Drop It In Like It's Hot: An Analysis of Persistent Memory as a Drop-in Replacement for NVMe SSDs

Maximilian Böther, Otto Kißig, Lawrence Benson, Tilmann Rabl
Hasso Plattner Institute, University of Potsdam
{maximilian.boether,otto.kissig}@student.hpi.de,{lawrence.benson,tilmann.rabl}@hpi.de

ABSTRACT

Solid-state drives (SSDs) have improved database system performance significantly due to the higher bandwidth that they provide over traditional hard disk drives. Persistent memory (PMem) is a new storage technology that offers DRAM-like speed at SSD-like capacity. Due to its byte-addressability, research has mainly treated PMem as a replacement of, or an addition to DRAM, e.g., by proposing highly-optimized, DRAM-PMem-hybrid data structures and system designs. However, PMem can also be used via a regular file system interface and standard Linux I/O operations. In this paper, we analyze PMem as a drop-in replacement for Non-Volatile Memory Express (NVMe) SSDs and evaluate possible performance gains while requiring no or only minor changes to existing applications. This drop-in approach speeds-up database systems like `Postgres`, without requiring any code changes. We systematically evaluate PMem and NVMe SSDs in three database microbenchmarks and the widely used TPC-H benchmark on `Postgres`. Our experiments show that PMem outperforms a RAID of four NVMe SSDs in read-intensive OLAP workloads by up to 4x without any modifications while achieving similar performance in write-intensive workloads. Finally, we give four practical insights to aid decision-making on when to use PMem as an SSD drop-in replacement and how to optimize for it.

ACM Reference Format:
Maximilian Böther, Otto Kißig, Lawrence Benson, Tilmann Rabl. 2021. Drop It In Like It's Hot: An Analysis of Persistent Memory as a Drop-in Replacement for NVMe SSDs. In *International Workshop on Data Management on New Hardware (DAMON'21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3465998.3466010

1 INTRODUCTION

Today's data-driven industry relies heavily on databases that efficiently process huge amounts of data. In the last decade, database deployments shifted from hard disk drives to flash-based storage (SSD), following the trend of faster available storage at high capacity. More recently, Non-Volatile Memory Express (NVMe) SSDs achieve even higher performance and are making their way into modern database deployments to enable even more data-intensive processing. However, the emergence of persistent memory (PMem) makes a new and fast storage technology available. PMem provides persistent data storage at SSD-like capacity while achieving close-to-DRAM performance. Many enterprise deployments rely on the stability of well-established database management systems like `Postgres`. These systems put stability and reliability over adoption of the latest research results. Thus, instead of changing the algorithms or storage layout, one way to speed up enterprise deployments without requiring code changes is exchanging the underlying hardware. PMem as the underlying storage layer has the potential to cause a new major database performance shift.

Recent work mostly views PMem as a *supplement* to DRAM, between SSDs and DRAM in the memory hierarchy [8, 22, 43]. The commercial release of Intel® Optane™ DC Persistent Memory in 2019 has enabled the first performance studies of real-world persistent memory DIMMs [11, 18, 49]. Many PMem-aware data structures that utilize PMem as a replacement or addition to DRAM have been proposed [8, 21, 23, 25, 29, 44]. In contrast, in this paper we evaluate PMem as a fast storage technology that may replace NVMe SSDs, not DRAM, for storing large data volumes in databases.

Recent performance studies observe the high bandwidth PMem offers, reaching up to 40 GiB/s for sequential reads [11]. Modern NVMe SSDs on the other hand are physically limited at 7.3 GiB/s by the maximum supported four PCIe 4.0 lanes. In light of this significant difference, we provide insights into how PMem performs when used as a drop-in replacement for SSD storage in typical database management workloads. To this end, we interact with PMem via a filesystem interface and standard Linux I/O operations. To provide results in more realistic system setups than previous work [43, 47], we evaluate six DIMMs and up to four NVMe SSDs in our benchmarks, which allows for higher degrees of parallelism in both storage technologies.

In this paper, we compare PMem and NVMe SSDs in three database workload microbenchmarks, *table scan*, *buffer management*, and *logging*. We then show the impact of PMem in a real database system by running TPC-H on `Postgres` with PMem and NVMe SSDs as the storage layer. We discuss our findings and propose how to integrate PMem into existing systems and how to optimize for it when used as an SSD drop-in replacement. In summary, we make the following contributions:

1) We evaluate realistic PMem and NVMe SSD setups in common database workloads.
2) We run TPC-H on `Postgres` with PMem and NVMe SSDs as the storage medium to show the impact of a PMem drop-in replacement in real database systems and workloads.
3) We summarize our findings in four practical insights to aid decision-making on when to use PMem as an SSD drop-in replacement and how to optimize for it.

DAMON'21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8556-5/21/06…$15.00
https://doi.org/10.1145/3465998.3466010

The remainder of this paper is structured as follows. In Section 2, we introduce PMem and NVMe SSDs. We evaluate the performance of PMem and NVMes in database workload microbenchmarks (Section 3) and the TPC-H benchmark (Section 4). We discuss our findings in Section 5 and our related work in Section 6, before concluding in Section 7.

2 BACKGROUND

In this paper, we compare *persistent memory* (PMem) and *Non-Volatile Memory Express Solid-State Drives* (NVMe SSDs). Both of these storage media are a type of *non-volatile memory*, which retain data even without power. In the following, we give a clear conceptual clarification of both technologies.

2.1 Persistent Memory

Persistent memory is the currently established term for non-volatile main memory. Previously, it has also been called storage class memory (SCM) or non-volatile memory (NVM/NVRAM), which frequently led to confusion, as PMem is not the only type of non-volatile memory. Unlike other non-volatile memory technologies, e.g., SSDs, PMem is byte-addressable like DRAM, but also persistent. The first widely available persistent memory technology currently available is Intel® Optane™ DC Persistent Memory, which comes in DIMMs in sizes of up to 512 GiB. Optane is the brand name for the 3D XPoint *non-volatile memory* technology, which is a general non-volatile memory medium. Based on this, Intel released both Optane persistent memory modules as well as Optane SSDs. In this paper, we refer to Optane DC Persistent Memory as *PMem*.

