PerMA-Bench: Benchmarking Persistent Memory Access

Lawrence Benson, Leon Papke, Tilmann Rabl
Hasso Plattner Institute, University of Potsdam
{firstname.lastname}@hpi.de

ABSTRACT

Persistent memory's (PMem) byte-addressability and persistence at DRAM-like speed with SSD-like capacity have the potential to cause a major performance shift in database storage systems. With the availability of Intel Optane DC Persistent Memory, initial benchmarks evaluate the performance of real PMem hardware. However, these results apply to only a single server and it is not yet clear how workloads compare across different PMem servers. In this paper, we propose PerMA-Bench, a configurable benchmark framework that allows users to evaluate the bandwidth, latency, and operations per second for customizable database-related PMem access. Based on PerMA-Bench, we perform an extensive evaluation of PMem performance across four different server configurations, containing both first- and second-generation Optane, with additional parameters such as DIMM power budget and number of DIMMs per server. We validate our results with existing systems and show the impact of low-level design choices. We conduct a price-performance comparison that shows while there are large differences across Optane DIMMs, PMem is generally competitive with DRAM. We discuss our findings and identify eight general and implementation-specific aspects that influence PMem performance and should be considered in future work to improve PMem-aware designs.

1 INTRODUCTION

Both research and industry have awaited the arrival of persistent memory (PMem) as a new layer in the storage hierarchy for many years. PMem promises byte-addressability and persistency at DRAM-like speed with SSD-like capacity. These characteristics have the potential to cause a major performance increase in storage systems, such as databases and key-value stores. Thus, research on system design incorporating PMem was published long before real PMem hardware was available [1, 46, 53]. Now that byte-addressable, persistent memory is finally available commercially, Intel's Optane DC Persistent Memory has received a lot of attention in initial performance evaluations [10, 13, 52, 56]. These

evaluations provide valuable insights into the general performance and unique characteristics of first-generation Optane.

Research on data structures [7, 36, 39] and storage systems [5, 8, 35] that incorporate these insights often have to perform additional hardware-specific micro benchmarks to understand the specific nuanced PMem behavior for their expected workloads. Initial research shows that Optane's performance is highly dependent on the workload with major differences between read and write behavior.

Due to limited availability and high prices, researchers often have access to only one PMem server. Thus, new systems built for PMem are designed, implemented, and optimized on a single server with a single combination of PMem, DRAM, and CPU. However, many factors impact PMem performance that are not yet well understood, e.g., the DIMMs' size and power budget or the number of DIMMs in the server. As PMem is a very new technology, it is unclear how well these initial designs generalize across PMem configurations. On top of various configurations, with the availability of second-generation Optane, new performance characteristics are introduced.

Based on the configuration space and workload-tailored micro-benchmarks of previous work, we identify the need for a comparable workload-driven analysis of PMem. We propose PerMA-Bench, a configurable benchmark framework that analyzes the bandwidth, latency, and operations per second for customizable database-related PMem access. In PerMA-Bench, we pre-define various workloads that cover the maximum achievable performance of core access patterns (sequential/random reads/writes), as well as a wide range of realistic, database-related access patterns, such as updates, lookups, and scans in tree and hash indexes. These complex patterns include pointer-chasing loads, mixed read/write access, and hybrid PMem/DRAM access. Additionally, PerMA-Bench allows users to run custom workloads tailored toward their design choices. With PerMA-Bench, we propose a tool that provides insight into the performance of PMem at a general and workload-specific level. Users can explore the performance of new access patterns but also validate existing designs. Based on these findings, users can validate their design choices without having to write their own benchmark application and find areas of improvement in existing designs.

Based on PerMA-Bench, we perform the first extensive evaluation of Optane for database workloads across various DIMM sizes of the first and second generation. We compare the performance of all three DIMM sizes of 100 Series Optane and one DIMM size of the 200 Series. Additionally, we show the impact of varying the number of DIMMs, DIMM power budgets, and memory bus speeds.

We validate our results with existing implementations and show that they do not fully utilize the performance improvements across Optane generations. We show that the choice of persist instruction has a high performance impact and that avoiding explicit flushes in eADR does not always yield the best results. Based on our results, we identify and discuss eight aspects that future work should take into account when designing PMem-aware systems. With the

Figure 1: Standard PMem access modes.

Figure 2: Writing to NVDIMM-Ps from the CPU.
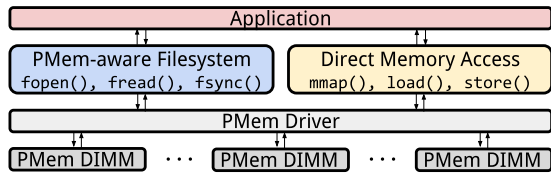
availability of more PMem hardware, research has to consider more than one setup to achieve general PMem-optimized designs.

In addition to PMem's performance, its price-performance is important to determine whether PMem is suitable for users' needs. In this paper, we perform a price-performance comparison of various server configurations. Our comparison shows that PMem's price-performance is competitive with that of DRAM and is often even better. Thus, in addition to providing persistence, PMem can act as a larger, cost-effective general memory when used correctly. In summary, we make the following contributions:

1) We propose PerMA-Bench, a configurable benchmark framework to analyze bandwidth, latency, and operations per second for customizable database-related PMem access.

2) We perform an extensive evaluation of PMem performance across four PMem servers and additional per-server configurations to show the impact of individual server setups on bandwidth utilization and latency.

3) We compare the price-performance for key workloads across all servers and show that while there are large differences across Optane, PMem is generally competitive with DRAM.

4) We discuss eight general and implementation-specific aspects that influence the performance of PMem and need to be taken into account for the design of future PMem-aware systems.

The remainder of the paper is structured as follows. In Section 2, we briefly introduce PMem and its access methods. We then introduce the PerMA-Bench framework in Section 3. In Section 4, we present PerMA-Bench results on various hardware configurations, which we then use in Section 5 to discuss the price-performance of PMem. Finally, we discuss our findings (Section 6) and related work (Section 7), before concluding in Section 8.

2 PERSISTENT MEMORY

In this section, we first give a short overview of PMem technologies. Then, we present how developers access and interact with PMem.

2.1 Persistent Memory Types

Large-scale persistent memory is currently based on one of two designs: 3D XPoint (NVDIMM-*P*) or DRAM + flash (NVDIMM-*N*). 3D XPoint, developed by Intel and Micron, is the underlying technology of Optane [19]. It is the only publicly available *true* PMem, in which a single storage medium allows for both byte-addressability and persistence. DRAM + flash storage designs are employed in PMem offered by, e.g., HPE [11]. These battery-backed NVDIMM-Ns flush their state to flash chips on power failure.

According to the JEDEC standards, NVDIMM-Ns are seen as regular DRAM by the server while NVDIMM-Ps are viewed as separate storage with additional changes to the DDR4 protocol [25, 26]. Future PMem technology is expected to follow the NVDIMM-P standard, as this allows for larger capacity and extended functionality,
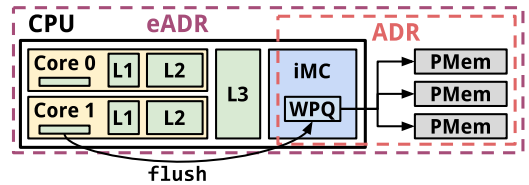
while NVDIMM-Ns are limited by DRAM [27]. Currently, Optane PMem is the only available NVDIMM-P implementation. Various other PMem designs have been announced or are actively developed. These include Nano-RAM [38], phase change memory [31, 47, 58], resistive RAM [4], and magnetoresistive RAM [14].

While NVDIMM-Ns have been available for many years, they have not achieved widespread adoption. On the other hand, Optane, as a new technology, has received a lot of attention in academia and initial use in industry, e.g., in SAP HANA [16]. As NVDIMM-Ns are essentially DRAM and have DRAM performance, we focus our evaluation on NVDIMM-Ps in the form of Optane.

2.2 Accessing Persistent Memory

The Storage Networking Industry Association (SNIA) defines an NVM Programming Model (NPM) [48], which specifies a unified access model for PMem. This model allows for the integration of a wide range of storage technologies. We show the two PMem access modes of this model in Figure 1. Applications either access PMem via regular filesystem interfaces (shown on left side) using calls such as `fopen`, `fread`, `fsync`, or they access PMem via memory-mapping (shown on right side) using calls such as `mmap`, `load`, `store`.

The filesystem access allows existing applications to use PMem as a drop-in replacement for common disk-based interaction, while the second mode allows applications to use PMem identically to DRAM. The programming model allows for memory mapping of files, i.e., combining both modes. In this case, files are used to logically structure raw memory chunks but PMem is accessed directly without the overhead of regular file I/O.

