J. Levandoski, P. Larson, and R. Stoica. Identifying hot and cold data in main-memory databases. In *ICDE*, pages 26–37, 2013.
[32] X. Liu and K. Salem. Hybrid storage management for database systems. *PVLDB*, 6(8):541–552, 2013.
[33] T. Luo, R. Lee, M. P. Mesnier, F. Chen, and X. Zhang. hStorage-DB: Heterogeneity-aware data management to exploit the full capability of hybrid storage systems. *PVLDB*, 5(10):1076–1087, 2012.
[34] N. Megiddo and D. S. Modha. ARC: a self-tuning, low overhead replacement cache. In *FAST*, 2003.
[35] S. Mittal and J. S. Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Trans. Parallel Distrib. Syst.*, 27(5):1537–1550, 2016.
[36] I. Oukid, D. Booss, W. Lehner, P. Bumbulis, and T. Willhalm. SOFORT: a hybrid SCM-DRAM storage engine for fast data recovery. In *DaMoN*, 2014.
[37] I. Oukid, D. Booss, A. Lespinasse, and W. Lehner. On testing persistent-memory-based software. In *DaMoN*, 2016.
[38] I. Oukid, D. Booss, A. Lespinasse, W. Lehner, T. Willhalm, and G. Gomes. Memory management techniques for large-scale persistent-main-memory systems. In *VLDB*, 2017.
[39] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *SIGMOD*, pages 371–386, 2016.
[40] I. Oukid, W. Lehner, T. Kissinger, T. Willhalm, and P. Bumbulis. Instant recovery for main memory databases. In *CIDR*, 2015.
[41] M. Stonebraker. How hardware drives the shape of databases to come. https://www.nextplatform.com/2017/08/15/hardware-drives-shape-databases-come/. Accessed: 2017-11-02.
[42] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, 2013.
[43] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, pages 61–75, 2011.
[44] S. Viglas. Write-limited sorts and joins for persistent memory. *PVLDB*, 7(5):413–424, 2014.
[45] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *PVLDB*, 7(10):865–876, 2014.
[46] F. Xia, D. Jiang, J. Xiong, and N. Sun. Hikv: A hybrid index key-value store for DRAM-NVM memory systems. In *USENIX ATC*, pages 349–362, 2017.
[47] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *FAST*, pages 167–181, 2015.

Chapter 3

Paper 2: Building Blocks for Persistent Memory

This paper investigates the characteristics of Intel's Optane DC Persistent Memory Modules and proposes new designs for efficient storage primitives. The paper is an extended version of a 2019 DaMoN short paper [vRVL+19].

The first part of the paper is comprised of a number of experiments to measure the basic performance characteristics of persistent memory: For instance, compared to DRAM, PMem's read latency is 3.3x higher and a persistent write takes around 100 ns in an otherwise idl system. Further, the multi-threaded read bandwidth for 6 memory modules is with 39.1 $\mathrm{GB\,s^{-1}}$ around 2.9x lower (113.8 $\mathrm{GB\,s^{-1}}$ for DRAM) and the write bandwidth is with 12.5 $\mathrm{GB\,s^{-1}}$ around 7.5x lower (92.5 $\mathrm{GB\,s^{-1}}$ for DRAM). Lastly, the paper unveils an interesting artifact: Unlike the 64 B write granularity on DRAM, PMem has a 256 B one.

The second part of the paper proposes a set of algorithms and programming techniques for efficiently manipulating data on PMem:

Page Propagation: I compare two algorithms for writing larger chunks of memory (4 kB) at a time in a failure atomic way. The first one is based on copy-on-write using out-of-place updates and the second one does in-place-updates using a write ahead log to ensure consistency. When tracking the dirty cache lines on a page, I found that the second algorithm can be beneficial for up to 28 dirty cache lines.

Logging: I propose *PopLog*, a logging algorithm that minimizes persistency barriers and compare it with two well-known copy-on-write-based techniques for SSDs and the RAWL algorithm [VTS11]. The experiment showed that for these small writes, the performance dominating factor is the number of persistency barriers.

In-place Updates: Similar to Köppen et al. [KTB+19], I propose a technique to update data on persistent memory in-place with a single persistency barrier. The idea is to keep two versions of the data in a 8 B block (atomic update) and a flag that indicates which one is valid. This approach can be very beneficial for write heavy portions of a software system such as the meta data in a persistent memory allocator.

Asynchronous Writing: Whereas on DRAM writes slowly trickle through the cache hierarchy, on PMem writes become synchronous operations, due to the persistency barriers. I propose to use user space preemptive scheduling of multiple logical threads of operation to avoid these stalls. In a sample implementation, based on coroutines, I achieve speedups of up to 6.2 times over regular execution.

Contributions: The implementation and evaluation of all algorithms as well as the writing of the paper itself was done by the first author. Co-authors helped with the algorithm design, prove reading, and identifying related work.

The VLDB Journal (2020) 29:1223–1241
https://doi.org/10.1007/s00778-020-00622-9

SPECIAL ISSUE PAPER

Building blocks for persistent memory

How to get the most out of your new memory?

Alexander van Renen[1] · Lukas Vogel[1] · Viktor Leis[2] · Thomas Neumann[1] · Alfons Kemper[1]

Received: 29 January 2020 / Revised: 6 July 2020 / Accepted: 15 July 2020 / Published online: 23 September 2020
© The Author(s) 2020

Abstract
I/O latency and throughput are two of the major performance bottlenecks for disk-based database systems. Persistent memory (PMem) technologies, like Intel's Optane DC persistent memory modules, promise to bridge the gap between NAND-based flash (SSD) and DRAM, and thus eliminate the I/O bottleneck. In this paper, we provide the first comprehensive performance evaluation of PMem on real hardware in terms of bandwidth and latency. Based on the results, we develop guidelines for efficient PMem usage and four optimized low-level building blocks for PMem applications: log writing, block flushing, in-place updates, and coroutines for write latency hiding.

Keywords Persistent memory · Systems · Databases

1 Introduction

Today, data management systems mainly rely on solid-state drives (NAND flash) or magnetic disks to store data. These storage technologies offer persistence and large capacities at low cost. However, due to the high access latencies, most systems also use volatile main memory in the form of DRAM as a cache. This yields the traditional two-layered architecture, as DRAM cannot solely be used due to its volatility, high cost, and limited capacity.

Novel storage technologies, such as Phase Change Memory, are shrinking this fundamental gap between memory and storage. Specifically, Intel's *Optane DC Persistent Memory Modules* (Optane DC PMM) offer an amalgamation of the best properties of memory and storage—though as we show in this paper, with some trade-offs. This Persistent Memory (PMem) is durable, like storage, and directly addressable by the CPU, like memory. The price, capacity, and latency lies between DRAM and flash.

PMem promises to greatly improve the latency of storage technologies, which in turn would greatly increase the performance of data management systems. However, because PMem is fundamentally different from existing, well-known technologies, it also has different performance characteristics to DRAM and flash. While we perform our evaluation in a database context, these introduced techniques are transferable to other systems, as evidenced by the fact that they are also implemented by the Persistent Memory Development Kit (PMDK) [43]. This paper is an extended version of an originally released paper at DaMoN 2019 [47]. While the experiments in the original paper were conducted on a prototype of Intel's PMem hardware, all numbers in this paper are measured on commercially available PMem hardware. Our contributions can be summarized as follows[1]:

– We provide the first analyses of actual (not prototyped or simulated) PMem based on Intel's Optane DC Persistent Memory Modules (PMM). We highlight the impact of

Alexander van Renen
renen@in.tum.de

Lukas Vogel
vogell@in.tum.de

Viktor Leis
viktor.leis@uni-jena.de

Thomas Neumann
neumann@in.tum.de

Alfons Kemper
kemper@in.tum.de

[1] Technical University of Munich, Munich, Germany

[2] Friedrich Schiller University Jena, Jena, Germany

[1] Source code: github.com/alexandervanrenen/pmembench.

Table 1 Server Platform—configuration of the used PMem server

CPU	Intel Xeon Gold 6212U
Frequency	2.40 GHz (3.90 GHz)
# Cores	24
L1 I+D Cache (per core)	64 kB
L2 Cache (per core)	1 MB
L3 Cache	35.8 MB
# AVX-512 Units	2
CPU Supported Memory	1 TB (DRAM + PMem)
DRAM	192 GB (6 × 32 GB)
PMem	768 GB (6 × 128 GB)

the physical properties of PMem on software and derive guidelines for efficient usage of PMem (Sect. 2).

- We investigate different algorithms for persisting large data chunks (database pages) in a failure atomic fashion to PMem. By combining a copy-on-write method with temporary delta files, we achieve significant speedups (Sect. 3.2).
- We introduce an algorithm for persisting small data chunks (transactional log entries) that reduces the latency by 2× compared to state-of-the-art algorithms (Sect 3.3).
- We introduce a new abstraction on top of PMem, called Failure-Atomic Memory (FAM) that allows for fast in-place updates on PMem (Sect. 3.4).
- We show how synchronous persistent writes to PMem can be interleaved using fibers to avoid stalling on the relatively high PMem latencies (Sect. 3.5).

2 PMem characteristics

In this section, we first describe the evaluation platform and how a PMem system can be configured. Next, we show experimental results for latency and bandwidth. Lastly, we evaluate the interference of concurrently running PMem and DRAM processes in a database-like workload. To provide a better overview, we summarized all important characteristics of our evaluation platform, which is used for all experiments conducted throughout this paper, in Table 1.

2.1 Setup and configuration

There are two ways of using PMem: memory mode and app direct mode. In *memory mode*, PMem replaces DRAM as the (volatile) main memory, and DRAM serves as an additional hardware managed caching layer ("L4 cache"). The advantage of this mode is that it works transparently for legacy software and thus offers a simple way of extending the main memory capacity at low cost. However, this does not utilize persistence, and performance may degrade due to the lower bandwidth and higher latency of PMem. In fact, as we show later, there is a $\approx$ 10% overhead for accessing data when DRAM acts as a L4 cache instead of normally.

Because it is not possible to leverage the persistency of PMem in memory mode, we focus on *app direct mode* in the remainder of this paper. App direct mode, unlike memory mode, leaves the regular memory system untouched. It optionally allows programs to make use of PMem in the form of memory mapped files. We describe this process from a developer point of view in the following:

We are using a two-socket system with 24 physical (48 virtual) cores on each node. The machine is running Fedora with a Linux kernel version 4.15.6. Each socket has 6 PMem DIMMs with 128 GB each and 6 DRAM DIMMs with 32 GB each.

To access PMem, the physical PMem DIMMs first have to be grouped into so-called *regions* with `ipmctl`[2]:

```
ipmctl create -f -goal -socket 0
    MemoryMode=0 \
PersistentMemoryType=AppDirect
```

To avoid complicating the following experiments with a discussion on NUMA effects (which are similar to the ones on DRAM) we run all our experiments on `socket 0`. Once a region is created, `ndctl`[3] is used to create a namespace on top of it:

```
ndctl create-namespace --mode fsdax
    --region 28
```

Next, we create a file system on top of this namespace (`mkfs.ext4`[4]) and mount it (`mount`[5]) using the `dax` flag, which enables direct cache-line-grained access to the device by the CPU:

```
mkfs.ext4 /dev/pmem28
mount -o dax /dev/pmem28 /mnt/pmem28/
```

Programs can now create files on the newly mounted device and map them into their address space using the `mmap`[6] system call:

```
fd = open(("/mnt/pmem28/file", O_RDWR, 0);
res = ftruncate(fd, SIZE);
ptr = mmap(nullptr, SIZE, PROT_WRITE,
    MAP_SHARED, fd, 0);
```

The pointer can be used to access the PMem directly, just like regular memory. Section 3 discusses how to ensure that a value written to PMem is actually persistent. In the remainder of this section, we discuss the bandwidth and latency of PMem.

[2] `ipmctl`: github.com/intel/ipmctl.

[3] `ndctl`: github.com/pmem/ndctl.

[4] `mkfs.ext4`: linux.die.net/man/8/mkfs.ext4.

[5] `mount`: linux.die.net/man/8/mount.

[6] `mmap`: man7.org/linux/man-pages/man2/mmap.2.html.

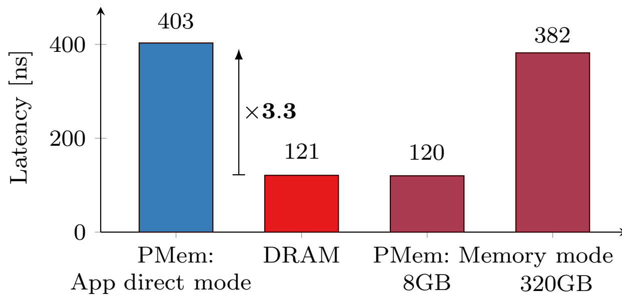


Fig. 1 Read latency—random access read latency

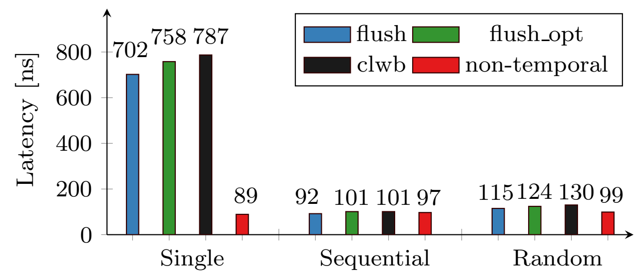


Fig. 2 Persistent write latency—access latency for writing cache lines persistently

2.2 Latency

In this first set of experiment, we want to investigate the latency of PMem. While bandwidth (discussed in the next section) is critical for OLAP-style applications, latency is much more important for OLTP-style workloads because the access pattern shifts from large scan operations (sequential I/O) to point lookups (random I/O), which are usually latency bound. OLTP-style applications are not limited to database systems, but extends to any data intense application, where the performance depends on random I/O.