Other PMem products are in development, which do not build on 3D XPoint technology, e.g., Fujitsu and Nantero plan to release PMem modules based on Nano-RAM (NRAM) technology, which utilizes carbon nanotubes [24, 26], and there are other technologies, like phase-change memory [19, 36, 50], resistive RAM [4], and MRAM [13]. Because of this future PMem could largely differ in its observed behavior and even embedding into the system.

PMem DIMMs are connected to the CPUs integrated memory controllers (iMCs) via memory channels, like regular DRAM. The iMC keeps a read and write pending queue (RPQ/WPQ) of operations, which mark the beginning of the asynchronous DRAM refresh domain, in which persistence is guaranteed. Optane memory has an internal granularity of 256 B, while the CPU has a cache-line granularity of 64 B, which causes write amplification for stores smaller than 256 B. The DIMMs are commonly configured interleaved, striping the data in 4 KiB blocks, to enable parallel access.

We employ the *App Direct Mode* of the DIMMs, which enables direct access (*dax*) to PMem, either using a character-device (*devdax*) or using a mounted filesystem (*fsdax*). PMem can also be used in *Memory Mode*, in which it provides a transparent extension of DRAM without persistency guarantees. The widespread filesystems ext4 and xfs are dax-aware, skipping the kernel I/O layer when `mmap`ing a region of or reading from a file stored on an fsdax device. Data access is possible on various levels of abstraction, from standard I/O functions (`open`, `read`, `write`), over memory-mapping operations (`mmap`, `msync`), to Intel ISA additions in order to explicitly flush cache lines (`clwb`, `ntstore`, `sfence`). An explicit call to ensure data persistence is necessary and depends on the type of I/O used (`fsync`, `msync`, `clwb`).

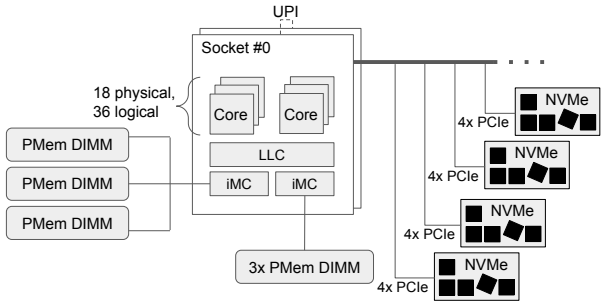

Figure 1: Overview of our system.

2.2 NVMe SSDs

SSDs are secondary storage devices that do not require mechanical components. These storage devices are commonly based on NAND flash chips. However, some newer SSDs use other media instead of flash, e.g., 3D XPoint. SSDs can be connected to the system using different interfaces, for example, NVM Express (NVMe), which is a register-level host-controller interface specifically optimized for accessing SSDs via the PCIe standard [16, 28]. NVMe SSDs are successors of the classical SATA SSDs, which commonly rely on AHCI as the host-controller interface. The physical connection of NVMe SSDs is not limited to PCIe slots and is oftentimes realized via M.2 connectors, a form factor standard for physical device layouts.

The NVMe interface allows for better exploitation of parallelism of the storage medium. Internally, SSDs feature built-in controllers, buffers, and caches, requiring complex access logic. NAND-based SSDs are organized in blocks, that are sub-divided in pages, with a common page size varying between 2 KiB to 16 KiB, and common block sizes from 256 KiB to 4 MiB [15]. From an operating system perspective, SSDs are block devices, not offering byte-addressable access. SSDs are inherently parallel, allowing them to serve multiple requests at once and reducing their sensitivity to access patterns [6, 7]. Typically, they are accessed using standard I/O functions or memory-mapping and, like PMem, require an explicit function call to ensure persistence (`fsync`, `msync`).

3 WORKLOAD MICROBENCHMARKS

To gain a better understanding of PMem as a drop-in replacement for NVMe SSDs, we run three workload microbenchmarks that reflect basic database operators. These are a table scan, a buffer management workload, and a logging workload. The workloads cover both sequential and random read and write tasks and, thus, provide a good basis for database-relevant performance.

3.1 Setup and Methodology

We show our system setup in Figure 1. All of our experiments are conducted on a server with two Intel Xeon Gold 5220S processors with 18 cores (36 logical cores in total with hyperthreading). We use hyperthreading and bind all experiments to one socket via `numactl` to avoid cross-socket access. We use Intel DC P4610 1.6 TiB SSDs, which are TLC 3D NAND NVMe SSDs, and Intel Optane DC

Table 1: Comparison of PMem and NVMe in our server.

	R/W Bandwidth GiB/s Sequential	Random	Capacity	$/GiB	Access Modes
PMem	40 / 12	30 / 9	750 GiB	5.43	FS, Block, Char
NVMe	3.2 / 2.1	2.6 / 0.9	1500 GiB	0.36	FS, Block

Persistent Memory 128 GiB modules. All NVMes are connected to the same socket using 4 PCIe lanes each. We use `fio` [3] to verify that there is no performance difference between the NUMA nodes when accessing the drives. PMem DIMMs are connected to the CPU via memory channels. Additionally, we benchmark a *pseudo-RAID 0* of four NVMes, in which data is striped across the NVMes, similar to interleaved PMem. We do not utilize a regular RAID 0 for the microbenchmarks as our approach gives us more fine-grained control on how data is distributed. To enable a drop-in replacement of SSDs, we access PMem using *fsdax*. The server runs Ubuntu 20.04 LTS with kernel 5.4.

