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

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.

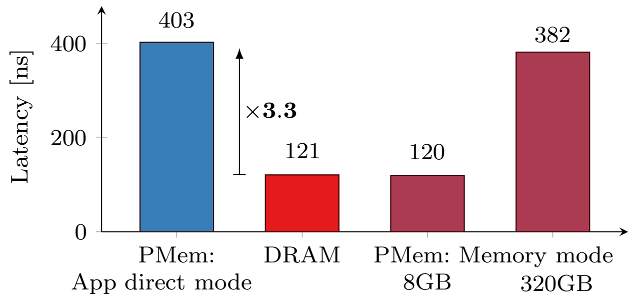


Fig. 1 Read latency—random access read latency

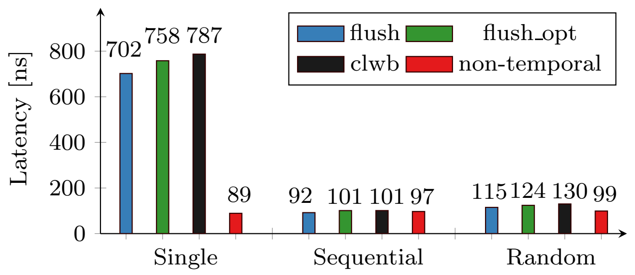


Fig. 2 Persistent write latency—access latency for writing cache lines persistently

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.

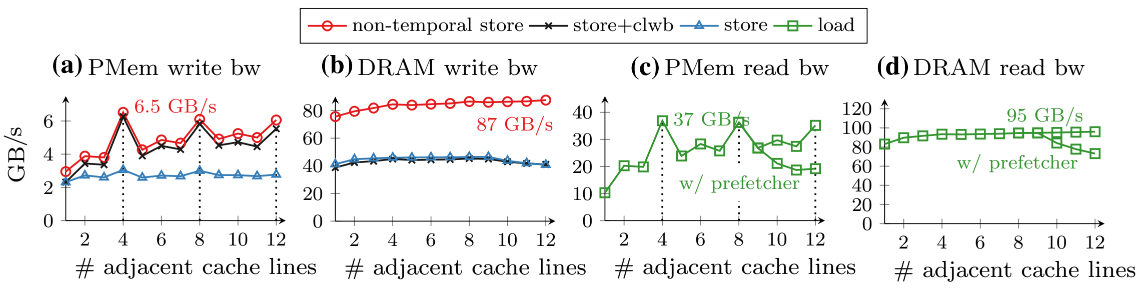


Fig. 3 PMem bandwidth: varying access granularity—PMem bandwidth (**a**, **c**) with 24 threads compared to DRAM bandwidth (**b**, **d**) with a varying number of adjacently accessed cache lines. We use a random access pattern that allows for out-of-order execution

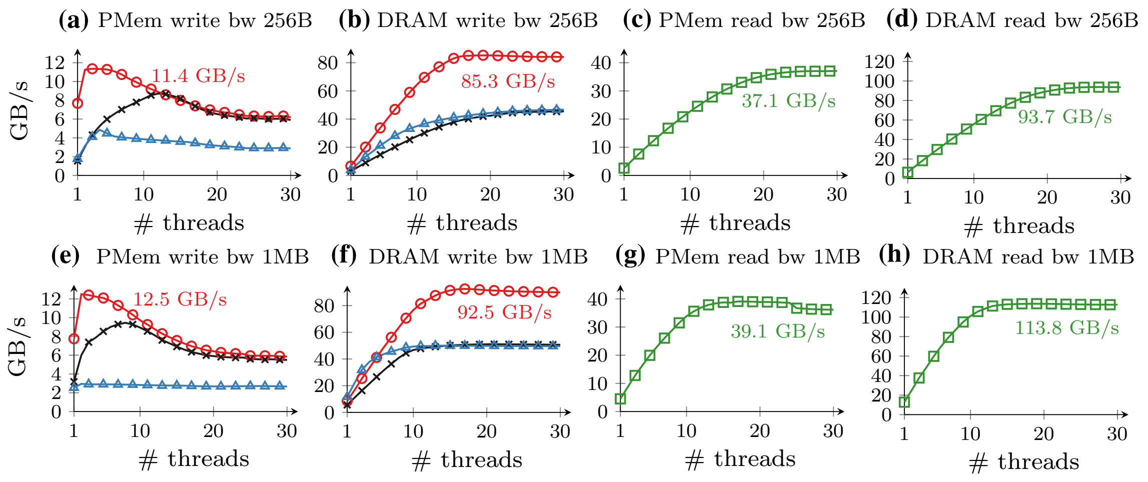


Fig. 4 PMem bandwidth: varying thread count—PMem bandwidth (**a**, **c**, **e**, **g**) compared to DRAM bandwidth (**b**, **d**, **f**, **h**) for 256-byte (4 adjacent cache lines) and 1 MB blocks with an increasing number of threads. We use a random access pattern that allows for out-of-order execution

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 over-saturation 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.