An important distinction between memory mapping files on disk and PMem is that traditionally data is copied to a page cache in DRAM, which is then modified and flushed back. When mapping PMem, the page is accessed directly and not copied to DRAM, which is possible through PMem's byte-addressability. This, however, changes the failure granularity of data modification. Filesystems provide crash consistency for file I/O, allowing developers to rely on the filesystem for atomic writes. When using PMem via memory mapping, there are no such guarantees, as data is accessed and modified via regular load/store instructions from the CPU. Thus, developers must explicitly control data persistence and handle low-level crash consistency in the application themselves.

To describe the necessary steps for developers to ensure correct data persistence and potential issues that arise around it, we show a simplified connection of a CPU and an NVDIMM-P in Figure 2. This model is based on Intel's Xeon processors [22] and Optane. A CPU contains one or more integrated memory controllers (iMCs), which are directly connected to PMem via memory channels. To write data to PMem, the CPU must flush cache lines to a write pending queue (WPQ) within an iMC. The WPQ then issues the write to the correct PMem device. PMem and the WPQs constitute the asynchronous

DRAM refresh domain (ADR), in which persistence is guaranteed. Once a cache line enters the WPQ it is guaranteed to be persisted even in case of power loss. While WPQs are Intel-specific, they are based on the NVDIMM-P standard that describes a write buffering mechanism. Future PMem is likely to work similarly.

Data in the caches is not persisted. Programmers must explicitly control cache line flushes to ensure persistence. On Intel CPUs, programmers can use, e.g., `clflush` (flush w/ invalidate), `clwb` (flush w/o invalidate), or `ntstore` (bypass caches). Additionally to explicit flushes, data might be randomly evicted from the cache, resulting in unexpected data persistence. As current CPUs provide only 8 Byte atomic writes, random 64 Byte cache line evictions may cause an inconsistent state after a crash for modifications larger than 8 Byte. Thus, programmers must carefully design fine-grained PMem data access to ensure application correctness.

Additionally, programmers have to ensure correct store ordering. Modern compilers and CPUs may re-order instructions to improve performance, e.g., through better pipelining. However, this may lead to re-ordering of persist instructions, resulting in correctness bugs [37]. To avoid such re-ordering, programmers must explicitly issue, e.g., an `sfence` instruction on x86 [23].

Correctly moving data from CPU caches to PMem burdens programmers due to these correctness issues. It also incurs performance penalties due to additional CPU instructions. However, solutions exist to mitigate the correctness issues and performance penalties. Intel's 3rd Generation Xeon processors introduce an *enhanced* ADR (eADR) [20]. This includes all caches in the ADR, i.e., ensuring the persistence of all cached data in case of power loss. This new design removes the necessity of explicit flushing but still encounters random (partial) eviction. A recent study finds that missing flushes are a common mistake in various PMem applications and libraries [42]. Thus, an eADR server protects the user from this class of bugs.

3 INTRODUCING PERMA-BENCH

In this section, we introduce PerMA-Bench, a benchmark framework for persistent memory access. When designing new systems or database components, it is important to know the performance of the underlying memory access. This understanding allows users to tune their system towards better PMem utilization. PerMA-Bench supports *basic* and *complex* memory access patterns to evaluate the performance of PMem. Basic access patterns determine the maximum achievable bandwidth utilization and latency by repeatedly executing the same operation, i.e., a simple read or write. Complex patterns allow users to evaluate specific designs via chained read/write access from/to DRAM and PMem with varying persist instructions and access sizes. Based on these complex patterns, users can model, e.g., new index structure designs and gain insight into their memory performance before implementing them.

We present the runtime of PerMA-Bench in Section 3.1. Then, we present options for workload customization in Section 3.2 and briefly discuss supported memory store semantics in Section 3.3.

3.1 Runtime

PerMA-Bench is designed as a standalone benchmark executable. Users interact with PerMA-Bench via configuration files and command line arguments. Based on these specified configuration parameters, individual benchmarks are created. We show the execution
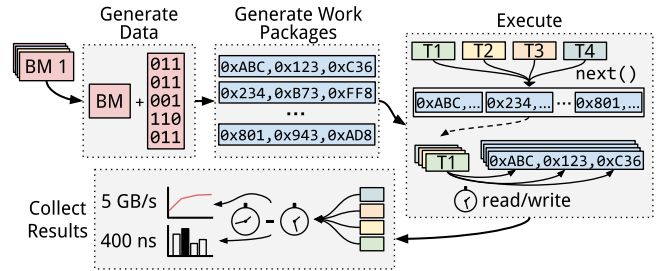

Figure 3: Execution cycle of a benchmark in PerMA-Bench.

cycle of PerMA-Bench in Figure 3. For each benchmark (BM) that is created, PerMA-Bench performs four steps.

First, all data files are prepared and filled with random data. The files can be located in PMem or DRAM to allow for hybrid setups, which are common in current PMem research. Next, individual work packages are generated, which contain a pointer for each operation that is to be executed on the data. Work packages of, e.g., a random read benchmark with 100 operations contain 100 pointers to random offsets in the data file. For sequential access, packages contain 100 pointers to contiguous addresses. This allows for execution in a tight loop instead of requiring logic per benchmark type. For raw performance benchmarks, all work packages are pre-generated to avoid the overhead of generation during execution.

During the execution, *N* threads are spawned ($N = 4$ in Figure 3). Each thread then continuously pulls a new work package from a shared queue and executes the requested operations on that work package. PerMA-Bench adopts this work package approach for two reasons. First, to avoid stragglers when statically assigning work to threads. During our evaluation, we observed that hyperthreading often leads to very unbalanced execution times, skewing the final results. Second, as the general concept of work-stealing is employed in many databases exactly to avoid execution skew, PerMA-Bench represents a common execution model where workers operate on small work packages, e.g., via morsels [33]. To avoid long-running packages and the skew this entails, work packages contain 64 MB worth of operations by default, which execute in less than 100 ms in most cases. All threads are synchronized via a barrier before the execution starts to ensure concurrent execution.

We find that results are not impacted by a warm-up phase within workloads, as they exceed cache and queue sizes. However, as pre-faulting pages before writing to them avoids kernel page zeroing during execution [21], we provide a "warm-up" pre-fault flag.

After all work packages have been processed, PerMA-Bench collects the results of all threads to calculate the final benchmark results. PerMA-Bench determines the total execution time as the time between all threads' earliest begin timestamp and latest end timestamp. This captures the entire execution duration but may underestimate the actual performance slightly, as some threads are already idle while others are finalizing their work. However, in PerMA-Bench, we perform workload-driven performance evaluation and from a higher-level perspective, this approach captures the total time it takes to complete a given workload. Based on the total number of processed bytes or operations and the total execution time, PerMA-Bench calculates the overall throughput in GB/s or operations/s. If specified by the user, PerMA-Bench also samples the latency of individual operations. The sampled values are added

Listing 1: Example config YAML file.

```
1 hash_index_update:
2  matrix:
3   number_threads: [ 1, 4, 16 ]
4  args:
5   custom_operations:
6    "r_256,w_64_cache_128,w_64_cache_-128"
7   total_memory_range: 10G
8   number_operations: 100000000
```

to a histogram and presented in the form of minimum, maximum, average, and multiple percentile latencies.

3.2 Custom Workloads and Configuration

Besides the pre-defined workloads, custom benchmarks can be configured via YAML files and command line arguments. In this section, we present configuration options provided by PerMA-Bench with which users can express their specific workloads' access patterns.

Configuration Files. Benchmarks in PerMA-Bench are configured via YAML files. This format allows users to specify workloads manually and programmatically. We show an example configuration in Listing 1. Each configuration file consists of two main parts, the matrix arguments and the general arguments. The matrix block (Lines 2-3) describes which dimensions should be evaluated in the benchmark. Each matrix argument is provided as a list, from which PerMA-Bench creates a benchmark for each combination in the cross product, i.e., three benchmarks in this example. The args block (Lines 4-8) describes which general arguments should be used for every combination. In this example, we configure a hash index update workload and evaluate it for 1, 4, and 16 threads.

Custom Operations. In Line 6, we show the definition of a custom operation. These model complex, pointer-chasing memory access patterns instead of simple, independent reads or writes. They are created in a chain in which each operation op is responsible for calling the next operation op' once complete. When op has read the random data d, it passes d to op', which then determines the next address based on d. By requiring data from op in op', PerMA-Bench ensures that op' is not executed before op was completed.

In the example, PerMA-Bench reads 256 Byte (r_256), e.g., a hash bucket, at a random location r_a within the allocated data range. Then, two 64 Byte *Cache* write instructions (w_64_cache) are executed. The first is performed with an offset of 128 Byte (_128 = r_a + 128), e.g., to store data in a hash bucket. The next write operation jumps back 128 Byte to the start of the bucket (_-128 = r_a) to update metadata. This pattern of storing data in a node and updating metadata afterwards is common in PMem data structures [5, 36, 39, 46]. As 64 Byte cache line flushes are combined to 256 Byte in Optane, it is important to model adjacent writes correctly instead of simulating them with writes to the same cache line while supporting different persist instructions. Varying these sizes also gives users insight into the impact of prefetching in PMem. Additionally, PerMA-Bench supports mixing DRAM and PMem for hybrid access, as used, e.g., in PMem B-Trees [7, 36, 46, 57].