Similar to `fio` [3], we set a `POSIX_FADV_DONTNEED` hint to prevent measuring cached data in DRAM. We randomly initialize all of the buffer files and randomly scramble the files between two runs, to prevent caching. It is not sufficient to rely on the `O_DIRECT` flag to prevent caching, as it has no effect on dax-enabled storage devices like PMem, and it disables prefetching mechanisms, which we analyze separately. Scrambling provides a clean system between runs, but enables the operating system to optimize within a single run. We run all benchmarks in our custom benchmark framework, which is publicly available on GitHub[1].

3.2 Hardware Comparison

To gain an intuition about performance of NVMe SSDs and PMem as used in our server, we compare one NVMe with a full PMem setup in Table 1. For NVMe, the values are based on the hardware specification [17]. As there is no official performance specification for fully-stocked PMem, we provide the peak performance of previous work [11, 18, 49]. With respect to bandwidth, we see that PMem is an order of magnitude faster than NVMe, for both sequential and random speeds. As discussed in Section 2.1, as PMem is byte-addressable, it can also be accessed as a character device. The performance of NVMe scales with additional devices while PMem requires additional CPUs to scale further. Last, we find that the price of PMem is an order of magnitude higher than that of the NVMe SSD, but note that pricing information for PMem is not stable yet, due to limited market availability [1].

3.3 Table Scan

The table scan workload sequentially reads a 5 GiB file per thread, similar to iterating over rows in a table. To reflect OS and common SSD page sizes, we test 4 KiB and 16 KiB pages with standard Linux I/O methods, i.e., `read/write`, as well as with `mmap`. We load all pages to the same DRAM buffer comprising a single page (4 KiB/16 KiB) to allow for L2 caching and to avoid unnecessary data replication in DRAM. This behavior is similar to the table scan implementation in `Postgres` [35]. In the following, we first give performance numbers, discuss the NVMe and PMem results, and compare Linux I/O to `mmap`.

Results. The results are shown in Figure 2. For all configurations, PMem outperforms the NVMes. With PMem, we reach a peak performance of 31.2 GiB/s when using Linux I/O with 64 threads and 16 KiB pages, which is equivalent to 2 Mops/s (million page operations per second). The NVMe pseudo-RAID reaches 11.2 GiB/s using Linux I/O with 64 threads, for both 4 KiB (3 Mops/s) and 16 KiB pages (734 Kops/s), while a single NVMe peaks at 2.8 GiB/s (184 Kops/s for 16 KiB pages and 734 Kops/s for 4 KiB pages) in the same setting. This means that PMem is more than 11x faster than a single NVMe.

NVMe Discussion. The pseudo-RAID needs at least 16 threads to achieve linear scaling over the single NVMe in the same setting. We do not directly assign threads to drives and access all drives in a round-robin fashion, which requires more threads to fully utilize the RAID. Furthermore, as a single NVMe scales with an increasing thread count, the NVMes benefit from the pseudo-RAID setup only once a single NVMe is saturated.

For the NVMes, we note that the Linux kernel prefetches data, which can be tuned using the `fadvise` syscall. Disabling prefetching (`FADV_RANDOM`) lowers the peak performance of the pseudo-RAID to 4.9 GiB/s and of the single NVMe to 2.5 GiB/s. Especially the pseudo-RAID configuration benefits from kernel prefetching. However, enforcing stronger prefetching (`FADV_SEQUENTIAL`) does not increase peak performance and increases performance in other settings by less than 0.5 GiB/s.

PMem Discussion. Compared to previous work [11, 49], we observe lower PMem peak bandwidth. Daase et al. measure ~40 GiB/s peak performance for sequential reads, which we can replicate on our platform using their benchmark tool. The reason that we measure only 31.2 GiB/s is that previous work uses manually unrolled, aligned, non-temporal AVX-512 `vmovntdqa` instructions to move data from PMem into CPU registers only, while we read data from PMem to DRAM using standard Linux/C++ methods. For Linux I/O, the `read` syscall uses the `__memcpy_mcsafe` kernel function to copy data from PMem to DRAM, which employs standard, non AVX-512, `movq` instructions. For `mmap`, we use the standard library `memcpy` function, which, even when hardcoding the page size, compiles to unaligned, and non-unrolled `vmovdq` instructions, as implemented in the glibc `__memmove_avx_unaligned` function. Also, Daase et al. observe slightly lower bandwidth pinning to NUMA nodes instead of individual cores. Thus, a combination of the above reasons causes a bandwidth gap when using common system functions.

Our results show that copying data from PMem to DRAM does not fully saturate PMem's bandwidth. Thus, we observe better performance for more threads and larger page sizes, which contrasts previous work, that observes that larger page sizes and more threads do not improve the performance of highly optimized sequential reads [11, 49]. This shows that insights gained with highly-optimized benchmarks are not necessarily transferable to real-world systems and parameters need to be carefully evaluated for the workload at hand. Lastly, as PMem is accessed directly (DAX) without paging to DRAM, `fadvise` syscalls have no effect.

Linux I/O vs `mmap`. There is a small difference between using Linux I/O and `mmap`. For 4 KiB pages, Linux I/O is a little faster

[1] `https://github.com/hpides/pmem-nvme-dropin`

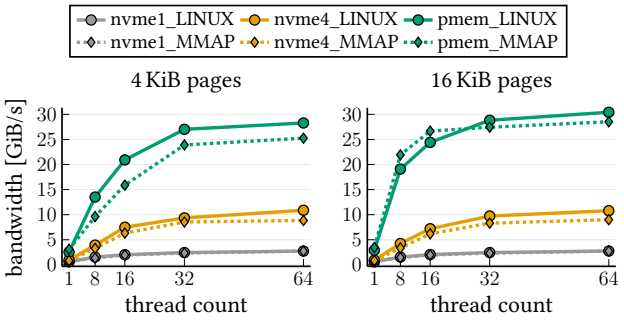

Figure 2: Bandwidth for the table scan workload.

