Persistent Memory I/O Primitives

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

Lukas Vogel
Technische Universität München
vogell@in.tum.de

Viktor Leis
Friedrich-Schiller-Universität Jena
viktor.leis@uni-jena.de

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

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

ABSTRACT

I/O latency and throughput is one of the major performance bottlenecks for disk-based database systems. Upcoming 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 one of the first performance evaluations of PMem in terms of bandwidth and latency. Based on the results, we develop guidelines for efficient PMem usage and two essential I/O primitives tuned for PMem: log writing and block flushing.

ACM Reference Format:
Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Persistent Memory I/O Primitives. In *International Workshop on Data Management on New Hardware (DaMoN'19), July 1, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3329785.3329930

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 about to shrink this fundamental gap between memory and storage. Specifically, Intel's upcoming *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. We also expect the price, capacity, and latency to lie 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. In this work, we show how to efficiently implement atomic log writing and page flushing—two critical I/O primitives for database systems. While we perform our evaluation in a database context, these two I/O primitives are transferable to other systems, as evidenced by the fact that they are also implemented by the Persistent Memory Development Kit (PMDK) [1]. The results reported are based on a prototype of Intel's Optane DC PMM rather than software or hardware-based emulation. Our contributions can be summarized as follows:

- We provide one of the first analyses of PMem on a prototype of Intel's Optane DC PMM. We highlight the impact of the physical properties of PMem on software and derive guidelines for efficient usage of PMem.
- 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.
- 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.

2 PMEM CHARACTERISTICS

In this section, we first describe how we configured our system before presenting latency and bandwidth results.

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 ≈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.

DaMoN'19, July 1, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-6801-8/19/07…$15.00
https://doi.org/10.1145/3329785.3329930

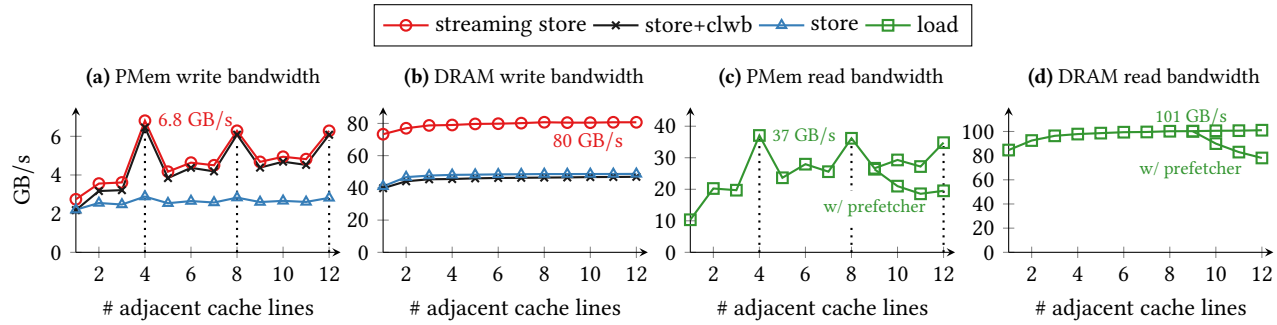


Figure 1: PMem Bandwidth: Varying Access Granularity – PMem bandwidth (**a**, **c**) with 24 threads compared to DRAM bandwidth (**b**, **d**) with a varying number of adjacently accessed cache lines. We use a random access pattern that allows for out-of-order execution.