In the experiments, we compare the latency of PMem to that of DRAM, as both can be used via direct load and store instructions by the CPU. With PMem being persistent, it can also act as a replacement for traditional storage devices such as SSDs and HDDs, which are orders of magnitude slower. Therefore, while we focus the discussion in this section on the comparison between PMem and DRAM, the extremely low latency PMem is able to speed up applications that require persistent storage such as logging, page propagation, or simple random reads/writes from/to a persistent medium.

To measure the latency for load operations on PMem, we use a single thread and perform loads from random locations. To study this effect, we prevented out-of-order execution by chaining the loads such that the address for the load in step i depends on the value read in step $i-1$. This approach is more reliable than using fencing operations (`lfence`), as these still allow for speculative loads and introduce a small CPU overhead. To minimize caching effects, we use an array sufficiently larger (8 GB) than the CPU cache (32 MB). The results of this experiment are shown in Fig. 1.

We can observe that DRAM read latency is lower than PMem by a factor of 3.3. Note that this does not mean that each access to PMem is that much slower, because many applications can usually still benefit from spatial or temporal locality (i.e., caching). When PMem is used in memory mode, it replaces DRAM as main memory and DRAM acts as an L4 cache. In this configuration, the data size is important: When using only 8 GB (as in the app direct experiment) the performance is similar to that of DRAM, because the DRAM cache captures all accesses. However, when we increase the data size to 360 GB, the DRAM cache (192 GB) is not hit that frequently and the performance degrades to what the PMem is actually capable of.

To store data persistently on PMem, the data have to be written (i.e., a store instruction), the cache line evicted, and then an `sfence` has to be used to wait for the data to reach PMem. This process is described in more detail in Sect. 3.1. To measure the latency for persistent store operations[7] on PMem, we use a single thread that persistently stores data to an array of 10 GB in size. Each store is aligned to a cache line (64 bytes) boundary. The results are shown in Fig. 2.

The four bars on the left show the results for continuously writing to the same cache line, in the middle we write cache lines sequentially, and on the right randomly. In each scenario, we use four different methods for flushing cache lines. From left to right: `flush`, `flushopt`, `clwb`, and non-temporal stores (`_mm512_stream_si512`).

When data are written to the same cache line, non-temporal stores should be preferred. This pattern appears in many data structures (e.g., array-like structures with a size field) or algorithms (e.g., a global counter for time-stamping) that have some kind of global variable that is often modified. Therefore, for efficient usage of PMem, techniques similar to the ones developed to avoid congestion in multi-threaded programming have to be applied to PMem as well. Among instructions without the non-temporal memory hint, there is no significant difference, because the Cascade Lake CPUs do not fully implement `clwb`. Intel has added opcode to allow software to use it, but implement it as `flush_opt` for now. Therefore, non-temporal operations and `clwb` should be preferred over `flush` and `flush_opt`.

2.3 Bandwidth

Having discussed latency of PMem in the previous section, we now want to investigate the bandwidth for both non-saturated and saturated PMem systems. For this, it is important to know that the PMem hardware internally

[7] We do not show non-persistent writes to PMem, as these are not latency bound because they are cached just like DRAM writes.

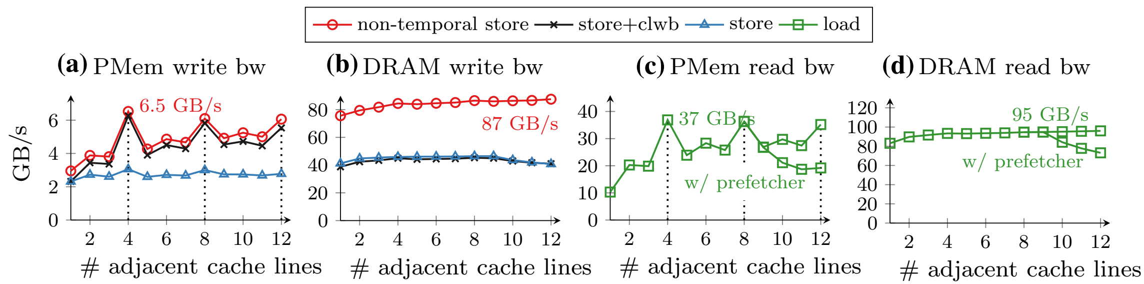


Fig. 3 PMem bandwidth: varying access granularity—PMem bandwidth (**a**, **c**) with 24 threads compared to DRAM bandwidth (**b**, **d**) with a varying number of adjacently accessed cache lines. We use a random access pattern that allows for out-of-order execution

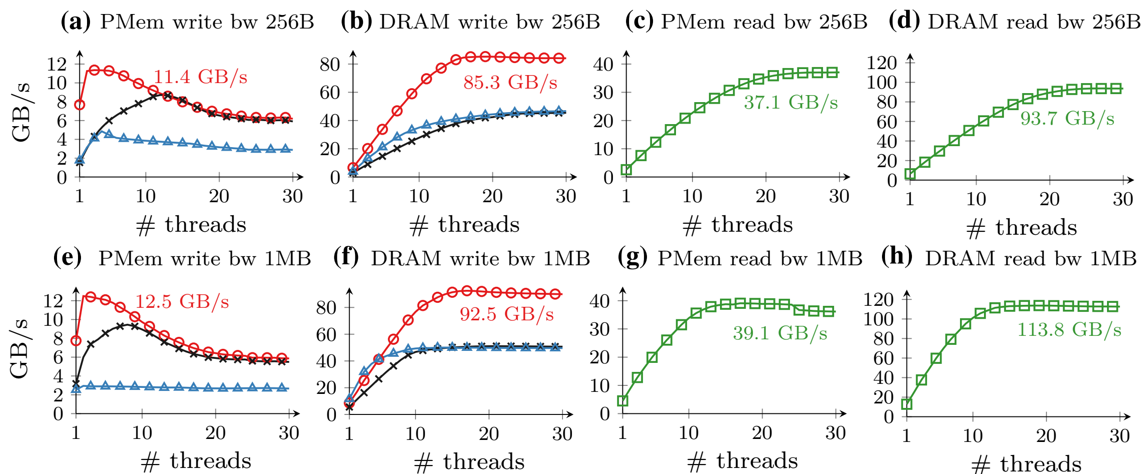


Fig. 4 PMem bandwidth: varying thread count—PMem bandwidth (**a**, **c**, **e**, **g**) compared to DRAM bandwidth (**b**, **d**, **f**, **h**) for 256-byte (4 adjacent cache lines) and 1 MB blocks with an increasing number of threads. We use a random access pattern that allows for out-of-order execution

works on 256-byte blocks. A small write-combining buffer is used to avoid write amplification, because the transfer size between PMem and CPU is, as for DRAM, 64 bytes (cache lines).

This block-based (4 cache lines) design of PMem leads to some interesting performance characteristics that we are discussing in this section. The first experiment (cf. Fig. 3) measures the bandwidth for loading/storing from/to independent random locations (allowing out-of-order execution) on PMem and DRAM. We use all 24 physical cores of the machine to maximize the number of parallel accesses. The figure shows store (PMem: a, DRAM: b) and load (PMem: c, DRAM: d) benchmarks. The performance depends significantly on the number of consecutively accessed cache lines on PMem, while there is no significant difference on DRAM. Peak throughput can only be reached when a multiple of the block size (4 cache lines = 256 bytes) is used, thus confirming the 256-byte block-based architecture of PMem. Obviously, this effect is mostly relevant for smaller chunk sizes as the write/read amplification is bound to three cache lines at most.

Before discussing the different write techniques, we want to use Fig. 4 to derive peak bandwidth numbers: In the experiment, we vary the number of threads on the horizontal axis instead of the number of cache lines loaded/stored. The first row (a, b, c, d) shows the bandwidth for writing PMem-block-sized chunks (256 bytes) to random locations. The second row (e, f, g, h) shows the same for 1 MB sized chunks. As one would suspect, by using larger chunk sizes a higher bandwidth can be achieved, the peaks are shown in Table 2.

Next to pure bandwidth numbers, the figure shows that the “ramp up’-phase’ (number of threads required to reach the peak bandwidth) is faster with larger chunk sizes. Additionally, we observe that the throughput peaks on PMem when using a small number of threads and then declines, while it flattens out on DRAM. We suspect that this is only an artifact of the first version of this new hardware and future versions of this product will be able to handle higher request rates without suffering in throughput.

This hole section is broken. There should NOT be a new line before the icons. In addition, the icons are blury. (–▲–), regular stores followed by a `clwb` instruction (–×–), and blind writes realized by a non-temporal (or streaming) store (i.e., `_mm512_stream_si512`) (–⊖–). For both, DRAM and PMem, the blind stores provide the best throughput because the modified cache lines do not have to be loaded

Table 2 Peak write and read bandwidth for DRAM and PMem with an optimal number of threads

	Peak read BW	Required #threads	Peak write BW	Required #threads
DRAM	113.8	15	92.5	17
PMem	39.1	17	12.5	3

first—thereby saving memory bandwidth. On PMem, however, there is an additional benefit when using non-temporal stores as they bypass the cache and force the data to the PMem DIMM directly.

To explain this, consider the stark performance difference between DRAM and PMem when using stores followed by `clwb` (−∗−) in Fig. 4: On DRAM, the extra instruction only adds additional CPU overhead to a very tight loop and thus causes a slowdown compared to regular stores (−△−). With an increasing number of threads this overhead no longer impacts the overall throughput, as the bottleneck shifts from CPU-bound to memory-bound. On PMem, in contrast, the performance of regular stores (−△−) can be increased by issuing a `clwb` instruction after each store (−∗−). By forcing the cache lines to PMem right after they are modified, we can ensure that the ones that are modified together also arrive at the PMem DIMMs together and can thus be written together by the write-combining buffers. In other words: By using the `clwb` instruction, we are preserving the spatial locality of the writes when going from the CPU to the PMem DIMMs.

Using `clwb` (−∗−) becomes more important with several threads than with a single one, because cache lines are evicted more randomly from the last level CPU cache, and thus arrive increasingly out of order at the PMem write-combining buffer. Starting at 12 threads for 256B chunks, regular stores followed by a `clwb` (−∗−) become as fast as non-temporal stores (−⊖−). However, this is likely due to the performance drop experienced by the non-temporal stores due to the oversaturation of PMem. Compared to DRAM, where there is only a difference between blind writes (−⊖−) and regular ones (−△−, −∗−), on PMem there is also a difference whether we ensure spatial locality of modified cache lines at the PMem DIMM (−∗−, −⊖−) or not (−△−). Thus, on PMem we end up with three discrete optimal throughput numbers (when considering the peaks) for regular stores (−△−), regular stores followed by a `clwb` instruction (−∗−), and non-temporal store (−⊖−). While there is a minor CPU overhead for using `clwb`, our experiments do suggest that the potential bandwidth benefit is worth it.

Lastly, we briefly want to show an interesting yet, PMem-unrelated finding: The read benchmarks (c, d) show throughput numbers with (*w/ prefetcher*) and without the hardware prefetcher[8]. For both, PMem and DRAM, there is a significant performance drop when the prefetcher is enabled starting at 10 consecutively accessed cache lines. This experiment illustrates a PMem unrelated, yet interesting effect: when reading chunks of more than 10 cache lines from random locations with many threads (oversubscribed system), the prefetcher can actually harm the effective bandwidth as it unnecessarily loads cache lines.

In summary, judging from our experimental results, we recommend the following guidelines for bandwidth-critical applications:

– Algorithms should no longer be designed to fit data on single cache lines (64 bytes) but rather cluster data on PMem blocks (256 bytes).
– When possible, non-temporal stores should be utilized, otherwise the regular stores should be followed by a `clwb` instruction.
– Over-saturating PMem can lead to reduced performance with as little as four threads.
– The experiments showed that currently the PMem read bandwidth is 2.9× lower and the write bandwidth 7.4× lower than DRAM. Therefore, performance-critical code should prefer DRAM over PMem (e.g., by buffering writes in a DRAM cache).

2.4 Interference

Next to bandwidth and latency, another important question to answer is how well DRAM, after decades of solitude, works alongside with PMem. In contrast to Yoshida et al. [19], who have already undertaken an extensive study in several microbenchmarks, we show interference effects in a simulated database workload. We simulate a database workload made up of four tasks: table scans (sequential reads: "Sq"), index lookups (random reads: "Rd"), logging (small sequential persistent writes: "Log") and page propagation (large random persistent writes: "Page"). Table scans and index lookups can be executed either on DRAM (volatile: "V") or on PMem (persistent: "P"). Page propagation and logging are always done on PMem as they need to be persistent. Figure 5 shows the relative slowdown of 14 threads performing table scans on DRAM (a), table scans on PMem (b), index lookups on DRAM (c) and index lookups on PMem (d) when executed together alongside with one other task (depicted on the horizontal axis). The other task uses 1, 5 or 10 threads.

We find a significant slowdown (around 50%) for table scans on DRAM (a) when large amounts of data are read from

[8] Intel hardware prefetcher can be disabled via `wrmsr -all 0x1a4 7` https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.