Benchmark Parameters. PerMA-Bench currently offers 19 configuration parameters that allow users to define a wide range of individual benchmarks without having to write C++ code for each of them. Users can specify, e.g., PMem/DRAM memory ranges,

access size, sequential/random execution, number of partitions and threads (for data parallelism), custom operations, work package size, runtime, and file pre-faulting.

Other Features. PerMA-Bench supports running different workloads as task-parallel benchmarks. Concurrent workloads might impact each other as one benefits from caching, while the other fills the cache with unwanted data. Users can also specify *NUMA-aware execution* of benchmarks on *far* or *near* CPUs to explore how data placement impacts their workloads and whether NUMA must be considered in their design. PerMA-Bench additionally allows users to run all benchmarks in DRAM as a performance reference.

3.3 Persist Instructions

PerMA-Bench supports four persist instructions, *Cache*, *CacheInvalidate*, *NoCache*, and *None*. *Cache* represents a temporal store (clwb), *CacheInvalidate* represents a temporal store that invalidates the cache line (clflushopt), *NoCache* represents a non-temporal store (ntstore), and *None* performs no explicit flush instruction. Temporal refers to the inclusion of data in the cache hierarchy with the assumption of future access, i.e., temporal locality is likely. When temporal locality is unlikely, non-temporal instructions can bypass the cache completely, avoiding cache pollution. Not explicitly flushing is useful when persistence is not required, e.g., when storing intermediate results in PMem or when eADR ensures persistence. For *Cache*, *CacheInvalidate*, and *NoCache*, we add a store fence (sfence) afterwards to guarantee correct write ordering. PerMA-Bench uses Intel's AVX512 extension to write an entire cache line, i.e., 64 Byte or 512 Bits, in one instruction using SIMD-registers [23].

4 PERMA-BENCH RESULTS

In this section, we present the results of various PerMA-Bench workloads on multiple PMem server configurations. Our results give insight into both raw and workload-specific PMem performance to better understand PMem's use in database workloads. We evaluate various configurations to show how comparable previous results are across PMem setups, as they are often run on only one configuration, e.g., on one DIMM size or with a partially stocked server. These configurations allow us to draw more general conclusions about PMem as well as provide insight into how previously published systems and results apply to other setups.

We describe our evaluation servers in Section 4.1. We then show the bandwidth and latency results of PerMA-Bench's raw performance workloads in Section 4.2 and Section 4.3 to gain an understanding of the maximum performance of current PMem hardware. In Section 4.4, we discuss the results of database-related workloads and index structures to gain insight into the performance of PMem for more complex access patterns in actual systems and implementations. Finally, we investigate the impact of configurations affecting a single server in Section 4.5, i.e., by varying the number of DIMMs or the memory bus speed, as well as by disabling the prefetcher.

4.1 Setup And Methodology

We perform our evaluation on the four server configurations presented in Table 1. We refer to the servers as named in the table or via their label, e.g., Apache-128 or A-128. Apache Pass is the code name for the first generation/100 Series Optane DIMMs. Barlow

Table 1: Evaluated servers (single socket). Apache/Barlow refer to the code names of 100/200 Series Optane.

Name *(Plot Label)*	CPU	PMem	DRAM	OS
Apache-128 *(A-128)*	Intel Xeon Cascade Lake 18 Cores @ 2.7 GHz	6x 128 GB Intel Optane 100 Series @ 2666 MT/s | 15 Watt	6x 16 GB DDR4	Ubuntu 20.04 (5.4 kernel)
Apache-256 *(A-256)*	Intel Xeon Cascade Lake 18 Cores @ 2.6 GHz	6x 256 GB Intel Optane 100 Series @ 2666 MT/s | 18 Watt	6x 16 GB DDR4	Ubuntu 20.04 (5.4 kernel)
Apache-512 *(A-512)*	Intel Xeon Cascade Lake 24 Cores @ 2.4 GHz	6x 512 GB Intel Optane 100 Series @ 2666 MT/s | 15 Watt	6x 64 GB DDR4	Ubuntu 20.04 (5.4 kernel)
Barlow-256 *(B-256/B-D)*	Intel Xeon Ice Lake 32 Cores @ 2.2 GHz	8x 256 GB Intel Optane 200 Series @ 3200 MT/s | 15 Watt	8x 32 GB DDR4	Ubuntu 20.04 (5.4 kernel)

Pass is the code name for the second generation/200 Series Optane DIMMs. All servers are equipped with Optane DC Persistent Memory DIMMs. All Optane DIMMs are configured interleaved, i.e., striped in 4 KB blocks and accessed in App Direct mode. All measurements are performed on a single socket. To avoid measuring zeroing of requested pages by the kernel, files are pre-allocated and pre-faulted by default before running the benchmark, as recommended [21]. Unless stated otherwise, we generate a fixed amount of random data for each benchmark, depending on the benchmark configuration. In all experiments, we use 1 GB = 2^{30} Byte.

The A-256 server is configured with an 18 Watt average power budget per DIMM, the other 100 Series servers are configured to 15 W. While Optane allows the power budget to be set from 12 to 15 W (A-128, B-256) or 18 W (A-256/512), it is set by the vendor on the evaluated servers and cannot be reconfigured. Analyzing the power range allows us to show that even within the same generation, server configuration has a large performance impact.

When drawing performance conclusions, we also consider official performance numbers provided by Intel [19, 20] and previously reported numbers in research [10, 13, 24, 52, 56]. To provide a reference to well-known performance numbers, we also evaluate all experiments in DRAM. We show the results of the DRAM runs on Barlow-256 (shown as *B-D* in the plots). The DRAM performance in the Apache servers is lower, so we omit them for space reasons. Due to space limitations, we present only selected results in this paper. The full results can be found in our repository.

4.2 Raw Performance Workloads – Bandwidth

In this section, we present the bandwidth results of PerMA-Bench's raw performance workloads. We investigate the bandwidth utilization of all servers for sequential and random reads and writes. As the first part of our evaluation, these workloads provide insight into which performance can be achieved with current PMem hardware. Based on this knowledge, users can make decisions about the feasibility of PMem-specific implementations and their expected performance range in bandwidth-heavy applications.

4.2.1 Sequential Reads. We first discuss the throughput of sequential reads across all servers, as they are a core database access pattern. In this benchmark, we perform a sequential read workload on 50 GiB of randomly generated data with 4096 Byte access size and a varying number of threads. We show our results in Figure 4a. Within the first generation, we observe a difference of up to 44%, ranging from 29 to 42 GB/s. According to the official product sheet, A-128 and A-256 have the same read bandwidth under equal power budgets [19]. So the 24% difference between A-512 and A-128 is
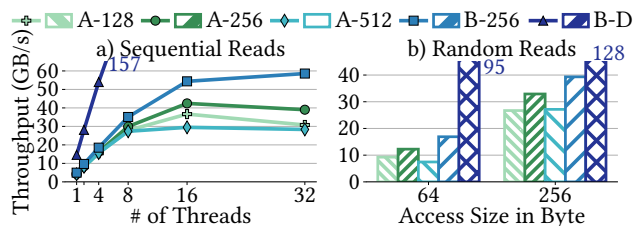
Figure 4: Sequential and random read bandwidth.
a) Fixed to 4096 Byte Access | b) Fixed to 16 Threads

based on the DIMMs, while the additional 15% improvement from A-128 to A-256 is based on the higher power budget.

B-256 achieves ~40% higher bandwidth than its first-generation counterpart A-256 with 58 GB/s and a 60% improvement over A-128. Barlow-256 is stocked with 8 DIMMs per socket instead of 6 DIMMs as in the 100 Series, leading to a 33% higher expected performance. Beyond this, we observe only a small improvement compared to the 18 W budget in A-256. But compared to A-128, which has the same read bandwidth as A-256 under equal power budgets [19], we see an additional 30% improvement. So for common 15 W setups, there is a notable performance increase between generations.

Our results show higher variance in throughput once hyperthreading is used and PMem limits are reached. This is observable for A-128 and A-256 with 32 threads, as both have only 18 physical cores. Apache-512 is more consistent, as it has 24 cores and B-256 even improves until 32 threads, which is the number of its physical cores. To achieve stable performance across servers, it is important to not exceed the number of physical cores when scanning data.

As a reference, B-DRAM's bandwidth reaches 98/145/157 GB/s for 8/16/32 threads, which is still significantly higher than PMem's. In the second generation of Optane DIMMs, the gap between PMem and DRAM even increases, from 2.3 to 2.7×. So while the bandwidth improved, there is still a clear advantage for DRAM in bandwidth-heavy applications. On the other hand, future systems must be able to process ~60 GB/s of data when reading from PMem, which is a major challenge when considering the cost of, e.g., random access data structures used in aggregations or joins. Thus, for most data-intensive applications, the bandwidth of sequentially accessing data stored in PMem is sufficient and does not constitute a bottleneck, unlike alternative secondary storage [10].