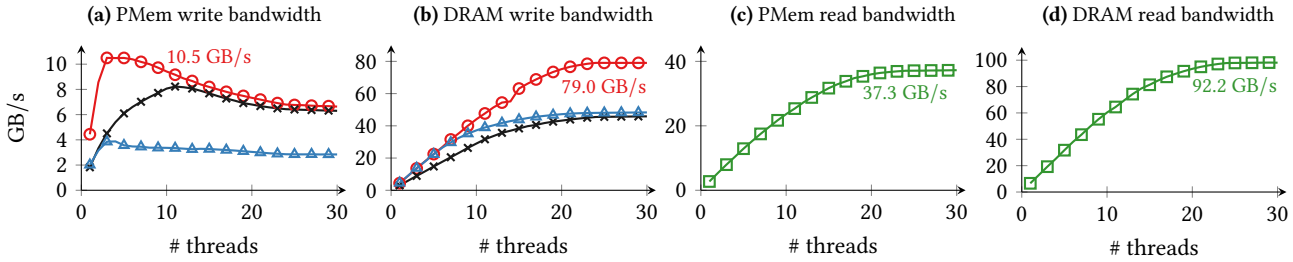


Figure 2: PMem Bandwidth: Varying Thread Count – PMem bandwidth (**a**, **c**) compared to DRAM bandwidth (**b**, **d**) for 4 adjacent cache lines with an increasing number of threads. We use a random access pattern that allows for out-of-order execution.

To access PMem, the physical PMem DIMMs first have to be grouped into so-called *regions* with `ipmctl`[1]:

```
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`[2] 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`[3]) and mount it (`mount`[4]) 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 `mmap`[5]:

```
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.

[1] **`ipmctl`:** https://github.com/intel/ipmctl
[2] **`ndctl`:** https://github.com/pmem/ndctl
[3] **`mkfs.ext4`:** https://linux.die.net/man/8/mkfs.ext4
[4] **`mount`:** https://linux.die.net/man/8/mount
[5] **`mmap`:** http://man7.org/linux/man-pages/man2/mmap.2.html

2.2 Bandwidth

It is important to know that the PMem hardware works internally 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 byte (cache lines).

The block-based (4 cache lines) design of PMem leads to some interesting performance characteristics that we show in Figure 1. The experiment measures the bandwidth for loading/storing from/to independent random locations on PMem and DRAM. We use all 24 physical cores of one socket 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 byte) is used.

As on DRAM, streaming (non-temporal) stores are more efficient on PMem because the modified cache lines do not have to be loaded first—thereby saving memory bandwidth. However, on PMem the performance of regular stores can be increased to that of streaming stores by issuing a `clwb` (cache line write back) instruction after each store. The `clwb` forces a dirty cache line in the data cache to be written to the underlying memory system (without evicting the cache line). While this is beneficial on PMem **(a)**, it does not change the throughput on DRAM **(b)**.

This effect is studied further in Figure 2, which shows the same experiment, but instead of varying the number of cache lines loaded/stored we vary the number of threads. It shows that the `clwb`

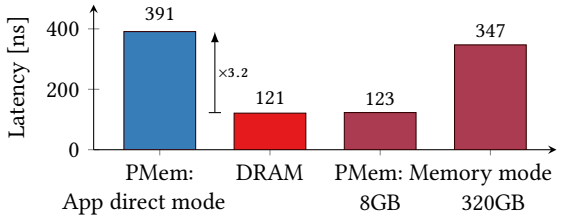


Figure 3: Read Latency – Random access read latency.

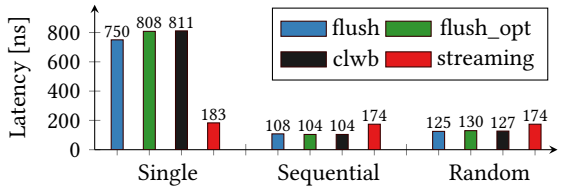


Figure 4: Persistent Write Latency – Access latency for writing cache lines persistently.

instruction only becomes necessary once several threads are writing to PMem: With more threads, 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. It seems that at a certain point (≈ 4 threads), the buffer is no longer able to combine the cache lines into a single PMem block write. Using the `clwb` instruction, we can force the order in which the cache lines arrive at the PMem write buffer and thus enable it to combine neighboring cache lines into a single block write.

Another effect we observe is that the throughput peaks at around 3 threads for streaming (and 12 for stores with `clwb`). Using additional threads decreases the throughput slightly.

Lastly, a largely unrelated but somewhat amusing effect of the hardware pre-fetcher is shown in Figure 1 **(c)** and **(d)**. Starting at 10 adjacent cache lines, the pre-fetcher becomes active and fetches additional cache lines. If these are not needed, as in our experiment, the effective throughput suffers.