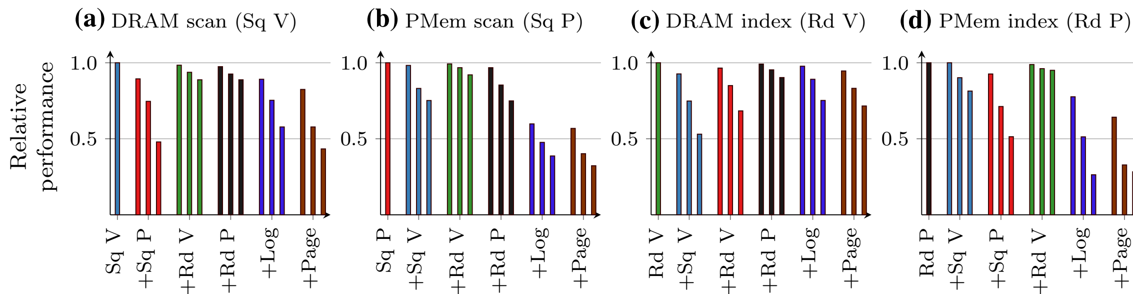


Fig. 5 PMem versus DRAM interference—the relative slowdown of sequential scans (Sq) and random accesses (Rd) on DRAM (V) and PMem (P) when executed together with other sequential scans, random lookups, log writing (Log), and page propagation (Page) with 1, 5, and 10 threads

or written to PMem. Concurrently performed index lookups, only cause a minor slowdown, because they do not require a lot of bandwidth. It is interesting to note, that writing data to PMem via logging ("Log") and page propagation ("Page") only consumes 6.4 GB/s and 9.2 GB/s of bandwidth, yet the sequential DRAM read bandwidth drops from 113.5 GB/s down to 65.6 GB/s and 49.2 GB/s, respectively. This leaves a significant gap of "unused" bandwidth. Yoshida et al. [19] suggest that this is caused by the CPU's memory controller prioritizing PMem write requests over DRAM requests.

The other way around, when performing table scans on PMem (b) the slowdown caused by DRAM reads ("Sq V") is not as pronounced. In this scenario, the only significant slowdowns happen when another task is writing to PMem at the same time. As before, this causes a significant slowdown of more than 50% from 40.3 GB/s down to 15.6 GB/s for logging and 13.0 GB/s for page propagation. Together with the results from the table scan on DRAM (a), this suggests that the amount of data that is written to PMem (page propagation consumes more bandwidth than logging) is more important than the number of persistency barriers (logging).

In the third scenario (c), the index lookups on DRAM are mostly slowed down by other DRAM uses. This can be explained by the smaller amount of bandwidth that is required for random reads, compared to sequential reads (a). However, this effect is not observable on PMem: for index lookups on PMem (d), the experiment shows even larger slowdowns than for table scans on PMem.

3 Building blocks for PMem

The low write latency of PMem (compared to other durable storage devices) makes it an ideal candidate for use in database systems, file systems, and other system software. However, due to the CPU cache, writes to PMem are only persisted once the corresponding cache line is flushed. Algorithms have to explicitly order stores and cache line flushes to ensure that a persistent data structure is always in a consistent state (in case of a crash). We call this property *failure atomicity* and discuss it in Sect. 3.1. Intel's Persistent Memory Development Kit (PMDK) [43], an open-source library for PMem, abstracts this complexity by providing two failure atomic I/O primitives: log writing (*libpmemlog*) and block-/page flushing (*libpmemblk*). In Sect. 3.3 and 3.2, we apply the guidelines developed earlier (Sect. 2) to these two problems and analyze their performance. Afterwards, in Sect. 3.4 we introduce Failure-Atomic Memory (FAM), an abstraction over persistent memory that enables fast in-place updates while guaranteeing failure atomicity. Lastly, we show how to use fibers[9] (implemented as C++20 coroutines) to avoid stalls on synchronous writes to PMem.

3.1 Failure atomicity

As mentioned earlier, when data are written to PMem, stores are not immediately propagated to the PMem device, instead they are buffered in the regular on-CPU cache. Therefore, a whole cache line cannot be written as an atomic operation. Only updates made to a cache line (in 8-byte blocks) by the CPU are atomic. While programs cannot prevent the eviction of a cache line, they can force it by using explicit write-back (`clwb`) or flush CPU instructions (`flush` or `flush_opt`). This implies that any persistent data structure on PMem always needs to be in a consistent (or recoverable) state, as any modification to the structure could become persistent immediately. Otherwise a system crash—interrupting an update operation—could lead to an inconsistent state after a restart. The following code snippet shows how an element is appended to a pre-allocated buffer:

```
1 struct Buffer {      | void append(Buffer* buf,
2   int eles[128]      |             int ele) {
3   int next           |   buf->eles[buf->next]=ele
4 }                    |   clwb(&buf->eles[buf->next])
5                      |   sfence()
6                      |   buf->next++
7                      |   clwb(&buf->next)
8                      |   sfence()
9                      | }
```

[9] user-land threads with cooperative multitasking.

The new element is first copied into the next free slot (line 3) and the corresponding cache line is forced to be written back to PMem (line 4). Instead of using a regular flush operation, `clwb` (cache line write back) is used, which is an optimized flush operation designed to evict the cache line without invalidating it. Before the buffer's size indicator (`next`) can be changed, an `sfence` (store fence) must be issued to prevent re-ordering by the compiler or hardware (line 5). Once `next` has been written (line 6), it is persisted to memory in the same fashion (line 7, 8). Note that persisting the `next` field is not necessary for the failure atomicity of a single append operation. However, it is convenient and often required for subsequent code (e.g., another append). Hereafter, we will use the term *persistency barrier* and *persist* for a combination of a `clwb` and a subsequent `sfence`:

```
void persist(void* ptr) { clwb(ptr); sfence(); }
```

Generally, a persistency barrier is an expensive operation ($\geq$ 100 ns, cf. Sect. 2.2), as it forces a synchronous write to PMem (or, more precisely, to its internal battery-backed buffers). Therefore, in addition to the guidelines laid out in Sect. 2, it is also important to minimize the number of persistency barriers while still maintaining failure atomicity. While a hand-tuned implementation for a specific problem can often outperform a generic library, the involved complexity and proneness to errors needs to be considered. In the following four sections, we introduce highly tuned building blocks for various problems when dealing with PMem: page propagation (3.2), logging (3.3), in-place updates (3.4), and asynchronous writes via fibers (3.5).

3.2 Page propagation

One of the most important components of a storage engine is the buffer manager. It is responsible for loading (swapping in) pages from the SSD/HDD into DRAM whenever a page is accessed by the query engine. When the buffer pool is full, the buffer manager needs to evict pages in order to serve new requests. When a dirty page (i.e., a modified one) is evicted, it needs to be flushed to storage before it can be dropped from the buffer pool, in order to ensure durability. This process has to be carefully coordinated with the transaction and logging controller, i.e., a page can only be flushed when the undo information of all non-committed modifications is persisted in the log file (otherwise a crash would lead to corrupted data).

Flushing pages to persistent storage is an inherent I/O-bound task. To reduce the latency for page requests, the buffer manager continuously flushes dirty pages to persistent storage in the background. This way, it can always serve requests without stalling on a page flush. This makes flushing pages (in a background thread) a mostly bandwidth-critical problem (compared to log writing, where latency is most important).

For SSDs/HDDs, this architecture is strictly necessary as pages have to be copied to DRAM before they can be read or written by the CPU. When PMem is used instead of SSDs/HDDs, the buffer pool becomes optional. However, as recent work [5,46] has shown, it is still beneficial to use a buffer pool, due to the lower latencies and reduced complexity when working on DRAM compared to PMem. In addition, this architecture is used in most existing disk-based database systems. Moreover, page propagation is also required in many other system software, as evidenced by the Persistent Memory Development Kit (PMDK [43]) offers page propagation in their `libpmemblk` library.

In the following, we first discuss why page propagation algorithms on PMem should be failure atomic. After that, we describe the two well-known page propagation algorithms (copy-on-write and log-based), show how they can be applied to PMem and propose potential optimizations.

3.2.1 Failure-atomic page propagation

In order to prevent data corruption, the page propagation algorithm needs to ensure that written data can always be recovered. This can either be achieved by making the propagation process failure atomic or by detecting and recovering inconsistent pages later on. Using a failure-atomic page propagation has the advantage that it reduces the complexity of the system: There is no need to detect torn writes during recovery and use a combination of logging and snapshotting to repair inconsistent pages. While this is desirable for most applications, one might argue that high-performance system software (such as database systems) implements these functionalities already and could therefore benefit from a faster *non*-failure-atomic page propagation algorithm. In fact, at any given point in time there is only a very small number of pages that might experience a torn write during a crash: The CPU cache is usually much smaller (tens of megabytes) than the page volume in a database (hundreds of gigabytes). However, in the following we show that failure-atomic page propagation is as efficient as detecting torn writes on PMem and therefore advantageous, due to its reduced complexity.

On SSDs and HDDs detecting torn writes relies on sequential write guarantees of the underlying hardware: A marker (bit pattern) at the beginning and end of a page can be used to validate if a page has been written completely. Without these guarantees on PMem (only ensures atomic 8-byte writes), we need to utilize persistency barriers to order stores and make torn writes detectable. By utilizing two (or more) persistency barriers torn writes could easily be detected on PMem (e.g., copy-on-write). However, such an algorithm already ensures

failure atomicity making torn write detection and recovery unnecessary.

While it is possible to detect torn writes with a single persistency barrier, it comes with some, arguably, unacceptable overheads for page propagation[10]: *PopLog* requires zero-initialized memory and thus twice the bandwidth, *RAWL* requires significant additional computation and some extra storage, and *FAM* requires roughly twice the amount storage.

Alternatively, probabilistic (also called: optimistic) techniques, as proposed by Lersch et al. [30], can be utilized: A check sum of the page's data is written as part of the page and can be used to validate the page's consistency during recovery. When an inconsistency is detected the log and an older snapshot (additional storage required) of the page is used to restore the page. Cryptographic hash functions, such as the Secure Hash Algorithm (SHA), make collisions practically impossible and should provide sufficient throughput in a multi-threaded scenario (throughput of state-of-the-art SIMD-optimized `SHA-256` implementations is reported[11] at roughly 3.5 GB/s per core). In our implementation we use `CRC32`, which is supported directly by modern CPUs (`_mm_crc32_u64`) and works almost at line rate (we measured a throughput of 10.3 GBs^{-1}). While `CRC32` does not provide as good of a collision resistance, it does model the best case scenario for the check-sum-based page propagation as it incurs the lowest overhead. However, our experimental results, even for `CRC32`, showed no performance advantage compared to the failure atomic copy-on-write implementation[12]. Therefore, we argue that the additional system complexity, recovery time, and storage overhead (for snapshots) is not worth it and failure-atomic page propagation should be preferred.

3.2.2 Copy-on-write

When writing a page back from DRAM to PMem, Copy-on-Write (CoW) does not overwrite the original PMem page. Instead, the modified page is written to an unused PMem page [4], thus avoiding any torn-write complications during copy process. Once the page is fully written, it is atomically set to valid.

Listing 1: **Failure-atomic Page Flushing** – Pseudo code to flush a DRAM page (vp) to a PMem page (pp).

```
            CoW                   |           µlog
 1  // 1. Write data              |  // 1. Invalidate µlog
 2  pp.tex = vp.tex               |  µlog.pid = INVALID;
 3  persist(pp)                   |  persist(log.pid);
 4                                |
 5  // 2. Set PMem                |  // 2. Write to µlog
 6  // page valid                 |  µlog <- vp.dirty_cls
 7  pp.pid = vp.pid               |  persist(µlog);
 8  sfence()                      |
 9  pp.pvn = vp.pvn               |  // 3. Set µlog valid
10  persist(pp.pid,               |  µlog.pid = vp.pid;
11          pp.pvn)               |  persist(µlog.pid);
12                                |
13                                |  // 4. Write to page
14                                |  pp <- vp.dirty_cls
15                                |  persist(pp);
```

This process is illustrated on the left of Listing 1. The pseudo code shows the page propagation of a DRAM resident volatile page (`vp`) to a used persistent page (`pp`). Once the volatile page (`vp`) is written (line 2) and persisted using a persistency barrier (line 3), it is marked as valid (line 7-11) and the old PMem page can be reused. During recovery, the headers of all PMem pages are inspected to determine the physical location of each logical page. To avoid invalidating unused pages before they can be written again, we use a per page monotonically increasing page version number (*pvn*) to determine the latest version of the page on disk.

We illustrate the use of the *pvn* in Fig. 6. Time progresses from left to right. The three physical page slots in the left most column (state 1) show the initial state on PMem: The page slots in row (1) and (2) contain the latest valid persistent copy of the logical page `B` and `A`, respectively. Both slots are shaded in green to indicate that they currently hold a valid page. The page slot in row (3) contains an older version `B`, which can be determined by inspecting the *pvn*: a lower *pvn* indicates an older page version. The different versions of page slot (3) show each step (cf. Listing 1) of flushing a new version of page `A` to this currently (state 1) unused slot. The line numbers in the pseudo code where the transition might

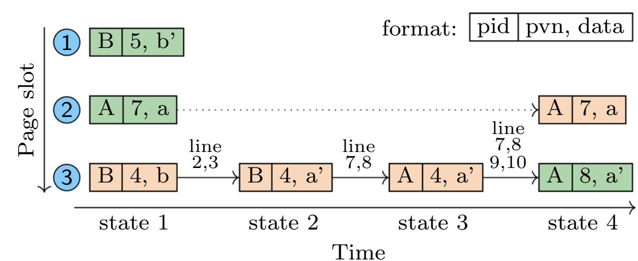


Fig. 6 CoW page propagation—the flush process is optimized by using a page version number (*pvn*) to avoid invalidating pages, which would require an additional persistency barrier

[10] These algorithms are discussed in Sect. 3.3 (logging) and Sect. 3.4 (in-place updates).