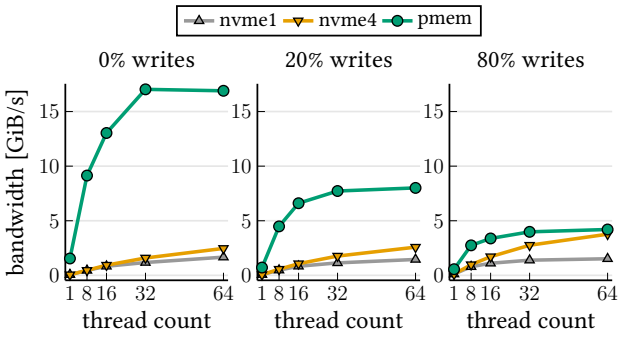

Figure 3: Bandwidth for the buffer management workload with 4 KiB pages.

than mmap in all configurations. For 16 KiB pages, the behavior for PMem is a little more nuanced, as mmap is better for lower thread counts, but beginning with 32 threads Linux I/O is faster. Both I/O methods perform direct access on PMem, and thereby do not differ in caching behavior. Therefore, the performance difference is again caused by the used move instructions and copy loops.

Summary. PMem significantly outperforms the single NVMe and pseudo-RAID for the read-only table scan workload. Unlike previous benchmarks that were heavily optimized and PMem-bound, we observe that a larger thread count and larger page sizes are preferable to increase performance for both NVMe and PMem. Linux I/O performs slightly better than mmap-based paging.

3.4 Buffer Management

The buffer management workload is a random mixed read-write workload, i.e., we both page in and page out data, from and to random pages within the data files. We use 5 GiB data files and let each thread execute a total workload of 5 GiB. We test write ratios of 0 %, 20 %, and 80 %. For a 20 % write ratio, every fifth operation is a page write. For the write operations, a pool of pages is generated randomly at the beginning of the benchmark and before a page is paged out, a randomly chosen byte is altered. Furthermore, unlike the table scan benchmark, we load data into a 1 GiB buffer, which prevents caching the entire data. In the following, we first discuss the performance results depending on page size and write ratio, and then compare Linux I/O to mmap+msync and libpmem2.

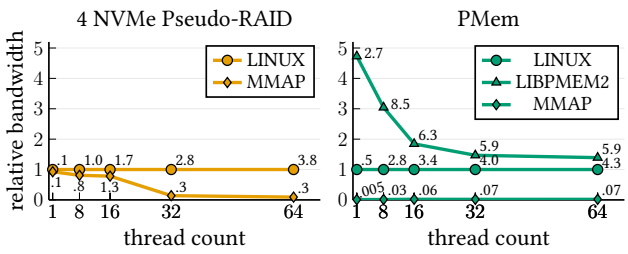

Figure 4: Relative bandwidth of mmap+msync and libpmem2 to Linux I/O for the buffer management workload, for 4 KiB pages and 80 % writes. Note that the plot shows relative performance, and absolute performance numbers in GiB/s for each data point are given.

Results. The results for Linux I/O and 4 KiB pages are shown in Figure 3. Larger page sizes exhibit similar trends with slightly higher bandwidths, as more data is read sequentially. Comparing 0 % writes to the previous table scan results, the bandwidth of PMem drops to 18.5 GiB/s (4.8 Mops/s) because the workload reads randomly selected pages instead of sequentially scanning, preventing prefetching of pages. Furthermore, the workload cannot utilize the L2 cache as in the table scan workload, because we force a write into DRAM. Moving up to 20 % write operations has a significant impact on PMem, resulting in a more than halved bandwidth of 8.2 GiB/s (2.1 Mops/s). For NVMe SSDs, increasing the write ratio increases performance, which indicates that random writes scale better than random reads in this workload. Our random access reads are memory-latency bound as each access requires the complete access cycle to the NVMe SSD without any prefetching mechanism. This high access latency also explains the bad scaling of the NVMe pseudo-RAID in our evaluation. To verify this latency bound, we run the benchmark with more than 64 threads and observe improved RAID performance, as the RAID's bandwidth is not saturated. On the other hand, writing is not latency-bound and can scale with multiple NVMes.

Linux I/O vs mmap+msync. Instead of Linux I/O, it is possible to use mmap and memcpy to page data in and out. To persist data, we use the msync syscall. In Figure 4, we compare the bandwidth of mmap relative to Linux I/O for 80 % writes and 4 KiB pages, i.e., the bandwidth of mmap divided by the bandwidth of Linux I/O. We see that for PMem, mmap performs poorly, only reaching 1-2 % of the bandwidth of Linux I/O. For NVMes with low thread counts, mmap reaches around 90 % of the Linux I/O bandwidth, but drops to 10 % of the bandwidth for higher thread counts. The msync syscall is known to result in poor performance, as (1) msync does not acknowledge non-temporal stores by memcpy, i.e., redundantly flushing data that has been flushed already and (2) msync requires locks of kernel-internal data structures, serializing multi-threaded code [31, 41]. Our findings are in line with previous work, which identifies the global I/O queue lock as a major bottleneck [5]. We observe the same performance drop in libpmem2 when forcing it to use msync, indicating that this is a general PMem issue.