4.2.2 Random Reads. As indexes are core database components and essential to query performance, we investigate an index-inspired workload consisting of small, random, read-only operations, as commonly performed in hash or tree indexes. The bandwidth for uniform 64 and 256 Byte access across 10 GiB of random data is
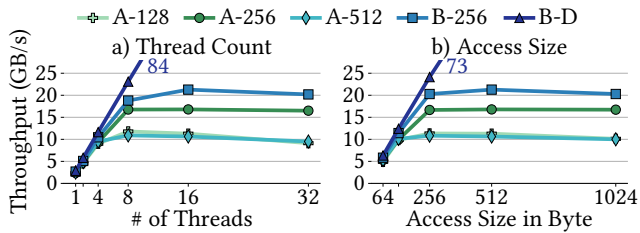
Figure 5: Thread and access size impact on sequential writes.
a) Fixed to 512 Byte Access | b) Fixed to 16 Threads

Figure 6: Impact of persist instruction on write bandwidth.
32 Threads | a) Sequential 512 Byte Write | b) Random 64 Byte Write

shown in Figure 4b. These access sizes represent the internal access granularity of Optane as well as standard cache-line-sized reads.

When considering PMem as random access memory as seen by the CPU, we see that it cannot achieve the same random to sequential ratio as DRAM. For 64 Byte, B-DRAM achieves 60% of the peak sequential performance, while the PMem servers achieve only 25%. Using the access granularity of Optane at 256 Byte, PMem achieves 67 − 92% and B-DRAM achieves 81%. Configurations with lower sequential performance achieve higher percentages in random access, reducing the gap from 40 to 20%. The second generation improves by 40% over A-256 and 80% over A-128. Similar to sequential access, we see little improvement over the higher-powered DIMMs but large gains compared to the lower wattage DIMMs.

When designing for PMem, it is important to keep the read amplification of small reads and the access granularity in mind, as 256 Byte access achieves more than two-thirds of the peak sequential performance. Compared to volatile DRAM, the performance is still significantly lower. But for applications that need fast access to small persistent records, e.g., point lookups in a key-value store or persistent index operations, 17+ GB/s of random 64 Byte access and 32+ GB/s of random 256 Byte access allow future systems to re-think the cost of persistence, especially when considering other bottlenecks such as network or alternative secondary storage.

4.2.3 Sequential Writes. Inspired by logging workloads in databases, we show a sequential write benchmark of 30 GiB in Figure 5. We investigate varying the number of threads and write size. We use *NoCache* writes, as logged data does not require temporal locality.

In Figure 5a, we evaluate the bandwidth for 512 Byte sequential writes. Within the 100 Series, we observe a large difference between A-128/512 and A-256. A-128/512 achieve around 12 GB/s sequential write bandwidth, as shown in previous work [10, 52, 56]. A-256, on the other hand, achieves close to 17 GB/s, due to the higher power budget. This is 40% higher than previously published results for 100 Series Optane. We verify this bandwidth utilization in VTune to confirm that there is a significant difference even within the first generation DIMMs. Thus, it is highly beneficial to configure PMem with a higher power budget of 18 W for write-heavy applications.

We observe a large improvement from the regular-powered 100 to the 200 Series. At its peak, B-256 achieves 21.6 GB/s, which is 75% higher than A-128/512 and goes beyond the expected 33% increase due to more DIMMs. The improvement for sequential writes is also higher than that of sequential reads. However, compared to the high-powered A-256, we observe only a 30% improvement, i.e., none beyond the extra DIMMs. Nonetheless, for common configurations used in previous research, there is a large increase that encourages utilizing PMem even more for sequential writes, e.g., when logging
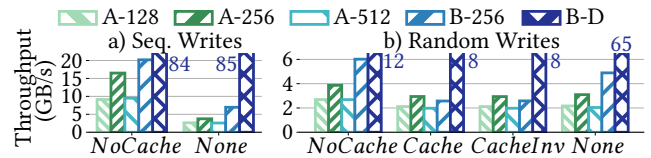
or using log-based storage systems. When persistence is not needed, PMem performs significantly worse than DRAM, which achieves more than 40/80 GiB/s for 16/32 threads, i.e., a difference of 4×.

When scaling the access size for 16 threads, as shown in Figure 5b, we observe that all servers require at least 256 Byte to achieve peak bandwidth. However, for A-256 and B-256, this is more important than for A-128/512. The latter two servers perform close to their maximum with 128 Byte, while the other two servers improve by at least 70% from 128 to 256 Byte. B-256 even improves from 256 Byte to 512 Byte, before dropping again slightly for larger sizes.

We also observe that all configurations decrease slightly when increasing the number of threads beyond a certain point. This point is at 16 threads for A-256 and B-256, while it is at 8 for A-128 and A-512. The ideal configuration of threads and access size depends on the server and differs across generations. While all 100 Series servers in our evaluation peak at 512 Byte access, B-256 peaks with 256 Byte access and 32 threads. These slight performance differences across all servers indicate that fine-tuning for the individual server yields higher performance and cannot be easily generalized.

4.2.4 Persist Instruction. We evaluate the impact of different persist instructions on the bandwidth for sequentially writing 30 GiB and randomly writing 10 GiB. The results are shown in Figure 6.

With Ice Lake CPUs, Intel offers persistence for all data in the eADR (cf. Section 2.2), making explicit flushing optional. However, we see that for sequential write access, not flushing data strongly decreases bandwidth utilization by up to 4× compared to explicit stores. Randomly evicted cache lines impair write-combining within the DIMMs, resulting in random-access-like write performance. Thus, explicitly flushing is beneficial for sequential writes, even with the second generation server and eADR. For B-DRAM, on the other hand, there is no difference between both options, as there is no write amplification when randomly evicting data from cache.

For random 64 Byte writes, we evaluate all four persist options, i.e., *NoCache*, *Cache*, *CacheInvalidate*, and *None*, as shown in Figure 6b. Explicitly bypassing the cache via non-temporal stores achieves the highest bandwidth in all servers. Non-temporal stores also surpass explicit flushing in DRAM, which shows that there is a ~25% overhead of passing stores through the cache hierarchy.

Issuing no flush (*None*) is only marginally better than explicit temporal stores for Apache servers, but nearly 2× better for B-256. Thus, in eADR servers, users benefit from reduced code complexity and higher bandwidth when not explicitly flushing. Based on the different performance characteristics of flushes in the 100 and 200 Series, future work should re-evaluate flushes in existing PMem-optimized index structures. Significant work has been done to reduce the number of flushes and to decide which instructions to use [36, 39, 46, 57, 59], but it is unclear whether the choices apply to future Optane or are tailored only towards 100 Series.
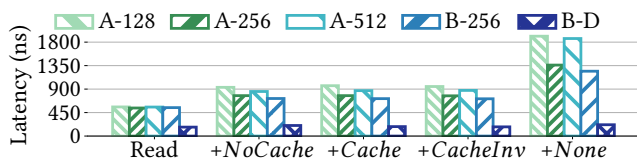
Figure 7: 256 Byte random read + write latency. *16 Threads.*

Figure 8: Double-flush latency. *64 Byte read + 2× 64 Byte write.*

4.3 Raw Performance Workloads - Latency

In this section, we evaluate the latency of raw PMem access across all servers. Understanding latency allows users to evaluate the feasibility of PMem-specific implementations in latency-critical applications and gain insight into the expected performance.

4.3.1 Operation Latency. In Figure 7, we show the average latency of five operations in PMem: a 256 Byte read and a 256 Byte read followed by a 256 Byte write to the same location with the supported persist instruction (*NoCache, Cache, CacheInvalidate, None*). We perform 100 million operations on 10 GiB of data and sample every 5000th operation. The results show that read latency is consistent across servers. While their read bandwidth differs significantly, there is no difference in latency. However, the latency is still 3× higher than in B-DRAM, which is also approximately the factor between both for random read bandwidth.

When following the read with a write operation, latency is not equal on all servers. We observe that A-256 and B-256 have lower latency than the other servers across all flush operations. While read latency is bound by the latency of physical media access, write latency is more nuanced, as writes do not need to be flushed to the medium to be considered complete. Flushing to a full write pending queue blocks the caller and has higher latency. Due to the servers' higher bandwidth, more writes are flushed, freeing up space in the queue. The latency across all explicit persist instructions is consistent within each server, with *NoCache* having a slightly lower latency. Not flushing when writing 256 Byte of data has the highest latency (and lowest bandwidth), as randomly evicted cache lines cause high write amplification, blocking the write pending queues.