[11] https://github.com/minio/sha256-simd.

[12] There is a slight (3 and 4%) advantage in a single threaded scenario.

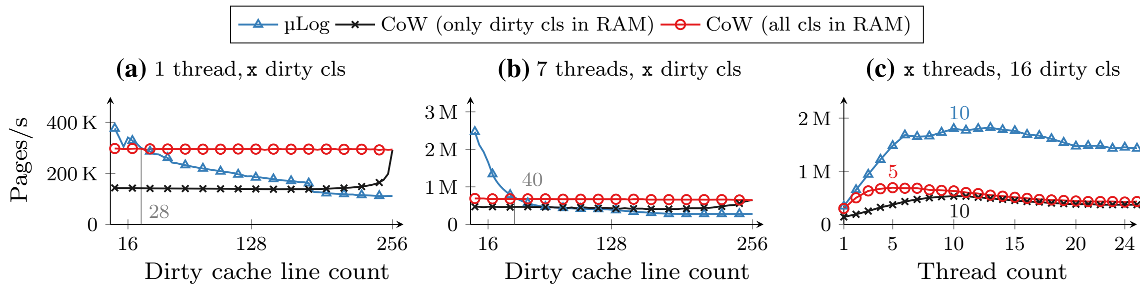


Fig. 7 Failure atomic page flush—flushing 16 kB pages (256cache lines*each*) in a failure atomic way from DRAM to PMem

occur are written over the arrows. In each step, the *pvn* can be used to determine the most recent version of each page.

In state 1, the page slot (3) contains the old (no longer needed) version of `B` and is therefore ready to be overwritten by a new version of page `A`. In state 2, the pseudo code is run until the persistency barrier in line 3 (`persist`). At this point only the payload has been updated and the page would therefore still be identified as an old version of `B`. Next, this newly written version of page `A` has to be made valid by updating the *pid* and *pvn*. It is crucial that the *pid* is updated before the *pvn*, otherwise there is a brief time window in which the updated *pvn* would identify the page as the latest version of page `B` despite it storing data of page `A`. We ensure this ordering by placing both (*pid* and *pvn*) on the same cache line and separating the two store operations with an `sfence`. This way, state 3 and state 4 are the only possible versions of the page during these updates. In each case, the page is correctly identified as an outdated version of page `B` or the new version of page `A`. The final persistency barrier in line 10 ensures that the update is completed before continuing. Note: If page slot (3) would have started out with a higher *pvn* than the one of `A` (e.g., 10), it would have already been identified as the latest version of `A` in state 3 (which is fine because the payload has already been updated).

Using the *pvn*, it becomes unnecessary to invalidate the old PMem page before writing the new one. This reduces the number of persistency barriers from three down to two (plus one `sfence`, which is much cheaper as it does not stall on a preceding `clwb`). Using this technique, we measured a $\approx$ 10% increase in throughput.

In the context of a database system, it would still be necessary to add a *pvn* next to the existing log sequence number *lsn*. Otherwise, the example would contain an invalid configuration in state 3: The log entries between 4 and 8 would be applied to an already updated page.

3.2.3 Micro log

The micro log technique (µlog) uses a small log file to record changes that are going to be made to the page. In order to know which cache lines have been changed, the page is required to track modified areas since its last flush. During recovery, all valid micro logs are reapplied, independent of the page's state. This forces us to invalidate the log (right-hand side of Listing 1, line 1-3) before changing the content (line 5-7), otherwise the changes would be applied to the previous page in case of a crash. Only once the changes are written, we set them to valid (line 8-10) and then apply them to the actual page (line 13-15).

3.2.4 Experiments

Figure 7 details the page flush performance. All techniques are implemented as a microbenchmark using non-temporal stores (also known as streaming stores), which have been shown to provide the highest throughput in Sect. 2. When using copy-on-write (*CoW*), we differentiate whether all cache lines are available in DRAM (-⊖-) or only the dirty ones (-×-). As a performance metric, we chose the number of pages that can be flushed to PMem per second. We vary the number of dirty cache lines in (a) for a single thread and in (b) for 7 threads. In (c), we vary the number of threads to show the scale-out behavior.

The results show that the micro log (*µLog*) is beneficial when the number of cache lines that have to be flushed is low. We can observe this effect for a single thread in (a): Using the *µLog* yields performance gains for up to 28 dirty cache lines. In a multi-threaded scenario (c), the *µLog's* advantage continues up to 40cache lines. Therefore, a hybrid technique based on a simple cost model should be used to choose the better technique, depending on the number of dirty cache lines (and single/multi threading).

The *CoW* approaches are largely independent of the number of dirty cache lines. As expected, the performance is lower when cache lines have to be loaded from PMem first (-×-). The throughput for *CoW* (-⊖-) with a single thread is almost at 300 thousand pages per second, which corresponds to a throughput of 4.9 GBs^{-1} and is thus significantly lower than the raw throughput for writing to PMem of 7.7 GBs^{-1} (as measured in Sect. 2.3). This can be explained by (1) the interference (cf. Sect. 2.4) when using DRAM and PMem in parallel (note: the bandwidth experiments in Sect. 2.3

do not load the data that is written first, but simple write register-resident constants to memory) and (2) the memory stalls of using persistency barriers ($\approx$ 6% slowdown). However, this gap tightens when using multiple threads (optimally 5): *CoW* is able to flush $\approx$ 700 thousand pages per second which corresponds to 11.3 GBs^{-1} and is therefore at 90% of the maximum throughput of PMem (using 2 threads: 12.5 GBs^{-1}).

In addition, as in the bandwidth experiments, we can see a performance degradation when too many threads are used: For optimal throughput it is important to tailor the number of writer threads to the system. As (b) shows, the performance degrades after reaching a peak at around 7-11 threads.

Lastly, the microbenchmarks in Sect. 2 suggested that non-temporal stores should be preferred over regular stores. We were able to confirm this finding in the page flushing experiment (not shown in the chart).

3.3 Logging

Write-ahead logging (WAL) is used to ensure the atomicity and durability of transactions in database systems as well as many other system software such as file systems. In this section, we device PMem techniques for efficient WAL logging. In WAL, the durability is achieved by recording (logging) the individual changes of a larger transaction in order to be able to undo them in the event of a crash or rollback. If any of the transaction's changes to the data are persisted while the transaction is still active, the log has to be persisted as well. Before a transaction is completed, all log entries of the transaction have to be written persistently (thereby guaranteeing to the user, that all changes of the transaction are durable). Logging allows a database to only persist the delta of the modifications: For example, consider an insert into a table stored as a B-Tree: Using logging, only the altered data needs to be persisted instead of all modified nodes (pages). During restart, the recovery component reads the log file, determines the most recent fully persisted log entry, and applies the log to the database.

Logging continues to constitute a major performance bottleneck in database systems [17] when using traditional storage devices (SSD/HDD): each transaction has to wait until the log entry, recording its changes, is written. As a mitigation, reduced consistency guarantees are offered and complex group commit protocols are implemented. However, using PMem, a low-latency logging protocol can be implemented that largely eliminates this problem.

In the following, we first explain the adoption of two well-known logging algorithms for SSD/HDD to PMem (*Classic* and *Header*). Next we discuss the *RAWL* algorithm [49] and introduce the *PopLog*, which are both designed for PMem. Lastly, we do an experimental evaluation of the described algorithms.

3.3.1 Algorithms

Classic represents a form of logging commonly used in database systems [45]. The following listing shows the algorithm in pseudo code (left) and the file layout grammar (right). For clarity, only information relevant to the protocol is depicted.

```
1 log << header << payload  |  LogFile -> Entry*
2 persist(log)              |  Entry -> header ↩
3 log << footer             |           payload ↩
4 persist(log)              |           footer
```

A log entry is flushed in two steps: First, the header and payload is appended to the log and persisted; second, the footer, which contains a copy of the log sequence number (*lsn*; an id given to each log entry). The *lsn* in the footer can be used during recovery to determine whether a log entry was completely written and therefore should be considered as valid and applied to the database. Note that it takes two persistency barriers. Without the first barrier, parts of the payload could be missing even if the footer is present in PMem, due to the flushes being reordered.

Header uses the same technique as *libpmemlog* in the PMDK [43]. It is similar to appending elements to an array:

```
1 log << header << payload  |  LogFile -> size ↩
2 persist(log)              |             Entry*
3 log.size += entry_size    |  Entry -> header ↩
4 persist(log.size)         |           payload
```

The log entry is also written in two steps: First, the header and payload are appended to the tail of the log and persisted. Next, the new size of the log is set in the header of the log file and persisted. This eliminates the need to scan the log file for the last valid entry during recovery because the valid size is directly stored in the header.

RAWL is a logging technique specifically developed for PMem in the context of the Mnemosyne library [49].

```
1 for(i=0;i<bit_cnt;i+=63)  |  LogFile -> Entry*
2 {                         |  Entry -> B64*
3   b = payload[i:i+62]     |  B64 -> 1bit-valid ↩
4   log << b                |      63-b-payload
5       << validity_bit     |
6 }                         |
7 persist(log)              |
```

In *RAWL*, the log file needs to be initialized to zero. This is commonly done by database systems (e.g., PostgreSQL) anyway to force the file system to actually allocate pages to the file. Unlike the first two techniques, *RAWL* requires only a single persistency barrier. This constitutes a large advantage in terms of performance because persistency barriers cause synchronous writes to PMem, which take around 100 ns as shown in Fig. 2. To still be able to guarantee atomicity, each 8-byte block (atomic write unit for PMem) in the log file

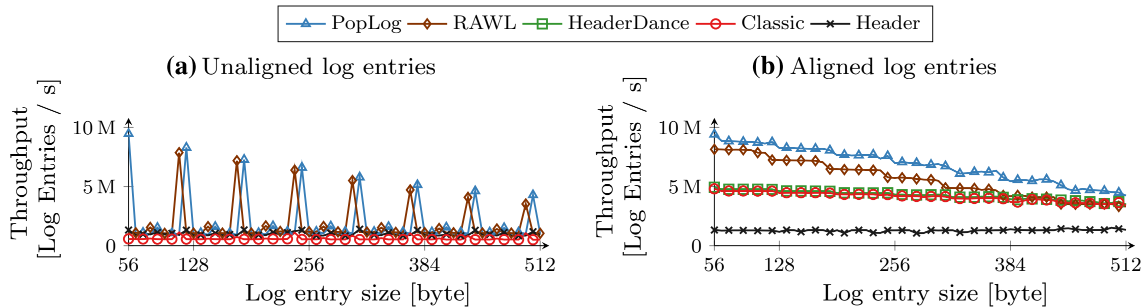


Fig. 8 Transaction log—the throughput for writing log entries of varying size to PMem

contains a validity bit and 63 bits of log data. When initially writing the log file, data is chunked into 63 bit ranges and concatenated with a set validity bit (`validity_bit=1`). Once the log file is full, it can be reused by flipping the validity bit.

PopLog is a novel technique we propose for PMem that requires only one persistency barrier:

```
1 cnt = pop_count(header,      |  LogFile -> Entry*
2                 payload)      |  Entry -> header ↩
3 log << header                 |  pop_cnt payload
4     << cnt << payload         |
5 persist(log)                  |
```

As in *RAWL*, before logging starts, the log file is initialized to zero. When a log entry is written, the number of set bits are counted (using the `popcnt` instruction). Next the header, data, and bit count (`cnt`) is written to the log and persisted together. Using the bit count, it is always possible to determine the validity of a log entry: Either the cache line containing the bit count was not flushed or it was. In the former case, the field contains the number zero (because the file was zeroed) and the entry is invalid. In the latter case, the bit count field can be used to determine whether all other cache lines belonging to the log entry have been flushed as well. Compared to *RAWL*, the code for writing and reading the log is less complex and only requires a logarithmic space overhead (`pop_count` field) instead of a linear one (1 validity bit per 63 bits of log data).

3.3.2 Experiments

In Sect. 2.2, we showed that there is a large performance penalty when the same cache line is persisted twice in a row. This effect is very relevant for latency-critical systems, as shown in Fig. 8. We use a micro-benchmark that measures the throughput of flushing log entries of varying sizes. The left chart shows a naive implementation, while the right one uses padding on each log entry to align entries to cache line boundaries and thus avoid subsequent writes to the same cache line (which have been shown to be slow: cf. Fig. 1). While padding wastes some memory[13], the throughput greatly increases ($\approx 8\times$). The correct alignment also happens by "accident" in the left chart when the log entries are just the right size. These performance spikes happen 8 bytes earlier for *RAWL* compared to *PopLog*, because of the validity bits used in *RAWL*. This gap would widen with larger log entries (first time at 512 bytes where two times 8 bytes are required).

As an alternative to padding, the cache lines that are persisted twice (for two subsequent log entries) could also be cached in a DRAM buffer and then flushed with a non-temporal (or streaming) store operation. This would avoid the need to re-load the evicted cache line and therefore avoid the slowdown. The additional work caused by copying the data into a DRAM buffer has a small performance penalty, thus making this a trade-off between used space and latency. In the shown experiment we used the padding approach and thus traded space for latency.

However, even with padding, the *Classic* approach still outperforms the *Header* one, because of the slowdown due to the writes to the same cache line in the header when the size is updated. This problem can be solved by using a *dancing* size field: We use several size fields on different cache lines in the header and only write one (round-robin) for each log entry. By using 64 of these dancing size fields, the throughput of *Header* can be increased to that of *Classic*. However, both of these techniques still require persistency barriers and therefore cannot compete with *RAWL* and *PopLog* ($\approx 2\times$ faster). *PopLog* slightly outperforms *RAWL* because less processing is required and slightly less memory has to be copied.