Linux I/O vs libpmem2. To avoid fsync and msync on PMem, we test an I/O layer based on libpmem2, which is provided by Intel's Persistent Memory Development Kit (PMDK) [32]. This layer is

aware of the underlying storage technology and uses memory store instructions to ensure persistence for dax-enabled devices instead of kernel-provided file functions. For NVMes, the results are identical to `mmap+msync` as PMDK also uses `msync`. However, for PMem this avoids a syscall and instead utilizes fast memory-store instructions to ensure persistence. In Figure 4, next to the relative performance of `mmap+msync` to Linux I/O, we also show the relative performance of `libpmem2` for PMem. The plots show the relative performance, i.e., they do not show how absolute performance behaves if the thread count is varied. The absolute numbers are given at the markers of the plot. The relative `libpmem2` performance approaches the performance of Linux I/O for more threads and the absolute performance of `libpmem2` decreases after a peak at 8 threads, because the write queues are overloaded, which is in line with previous results [11, 49]. We find that for 80 % writes, `libpmem2` outperforms standard Linux I/O up to a factor of 4.7. Absolute `libpmem2` performance peaks at 8 threads with 8.5 GiB/s (2.2 Mops/s) and then slightly decreases to a plateau of 6 GiB/s (1.6 Mops/s). We note that the single-threaded performance of `libpmem2` is 2.7 GiB/s (707 Kops/s), while Linux I/O only achieves 500 MiB/s (128 Kops/s). For read-only access, there is no performance difference between `libpmem2` and `mmap+msync`.

Summary. For a drop-in replacement using standard I/O, we observe that as soon as the buffer-management workload becomes write-heavy and uses enough threads to saturate the NVMe RAID bandwidth, PMem's performance advantage becomes marginal. However, for purely random reads, PMem outperforms SSDs due to its significantly lower access latency. Compared to Linux I/O, we see that `mmap+msync` is not suitable for multi-threaded scenarios over 16 threads, due to the overhead incurred by `msync`, especially for PMem. However, introducing a storage layer-aware library, such as `libpmem2`, can improve standard Linux I/O single-threaded performance by 4.7x for PMem. For multi-threaded write-intensive workloads, `libpmem2` improves PMem performance by 1.4x, but unlike the table scan workload, does not significantly outperform the NVMe RAID. Overall, a PMem drop-in replacement offers little to no improvement over NVMe drives, if the workload mostly consists of writes and no storage layer-aware library can be used.

3.5 Logging

The logging workload consists entirely of small writes that need to be persisted in a log file. We test three logging methods, (1) a Linux I/O approach, in which we append log entries to a newly created file, (2) a Linux I/O approach that writes log entries into a recycled log file, and (3) logging into a recycled log file using the optimized `libpmemlog` library, provided by PMDK [32]. By comparing (1) and (2), we isolate the benefit of physically allocating log files beforehand, as it is common to recycle log files in database workloads [33, 37]. Previous work also shows the negative performance impact of zeroing new pages in the kernel before writing to them in PMem [11], making a recycling approach relevant also to PMem. Each thread has its own log file to avoid measuring concurrency and locking effects. For the pseudo-RAID, each thread is assigned one NVMe in a round-robin fashion, such that the workload is distributed across all drives. We test both smaller log entry sizes (128 B), similar to previous work on logging techniques for PMem [44], and also larger log entry sizes (8 KiB), as `Postgres` logs

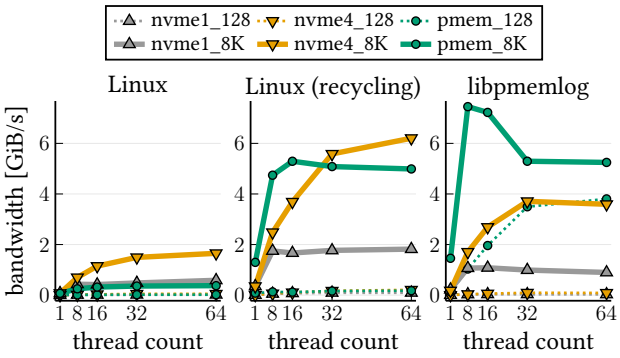

Figure 5: Bandwidth for the logging workload.

in 8 KiB pages by default [34, 39]. Furthermore, 8 KiB entries can also be understood as logging to a DRAM-backed buffer, which gets paged-out to disk once full.

Linux-based Logging. The results are shown in Figure 5. For both Linux I/O-based approaches, the NVMe RAID achieves the highest bandwidth of up to 6 GiB/s (790 Kops/s) when logging 8 KiB entries into an existing log file. Without using an existing file, even a single NVMe outperforms PMem both for 128 B and 8 KiB log entries. The performance improvement gained by recycling log files is significant. Compared to using a new file, the peak performance improves by 3.1x for the single NVMe, by 3.8x for the RAID, and by 14.1x for PMem. Noticeably, while PMem peaks at 5.4 GiB/s (700 Kops/s) using 16 threads and recycling and outperforms the RAID until that point, when employing 32 or 64 threads, the RAID scales and achieves 1.2x higher bandwidth than PMem for 64 threads.

`Libpmemlog` Results. When using `libpmemlog`, PMem outperforms the single NVMe by up to an order of magnitude for small log entries. This is because the byte-addressability of PMem in combination with a library that leverages more efficient user-space store instructions on PMem, e.g., `ntstore` or `clwb`, enables efficient logging compared to relying on expensive kernel-space `fsync` calls for persistence. Block devices, such as NVMe, that are not byte-addressable cannot log small entries as efficiently. Using `libpmemlog`, the gap between the log entry sizes on PMem is decreased, as for 128 B log entries, PMem peaks at around 3.9 GiB/s (33 Mops/s) (improvement of 21.7x compared to recycled file), and for 8 KiB entries at around 7.6 GiB/s (1 Mops/s, 1.4x improvement). We observe a performance decrease for more than 8 threads for 8 KiB log entries as, similar to the buffer management workload, the write queues are overloaded.

However, the NVMe performance drops compared to Linux I/O, where both the single NVMe and the RAID are 1.7x faster at peak. The reason for this is that `libpmemlog` on NVMes relies on `msync` instead of `fsync`, which performs worse, as shown in Section 3.4.