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 byte) but on PMem blocks (256 byte).
- Streaming operations should be utilized when possible, otherwise stores should be followed by `clwb`.
- Over-saturating PMem can lead to reduced performance.
- The experiments showed that the PMem read bandwidth is 2.6× lower and the write bandwidth 7.5× lower than DRAM. Therefore, performance-critical code should prefer DRAM over PMem (e.g., by buffering writes in a DRAM cache).

2.3 Latency

While bandwidth is critical for OLAP-style applications, latency is much more important for OLTP workloads because the access pattern shifts from large scan operations to (sequential I/O) to point lookups, which are essentially random accesses into memory. The performance of these random accesses is dominated by the latency of the underlying device.

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 $i-1$. The results are shown in Figure 3.

We can observe that DRAM read latency is lower than PMem by a factor of 3.2. Note that this does not mean that each access to PMem is that much slower, because many applications can still benefit from the regular on-CPU L3 cache. 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 8 GB (as in the other modes) 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 (around 200 GB on the socket we use) is not hit that frequently and the performance degrades.

To store data persistently on PMem, the data has to be written, 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 Section 3.1. To measure the latency for persistent store operations 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 byte) boundary. The results are shown in Figure 4.

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 streaming stores).

When data is written to the same cache line, streaming 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 non-streaming instructions, 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, streaming operations and `clwb` should be preferred over `flush` and `flush_opt`.

3 STORAGE PRIMITIVES FOR PMEM

The low write latency of PMem (compared to other storage devices) makes it an ideal candidate for use in database systems, file systems, and other systems software. However, due to the CPU cache, writes to PMem are only persistent 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 Section 3.1. Intel's Persistent Memory Development Kit (PMDK) [1], an open-source library for Pmem, abstracts from this complexity by providing two failure atomic I/O primitives: log writing (*libpmemlog*) and block/page flushing (*libpmemblk*). In Section 3.3 and Section 3.2, we apply the guidelines developed earlier (Section 2), apply them to these two problems, and analyze their performance.