The log implementation (*libpmemlog*) of PMDK [43] uses the same approach as our naive *Header* implementation without alignment and dancing. Therefore, it also yields the same throughput, when its support for multi-threading is disabled (not shown in the charts). It has the advantage that the log file is dense and can be presented to the user as one continuous memory segment. However, this leaves the user with the task of reconstructing log entry boundaries manually. By moving

[13] At most 1 cache line for *PopLog*, *RAWL*, and *Header*; up to 2 cache lines for *Classic*.

Table 3 Performance characteristics of in-place update techniques

	Required	#Cache lines written			#Persists
	Size (byte)	16B	32B	64B	
CoW	$2n+1$	2	2	3	2
Log	$n+c$	2	2	3	2
FAM	$\lceil 8n/31 \rceil * 8$	1	2	3	1

this functionality into the library, a better logging strategy can be implemented and the usability increased.

For validation, we have integrated all techniques into our storage engine prototype HyMem [46]. Running a write-heavy (100%) YCSB benchmark [11] on a single thread with a DRAM-resident table, *PopLog*, *Header*, and *Classic* achieves a throughput of 2 M, 1.7 M, and 1.5 M transactions per second, respectively.

3.4 In-place updates

In the previous two sections, we discussed page propagation (writing out large chunks of data to random locations) and log writing (writing out small pieces of data sequentially). In the following, we want to investigate small (16- to 64-byte) failure-atomic in-place updates, which are important as they are the persistent equivalent to *simply writing* to volatile memory. PMem only supports 8-byte failure-atomic writes. Any data up to this size can simply be updated by a regular store instruction followed by a persistency barrier (`clwb` and `sfence`). For larger in-place updates, as commonly used in any PMem-based data structure [2,9,15,27,48,52], either copy-on-write or log-based techniques are used. Both techniques require at least two persistency barriers, thus slowing down the update throughput.

In the following, we first detail these existing techniques and then introduce a new approach that is able to perform in-place updates with a single persistency barrier. To simplify the examples, we will focus our discussion on updating 16 bytes. In the evaluation, toward the end of this section, we evaluate the techniques for a variety of data sizes.

Table 3 shows a summary of the performance characteristics of the three approaches. The first column displays the size (in bytes) of the data structure to store n bytes of user data. The next three columns, show how many cache lines need to be written to PMem per update for three data sizes (16 B, 32 B, and 64 B). The last column shows how many persistency barriers are required per update. Note that 8 B can be updated atomically by hardware and, therefore, do not to be handled by a special algorithm.

3.4.1 CoW-based

Similar to Sect. 3.2.2 where copy-on-write (CoW) was used for page propagation, the new data is first written to an unused location, persisted, and then set valid:

```
1 struct CowBased { |  void update(char[16] new) {
2   bool active     |   if(active) {
3   char a[16]      |     b = new; persist(b)
4   char b[16]      |   } else {
5 }                 |     a = new; persist(a)
6                   |   }
7                   |   active = !active
8                   |   persist(active)
```

In order to do this in-place, we need roughly twice the required data plus a single boolean value that indicates which field (`a` or `b`) is currently active. The memory consumption could be optimized by sharing the "unused" buffer over multiple CoW structures. However, this would incur an additional cache line miss (pointer chase). Additionally, by moving the actual data behind a pointer (out-of-place), we would avoid the actual issue we are trying to solve here: in-place updates. Therefore, the depicted algorithm keeps both versions in-place and could be used on a single node in a tree-like data structure (thus avoid memory allocation and reclamation issues and also keeping it in a flat memory format that can be easily written to disc). The update process inherently requires two persistency barriers to avoid any corruption in case of a crash, because the new data needs to be fully written before it can be set valid. For both, reading and writing, only one cache line has to be touched for 16 bytes of data.

3.4.2 Log-based

Similar to Sect. 3.2.3 about the µlog for page propagation, the data is first written to a log, persisted, and then modified in-place. In case of a crash, the log is used to undo or redo the pending changes.

```
1 struct LogBased { |  void update(char[16] new) {
2   PopLog* log     |   log->Append(new)
3   char data[16]   |   log->Persist()
4 }                 |   data = new
5                   |   persist(data)
6                   |  }
```

Unlike in the *CoW*-based technique, only a single log file is required for all in-place updatable fields in a data structure (or the entire program). Therefore, the space overhead is reduced to a constant amount (depending on the data size). However, an update operation now touches at least two distinct cache lines and still requires at least two (depending on the logging technique) persistency barriers: one for the log

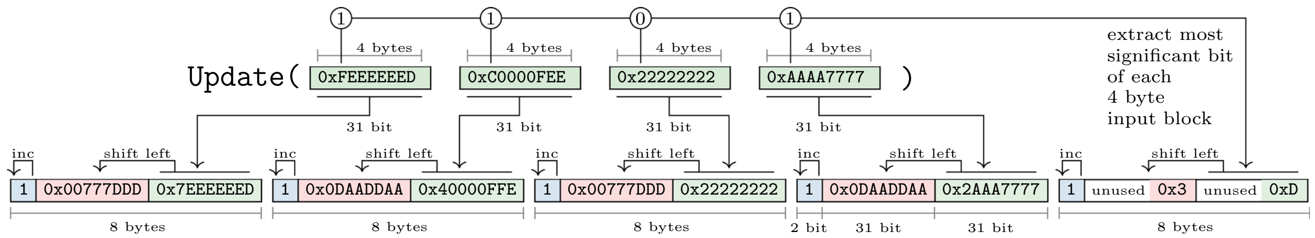


Fig. 9 In-place updates with a single persistency barrier—the input (top) is split into 31-bit blocks and stored in 8-byte blocks (bottom), which are of which stores the previous (red) and new (green) state. In case of a crash, the version number (blue) can be used to recover the old state

and on for the data. To minimize the number of used persistency barriers we used *PopLog* for the log-based in-place updates.

Note that it is not possible to simply use *PopLog* or *RAWL* for in-place updates directly: *PopLog* would require to reset the data to zero, before writing the new data. If the system crashes during these two steps, the old data would not be recoverable. *RAWL* does not have this limitation. However, if the system crashes after a few 8-byte updates, these could also not be recovered. The fact that the memory that we are writing already contains valid information makes the in-place update problem distinct from logging.

3.4.3 Failure-atomic memory (FAM)

While *CoW*-based and logging-based techniques are well known in the field and have been used for decades, PMem allows for some novel algorithms. Here we introduce *failure-atomic memory* (FAM), which is an in-place update algorithm tuned for actual PMem hardware. It improves the write latency and throughput of small in-place updates, at the cost of additional storage (compared to logging). FAM has an advantage over the CoW- and log-based approaches because it only requires a single persistency barrier per update. The key idea is to split the user data into smaller chunks and store them in recoverable 8-byte blocks (hereafter referred to as FAM blocks or FAMBs). Each FAMB is able to store 31 bits of user data and can be written in a failure-atomic way, due to the failure-atomic write granularity of PMem (8 bytes). To make the FAMB recoverable, it stores a version number (2 bits), the old version (31 bits), and the new version (31 bits) of the user data. This allows for updates with only a single persistency barrier, but requires more computation and memory bandwidth when reading the data. For FAM to be able to store 16 bytes of user data, 5 FAMBs (40 bytes) are required:

```
1  struct FAMB {          |  void updateFAMB(i, val) {
2    int2 version         |   l = units[i]
3    int31 old            |   l.version++
4    int31 new            |   l.old = l.new
5  }                      |   l.new = val
6                         |   units[i] = l
7  struct FAM {           |  }
8    FAMB units[5]        |
9  }                      |  void update(char[16] new) {
10                        |   new = cast<int32>(new)
11                        |   for(i=0; i<4; i++) {
12                        |    updateFAMB(i, new[i])
13                        |    high |= new[i] >> 31
14                        |    high = high << 1
15                        |   }
16                        |   updateFAMB(4, high)
17                        |   persist()
18                        |  }
```

The update of a single FAMB is shown in line 1-6: The entire FAMB (8 bytes) is first loaded into memory. It is important that it is copied into a local variable so that any intermediate changes are not written back to memory. Next (order irrelevant), the version is incremented (line 3), the currently stored user data (`new`) is copied to a backup location (`old`) (line 4), and the new user data is written (line 5). Once the FAMB is updated it is written back to memory (6). This process is performed for each 4-byte block of the user data. Because FAMBs only store 31 bits of user data, the most significant bits of each 4-byte input block are extracted (line 13-14) and stored in an additional fifth FAMB (line 16).

This whole process is visualized in Fig. 9: The four blocks at the top visualize the user data (within the `Update()` call) and the five blocks toward the bottom show the five FAMBs. Our algorithm only ensures that no intermediate state of a single FAMB is leaked to PMem, however individual FAMBs of one FAM may be written back before others. In case of a crash before everything is committed to PMem (`persist`), the program can inspect the 2 bit version number during recovery: If the version numbers of all FAMBs match, the FAM is in a consistent state (either old or new). Otherwise, only some FAMBs have been persisted and need to be rolled back. The version number (2 bit) provides 4 states (0, 1, 2, and 3) and increments may trigger overflows (`inc(3) = 0`), making it possible to determine which FAMB has the more advanced

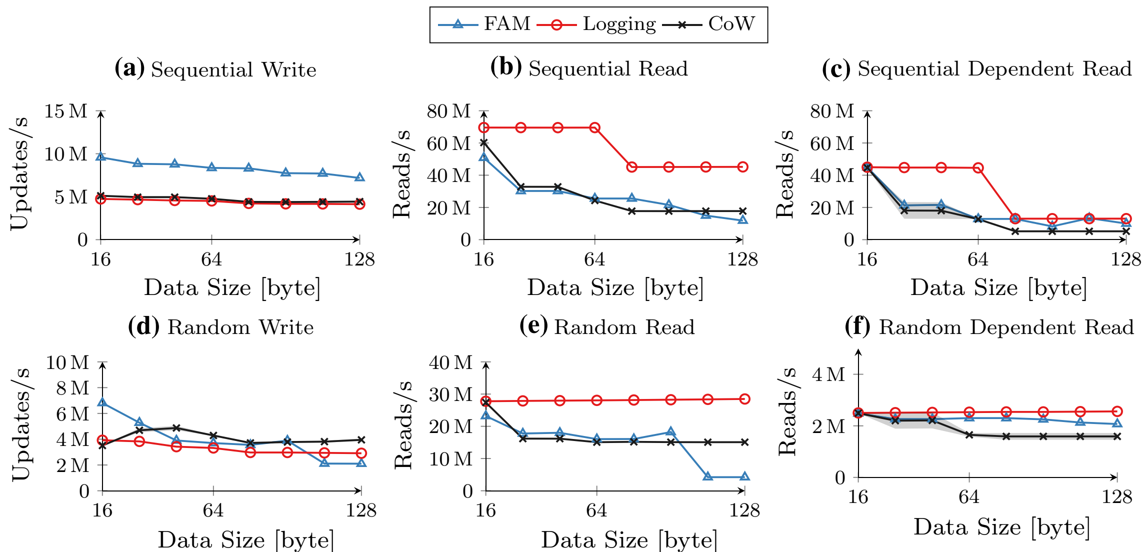


Fig. 10 In-place updates—performance of Failure-Atomic Memory (FAM) in comparison with CoW- and log-based updates and reads. While the lines show the average throughput, the highlighted areas (mostly in (c) and (f)) indicate the 95% percentile over multiple runs

version and needs to be rolled back. A rollback requires the version number to be decremented (with underflows: `dec(0)= 3`) and the recovery of the old version (`l.new = l.old`). In case of repeated crashes, a single FAMB is only rolled back once because during subsequent recoveries the version number already matches the other FAMBs. Hence, the recovery of FAM is idempotent and guarantees progress as rollback actions do not need to be repeated.

FAM reduces the number of required persistency barriers (roughly 100 ns) from two to one, by making use of the failure-atomic 8-byte block on PMem. The additional processing required for FAM can largely be hidden by the high access latency of PMem. However, FAM requires one 8-byte FAMB for each 31 bits of user data. Both the processing and storage overhead (for smaller inputs) could be reduced if the application is not using the entire domain of the data (only 31 bit out of the 32 bit). In our evaluation, we ignore this optimization potential and measure the most generic version of FAM that deals with opaque user data. Appendix A provides and discusses a highly-optimized SIMD (AVX2) implementation of FAM.

3.4.4 Experiments

Figure 10 shows the performance evaluation of the three introduced approaches for in-place updates. We plot the throughput (vertical axis) over the size of the updates in byte (horizontal axis). For each approach, we create a flat array of 10 GB and perform 100 M operations on it. All in-place updatable structures are aligned to cache lines boundaries to minimize the number of cache line write backs (`clwb`). The logging-based approach is assigned a sufficiently large log file for the workload, such that it never has to be re-initialized. The figure shows performance for a sequential access pattern (a, b, c) and a random access one (d, e, f). The throughput of writes (a, d), reads (b, e), and dependent reads (c, f) are depicted from left to right. For dependent reads, out of order execution is prevented by making a read location dependent on the previously read value.

Overall, the results show that the reduction in persistency barriers of FAM pays off for write operations and still offers reasonable performance when reading. Especially for small data sizes (16 bytes), the FAM offers large performance gains (2× for sequential and 1.6× for random). Logging, performs especially well in sequential reads (b, c), because there is no indirection (CoW) or any other processing (FAM) required. However, this advantage is largely lost for random reads as those are dominated by access latency.