Summary. Recycling log files is critical for both PMem and NVMes. The RAID scales better in multi-threaded scenarios, outperforming PMem by up to 1.2x with log file recycling and 4.4x without recycling for multiple threads. The `libpmemlog` experiments show that the switch from a simple drop-in replacement to a library that provides a similar API, but differentiates between the underlying storage technology, leads to a significant performance increase of up to 21.7x for 128 B log entries on PMem, compared to Linux logging on a recycled file.

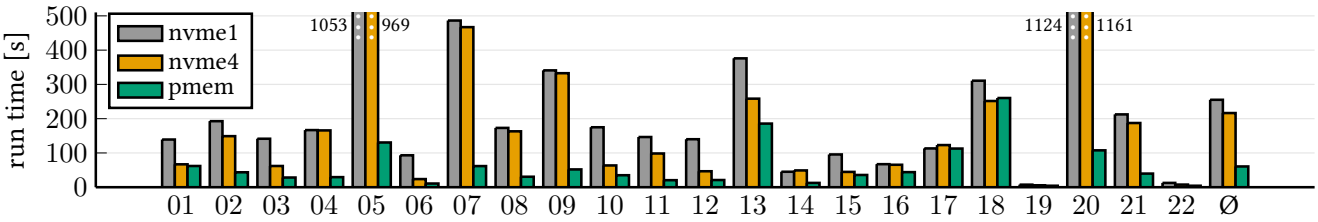

Figure 6: TPC-H results (scale factor 100) on `Postgres` running on a single NVMe, a RAID 0 of four NVMes, and PMem.

4 TPC-H BENCHMARK

To better understand the performance impact of NVMes and PMem in real-world database systems, we run the end-to-end TPC-H benchmark [42] on `Postgres`. We run `Postgres` 13.2 and tune it towards the memory and CPU specs of our machine based on the suggestions of PGTune [45] and tuning guides [38, 40]. We vary the storage backend of `Postgres` between a single NVMe, a real RAID 0 of four NVMes (via `mdadm`), and PMem. After each query, we restart `Postgres` and clear its caches to prevent DRAM caching.

We run the TPC-H benchmark with a scale factor of 100. We use the data schemes and indexes for `Postgres` as provided by OLTPBench [10, 12] and rely on the default TPC-H `dbgen` tool for data- and query-generation. After loading the data, we run a `VACUUM(ANALYZE)` command to let `Postgres` obtain statistics on the data on the underlying storage medium.

Results. The results for all queries and the average query time are shown in Figure 6. We observe that, except for query 18, all queries run faster on PMem. However, the speedup varies greatly between almost no speedup for query 18 and a 10.4x speedup for query 20. We discuss three queries (#3, #5, #18) in more detail as these represent (1) queries where RAID is faster than a single NVMe and PMem is faster than the RAID, (2) queries where PMem is significantly faster than both RAID and single NVMe, and (3) queries where all storage technologies perform nearly identically.

For query 3, the dominating component is a parallel sequential scan that reads 86 GiB using 9 parallel workers. The query plan is identical for all storage components. The sequential scan takes around 30 s for the NVMe RAID and 12 s for PMem. As the query is dominated by a sequential read, we confirm the observations from Section 3.3 that the NVMe RAID scales well for sequential workloads and that PMem achieves higher bandwidth than it.

Query 5 is dominated by a parallel index scan, reading 430 GiB of data using 4 workers. This takes around 15:30 minutes for the RAID and only 1:20 minutes for PMem. As the index scan is a random read workload, we confirm our observation from Section 3.4, for which the RAID provides almost no performance improvement due to the latency-bound NVMe SSDs. Due to its low access latency, PMem outperforms the RAID by a factor of 7.4x for this query.

For query 18, an analysis of the query plan shows that the workload is bound by aggregations and hash joins, not by I/O. While a sequential scan is performed, this does not dominate the run time and ~70 % of the run time is spent on aggregating and joining. As this query is compute-bound, PMem cannot improve its runtime.

As most queries are random I/O heavy and PMem especially improves on the latency-bound random read performance on NVMe SSDs, `Postgres` executes the average TPC-H query on PMem 4.2x faster than on a single NVMe and 3.6x faster than on the RAID. The RAID improves the average run time by only 1.2x compared to a single NVMe. This verifies the results of our microbenchmarks and shows that a drop-in replacement of PMem significantly improves the performance of read-heavy database workloads.

5 DISCUSSION

In this section, we summarize our results and give four practical insights to aid decision-making on when to use PMem as an SSD drop-in replacement and how to optimize for it.

Insight 1. For read-intensive workloads, PMem outperforms a single NVMe by up to 11x and a four-NVMe RAID by 2.8x. Thus, without application changes a PMem drop-in replacement significantly speeds up sequential read workloads.

Insight 2. While previous work suggests specific configurations for maximum PMem performance, standard Linux/C++ operations do not fully utilize PMem's bandwidth. When using these operations, one should use as many threads as possible and prefer larger access sizes to maximize the performance of PMem as the storage medium. However, to reach its full potential in the future, Linux and glibc operations should be optimized, e.g., by more targeted use of SIMD move instructions.

Insight 3. As PMem write bandwidth is similar to the NVMe RAID, a drop-in replacement of PMem for write-intensive workloads does not lead to large performance improvements. However, introducing a PMem-aware I/O layer into the application significantly increases PMem write performance, due to efficient memory store instructions instead of expensive file sync system calls. In the buffer management workload, `libpmem2` outperforms Linux I/O by up to 4.7x, and `libpmemlog` improves the logging performance of small 128 B entries by 21.7x.

Insight 4. In random read workloads, NVMes are latency-bound and therefore do not scale well in RAID configurations. The latency-boundedness has a severe impact on both microbenchmark and end-to-end TPC-H performance. In the buffer management read-only workload with 64 threads, the NVMe RAID is only 1.5x faster than a single NVMe, while PMem is 10.1x faster. When choosing between expanding the storage layers with several NVMes or adding PMem, one should consider PMem if the workload at hand is dominated by random I/O, as these suffer from the higher latency of NVMes.