3.1 Failure Atomicity

As mentioned earlier, when data is written to PMem, stores are not immediately propagated to the PMem device, but instead buffered in the regular on-CPU cache. While programs cannot prevent the eviction, they can force it using explicit write-back or flush instructions. This implies that any persistent data structure on PMem always needs to be in a consistent state, 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 {
  int eles[128];
  int next;
};
```

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

The new element is first copied into the next free slot and the corresponding cache line is forced to be written back to PMem. Instead of using a regular flush operation, `clwb` (cache line write back) is used, which is an efficient flush operation designed for PMem that flushes the cache line without invalidating it. Before the buffer's size indicator (`next`) can be changed, a `sfence` (store fence) must be issued to prevent re-ordering by the compiler or hardware. Once `next` has been written, it is persisted to memory in the same fashion. 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). In the following, 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 speaking, a persistency barrier is an expensive operation, 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 Section 2, it is also important to minimize the number of persistency barriers while still maintaining failure atomicity. In the following two sections, we show a manually-tuned implementation for logging and page flushing.

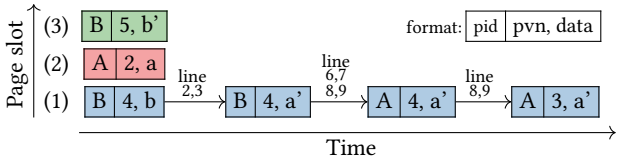
3.2 Page Propagation

Besides logging, the other essential storage engine component that requires I/O is the buffer manager. It is responsible for loading (swapping in) pages from 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 is evicted and has been modified, 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 corrupt data). In addition, flushing a page needs to be failure atomic: After a crash, the recovery component needs a consistent snapshot of the page.

Flushing pages to persistent storage is an inherently I/O-bound task. To reduce the latency for pages requests, the buffer manager constantly flushes dirty pages to persistent storage in the background. This way, it can always serve requests without needing to flush a page first. In addition, this makes flushing pages (on 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, the buffer pool becomes optional. However, as recent work [6, 33] 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. In order to integrate PMem into existing systems, the page flushing algorithm needs to be correct (failure atomicity) and efficient (high bandwidth). In the following, we describe two algorithms for failure atomic page flushing and then evaluate them.

3.2.1 Copy-on-Write. **CoW** does not overwrite the original PMem page, but instead writes the DRAM page to an unused PMem page [5] (left-hand side of Listing 1, line 1-3). Once the new PMem page is persisted, it is marked as valid (line 5-10) 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. By adding a page version number (*pvn*) that is increased after each flush, we can identify the latest version of a page. Using the *pvn*, it becomes unnecessary to invalidate the old PMem page before writing the new one. This lowers the number of required persistency barriers from three to two and thus yields ≈ 10% increased throughput. We illustrate the *pvn* in the following example:



The green page slot (3) contains the latest persistent copy of page `B`. The red one (2) contains the original version of page `A`. The different versions of the blue page slot ((1)) show each step of flushing a new version of page `A`. The line numbers where the transition might occur are written over the arrow. In each step, the *pvn* can be used to figure out the most recent version of each page. In database systems, the log sequence number *lsn* could be used instead of the *pvn*, however if the system crashes in line 6, log entries might be reapplied to a page.

3.2.2 Micro Log. The micro-log technique 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).

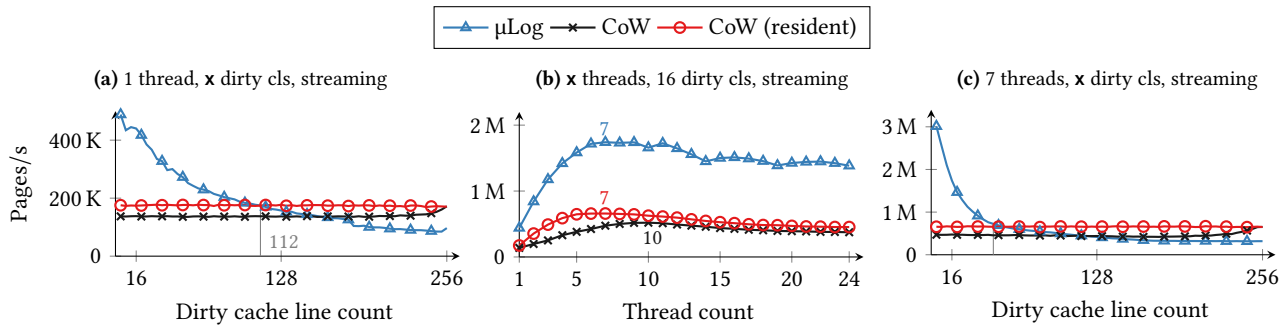


Figure 5: Failure Atomic Page Flush – Flushing 16 kB pages (256 cache lines each) in a failure atomic way from DRAM to PMem.

Listing 1: Failure Atomicity – Pseudo code to flush a DRAM page (page_v) to a PMem page (page_nv). CoW: left, µLog: right.

```
// 1. Write data
page_nv.data = page_v.data;
persist(page_nv);
// 2. Make PMem page valid
page_nv.pid = page_v.pid;
sfence();
page_nv.pvn = page_v.pvn;
persist(page_nv.pid
      , page_nv.pvn);
```