3.5 Coroutines

To avoid stalling the active thread of a program on high-latency operations (like disk or network I/O), many libraries implement asynchronous APIs. Internally, these libraries can use multi-threading or work queues and offer some event- or polling-based mechanism for the user to learn about completed operations. Alternatively, the user can utilize kernel or user-land threads with a synchronous API. This has several advantages, as the state of the thread of execution at the time of the API call is automatically preserved and does not have to be restored manually. Independent of how the asynchronous

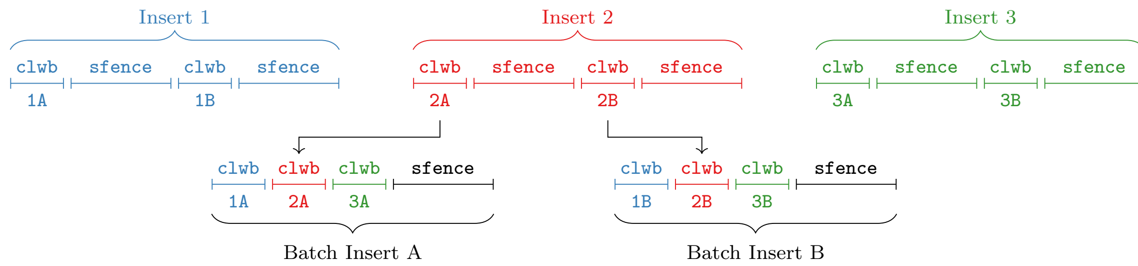


Fig. 11 Coroutines: interleaved inserts—by interleaving n write operation (`clwb`) and sharing one synchronization barrier (`sfence`), the number of memory stalls can be reduced from n to 1

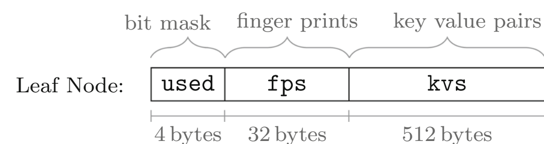
execution is realized, the advantage is that the program can make progress while waiting on a slow I/O operation.

To avoid stalling on low-latency operations (such as DRAM or PMem reads during pointer chasing), an asynchronous API or kernel threads are infeasible as their overhead would be to great. However, lightweight user-land threads with cooperative multitasking have successfully been used to mitigate the impact of memory stalls. The early works of Chen et al. on *group pre-fetching* [8] and Koçberber et al. on *asynchronous memory access chaining (AMAC)* [26] implement the switching between different tasks by hand. With the release of C++20, a low-overhead implementation of cooperative multitasking in the form of coroutines has become available as a language feature. Jonathan et al. [22] showed that the performance of coroutines is competitive with earlier manual implementations and greatly reduces complexity. Given that memory latencies are already an issue for data structures with random access patterns on DRAM, this issue is only intensified on PMem due to the higher latencies. Coroutines have successfully been shown to mitigate the high latencies on PMem by Psaropoulos et al. [44] for index joins and tuple reconstruction in database systems.

On DRAM, only read operations are synchronous and write operations can always be performed in an asynchronous way. On PMem, however, write operations become synchronous as well when followed by a persistency barrier. This leaves the CPU stalling until the written data has reach a persistent location. To mitigate this, we propose to use coroutines (or cooperative multitasking, in general) for interleaving a number of update operations. In the following section, we introduce the FP-Tree [37], which will be used as an example and in the evaluation. Afterwards, we discuss the use of coroutines for read and write operations.

3.5.1 FP-tree implementation

The FP-Tree is a B-Tree-like data structure designed for PMem. It uses sorted inner nodes that are placed in DRAM to speed up the traversal. These volatile inner nodes can be recovered after a crash from the leaf nodes which are placed in PMem. Leaf nodes are not sorted but use hash-based finger prints for efficient point lookups instead:



Each leaf node has a bit mask (`used`) indicating which slots are filled, an array of finger prints (`fps`) that stores a 1-byte hash of each key, and an array with key-value pairs (`kvs`). We use these leaf nodes to measure the effect of interleaving lookups as well as interleaving inserts.

3.5.2 Lookup implementation

A lookup is done by hashing the search key and comparing it to each used finger print in the node. If there is a match, the actual key for this finger print is retrieved and used to validate the match before returning the result. For simplicity, we only evaluated positive lookups and avoid control flow divergence of the executed coroutines by prohibiting multiple matching fingerprints. The non-interleaved lookup code (left) can easily be extended using coroutines (right):

```
 1 Val lookup(k) {            |  Val lookup(k) {
 2  fp = hash(k)              |   fp = hash(k)
 3                            |   _mm_prefetch(used)
 4                            |   co_await sched
 5                            |
 6  for(s=0; s<32; s++) {     |   for(s=0; s<32; s++) {
 7   ok = used[s]             |    ok = used[s]
 8   if(ok&&fps[s]==fp) {     |    if(ok&&fps[s]==fp) {
 9                            |    _mm_prefetch(kvs[s])
10                            |    co_await sched
11                            |
12    if(k == kvs[s].k) {     |     if(k == kvs[s].k) {
13     return kvs[s].v        |      return kvs[s].v
14 }}}}                       | }}}}
```

Before potential memory stalls we issue a prefetch (`_mm_prefetch`) instruction to get the requested cache

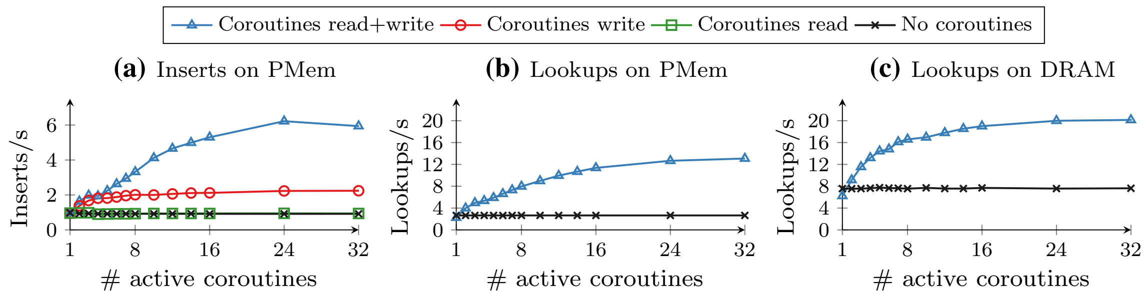


Fig. 12 Coroutines on PMem—we use coroutines to hide read and write latencies of PMem for lookups and inserts on FP-Tree leaf nodes

line from the underlying memory (DRAM or PMem). Instead of waiting for the cache line to be loaded, we use `co_await` to return the control flow to the caller. The caller can then continue execution by resuming the next active coroutine or starting a new one. This way any number of lookups can be executed in an interleaved fashion and while one is waiting for memory to be loaded, another one can progress.

3.5.3 Insert implementation

As described in Sect. 3.4, PMem-based data structures use a two phased update process: (1) write the new data and persist it; (2) set a flag to mark the new data as valid and persist the flag. In case of the FP-Tree, the flag is the `used` bit mask and the data is the key-value pair (`kvs`) and the key's fingerprint `fps`. We can interleave multiple writes, by issuing the write and cache line write back instruction normally and then using one storage fence (`sfence`) for a group of inserts to force the data to PMem. Hence, the algorithm only has to wait once for the completion of all cache evictions, but each individual insert operation still has the guarantee that its data was persisted before it continues. Figure 11 illustrates three inserts with individual fences (top) and shared ones (bottom).

3.5.4 Experiments

Figure 12 shows the experimental results for inserts on PMem (a), lookups on PMem (b), and lookups on DRAM (c). There is no experiment for inserts on DRAM because writes on DRAM are not persistent and, therefore, do not require interleaving. The horizontal axis shows the group size: how many coroutines are active at the same time. For both, reads and writes the curve flattens out at around 20 active coroutines. Due to the higher latencies of PMem, the impact of using an interleaved execution to prevent stalls is more significant than on DRAM: 6.2× for inserts on PMem and 5× speed up for lookups, compared to 2.6× for lookups on DRAM.

The insert experiment (a) shows three different usages of coroutines: only for reads (-⊟-), only for writes (-⊖-) and for both (-▲-). There is no benefit in using coroutines to interleave only reads (-⊟-), because the CPU uses out-of-order execution in the normal code path to prefetch data across persistency barriers. This is possible, because a persistency barrier is made up of a cache line write back (`clwb`) and storage fence (`sfence`) instruction. The `sfence` instruction allows for reordering of loads and only "fences" stores. To test this hypothesis, we used a memory fence (`mfence`) instruction instead of the `sfence`. The `mfence` does not allow for any reordering. In this scenario, the performance of inserts with interleaved reads (-⊟-) becomes similar to that of inserts with interleaved writes (-⊖-).

4 Related work

With PMem only being released recently, this is one of the two [21] initial studies that have been performed on the actual hardware. While our work proposes low-level optimizations, Swanson et al. evaluate PMem with various storage engines as well as file systems. Until now, software or hardware-based simulations, or emulations based on speculative performance characteristics, have been used to evaluate possible system architectures [3,36,38,40]. The number of persistent index structures [2,9,15,27–29,48,52,53] is large, and has been summarized by Götze et. al [16]. Similar techniques have been used to build storage engines directly on PMem [4,35]. These approaches use in-place updates on PMem, which suffers from the lower-than-DRAM performance. Therefore, a number of indexes [37,51] as well as storage engines [1,7,12,23,24,32,33] integrate PMem as a separate storage layer or an extension to the recovery component [39,41]. Furthermore, buffer-managed architectures [5,25,46] have been proposed to use PMem more adaptively. Recovery has always been an essential (and performance-critical) component of database systems [45]. Several designs have been proposed for database-specific logging [6,14,18,42,50] and file systems [13]. There is also a great body of work that researches transactional semantics for PMem as a library to be easily used by other programs [31,34]

(similar to the PMDK). While we focus on the currently available hardware, another interesting line of research considers possible extensions to PMem, such as extending the persistency domain to include CPU caches [10,20].

5 Conclusion

This is the first comprehensive evaluation of PMem on "real" (non prototype) hardware. In our evaluation, we found several guidelines for using PMem efficiently (cf. Sects. 2.3 and 2.2): (1) Instead of optimizing for cache lines (64 bytes) as on DRAM, we have to optimize for PMem blocks (256 bytes). (2) As in multi-threaded programming, writes to the same cache line in close temporal proximity should be avoided. (3) Forcing the data out of the on-CPU cache (using `clwb` or non-temporal stores) is essential for a high write bandwidth. (4) When using PMem and DRAM at the same time, there are interference effects cause significant slowdowns.

Furthermore, we proposed and evaluated algorithms for logging, page propagation, in-place updates, and interleaved execution of PMem writes:

(1) Our logging experiments have shown that latency-critical code should minimize the number of persistency barriers and avoid subsequent writes to the same cache line.
(2) Our *PopLog* algorithm reduces the required persistency barriers from two to one, thus doubling the throughput.
(3) For flushing database pages, a small log (μLog) can be used to flush only dirty cache lines. The I/O primitives introduced use an interface similar to the one in PMDK [43], making them widely applicable.
(4) We introduced Failure-Atomic Memory (FAM), which enables in-place updates with a single persistency barrier.
(5) We showed how cooperative multitasking (via coroutines) can be utilized to not only interleave loads but also stores on PMem.

Acknowledgements This work was supported by Fujitsu Laboratories LDT. and the DFG project KE401/22. Further, the authors would like to dearly and explicitly thank Satoshi Imamura, Kazuichi Oe, Mitsuru Sato, and Dieter Kasper from Fujitsu Laboratories for their continuous support. Lastly, we want to express our sincere gratitude to our personal friend Stefan Marcik for his keen insights into SIMD programming.

Funding Open Access funding provided by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

Appendix

A FAM implementation details

Listing 2 contains the AVX2 SIMD code for performing updates with the FAM algorithm for 16 bytes. For better readability the comments are provided inline.