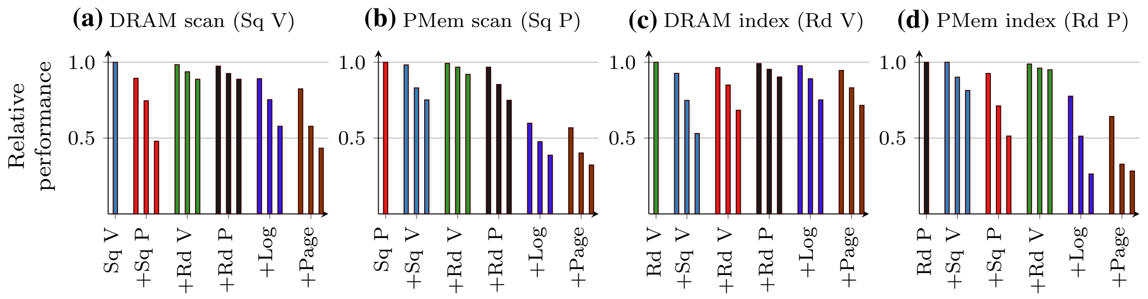


Fig. 5 PMem versus DRAM interference—the relative slowdown of sequential scans (Sq) and random accesses (Rd) on DRAM (V) and PMem (P) when executed together with other sequential scans, random lookups, log writing (Log), and page propagation (Page) with 1, 5, and 10 threads

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:

```
struct Buffer {      |  void append(Buffer* buf,
  int eles[128]      |              int ele) {
  int next           |   buf->eles[buf->next]=ele
}                    |   clwb(&buf->eles[buf->next])
                     |   sfence()
                     |   buf->next++
                     |   clwb(&buf->next)
                     |   sfence()
                     |  }
```

[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.

[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.

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

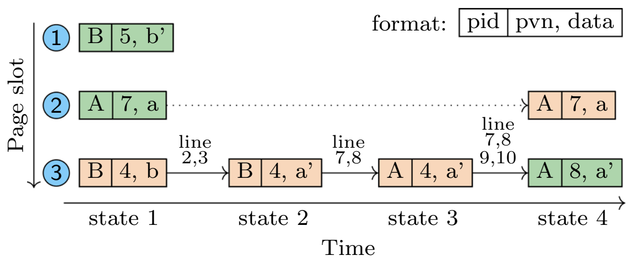


Fig. 6 CoW page propagation—the flush process is optimized by using a page version number (*pvn*) to avoid invalidating pages, which would require an additional persistency barrier

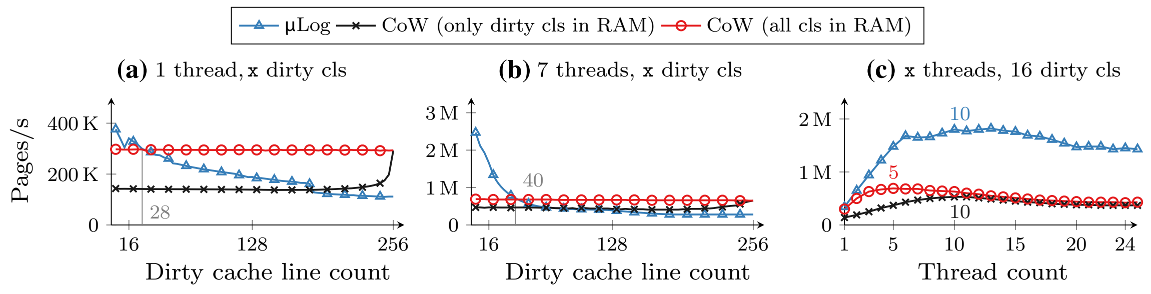


Fig. 7 Failure atomic page flush—flushing 16 kB pages (256cache lines*each*) in a failure atomic way from DRAM to PMem