```
// 1. Invalidate µlog
µlog.pid = INVALID;
persist(log.pid);
// 2. Write to µlog
µlog <- page_v.dirty_cls
persist(µlog);
// 3. Set µlog valid
µlog.pid = page_v.pid;
persist(µlog.pid);
// 4. Write to page
page_nv <- page_v.dirty_cls
persist(page_nv);
```

3.2.3 Experiments. Figure 5 details the page flush performance. All techniques are implemented as a micro benchmark using streaming (also known as non-temporal) writes, which have been shown to provide the highest throughput in Section 2. When using copy-on-write, 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 **(c)** for 7 threads. In **(b)**, we vary the number of threads to show the scale-out behavior.

The results show that the micro log is efficient 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 micro log yields performance gains for up to 112 dirty cache lines. A multi-threaded experiment is shown in **(c)**. Here the micro log only offers throughput gains, when fewer than 32 cache lines are dirty. 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 micro benchmarks in Section 2 suggested that streaming instructions should be preferred over regular stores. We were able to confirm this finding in the page flushing experiment (not shown in chart). 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.

3.3 Logging

In database systems, write-ahead logging is used to ensure the atomicity and durability of transactions. This 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 rollback. If any of the 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 (thereby guaranteeing to the user, that all changes of the transaction are durable), all log entries of the transaction are written persistently. 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 constitutes a major performance bottleneck in database systems using traditional storage devices (SSD/HDD) because 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.

3.3.1 Algorithms. In the following, we first explain and then evaluate three logging techniques: *Classic*, *Header*, and *Zero*:

Classic represents a form of logging commonly used in database systems [32]. 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
persist(log);
log << footer
persist(log);
```

```
LogFile -> Entry*
Entry -> header ⟷
         payload 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.

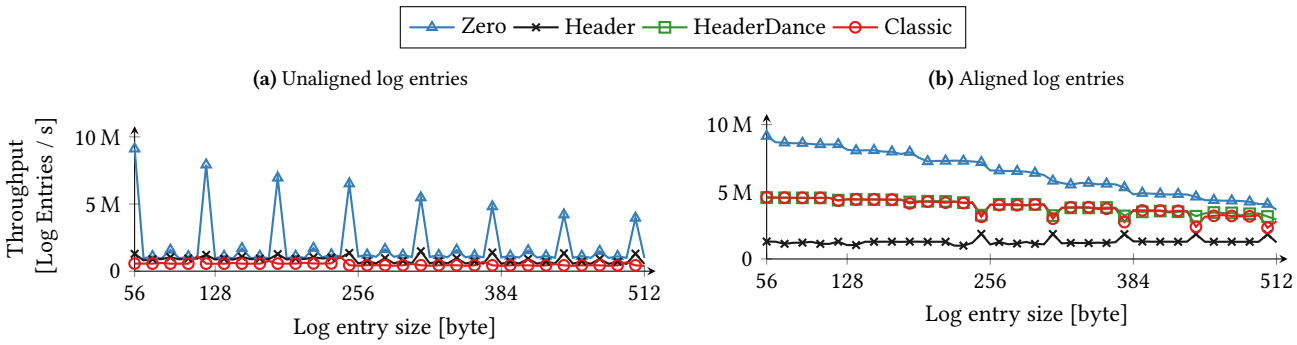


Figure 6: Transaction Log – The throughput for writing log entries of varying size to PMem.

Header uses the same technique as *libpmemlog* in the PMDK [1]. It is similar to appending elements to an array:

```
log << header << payload
persist(log);
log.size += entry_size
persist(log.size);
```

```
LogFile -> size, Entry*
Entry -> header 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.

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

```
cnt = pop_count(header, payload)
log << header << cnt << payload
persist(log);
```

```
LogFile -> Entry*
Entry -> header ⟷
         pop_cnt payload
```

Before logging starts, each log file is initialized to zero. This is commonly done anyway by database systems (e.g., PostgreSQL) to enforce that the file system actually allocates pages to the file. 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.

3.3.2 Experiments. In Section 2.3, 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 Figure 6. 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. While padding wastes some memory[6], the throughput greatly increases (≈ 8×).

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 *Zero* logging (≈ 2× faster).

The log implementation (*libpmemlog*) of the PMDK [1] uses the same approach (and therefore yields the same throughput, when locking is disabled) as our naive *Header* implementation without alignment and dancing. 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 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 [33]. Running a write-heavy (100%) YCSB benchmark [10] on a single thread with a DRAM-resident table, *Zero* logging, *Header*, and *Classic* achieves a throughput of 2 M, 1.7 M, and 1.5 M transactions per second, respectively.

4 RELATED WORK

With PMem only being released recently, this is one of the two [17] 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 [4, 25, 27, 29]. The number of persistent index structures [3, 9, 14, 21, 34, 37] is large, and has been summarized by Götze et. al [15]. Similar techniques have been used to build storage engines directly on PMem [5, 24]. These approaches use in-place updates on PMem, which suffers from the lower-than-DRAM performance. Therefore, a number of indexes [26, 36] as well as storage engines [2, 8, 11, 18, 19, 22, 23] integrate PMem as a separate storage layer or an extension to the recovery component [28, 30]. Furthermore, buffer-managed architectures [6, 20, 33] have been proposed to use PMem more adaptively. Recovery has always been an essential (and performance-critical) component of database systems [32]. Several designs have been proposed for database-specific logging [7, 13, 16, 31, 35] and file systems [12].