Summary. Overall, the choice of PMem as a drop-in replacement for NVMe SSDs should be made based on the workload that is supported by the storage layer. If higher performance without major code changes is important, then it is possible to employ PMem and achieve significant speedup. In the future, when access methods

are improved to utilize maximum PMem bandwidth, a drop-in replacement will become even more reasonable.

6 RELATED WORK

In this section, we discuss related work concerning PMem, NVMe SSDs, and their use in database systems.

PMem Benchmarks. Various performance studies of PMem modules have been conducted [11, 18, 30, 46, 49]. However, these papers utilize PMem as a replacement or extension to DRAM, and do not run typical database-like operations, but highly-tuned benchmark functions. These papers show that previous assumptions, e.g., PMem is just slower DRAM [2, 29], do not always hold, while outlining various unique performance characteristics of PMem.

PMem and NVMe for Databases. Recent work on modern-storage optimized database systems shows the importance of hardware-conscious designs. Leis et al. propose LeanStore [20], a database buffer management design for NVMe storage. Umbra is a modern in-memory RDBMS built with LeanStore to support larger-than-DRAM state [27]. A hierarchical DRAM/PMem/SSD buffer manager was proposed by van Renen et al., to close the gap between DRAM and SSD performance [43]. Other work presents hybrid DRAM-PMem key-value store designs and shows the benefit of incorporating PMem as persistent storage [9, 22].

Storage-aware Database Benchmarks. Wu et al. conduct performance evaluations of PMem and SSDs for the usage in DMBS using microbenchmarks and TPC-C and TPC-H workloads [47]. While their methodological approach is similar to ours, we conduct more nuanced micro-benchmarks that are tailored towards DBMS use-cases, explore the design space further, and use a more realistic setup with multiple PMem DIMMs and NVMes. Xu et al. investigate the TPC-C performance of NVMe SSDs, SATA SSDs, and HDDs for various systems [48]. Haas et al. maximize the performance of an NVMe array by analyzing the I/O stack in various microbenchmarks and the TPC-C benchmark [14].

7 CONCLUSION

In this paper, we analyze persistent memory as a drop-in replacement for NVMe SSDs in database workloads. To this end, we run a set of microbenchmarks as well as the TPC-H benchmark on `Postgres`. Our evaluation shows that PMem significantly speeds up read-intensive workloads by up to 4x in comparison to an NVMe RAID and up to 11x for a single NVMe. On the other hand, the speed-up for write-intensive workloads is negligible, unless a storage layer-aware library like `libpmem2` is used. Following our results, we give four practical insights to aid decision-making on when to use PMem as an SSD drop-in replacement and how to optimize for it. With the continuing development of PMem and its price stabilization, we expect it to become an increasingly relevant drop-in replacement of NVMe SSDs.

ACKNOWLEDGMENTS

This work was partially funded by the European Union's Horizon 2020 research and innovation programme (ref. 957407).

REFERENCES

[1] P. Alcorn. 2019. Intel Optane DIMM Pricing. https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html Accessed on 2021/03/10.
[2] J. Arulraj, J. Levandoski, U. F. Minhas, and P. Larson. 2018. BzTree: A High-Performance Latch-Free Range Index for Non-Volatile Memory. *Proceedings of the VLDB Endowment* 11, 5 (2018).
[3] J. Axboe. 2021. fio Repository: filesetup.c. https://github.com/axboe/fio/blob/014ab48afcbcf442464acc7427fcd0f194f64bf4/filesetup.c Accessed on 2021/03/17.
[4] I.G. Baek, M.S. Lee, S. Sco, M.J. Lee, D.H. Seo, D.-S. Suh, J.C. Park, S.O. Park, H.S. Kim, I.K. Yoo, U.-I. Chung, and J.T. Moon. 2004. Highly scalable non-volatile resistive memory using simple binary oxide driven by asymmetric unipolar voltage pulses. In *Proceedings of the 50th IEEE International Electron Devices Meeting (IEDM)*.
[5] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet. 2013. Linux Block IO: Introducing Multi-Queue SSD Access on Multi-Core Systems. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR)*.
[6] F. Chen, B. Hou, and R. Lee. 2016. Internal Parallelism of Flash Memory-Based Solid-State Drives. *ACM Transactions on Storage* 12, 3 (2016).
[7] F. Chen, R. Lee, and X. Zhang. 2011. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*.
[8] Y. Chen, Y. Lu, K. Fang, Q. Wang, and J. Shu. 2020. uTree: A Persistent B+-Tree with Low Tail Latency. *Proceedings of the VLDB Endowment* 13, 12 (2020).
[9] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[10] C. A. Curino, D. E. Difallah, A. Pavlo, and P. Cudre-Mauroux. 2012. Benchmarking OLTP/web databases in the cloud: The OLTP-bench framework. In *Proceedings of the 4th International Workshop on Cloud Data Management (CloudDB)*.
[11] B. Daase, L. J. Bollmeier, L. Benson, and T. Rabl. 2021. Maximizing Persistent Memory Bandwidth Utilization for OLAP Workloads. In *Proceedings of the International Conference on Management of Data (SIGMOD)*.
[12] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. 2013. OLTP-Bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment* 7, 4 (2013).
[13] X. Guo, E. Ipek, and T. Soyata. 2010. Resistive computation: Avoiding the power wall with low-leakage, STT-MRAM based computing. *ACM SIGARCH Computer Architecture News* 38, 3 (2010).
[14] G. Haas, M. Haubenschild, and V. Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *Proceedings of the 10th Conference on Innovative Data Systems Research (CIDR)*.
[15] J. Hruska. 2021. How Do SSDs Work? https://www.extremetech.com/extreme/210492-extremetech-explains-how-do-ssds-work Accessed on 2021/03/14.
[16] A. Huffman. 2012. NVM Express. https://www.nvmexpress.org/wp-content/uploads/NVM-Express-1_1.pdf
[17] Intel. 2021. Intel® SSD DC P4610 Series. https://ark.intel.com/content/www/us/en/ark/products/140103/intel-ssd-dc-p4610-series-1-6tb-2-5in-pcie-3-1-x4-3d2-tlc.html Accessed on 2021/03/10.
[18] 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. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR* abs/1903.05714 (2019).
[19] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. 2009. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*.
[20] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *Proceedings of the 34th IEEE International Conference on Data Engineering (ICDE)*.
[21] L. Lersch, X. Hao, I. Oukid, T. Wang, and T. Willhalm. 2019. Evaluating persistent memory range indexes. *Proceedings of the VLDB Endowment* 13, 4 (2019).
[22] L. Lersch, I. Schreter, I. Oukid, and W. Lehner. 2020. Enabling low tail latency on multicore key-value stores. *Proceedings of the VLDB Endowment* 13, 7 (2020).
[23] J. Liu, S. Chen, and L. Wang. 2020. LB+Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *Proceedings of the VLDB Endowment* 13, 7 (2020).
[24] Z. Liu. 2018. Fujitsu Targets 2019 for NRAM Mass Production. https://www.tomshardware.com/news/fujitsu-nram-nantero-carbon-nanotube,37437.html Accessed on 2021/03/20.
[25] B. Lu, X. Hao, T. Wang, and E. Lo. 2020. Dash: Scalable Hashing on Persistent Memory. *Proceedings of the VLDB Endowment* 13, 10 (2020).
[26] Nantero. 2021. Nantero Website. https://www.https://nantero.com/ Accessed on 2021/03/10.