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 (-o-) or only the dirty ones (-x-). 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 (-x-). The throughput for *CoW* (-o-) 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 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.

```
log << header << payload  |  LogFile -> Entry*
persist(log)              |  Entry -> header ↩
log << footer             |           payload ↩
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:

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

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

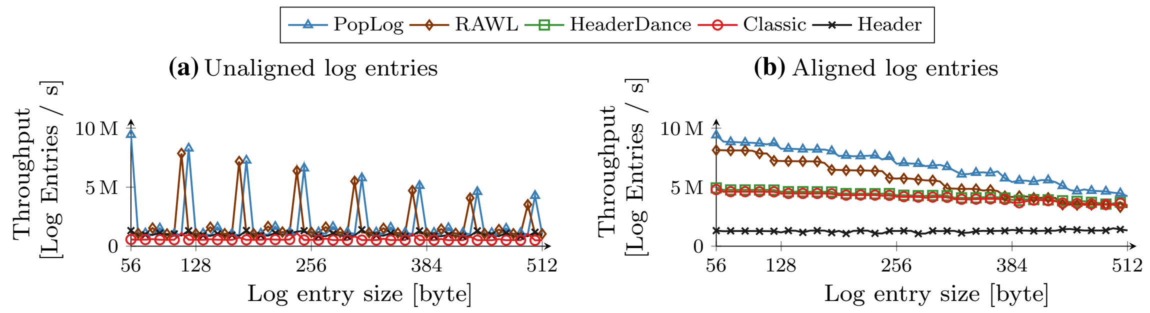


Fig. 8 Transaction log—the throughput for writing log entries of varying size to PMem

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:

```
cnt = pop_count(header,    |  LogFile -> Entry*
                payload)   |  Entry -> header ↩
log << header              |  pop_cnt payload
    << cnt << payload      |
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:

```
struct CowBased { |  void update(char[16] new) {
  bool active     |   if(active) {
  char a[16]      |     b = new; persist(b)
  char b[16]      |   } else {
}                 |     a = new; persist(a)
                  |   }
                  |   active = !active
                  |   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.

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

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

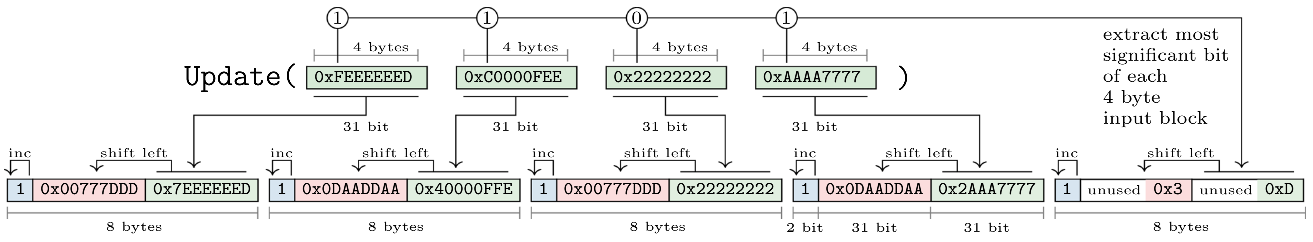


Fig. 9 In-place updates with a single persistency barrier—the input (top) is split into 31-bit blocks and stored in 8-byte blocks (bottom), which are of which stores the previous (red) and new (green) state. In case of a crash, the version number (blue) can be used to recover the old state

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

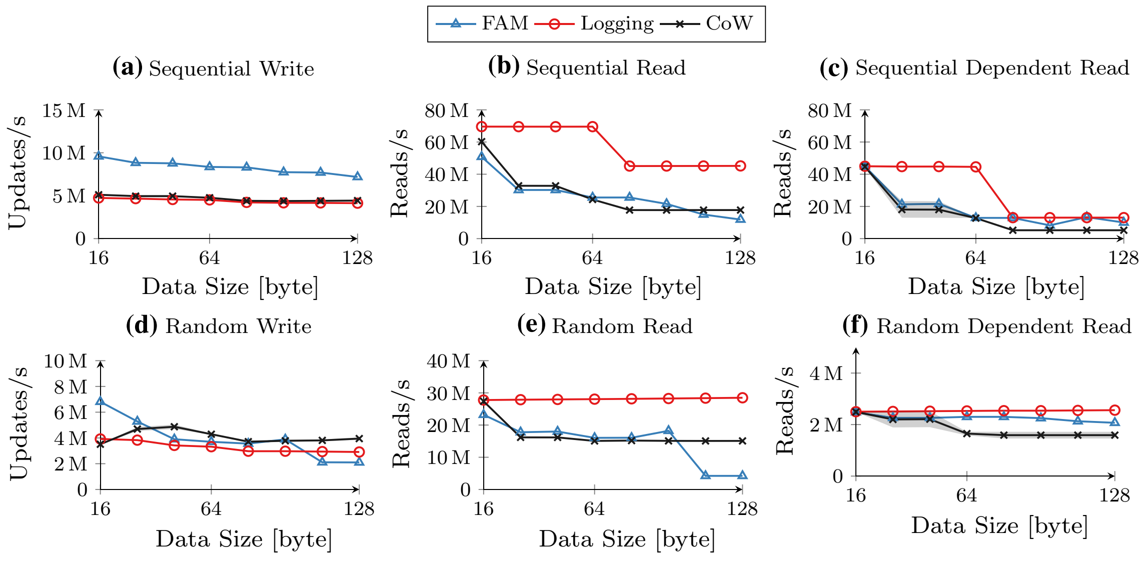


Fig. 10 In-place updates—performance of Failure-Atomic Memory (FAM) in comparison with CoW- and log-based updates and reads. While the lines show the average throughput, the highlighted areas (mostly in (c) and (f)) indicate the 95% percentile over multiple runs

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 ($2\times$ for sequential and $1.6\times$ 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

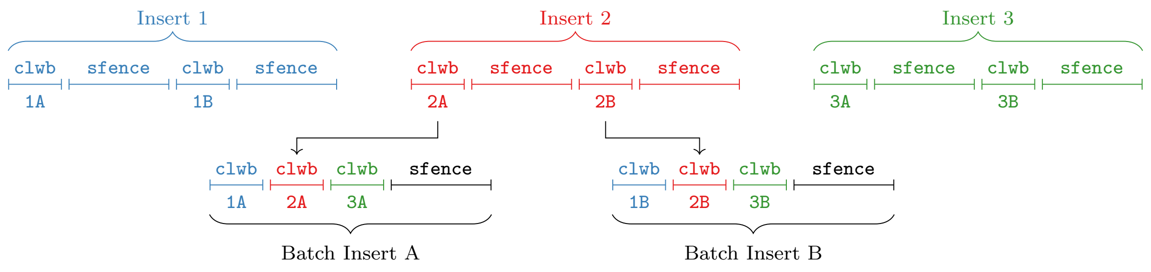


Fig. 11 Coroutines: interleaved inserts—by interleaving *n* write operation (`clwb`) and sharing one synchronization barrier (`sfence`), the number of memory stalls can be reduced from *n* to 1

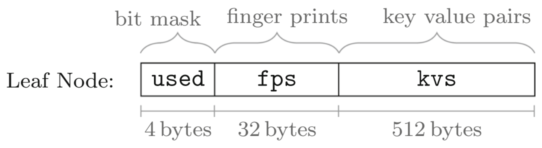
execution is realized, the advantage is that the program can make progress while waiting on a slow I/O operation.

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):