When running the same experiment with 64 Byte access, the latency is nearly identical for all instructions except *None*. For *None*, omitting the flush for consecutive memory addresses prevents efficient write combining. But for 64 Byte writes, write combining cannot be performed, so there is no disadvantage. For latency-critical applications, the choice of persist instruction is not relevant from a performance perspective. However, when writing more than 64 consecutive Bytes, an explicit flush should be used to benefit from write combining, even in eADR servers such as B-256.

While we observe a bandwidth increase across generations, latency has not improved. Additionally, we notice a correlation between higher available bandwidth and reduced write latency. However, most research focuses on bandwidth as a limiting factor of PMem compared to PMem. Our results raise the question of whether future designs should shift their focus towards avoiding latency instead. Especially in latency-bound applications, new approaches may sacrifice bandwidth to reduce latency, e.g., by writing additional data, which in turn reduces additional random lookups.

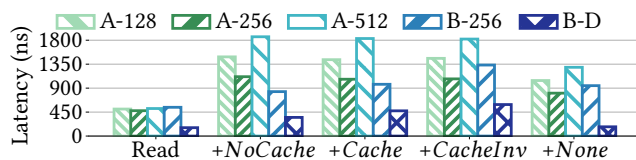*4.3.2 Double Flush Latency.* Current PMem systems often store metadata for tree nodes, hash buckets, or storage pages in a single cache line and repeatedly update, e.g., counters, bitsets, or locks in that cache line [5, 39, 46]. While this practice is often used, some authors discourage it due to high latency [29, 51]. In this section, we evaluate the impact of different persist instructions when flushing the same cache line twice (*double-flush*). We perform 100 million operations on a 10 GiB range and sample every 5000th operation.

In Figure 8, we see that the double-flush latency for A-128 and A-512 is 2× of the single flush latency shown in Figure 7. In comparison, the double-flush latency of A-256 and B-256 is only marginally higher than the single flush. Under high load, these servers achieve higher bandwidth, which results in less pressure on the write queue and, in turn, reduces the latency of individual writes.

By comparing *Cache* and *CacheInvalidate*, we see that in the second generation Optane DIMMs there is actually a difference between the used persist instruction. The 2nd generation Xeon CPUs in the Apache servers do not fully implement `clwb`, internally mapping it to `clflushopt` instructions instead. As B-256's CPU supports true `clwb`, we observe that invalidating flushes (via `clflushopt`) have a higher latency due to the required memory read between the writes. While the latency of B-256 is slightly higher than that of A-256 for *None* and *CacheInvalidate*, it is lower for *Cache* and *NoCache* stores. Thus, we conclude that using non-invalidating flush operations on the same cache line is preferable for 200 Series Optane, as it does not include the penalty of invalidating the cache line, which occurred in the 100 Series.

4.4 Database-Related Workloads

In this section, we present the results of database-related workloads modeled in PerMA-Bench. The memory access patterns in these workloads are based on actual implementations of PMem index structures. Expressing complex memory access patterns in PerMA-Bench allows users to gain insight into their design choices at a memory-performance level before having to implement numerous options. This also helps to understand where performance is lost and where operations are close to the raw memory performance. Our results show that both existing systems that were designed for a specific server configuration and systems that were designed pre-Optane do not fully utilize the performance improvements of second-generation Optane.

First, we discuss the performance of PMem-aware index-inspired workloads compared to a DRAM-only version in Section 4.4.1. Then, we evaluate the performance of actual PMem-aware implementations based on our findings in PerMA-Bench. We show that current designs often cannot fully utilize the performance improvements of 200 Series Optane and avoiding explicit flushes in eADR does not always yield the best results. To provide more general solutions, future work on PMem-aware systems must expand beyond designs evaluated on a single setup and reconsider design choices that may have been altered by newer characteristics of 200 Series Optane.
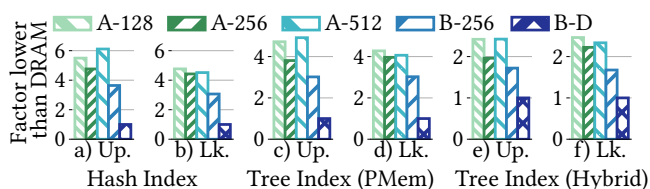
Figure 9: Factor *X* lower throughput than DRAM of updates (Up.) and lookups (Lk.) in PMem index workloads. *32 Threads.*

Figure 10: Hash index in PerMA and Dash. *16 threads. PerMA: 512 Byte lookup + 2× 64 Byte Cache update. Dash: 8/8 Byte key/value.*

4.4.1 Database Index Operations. In this section, we cover a wide range of database-related index workloads in PerMA-Bench, run with 32 threads. When designing PMem systems, a DRAM-based version is often used as a comparison to show the efficiency of the chosen design. To cover this, we model our access based on PMem but run the experiments in both PMem and DRAM. We use the throughput of all access in DRAM as a baseline and show the factor *X* by which the throughput of the same access in PMem is lower. From our evaluation in the previous section, we know that DRAM's random read bandwidth is 5 – 10× higher than PMem's for 64 Byte access and 3 – 4.5× for 256 Byte access (cf. Section 4.2.2). DRAM's random write bandwidth is up to 2 – 6× higher for 64 Byte with explicit flushes, and 3 – 6× higher for 256 Byte writes. Unflushed writes have an up to 30× higher bandwidth. This shows the strong imbalance between DRAM and PMem for random access data structures, especially if only tiny amounts of data are changed, e.g., an 8 Byte pointer in an index. Understanding these differences is important to optimize access to each memory type accordingly. We perform 100 million operations across 10 GiB in all workloads.

We model the access patterns for a hash index like Dash [39]. For lookups, its access consists of a 512 Byte read, representing two adjacent 256 Byte buckets, followed by two 64 Byte cache flushes for updates. In Figures 9a and b, we see how the improved random access performance of B-256 closes the gap to DRAM. As the access pattern of a hash index is $O(1)$ by design, the improvement should apply directly to real workloads. However, PMem still performs significantly worse than DRAM, as small updates to the index have a high write amplification, e.g., 16× for 16 Byte updates.

We model a PMem-only tree index after FAST+FAIR [18]. Based on the authors' implementation, we issue 3× 512 Byte random reads for a lookup and 4× 64 Byte cache flushes for an insert, as 50% of a node has to be moved on average when inserting a value into a leaf in FAST+FAIR. We see an average performance of around 4× in the 100 Series and 3× in the 200 Series. As this design operates on 512 Byte nodes, we observe the expected ~4× higher DRAM bandwidth for lookups. Updates perform slightly worse, but better than the raw DRAM bandwidth would suggest. However, four flushes per update (on average) are expensive, even in DRAM, as each one entails an sfence that clears all write buffers.

We represent a hybrid DRAM-PMem tree through FPTree [46]. PerMA-Bench issues 2× 2048 Byte DRAM reads and 1× 1024 Byte PMem read for lookups, followed by 3× 64 cache flushes for updates, based on the node sizes in the implementation that we use [15]. We see that the hybrid tree is closer to DRAM in relative performance, as most of the random lookups occur in DRAM. Thus, we see an overhead of 1.7× for placing the leaves in PMem on B-256. The Apache servers have a higher overhead, as their random read PMem bandwidth is lower and their DRAM is generally slower.
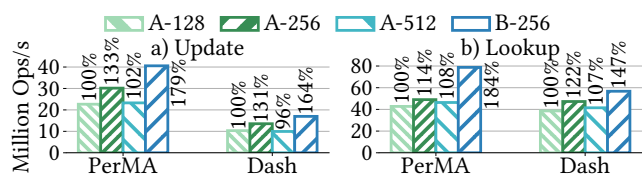
Overall, we see that random access patterns in PMem index structures perform significantly worse than in DRAM. Especially with small random writes, performance drops compared to PMem. We also note that these patterns are not optimized for DRAM, meaning that without the explicit flushes, even higher performance is observed. In the case of the hash index, we observe a 30% increase in DRAM when omitting the flush. But while PMem-based indexes cannot achieve the raw performance of DRAM, they offer (full) persistence and recovery with an average performance drop of only 4×. In addition, the price per GB of PMem is 4 – 6× lower than DRAM, striking a balance in price-performance.

4.4.2 Hash Index. In this section, we compare the PMem-aware hash index Dash [39] and the corresponding operations' memory access pattern in PerMA-Bench. We prefill Dash with 100 million entries before performing 100 million operations using the benchmark tool provided by the authors. The results are shown in Figure 10.

PerMA-Bench provides a good upper bound estimate of performance based on memory access alone. For all Apache servers, the relative performance in raw access transfers directly to the relative performance of Dash. The insert performance is more complex in Dash, as it includes regular inserts, displacement, overflow buckets, and resizing. As these depend heavily on the implementation, it is not possible to model all in one custom operation. For our comparison, we assume an idealized insert without resizing and displacement and, thus, overestimate the insert performance. Developers can model all operations individually and run the benchmarks separately. This provides a good overview of the individual operations' performance and the results can be combined to determine the overall performance depending on the configuration parameters of the desired implementation. If a displacement takes *X* μs, depending on the ratio of inserts to displacements, e.g., 3:1, we can add *X*/3 μs to each insert to combine both operations.