[27] T. Neumann and M. J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *Proceedings of the 10th Conference on Innovative Data Systems Research (CIDR)*.

[28] NVM-Express. 2012. NVM Express Explained. https://nvmexpress.org/wp-content/uploads/2013/04/NVM_whitepaper.pdf Accessed on 2021/03/10.

[29] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the International Conference on Management of Data (SIGMOD)*.

[30] I. B. Peng, M. B. Gokhale, and E. W. Green. 2019. System Evaluation of the Intel Optane byte-addressable NVM. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*.

[31] Persistent Memory Knowledge Base. 2020. Why msync() is less optimal for persistent memory. https://kb.pmem.io/development/100000025-Why-msync-is-less-optimal-for-persistent-memory/ Accessed on 2021/03/22.

[32] PMDK Team. 2021. Persistent Memory Development Kit. https://pmem.io/pmdk/ Accessed on 2021/03/18.

[33] Postgres. 2021. Postgres Docs: WAL Configuration. https://www.postgresql.org/docs/9.5/wal-configuration.html Accessed on 2021/03/25.

[34] Postgres. 2021. Postgres Docs: WAL Internals. https://www.postgresql.org/docs/current/wal-internals.html Accessed on 2021/03/18.

[35] Postgres. 2021. Postgres Repository: Buffer Readme. https://github.com/postgres/postgres/blob/15639d5e8f6f278219681fec8a5668a92fb7e874/src/backend/storage/buffer/README#L218-L230 Accessed on 2021/03/16.

[36] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*.

[37] SQLite. 2021. SQLite Docs: Write-Ahead Logging. https://sqlite.org/wal.html Accessed on 2021/03/25.

[38] S. Suryawanshi. 2019. Comprehensive guide on how to tune database parameters and configuration in PostgreSQL. https://www.enterprisedb.com/postgres-tutorials/comprehensive-guide-how-tune-database-parameters-and-configuration-postgresql Accessed on 2021/03/22.

[39] H. Suzuki. 2020. *The Internals of Postgres (Chapter 9 - Write Ahead Logging)*. https://www.interdb.jp/pg/pgsql09.html

[40] T. Swatz. 2020. Optimize PostgreSQL Server Performance Through Configuration. https://blog.crunchydata.com/blog/optimize-postgresql-server-performance Accessed on 2021/03/22.

[41] L. Torvalds. 2000. Re: mmap/mlock performance versus read [linux-kernel mailing list]. https://marc.info/?l=linux-kernel&m=95496636207616&w=2 Accessed on 2021/03/25.

[42] Transaction Processing Performance Council (TPC). 1993. TPC Benchmark H (TPC-H) - Standard Specification.

[43] A. van Renen, V. Leis, A. Kemper, T. Neumann, T. Hashida, K. Oe, Y. Doi, L. Harada, and M. Sato. 2018. Managing Non-Volatile Memory in Database Systems. In *Proceedings of the International Conference on Management of Data (SIGMOD)*.

[44] A. van Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper. 2020. Building blocks for persistent memory. *The VLDB Journal* 29, 6 (2020).

[45] A. Vasiliev. 2021. PGTune. https://pgtune.leopard.in.ua/#/

[46] M. Weiland, H. Brunst, T. Quintino, N. Johnson, O. Iffrig, S. Smart, C. Herold, A. Bonanni, A. Jackson, and M. Parsons. 2019. An early Evaluation of Intel's Optane DC Persistent Memory Module and its Impact on High-Performance Scientific Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[47] Y. Wu, K. Park, R. Sen, B. Kroth, and J. Do. 2020. Lessons Learned from the Early Performance Evaluation of Intel Optane DC Persistent Memory in DBMS. In *Proceedings of the 16th International Workshop on Data Management on New Hardware (DaMoN)*.

[48] Q. Xu, H. Siyamwala, M. Ghosh, T. Suri, M. Awasthi, Z. Guz, A. Shayesteh, and V. Balakrishnan. 2015. Performance analysis of NVMe SSDs and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*.

[49] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*.

[50] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. 2009. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*.