Listing 2: **Failure-Atomic Memory (FAM) Implementation** – Pseudo code for updating a 16-byte FAM.

```
__m256i ADD, AND_1, AND_2, AND_3;
ADD = _mm256_set1_epi64x(0x4000000000000000L);
AND_1 = _mm256_set1_epi64x(0xC000000000000000L);
AND_2 = _mm256_set1_epi64x(0x3FFFFFFFFFFFFFFFL);
AND_3 = _mm_set1_epi32(0x7FFFFFFF);

Update128(int32_t* values) {
 __m128i input32, msbs32;
 __m256i input64, fambs64, version, new_states, both;

 // Load input 'values' expand to 256bit register
 input32 = _mm_loadu_si128((__m128i*) values);
 input32 = _mm_and_si128(input32, AND_3);
 input64 = _mm256_cvtepu32_epi64(input32);

 // Load FAMBs (fambs[0] stores the high bits)
 fambs64 = _mm256_loadu_si256((__m256i*) &fambs[1]);

 // Extract version bits
 version = _mm256_add_epi64(fambs64, ADD);
 version = _mm256_and_si256(version, AND_1);

 // Shift old state and write new state
 new_states = _mm256_slli_epi64(fambs64, 31);
 new_states = _mm256_and_si256(new_states, AND_2);
 new_states = _mm256_or_si256(new_states, input64);

 // Write
 both = _mm256_or_si256(version, new_states);
 _mm256_storeu_si256((__m256i *) &fambs[1], both);

 // Deal with most significant bits (msbs)
 msbs32 = _mm_castsi128_ps(input32);
 uint64_t msbs = _mm_movemask_ps(msbs32);

 // Update msb FAMB
 uint64_t famb0 = fambs[0];
 uint64_t new_version = (msbs + 1L<<62) & (3L<<62);
 uint64_t old_msbs = famb0<<31;
 fambs[0] = new_version | old_msbs | msbs;

 // Done: persist
 persist();
}
```

References

1. Andrei, M., Lemke, C., Radestock, G., Schulze, R., Thiel, C., Blanco, R., Meghlan, A., Sharique, M., Seifert, S., Vishnoi, S., Booss, D., Peh, T., Schreter, I., Thesing, W., Wagle, M., Willhalm, T.: SAP HANA adoption of non-volatile memory. PVLDB **10**(12), 1754–1765 (2017)
2. Arulraj, J., Levandoski, J.J., Minhas, U.F., Larson, P.: Bztree: a high-performance latch-free range index for non-volatile memory. PVLDB **11**(5), 553–565 (2018)
3. Arulraj, J., Pavlo, A.: How to build a non-volatile memory database management system. In: SIGMOD (2017)
4. Arulraj, J., Pavlo, A., Dulloor, S.: Let's talk about storage and recovery methods for non-volatile memory database systems. In: SIGMOD, pp. 707–722 (2015)
5. Arulraj, J., Pavlo, A., Malladi, K. T.: Multi-tier buffer management and storage system design for non-volatile memory. arXiv (2019)
6. Arulraj, J., Perron, M., Pavlo, A.: Write-behind logging. PVLDB **10**(4), 337–348 (2016)
7. Canim, M., Mihaila, G.A., Bhattacharjee, B., Ross, K.A., Lang, C.A.: SSD bufferpool extensions for database systems. PVLDB **3**(2), 1435–1446 (2010)
8. Chen, S., Ailamaki, A., Gibbons, P.B., Mowry, T.C.: Improving hash join performance through prefetching. ACM Trans. Database Syst. **32**, 17 (2007)
9. Chen, S., Jin, Q.: Persistent B+-trees in non-volatile main memory. PVLDB **8**(7), 786–797 (2015)
10. Cohen, N., Aksun, D.T., Avni, H., Larus, J.R.: Fine-grain checkpointing with in-cache-line logging. In: ASPLOS (2019)
11. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: SoCC, pp. 143–154 (2010)
12. Do, J., Zhang, D., Patel, J.M., DeWitt, D.J., Naughton, J.F., Halverson, A.: Turbocharging DBMS buffer pool using SSDs. In: SIGMOD (2011)
13. Dulloor, S.R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., Jackson, J.: System software for persistent memory. In: EuroSys (2014)
14. Fang, R., Hsiao, H., He, B., Mohan, C., Wang, Y.: High performance database logging using storage class memory. In: ICDE, pp. 1221–1231 (2011)
15. Götze, P., Baumann, S., Sattler, K.: An NVM-aware storage layout for analytical workloads. In: ICDE Workshops (2018)
16. Götze, P., van Renen, A., Lersch, L., Leis, V., Oukid, I.: Data management on non-volatile memory: a perspective. Datenbank-Spektrum **18**(3), 171–182 (2018)
17. Harizopoulos, S., Abadi, D.J., Madden, S., Stonebraker, M.: OLTP through the looking glass, and what we found there. In: SIGMOD, pp. 981–992 (2008)
18. Huang, J., Schwan, K., Qureshi, M.K.: NVRAM-aware logging in transaction systems. PVLDB **8**(4), 389–400 (2014)
19. Imamura, S., Yoshida, E.: The analysis of inter-process interference on a hybrid memory system. In: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region Workshops, HPCAsia2020, pp. 1–4. Association for Computing Machinery, New York (2020). https://doi.org/10.1145/3373271.3373272 **(ISBN: 9781450376501)**
20. Izraelevitz, J., Kelly, T., Kolli, A.: Failure-atomic persistent memory updates via JUSTDO logging. In: Conte, T., Zhou, Y. (eds.) ASPLOS (2016)
21. Izraelevitz, J., Yang, J., Zhang, L., Kim, J., Liu, X., Memaripour, A., Soh, Y.J., Wang, Z., Xu, Y., Dulloor, S.R., Zhao, J., Swanson, S.: Basic performance measurements of the intel optane DC persistent memory module. In: CoRR (2019)
22. Jonathan, C., Minhas, U.F., Hunter, J., Levandoski, J.J., Nishanov, G.V.: Exploiting coroutines to attack the "killer nanoseconds". In: PVLDB (2018)
23. Kang, W., Lee, S., Moon, B.: Flash as cache extension for online transactional workloads. VLDB J. **25**(5), 673–694 (2016)
24. Karnagel, T., Dementiev, R., Rajwar, R., Lai, K., Legler, T., Schlegel, B., Lehner, W.: Improving in-memory database index performance with intel transactional synchronization extensions. In: HPCA (2014)
25. Kimura, H.: FOEDUS: OLTP engine for a thousand cores and NVRAM. In: SIGMOD, pp. 691–706 (2015)
26. Koçberber, Y.O., Falsafi, B., Grot, B.: Asynchronous memory access chaining. In: PVLDB (2015)
27. Lee, S.K., Lim, K.H., Song, H., Nam, B., Noh, S.H.: WORT: write optimal radix tree for persistent memory storage systems. In: FAST, pp. 257–270 (2017)
28. Lee, S.K., Mohan, J., Kashyap, S., Kim, T., Chidambaram, V.: RECIPE: converting concurrent DRAM indexes to persistent-memory indexes. In: SOSP (2019)
29. Lersch, L., Hao, X., Oukid, I., Wang, T., Willhalm, T.: Evaluating persistent memory range indexes. In: PVLDB (2019)
30. Lersch, L., Lehner, W., Oukid, I.: Persistent buffer management with optimistic consistency. In: DaMoN (2019)
31. Liu, M., Zhang, M., Chen, K., Qian, X., Wu, Y., Zheng, W., Ren, J.: Dudetm: building durable transactions with decoupling for persistent memory. In: ASPLOS (2017)
32. Liu, X., Salem, K.: Hybrid storage management for database systems. PVLDB **6**(8), 541–552 (2013)
33. Luo, T., Lee, R., Mesnier, M.P., Chen, F., Zhang, X.: hStorage-DB: heterogeneity-aware data management to exploit the full capability of hybrid storage systems. PVLDB **5**(10), 1076–1087 (2012)
34. Memaripour, A., Badam, A., Phanishayee, A., Zhou, Y., Alagappan, R., Strauss, K., Swanson, S.: Atomic in-place updates for non-volatile main memories with kamino-tx. In: EuroSys (2017)
35. Oukid, I., Booss, D., Lehner, W., Bumbulis, P., Willhalm, T.: SOFORT: a hybrid SCM-DRAM storage engine for fast data recovery. In: DaMoN (2014)
36. Oukid, I., Booss, D., Lespinasse, A., Lehner, W., Willhalm, T., Gomes, G.: Memory management techniques for large-scale persistent-main-memory systems. In: PVLDB (2017)
37. Oukid, I., Lasperas, J., Nica, A., Willhalm, T., Lehner, W.: FPTree: a hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In: SIGMOD, pp. 371–386 (2016)
38. Oukid, I., Lehner, W.: Data structure engineering for byte-addressable non-volatile memory. In: SIGMOD (2017)
39. Oukid, I., Lehner, W., Kissinger, T., Willhalm, T., Bumbulis, P.: Instant recovery for main memory databases. In: CIDR (2015)
40. Oukid, I., Lersch, L.: On the diversity of memory and storage technologies. Datenbank-Spektrum **18**(2), 121–127 (2018)
41. Oukid, I., Nica, A., Bossle, D.D.S., Lehner, W., Bumbulis, P., Willhalm, T.: Adaptive recovery for SCM-enabled databases. In: ADMS (2017)
42. Pelley, S., Wenisch, T.F., Gold, B.T., Bridge, B.: Storage management in the NVRAM era. In: PVLDB (2013)
43. PMDK.: Persistent memory development kit. http://www.pmem.io. Accessed 26 03 2019
44. Psaropoulos, G., Oukid, I. Legler, T., May, N., Ailamaki, A.: Bridging the latency gap between NVM and DRAM for latency-bound operations. In: DaMoN (2019)
45. Sauer, C.: Modern techniques for transaction-oriented database recovery. PhD thesis, Kaiserslautern University of Technology, Germany (2017)
46. van Renen, A., Leis, V., Kemper, A., Neumann, T., Hashida, T., Oe, K., Doi, Y., Harada, L., Sato, M.: Managing non-volatile memory in database systems. In: SIGMOD (2018)

47. van Renen, A., Vogel, L., Leis, V., Neumann, T., Kemper, A.: Persistent memory I/O primitives. In: DaMoN (2019)
48. Venkataraman, S., Tolia, N., Ranganathan, P., Campbell, R.H.: Consistent and durable data structures for non-volatile byte-addressable memory. In: FAST, pp. 61–75 (2011)
49. Volos, H., Tack, A.J., Swift, M.M.: Mnemosyne: lightweight persistent memory. In: ASPLOS (2011)
50. Wang, T., Johnson, R.: Scalable logging through emerging non-volatile memory. PVLDB **7**(10), 865–876 (2014)
51. Xia, F., Jiang, D., Xiong, J., Sun, N.: Hikv: a hybrid index key-value store for DRAM-NVM memory systems. In: USENIX ATC, pp. 349–362 (2017)
52. Yang, J., Wei, Q., Chen, C., Wang, C., Yong, K.L., He, B.: NV-tree: reducing consistency cost for NVM-based single level systems. In: FAST, pp. 167–181 (2015)
53. Zhou, X., Shou, L., Chen, K., Hu, W., Chen, G.: DPTree: differential indexing for persistent memory. In: PVLDB (2019)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Bibliography

[AJ89] R. Agrawal and H. V. Jagadish. *Recovery Algorithms for Database Machines with Non-Volatile Main Memory*, page 269–285. 1989.

[AKW+13] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, Alexander Driskill-Smith, and Mohamad Krounbi. Spin-transfer torque magnetic random access memory (STT-MRAM). *ACM J. Emerg. Technol. Comput. Syst.*, 2013.

[Alc] Paul Alcorn. Intel optane DIMM pricing. https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html. Accessed: 2021-06-08.

[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. *VLDB*, 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. *VLDB*, 2017.

[APD15] Joy Arulraj, Andrew Pavlo, and Subramanya Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *SIGMOD*, 2015.

[APM19] Joy Arulraj, Andy Pavlo, and Krishna Teja Malladi. Multi-tier buffer management and storage system design for non-volatile memory. *CoRR*, 2019.

[APP16] Joy Arulraj, Matthew Perron, and Andrew Pavlo. Write-behind logging. *VLDB*, 2016.

[BBS16] Stephan Baumann, Peter A. Boncz, and Kai-Uwe Sattler. Bitwise dimensional co-clustering for analytical workloads. *VLDB J.*, 2016.

[BC16] Hans-Juergen Boehm and Dhruva R. Chakrabarti. Persistence programming models for non-volatile memory. In *SIGPLAN*, 2016.

[BGJS21] Alexander Baumstark, Philipp Götze, Muhammad Attahir Jibril, and Kai-Uwe Sattler. Instant graph query recovery on persistent memory. In *DaMoN*, 2021.

[BHC+13] Katelin A. Bailey, Peter Hornyack, Luis Ceze, Steven D. Gribble, and Henry M. Levy. Exploring storage class memory with key value stores. In *INFLOW*, 2013.

[BM70] Rudolf Bayer and Edward McCreight. Organization and maintenance of large ordered indices. In *SIGFIDET*, 1970.

[BZN05] Peter A. Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, 2005.

[CCA+] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. nv-heaps.

[CCV15] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. REWIND: recovery write-ahead system for in-memory non-volatile data-structures. *VLDB*, 2015.

[CG21] Sakib Chowdhury and Wojciech M. Golab. A scalable recoverable skip list for persistent memory. In *SPAA*, 2021.

[CGN11] Shimin Chen, Phillip B. Gibbons, and Suman Nath. Rethinking database algorithms for phase change memory. In *CIDR*, 2011.

[CJ15] Shimin Chen and Qin Jin. Persistent B+-trees in non-volatile main memory. *VLDB*, 2015.

[CLZ+21] Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. Scalable persistent memory file system with kernel-userspace collaboration. In *USENIX FAST*, 2021.

[CNF+09] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin C. Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *SOSP*, 2009.

[Com79] Douglas Comer. The ubiquitous b-tree. *ACM Comput. Surv.*, 1979.

[DAP+] Justin A. DeBrabant, Joy Arulraj, Andrew Pavlo, Michael Stonebraker, Stanley B. Zdonik, and Subramanya Dulloor. A prolegomenon on OLTP database systems for non-volatile memory. In *ADMS, year = 2014,*.

[DDGZ18] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. In *USENIX ATC*, 2018.

[DHK+15] Biplob Debnath, Alireza Haghdoost, Asim Kadav, Mohammed G. Khatib, and Cristian Ungureanu. Revisiting hash table design for phase change memory. In *INFLOW*, 2015.

[DLCL21] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. Fast, flexible, and comprehensive bug detection for persistent memory programs. In Tim Sherwood, Emery Berger, and Christos Kozyrakis, editors, *ASPLOS*, 2021.

[dra07] Process integration, devices, and structures. international technology roadmap for semiconductors. 2007.

[DS10] Alexander Driskill-Smith. Latest advances and future prospects of STT-RAM. In *Non-Volatile Memories Workshop*, 2010.