```
Val lookup(k) {           |  Val lookup(k) {
 fp = hash(k)             |   fp = hash(k)
                          |   _mm_prefetch(used)
                          |   co_await sched
                          |
 for(s=0; s<32; s++) {    |   for(s=0; s<32; s++) {
  ok = used[s]            |    ok = used[s]
  if(ok&&fps[s]==fp) {    |    if(ok&&fps[s]==fp) {
                          |    _mm_prefetch(kvs[s])
                          |    co_await sched
                          |
    if(k == kvs[s].k) {   |     if(k == kvs[s].k) {
     return kvs[s].v      |      return kvs[s].v
}}}}                      | }}}}
```

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

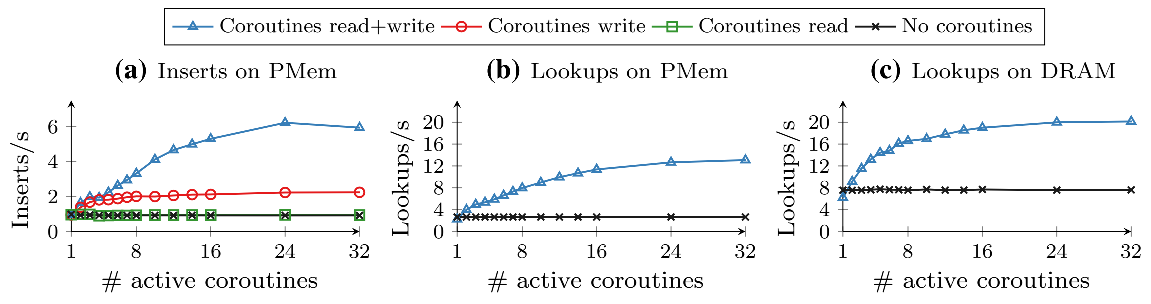


Fig. 12 Coroutines on PMem—we use coroutines to hide read and write latencies of PMem for lookups and inserts on FP-Tree leaf nodes

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.

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)