[6] Up to 1 cache line for *Zero* and *Header*; up to 2 cache lines for *Classic*

5 CONCLUSION

In our evaluation, we found several guidelines for using PMem efficiently (cf. Section 2.2 and 2.3): (1) Instead of optimizing for cache lines (64 byte) as on DRAM, we have to optimize for PMem blocks (256 byte). (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 (`clwb` or streaming) is essential for a high write bandwidth. Furthermore, we evaluated algorithms for logging and page propagation: (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 *zero* logging 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 [1], making them widely applicable.

REFERENCES

[1] PMDK: Persistent memory development kit. http://www.pmem.io. Accessed: 2019-03-26.

[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 and A. Pavlo. How to build a non-volatile memory database management system. In *SIGMOD*, 2017.

[5] 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.

[6] J. Arulraj, A. Pavlo, and K. T. Malladi. Multi-tier buffer management and storage system design for non-volatile memory. *arXiv*, 2019.

[7] J. Arulraj, M. Perron, and A. Pavlo. Write-behind logging. *PVLDB*, 10(4):337–348, 2016.

[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] S. Chen and Q. Jin. Persistent B+-trees in non-volatile main memory. *PVLDB*, 8(7):786–797, 2015.

[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154, 2010.

[11] J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson. Turbocharging DBMS buffer pool using SSDs. In *SIGMOD*, 2011.

[12] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *EuroSys*, 2014.

[13] 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.

[14] P. Götze, S. Baumann, and K. Sattler. An NVM-aware storage layout for analytical workloads. In *ICDE Workshops*, 2018.

[15] P. Götze, A. van Renen, L. Lersch, V. Leis, and I. Oukid. Data management on non-volatile memory: A perspective. *Datenbank-Spektrum*, 18(3), 2018.

[16] J. Huang, K. Schwan, and M. K. Qureshi. NVRAM-aware logging in transaction systems. *PVLDB*, 8(4):389–400, 2014.

[17] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, 2019.

[18] W. Kang, S. Lee, and B. Moon. Flash as cache extension for online transactional workloads. *VLDB Journal*, 25(5):673–694, 2016.

[19] 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.

[20] H. Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *SIGMOD*, pages 691–706, 2015.

[21] 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.

[22] X. Liu and K. Salem. Hybrid storage management for database systems. *PVLDB*, 6(8):541–552, 2013.

[23] 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.

[24] 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.

[25] I. Oukid, D. Booss, A. Lespinasse, W. Lehner, T. Willhalm, and G. Gomes. Memory management techniques for large-scale persistent-main-memory systems. *PVLDB*, 2017.

[26] 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.

[27] I. Oukid and W. Lehner. Data structure engineering for byte-addressable non-volatile memory. In *SIGMOD*, 2017.

[28] I. Oukid, W. Lehner, T. Kissinger, T. Willhalm, and P. Bumbulis. Instant recovery for main memory databases. In *CIDR*, 2015.

[29] I. Oukid and L. Lersch. On the diversity of memory and storage technologies. *Datenbank-Spektrum*, 18(2), 2018.

[30] I. Oukid, A. Nica, D. D. S. Bossle, W. Lehner, P. Bumbulis, and T. Willhalm. Adaptive recovery for scm-enabled databases. In *ADMS*, 2017.

[31] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge. Storage management in the NVRAM era. *PVLDB*, 2013.

[32] C. Sauer. *Modern techniques for transaction-oriented database recovery*. PhD thesis, Kaiserslautern University of Technology, Germany, 2017.

[33] A. van Renen, V. Leis, A. Kemper, T. Neumann, T. Hashida, K. Oe, Y. Doi, L. Harada, and M. Sato. Managing non-volatile memory in database systems. In *SIGMOD*, 2018.

[34] 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.

[35] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *PVLDB*, 7(10):865–876, 2014.

[36] 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.

[37] 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.