[Dul16] Subramanya R. Dulloor. *Systems and applications for persistent memory*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 2016.

[FBW+20] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. Nvtraverse: in NVRAM data structures, the destination is more important than the journey. In *PLDI*, 2020.

[FCP+11] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA database: data management for modern business applications. *SIGMOD*, 2011.

[FHH+11] Ru Fang, Hui-I Hsiao, Bin He, C. Mohan, and Yun Wang. High performance database logging using storage class memory. In *ICDE*, 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.*, 2012.

[Fog] Agner Fog. Instruction tables. https://www.agner.org/optimize/instruction_tables.pdf. Accessed: 2021-06-14.

[GBS18] Philipp Götze, Stephan Baumann, and Kai-Uwe Sattler. An NVM-aware storage layout for analytical workloads. In *ICDE*, 2018.

[gdb] GDB: The GNU project debugger. https://www.gnu.org/software/gdb/. Accessed: 2021-06-20.

[GDV13] Ellis Giles, Kshitij A. Doshi, and Peter J. Varman. Bridging the programming gap between persistent and volatile memory using wrap. In *Computing Frontiers Conference*, 2013.

[GKK12] Goetz Graefe, Hideaki Kimura, and Harumi A. Kuno. Foster b-trees. *ACM TODS*, 2012.

[GKL20] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. Understanding the idiosyncrasies of real persistent memory. *VLDB*, 2020.

[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 *SIGMOD*, 1987.

[GTS20] Philipp Götze, Arun Kumar Tharanatha, and Kai-Uwe Sattler. Data structure primitives on persistent memory: an evaluation. In *DaMoN*, 2020.

[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. *VLDB*, 2014.

[GvRL+18] Philipp Götze, Alexander van Renen, Lucas Lersch, Viktor Leis, and Ismail Oukid. Data management on non-volatile memory: A perspective. *Datenbank-Spektrum*, 2018.

[HAMS08] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, 2008.

[HCW+21] Daokun Hu, Zhiwen Chen, Jianbing Wu, Jianhua Sun, and Hao Chen. Persistent memory hash indexes: An experimental evaluation. *VLDB*, 2021.

[HFP02] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *DISC*, 2002.

[HHL20] Gabriel Haas, Michael Haubenschild, and Viktor Leis. Exploiting directly-attached NVMe arrays in DBMS. In *CIDR*, 2020.

[HKWN18] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *FAST*, 2018.

[HLWO20] Xiangpeng Hao, Lucas Lersch, Tianzheng Wang, and Ismail Oukid. PiBench online: Interactive benchmarking of persistent memory indexes. *PVLDB*, 2020.

[HR10] Heather Hanson and Karthick Rajamani. What computer architects need to know about memory throttling. In *Computer Architecture - ISCA 2010 International Workshops A4MMC, AMAS-BT, EAMA, WEED, WIOSCA*, Lecture Notes in Computer Science, pages 233–242. Springer, 2010.

[HS08] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[IGN+12] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 2012.

[IKK16] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via JUSTDO logging. In *ASPLOS*, 2016.

[Inta] Intel. Intel intrinsics guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/. Accessed: 2021-06-20.

[Intb] Intel. Product brief: Intel optane dc persistent memory. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-dc-persistent-memory-brief.pdf. Accessed: 2021-03-26.

[Intc] Intel. Product brief: Intel optane dc persistent memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html. Accessed: 2021-09-10.

[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. *arXiv*, 2019. Accessed: 2021-05-27.

[JBGS21] Muhammad Attahir Jibril, Alexander Baumstark, Philipp Götze, and Kai-Uwe Sattler. JIT happens: Transactional graph processing in persistent memory meets just-in-time compilation. In *EDBT*, 2021.

[JED] JEDEC. JEDEC announces support for nvdimm hybrid memory modules. https://www.jedec.org/news/pressreleases/jedec-announces-support-nvdimm-hybrid-memory-modules. Accessed: 2021-09-09.

[KBG+18] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redesigning lsms for non-volatile memory with novelsm. In *USENIX ATC*, 2018.

[KFH+17] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas E. Anderson. Strata: A cross media file system. In *SOSP*, 2017.

[Kim15] Hideaki Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *SIGMOD*, 2015.

[KLN+19] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young-ri Choi. SLM-DB: single-level key-value store with persistent memory. In Arif Merchant and Hakim Weatherspoon, editors, *FAST*, 2019.

[KN11] Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *ICDE*, 2011.

[KPS+16] Aasheesh Kolli, Steven Pelley, Ali G. Saidi, Peter M. Chen, and Thomas F. Wenisch. High-performance transactions for persistent memories. In *ASPLOS*, 2016.

[KTB+19] Marcel Köppen, Jana Traue, Christoph Borchert, Jörg Nolte, and Olaf Spinczyk. Cache-line transactions: Building blocks for persistent kernel data structures enabled by AspectC++. In *SOSP*, 2019.

[LC97] David E. Lowell and Peter M. Chen. Free transactions with rio vista. In *SOSP*, 1997.

[LC19] Jihang Liu and Shimin Chen. Initial experience with 3d xpoint main memory. In *ICDE*, 2019.

[LCK+20] Jihwan Lee, Won Gi Choi, Doyoung Kim, Hanseung Sung, and Sanghyun Park. TLSM: tiered log-structured merge-tree utilizing non-volatile memory. *IEEE Access*, 2020.

[LCW20] Jihang Liu, Shimin Chen, and Lujun Wang. Lb+-trees: Optimizing persistent index performance on 3dxpoint memory. *VLDB*, 2020.

[Ler21] Lucas Lersch. *Leveraging Non-Volatile Memory in Modern Storage Management Architectures*. PhD thesis, Dresden University of Technology, Germany, 2021.

[LHKN18] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. Leanstore: In-memory data management beyond main memory. In *ICDE*, 2018.

[LHO+19] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. Evaluating persistent memory range indexes. *VLDB*, 2019.

[LHWL20] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable hashing on persistent memory. *VLDB*, 2020.

[LIMB09] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *ISCA*, 2009.

[Liu17] Qingyue Liu. *Ouroboros wear-leveling: a two-level hierarchical wear-leveling model for NVRAM*. PhD thesis, 2017.

[LKN13] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *ICDE*, 2013.

[LLO19] Lucas Lersch, Wolfgang Lehner, and Ismail Oukid. Persistent buffer management with optimistic consistency. In Thomas Neumann and Ken Salem, editors, *DaMoN*, 2019.

[LLS13] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. The bw-tree: A b-tree for new hardware platforms. In *ICDE*, 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 *USENIX FAST*, 2017.

[LMK+19] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: converting concurrent DRAM indexes to persistent-memory indexes. In *SOSP*, 2019.

[LOLS17] Lucas Lersch, Ismail Oukid, Wolfgang Lehner, and Ivan Schreter. An analysis of LSM caching in NVRAM. In *DaMoN*, 2017.

[Lom18] David B. Lomet. Cost/performance in modern data stores: how data caching systems succeed. In *DaMoN*, 2018.

[Lom19] David B. Lomet. Cost/performance in modern data stores: How data caching systems succeed. In *ICDE*, 2019.

[LOSL17] Lucas Lersch, Ismail Oukid, Ivan Schreter, and Wolfgang Lehner. Rethinking DRAM caching for lsms in an NVRAM environment. In *ADBIS*, 2017.

[LSOL20] Lucas Lersch, Ivan Schreter, Ismail Oukid, and Wolfgang Lehner. Enabling low tail latency on multicore key-value stores. *PVLDB*, 2020.

[McC] John C. McCallum. Memory prices 1957+. `https://jcmit.net/memoryprice.htm`. Accessed: 2021-06-08.

[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 Trans. Database Syst.*, 1992.

[MKM12] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In Pascal Felber, Frank Bellosa, and Herbert Bos, editors, *EuroSys*, 2012.

[NCC+19] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H. Noh, and Beomseok Nam. FAST. 2019.

[NF20] Thomas Neumann and Michael J. Freitag. Umbra: A disk-based system with in-memory performance. In *CIDR*, 2020.

[NH12] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. In *ASPLOS*, 2012.

[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 *DISC*, 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 *DaMoN*, 2014.

[OBLL16] Ismail Oukid, Daniel Booss, Adrien Lespinasse, and Wolfgang Lehner. On testing persistent-memory-based software. In *DaMoN*, 2016.

[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 *SIGMOD*, 2016.

[Ouk18] Ismail Oukid. *Architectural Principles for Database Systems on Storage-Class Memory*. PhD thesis, Dresden University of Technology, Germany, 2018.

[Ouk19] Ismail Oukid. Architectural principles for database systems on storage-class memory. In *BTW*, 2019.

[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 *CIDR*, 2017.

[PMD] PMDK: Persistent memory development kit. http://www.pmem.io. Accessed: 2021-06-20.

[QKF+09] Moinuddin K. Qureshi, John P. Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis A. Lastras, and Bülent Abali. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *(MICRO-42*, 2009.

[QSR] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *ISCA), year = 2009,*.

[RBB+08] Simone Raoux, Geoffrey W. Burr, Matthew J. Breitwisch, Charles T. Rettner, Yi-Chou Chen, Robert M. Shelby, Martin Salinga, Daniel Krebs, Shih-Hung Chen, Hsiang-Lan Lung, and Chung Hon Lam. Phase-change random access memory: A scalable technology. *IBM J. Res. Dev.*, 52, 2008.

[RKK+14] Dulloor Subramanya Rao, Sanjay Kumar, Anil S. Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *EuroSys*, 2014.

[SDUP15] David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. Nvc-hashmap: A persistent and concurrent hashmap for non-volatile memories. In *IMDM*, 2015.

[SSSW08] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. The missing memristor found. *nature*, 453, 2008.

[Sto] Michael Stonebraker. How hardware drives the shape of databases to come. https://www.nextplatform.com/2017/08/15/hardware-drives-shape-databases-come/. Accessed: 2017-11-02.

[TD] Universität Trier and Schloss Dagstuhl. Digital bibliography & library project. https://dblp.org/. Accessed: 2021-08-01.

[val] Valgrind. https://www.valgrind.org/. Accessed: 2021-06-20.

[VMCL15] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li. Quartz: A lightweight performance emulator for persistent memory software. In *Proceedings of the 16th Annual Middleware Conference*, 2015.

[vN45] John von Neumann. First draft of a report on the EDVAC. 1945.

[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 *SIGMOD*, 2018.

[vRVL+19] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. Persistent memory I/O primitives. In *DaMoN*, 2019.

[vRVL+20] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. Building blocks for persistent memory. *VLDB J.*, 2020.

[VTRC11] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *USENIX FAST*, 2011.

[VTS11] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *ASPLOS*, 2011.

[WAA19] Kan Wu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Towards an unwritten contract of intel optane SSD. In Daniel Peek and Gala Yadgar, editors, *USENIX HotStorage*, 2019.

[WC08] R. Clint Whaley and Anthony M. Castaldo. Achieving accurate and context-sensitive timing for code optimization. *Softw. Pract. Exp.*, 2008.

[Weba] Intel Website. Intel optane technology delivers new levels of endurance. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-technology/delivering-new-levels-of-endurance-article-brief.html. Accessed: 2021-06-10.

[Webb] SQLite Website. Most widely deployed and used database engine. https://sqlite.org/mostdeployed.html. Accessed: 2020-09-30.

[WLL18] Tianzheng Wang, Justin J. Levandoski, and Per-Åke Larson. Easy lock-free indexing in non-volatile memory. In *ICDE*, 2018.

[WLZ+21] Haitao Wang, Zhanhuai Li, Xiao Zhang, Xiaonan Zhao, and Song Jiang. WOBTree: a write-optimized b+-tree for non-volatile memory. *Frontiers Comput. Sci.*, 2021.

[WQR13] XiaoJian Wu, Sheng Qiu, and A. L. Narasimha Reddy. SCMFS: A file system for storage class memory and its extensions. *ACM Trans. Storage*, 2013.

[WRK+10] H.-S. Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P. Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E. Goodson. Phase change memory. *Proc. IEEE*, 2010.

[XJXS17] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. HiKV: A hybrid index key-value store for DRAM-NVM memory systems. In Dilma Da Silva and Bryan Ford, editors, *USENIX ATC*, 2017.

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

[XZM+17] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. NOVA-Fortis: A fault-tolerant non-volatile main memory file system. In *SOSP*, 2017.

[YCJ+21] Baoyue Yan, Xuntao Cheng, Bo Jiang, Shibin Chen, Canfang Shang, Jianying Wang, Kenry Huang, Xinjun Yang, Wei Cao, and Feifei Li. Revisiting the design of lsm-tree based OLTP storage engine with persistent memory. *VLDB*, 2021.

[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 Sam H. Noh and Brent Welch, editors, *USENIX FAST*, 2020.

[You07] Matt T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE Computer Society, 2007.

[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 *USENIX FAST*, 2015.

[ZAPC21] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. Spitfire: A three-tier buffer manager for volatile and non-volatile memory. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD*, 2021.

[ZH18] Pengfei Zuo and Yu Hua. A write-friendly and cache-optimized hashing scheme for non-volatile memory systems. *IEEE Trans. Parallel Distributed Syst.*, 2018.

[ZHW18] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *OSDI*, 2018.

[ZLJW20] Junchen Zhang, Yongping Luo, Peiquan Jin, and Shouhong Wan. Optimizing adaptive radix trees for nvm-based hybrid memory architecture. In *IEEE International Conference on Big Data*, 2020.

[ZSC+19] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. Dptree: Differential indexing for persistent memory. *Proc. VLDB Endow.*, 2019.