PerMA-Bench also provides insight into potential areas of improvement. Dash achieves close-to-raw performance for lookups, which indicates that there is not much room for optimization in designs that access at most two buckets to retrieve an entry.

Finally, we observe that Dash underperforms on B-256. Unlike the 100 Series servers, Dash's relative raw lookup performance in PerMA-Bench is 40% higher than the actually achieved throughput. When investigating the performance in more detail, we see that Dash spends nearly 20% of all cycles on machine clears caused by memory ordering violations, which do not occur on the Apache servers. While we use Dash in this experiment, this problem is not Dash-specific but a general issue in current systems designed for PMem. Due to the high price, researchers often have access to only one server, resulting in current research focusing on a single configuration during development. Our results show that its performance is not yet understood well enough to generalize
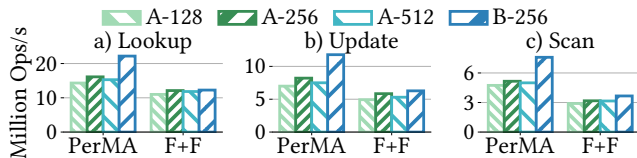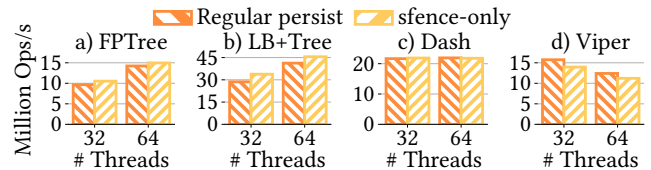
Figure 11: Tree index in PerMA and FAST+FAIR. *16 threads.*

Figure 12: Impact of eADR on write performance of different PMem key-value storage designs *(Barlow-256). 32/64 threads.*

from one server to all, especially across generations. Now that the second generation Optane DIMMs are available, it is beneficial to consider more than one server to develop more general PMem-aware solutions in the future. In the following section, we show that performance limitations occur also in other index structures.

4.4.3 Tree Index. In Figure 11, we show the FAST+FAIR BTree implementation [18] and its modeled memory access in PerMA-Bench. FAST+FAIR is a popular PMem-only BTree implementation that was designed pre-Optane. We prefill 100 million records before performing 100 million operations using the benchmark tool provided by the authors. The ideal-insert assumption as in Dash also applies to this benchmark. Across all operations, we see that the raw performance of B-256 is significantly higher than that of the 100 Series servers. However, this does not translate to FAST+FAIR, as its performance improves only marginally across generations.

As it was designed pre-Optane, it does not include various optimizations made in later designs. When taking a closer look at the execution, we see that 30% of all cycles are consumed by bad speculations, front-end stalls, and computation. These 30% are reflected in the performance difference between PerMA-Bench and FAST+FAIR. Compared to more recent work, FAST+FAIR also makes use of heavy-weight locking instead of atomics or hardware transactional memory. This overhead prevents FAST+FAIR from scaling with the higher performance of the newer Optane DIMMs.

These results indicate that general implementations without explicit knowledge of the underlying PMem technology do not scale well with better hardware. The memory access in FAST+FAIR is not optimized towards Optane, e.g., by requiring many flushes for updates due to sorted nodes. In another experiment, we observe better scaling results for the pre-Optane FPTree [46], as we used a version that was re-implemented more recently on Optane [15]. Adding to our insights on the Optane-tuned hash index Dash in Section 4.4.2, we conclude that it is also not viable to rely only on general PMem assumptions, as done in pre-Optane designs. It is beneficial to tune PMem-aware systems across a range of current hardware to capture the intricacies of Optane without optimizing solely for one server configuration. Especially with increasing PMem performance as in B-256, bottlenecks may shift from PMem access to, e.g., CPU or DRAM, requiring a balance between them.

4.4.4 Impact of eADR. In 200 Series Optane, eADR guarantees the persistence of data that resides in the cache, thus, making explicit flushes unnecessary. In this section, we evaluate the impact of omitting explicit flushes and issuing only `sfence` instructions[1] on various PMem-aware key-value storage designs. In this evaluation, we also include LB+Tree [36], a hybrid DRAM-PMem B+Tree optimized highly and explicitly for Optane, and Viper [5], a hybrid

DRAM-PMem log+index key-value store that is designed for operations on larger items than 8 Byte index entries. This gives us a broader overview of PMem storage designs. We show the results of 32 and 64 thread runs on B-256 for FPTree, LB+Tree, Dash, and Viper in Figure 12. For the three index structures, we use 8/8 Byte key/values. For Viper, we use 16/200 Byte key/values. We evaluate FPTree with pibench [34]. For LB+Tree, Dash, and Viper we use the respective benchmark tools provided by the authors.

Our results show that for the tree-based designs, removing explicit flushing improves performance. However, for Dash, we observe no improvement and even a slight decrease for 64 threads. When storing larger records in Viper, we observe that not explicitly persisting reduces performance by 10%. Viper is designed to leverage sequential PMem writes, which are lost through random cache line eviction when omitting flushes (cf. Section 4.2.4).

Our results show that eADR is not a silver bullet for future PMem-system design. Developers must still understand their access patterns and evaluate whether they benefit from explicit flushes or not. Based on this, we encourage future work that explicitly compares low-level flush performance in PMem index/storage designs to provide an overview of benefits and downsides in this space.

4.5 Single Server Performance

In this section, we investigate configurations and settings that impact a single server. First, we show the impact of Intel's hardware prefetchers on memory bandwidth and discuss the implications this has for general PMem-aware system design (Section 4.5.1.) Next, to provide insight into the performance of partially stocked servers with older or lower-end components, we evaluate the performance of a single server with varying configurations. This is important, as buying PMem in large quantities is currently still very expensive, which is why users may not always choose fully stocked servers (one DIMM per available slot) with the highest configuration and latest components. To this end, we investigate the impact of the memory bus speed on PMem bandwidth (Section 4.5.2), followed by the impact of the number of DIMMs (Section 4.5.3).

4.5.1 Prefetcher. Throughout our evaluation, we observe workloads for which PerMA-Bench achieves lower performance than expected due to Intel's hardware prefetching behavior. This observation has been made in previous work [10] and we investigate it further in Figure 13. We actively disable all hardware prefetchers and measure the bandwidth utilization of 200 million random reads across 10 GiB with varying sizes. In the top row, we see that Apache-256 performs worse when the prefetcher is active for 1024 Byte reads with both 16 (a) and 32 threads (b). On the other hand, Barlow-256 performs worse for 512 and 1024 Byte but only with 16 threads (c). With 32 threads, the prefetcher impacts the
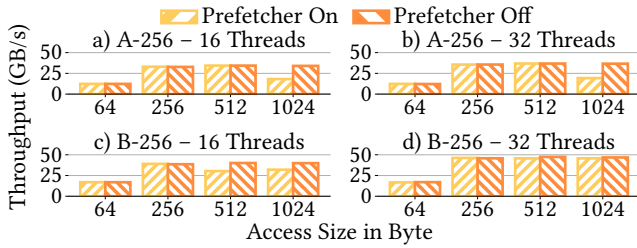
---
[1]`sfence` is still required to ensure correct ordering.

Figure 13: Impact of prefetcher on random read bandwidth.

Figure 14: Performance-impact of varying memory bus speeds (in MT/s) on *Apache-256* and *Barlow-256*. *32 threads.*

performance only marginally (d). For the impacted runs, we observe higher bandwidth utilization in VTune, which indicates that the prefetcher is mistakenly fetching unnecessary data and thus reducing the effective bandwidth.

To transfer these insights to a real system, we run a micro-benchmark on Barlow-256 with the key-value store Viper. In this, we observe that disabling the prefetcher for 200 Byte values results in a 40% performance increase for get requests with 32 threads. But for 64 threads, a disabled prefetcher reduces performance by 30%. So while it is not generally advisable to disable the hardware prefetchers, its impact should be taken into account when designing, profiling, and optimizing systems that operate on larger data chunks, e.g., buffer managers or storage engines.

4.5.2 Memory Bus Speed. Optane modules share the memory bus with regular DRAM DIMMs, so they must run at the same memory bus speed. While DRAM often supports higher speeds than Optane, this is not always the case, e.g., when using older DRAM modules, requiring users to reduce the speed of their PMem. To investigate which impact this has on PMem performance, we configure the memory bus in Apache-256 and Barlow-256 to different speeds. In Figure 14, we show the performance of sequential reads and writes, random reads, as well as read latency, and custom hash and tree index operation throughput. We use the same configurations as in the previous benchmarks. Apache-256's DRAM supports up to 2933 MT/s but is limited by PMem at 2666 MT/s, which we choose as the baseline. We also configure the bus speed to 2400 and 2133 MT/s to artificially slow down the server. Barlow-256's DRAM and PMem both support 3200 MT/s and we compare this to 2933 MT/s[2].

Our results show that PMem read bandwidth is impacted only marginally by reduced memory speed and not at all for write bandwidth. With a bus speed of 2666 MT/s, the theoretical bandwidth limit is ~20 GiB/s (= $2666 \times 10^6 \times 8$ Byte). In a server with six DIMMs, this allows for a theoretical maximum of ~120 GiB/s. Reducing the bus speed to 2133 MT/s results in a limit of ~16 GiB/s per DIMM and ~96 GiB/s across all DIMMs, i.e., a drop of 20%. However, PMem cannot supply data at this rate and stays significantly below the limit. The marginal difference in performance across the configurations is a result of slightly increased access latency due to fewer transfers per second. For DRAM, on the other hand, we observe a 20% bandwidth drop as it can provide data at the maximum frequency. Overall, we observe little to no performance drop and conclude that the selected memory bus speed is negligible for current Optane PMem. This also holds for 200 Series Optane, where it may be more common to have DRAM that limits the bus speed, as 3200 MT/s is also the current speed supported by DRAM.

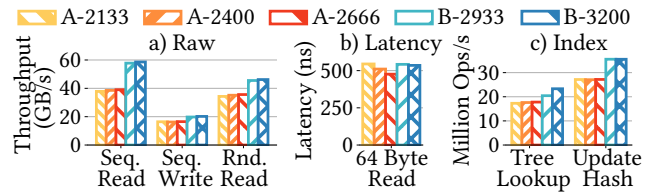[2]The server's BIOS does not allow configurations below 2933 MT/s.

4.5.3 Number of DIMMs. To provide insight into the performance of servers with only partially filled PMem slots, we evaluate PMem with a varying number of DIMMs. This allows us to both draw conclusions about how to stock a PMem server and also to make results of recent studies [30, 36, 54], which ran experiments on partially stocked servers, comparable to a full server. All experiments are run on a single socket of Apache-256 with 16 threads, we do not measure cross-socket performance. We run the experiment with all supported configurations, i.e., with 1/2/4/6 DIMMs. We physically remove the unused PMem DIMMs but keep all six DRAM DIMMs, following the official memory population guide [12]. In Figure 15, we show the bandwidth utilization of sequential and random read and write workloads, as well as random read latency and operations per second for tree index lookups and hash index updates.

In Figures 15a and b, we show the absolute bandwidth and relative improvement over the *1 DIMM* configuration. For reads and writes, we observe two different patterns. For write bandwidth, we observe a near-perfect linear scale. Each DIMM is fully saturated and constitutes a bottleneck. By adding more DIMMs, we evenly distribute the load across all available DIMMs until we have reached the maximum bandwidth. For sequential writes, we see a slightly super-linear scale. With fewer DIMMs, the load on the individual DIMMs is higher and the write combining buffers receive more requests. They cannot combine adjacent stores as efficiently, resulting in increasing write amplification. When using 32 threads, this effect is even stronger, as the buffers are overloaded with requests. In this case, we observe an 11.9× increase from one to six DIMMs.

Read bandwidth utilization shows super-linear scaling from one to two and from two to four DIMMs. On currently supported CPUs, configuring a server with fewer than six PMem DIMMs results in an *unbalanced* configuration. While these unbalanced setups are supported, they are not recommend [44]. Memory controllers cannot optimize the memory layout and must create multiple interleave sets, resulting in worse performance. Additionally, the server must run in single-channel mode when using a single DIMM, which further reduces performance. For sequential reads with six DIMMs, PMem achieves 42 GiB/s or 7 GiB/s per DIMM. With 1 DIMM, it achieves only 5 GiB/s, which is ~30% worse.

Read latency decreases with more DIMMs, as the contention on each is reduced. Once the load is distributed, latency is not affected as much. This translates directly to the tree and hash index operations shown in Figure 15d. The tree lookup is impacted more by latency, so its performance does not improve as much as raw bandwidth. The hash index updates reflect the scaling of random writes with an influence of increased preceding read bandwidth.

Overall, we see a predictable performance pattern when using more than one DIMM, which allows us to transfer the results of previous work by approximately scaling the used number of DIMMs to
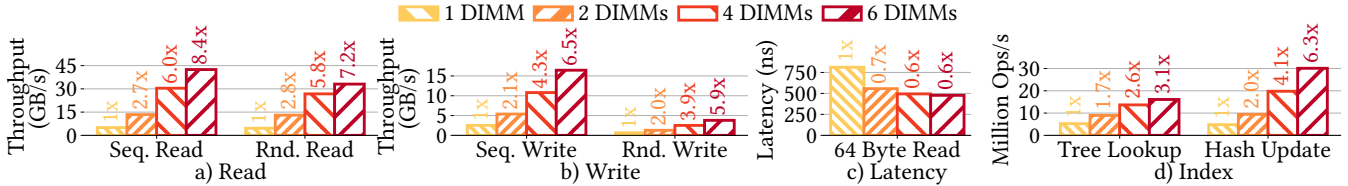
Figure 15: Impact of number of DIMMs in the server *(Apache-256)*. All runs with 16 threads. Sequential/random reads with 4096/256 Byte access size. Sequential/random writes with 512/64 Byte access size and *NoCache*. Tree with 512 Byte nodes. Hash with 256 Byte buckets.

six. Our results show that Optane PMem should be configured fully stocked and *balanced* to achieve maximum performance. However, if this is not possible, prefer multiple smaller DIMMs, e.g., 4× 128 GB = 512 GB, over fewer larger ones, e.g., 1× 512 GB to achieve a *near-balanced* configuration and reduce the load on the individual DIMMs, which otherwise quickly become over-saturated.

5 SERVER PRICE-PERFORMANCE

A major selling point of persistent memory is its lower price for higher capacity than DRAM. However, there is very little actual price-performance analysis in existing research. To provide insight into this, we perform a price-performance comparison across all evaluated servers. As major cloud vendors do yet not offer PMem, we base our analysis on the price as listed by Dell when configuring a server with Optane [49]. We note that the actual price of PMem differs slightly depending on the source, country, and currency[3], but the relative difference between them is consistent. As such, our focus is not on the exact monetary values but rather on the relative difference between the servers. We base the performance-related values on the price per GB to explicitly exclude the price of higher capacity. As the A-128 server does not support 18 Watt, a comparison against the 18 Watt A-256, which supports this improvement, does not yield unfair results. However, an 18 W 512 GB server may achieve better price-performance than in our evaluation. To the best of our knowledge, this is the first extensive price-performance comparison of PMem across various configurations.

We show the price-performance results in Table 2. We first compare the PMem servers and then draw an overall comparison to DRAM. The price per GB capacity increases with the DIMM size, resulting in an up to ~80% difference within the 100 Series. Across generations, i.e., Apache-256 to Barlow-256, the price per GB increases by ~20%. The price for sequential and random read throughput differs only slightly between the various PMem servers. However, Apache-512 is an outlier, as it offers the lowest performance (cf. Section 4.2) but the highest price, even when considering a 20% performance improvement through an 18 W configuration.

We see a wider range in the price-performance for both sequential and random writes. They differ by up to 1.5× and 2.9×, respectively. Due to significantly higher write bandwidth (cf. Section 4.2), it becomes apparent why both Apache-256 and Barlow-256 achieve up to 25/40% lower prices than the cheapest server (Apache-128) for sequential/random writes.

The price difference for hash index updates (normalized to 1 million operations) is a mix of the random read and write prices. Due to the low random read variance, the price variance across servers is not as significant as for pure random writes. The normalized 1 million tree lookups represent a random read workload and their relative price-performance ratio does not differ significantly from the random read ratios.

Regarding bandwidth and latency, DRAM outperforms PMem significantly (cf. Sections 4.2 and 4.3). However, DRAM's price per GB is up to 6.4× higher than PMem's. Our results show that PMem is competitive with DRAM in most of the raw access patterns, i.e., sequential/random reads and sequential writes. DRAM outperforms PMem for random writes, as PMem's bandwidth is significantly lower for such workloads. This read/write split also extends to more complex access in data structures, where DRAM outperforms PMem for write-intensive operations but offers little to no benefit for read-only access.

To optimize the price-performance of a server for a given workload, users have to choose which and how many DIMMs to buy. While this choice is heavily workload-dependent, our results show that users can use the following rule of thumb: *maximize the number of DIMMs for a target capacity.* If the workload fits into DRAM, there is no need for PMem, as this requires special CPUs and increases the overall cost. If the workload exceeds DRAM, users should use *n* DIMMs of the smallest size that offer the needed capacity, i.e., users should prefer 4× 128 GB over 2× 256 GB over 1× 512 GB. Our results in Section 4.5.3 show that the performance scales almost linearly, so while 256 GB DIMMs have a better individual performance than the 128 GB DIMMs, two 128 GB DIMMs outperform one 256 GB DIMM while providing the same capacity at a lower overall price. Thus, use larger DIMMs only when the capacity is needed, as the price grows disproportionately higher for larger capacity.

---

[3]We checked dell.de, dell.com, and hpe.com in February 2022 and December 2021.

Table 2: PMem price-performance comparison in Euro (€). See Table 1 for server info.

System	€ per DIMM	€/System *(GB capacity)*	€/GB capacity	€/GB/s seq. read	€/GB/s rnd. read	€/GB/s seq. write	€/GB/s rnd. write	€/100ns latency	€/update hash index	€/lookup tree index
Apache-128	1180	7080 *(768)*	**9.21**	0.25	0.34	0.78	3.43	**46.34**	0.39	0.56
Apache-256	2750	16500 *(1536)*	10.74	0.25	**0.33**	0.64	2.78	52.38	0.36	0.60
Apache-512	8500	51000 *(3072)*	16.60	0.56	0.61	1.52	6.18	83.94	0.71	0.96
Barlow-256	3270	26160 *(2048)*	12.77	**0.22**	**0.33**	**0.60**	2.12	64.49	**0.27**	**0.42**
B-DRAM	1900	15200 *(256)*	59.37	0.38	0.46	0.70	**0.91**	80.61	0.46	0.84

6 DISCUSSION

In this section, we present key takeaways from running PerMA-Bench and PMem-aware systems on various server configurations.

PMem Configurations. Our results show that the exact server configuration has a large impact on PMem performance. We identify four aspects that have not yet been studied in detail.

1) **DIMM size:** We show that the choice of DIMM size does not only impact capacity but also performance, especially as only the 256 and 512 GB DIMMs support higher power budgets.

2) **Power budget:** The 18 Watt power budget of Apache-256 improves write bandwidth by up to 40%, which is a major improvement considering PMem's otherwise limited write bandwidth. If possible (only for 256 and 512) and supported by the server, users should increase the power budget of their DIMMs.

3) **Number of DIMMs:** Varying the number of DIMMs has a predictable, close-to-linear impact on performance unless only a single DIMM is used. This causes an imbalanced memory configuration and a fallback to single-channel execution. For maximum performance, fully stocked servers should be chosen.

4) **Memory bus speed:** While DRAM and PMem must run with the same memory speed, we show that this does not impact PMem. The theoretical limits exceed PMem's performance, so users can reduce the speed if needed without losing performance.

Future PMem Research. We identify four additional aspects that impact PMem-aware implementations. With the increasing performance of PMem, previous bottlenecks may shift away from PMem to, e.g., the CPU, requiring more advanced and specialized implementations. As PMem is a very new and evolving memory technology, a detailed understanding and optimization level known from DRAM must still be developed for it.

5) **Hardware utilization:** We observe that existing indexes do not fully utilize the performance improvements of the second generation. With more Optane configurations available, it is essential to tune future designs across a wider range of servers to achieve more stable performance.

6) **Persist instruction:** While the choice of persist instruction for random writes impacts bandwidth by only 30% in the 100 Series, it makes a difference of up to 2.5× in the 200 Series. Future work has to consider this and re-evaluate which choice of persist instruction is best-suited for different designs. Especially, now that the 200 Series allows for two different cache flushes, non-temporal stores, and no stores via eADR.

7) **eADR:** We show that omitting flushes due to eADR does not always yield the best performance. It remains important to understand when explicit flushes improve bandwidth utilization and latency, and when they do not.

8) **Prefetcher:** The prefetcher has an unexpected negative impact on certain workloads. While it should not be disabled, developers have to be aware that their system may be influenced by it.

Price-Performance. Within the first Optane generation, we identify 512 GB DIMMs to have the worst price-performance by a large margin. But overall, we show that PMem's price-performance is generally competitive with DRAM or even better. This allows PMem to be used as both explicit persistent memory or as cheaper and larger volatile memory, potentially even allowing for in-memory processing of workloads that previously did not fit into DRAM.

7 RELATED WORK

In this section, we briefly discuss related work around PMem.

Persistent Memory Analysis. Various studies on the performance of PMem have been conducted. Earlier work focuses on performance assumptions and latency ranges to evaluate PMem in the context of various applications [1, 46, 53]. More recently, the performance of Intel's Optane DC Persistent Memory is investigated in more detail [6, 10, 13, 24, 51, 56]. These studies provide insight into the performance details of individual servers. In this work, we evaluate and compare the performance of PMem across various setups and show that this is needed to gain a better understanding of overall PMem behavior. These early benchmarks and existing tools such as fio [3] often run hard-coded queries or cannot represent complex access patterns and varying persist instructions, which are both essential to understand the performance of PMem for database components. To represent access patterns of current PMem systems, PerMA-Bench offers customizable, mixed PMem-DRAM pointer-chasing with locality-aware store instructions.

Persistent Memory Applications. The use of PMem is widely studied in index structures [9, 17, 32, 36, 39, 41, 46, 59], key-value stores [5, 8, 35], database systems [1, 2, 40, 40, 45, 50], and filesystems [28, 43, 55]. We extract common access patterns from this work and define the workloads in PerMA-Bench based on them.

8 CONCLUSION

In this paper, we propose PerMA-Bench, a configurable benchmark framework that allows users to evaluate the bandwidth, latency, and operations per second for customizable database-related PMem access. We perform an extensive analysis across four PMem servers of the first and second Optane generation, with varying configuration options, such as DIMM power budget, memory bus speed, and number of DIMMs per server. We show their performance impact and raise awareness for the configuration space of PMem. We validate our results with existing implementations and show that they do not fully utilize the performance improvements across Optane generations. We show that the choice of persist instruction has a high performance impact and that avoiding explicit flushes in eADR does not always yield the best results. Finally, we perform a price-performance comparison across all evaluated servers. While there are great differences between Optane DIMMs, PMem is generally competitive with DRAM. This allows PMem to be used as both explicit *persistent* memory or cheaper and larger *volatile* memory.

PMem is still a new and evolving technology and research into PMem-aware databases is still in its infancy compared to DRAM. We present directions for future designs, implementations, and evaluation of PMem solutions that are needed to fully understand and utilize the hardware. We make our evaluation results available and with PerMA-Bench, we hope to lead the way to a common understanding of PMem performance by gathering and comparing various existing configurations and future PMem hardware.

REFERENCES

[1] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. 2015. Let's talk about storage & recovery methods for non-volatile memory database systems. In *SIGMOD '15*. ACM, 707–722. https://doi.org/10.1145/2723372.2749441

[2] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-behind logging. *Proceedings of the VLDB Endowment* 10, 4, 337–348. https://doi.org/10.14778/3025111.3025116

[3] Jens Axboe. 2022. fio: Flexible I/O Tester. https://github.com/axboe/fio

[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 *IEDM Technical Digest. IEEE International Electron Devices Meeting*. IEEE. https://doi.org/10.1109/iedm.2004.1419228

[5] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. 2021. Viper: An Efficient Hybrid PMem-DRAM Key-Value Store. *Proceedings of the VLDB Endowment* 14, 9, 1544 – 1556. https://doi.org/10.14778/3461535.3461543

[6] Maximilian Böther, Otto Kißig, Lawrence Benson, and Tilmann Rabl. 2021. Drop It In Like It's Hot: An Analysis of Persistent Memory as a Drop-in Replacement for NVMe SSDs. In *DaMoN '21*. ACM. https://doi.org/10.1145/3465998.3466010

[7] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. 2020. uTree: a persistent B+-tree with low tail latency. *Proceedings of the VLDB Endowment* 13, 12, 2634–2648. https://doi.org/10.14778/3407790.3407850

[8] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *ASPLOS '20*. ACM, 1077–1091. https://doi.org/10.1145/3373376.3378515

[9] Zhangyu Chen, Yu Huang, Bo Ding, and Pengfei Zuo. 2020. Lock-free Concurrent Level Hashing for Persistent Memory. In *ATC '20*. USENIX Association, 799–812.

[10] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. 2021. Maximizing persistent memory bandwidth utilization for OLAP workloads. In *SIGMOD '21*. ACM. https://doi.org/10.1145/3448016.3457292

[11] Hewlett Packard Enterprise. 2021. HPE NVDIMMs Memory – Overview. https://support.hpe.com/hpesc/public/docDisplay?docId=c05302373

[12] Hewlett Packard Enterprise. 2021. Server memory and persistent memory population rules for HPE Gen10 servers with Intel Xeon Scalable processors technical white paper. https://www.hpe.com/psnow/doc/a00017079enw

[13] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. 2020. Understanding the idiosyncrasies of real persistent memory. *Proceedings of the VLDB Endowment* 14, 4, 626–639. https://doi.org/10.14778/3436905.3436921

[14] Xiaochen Guo, Engin Ipek, and Tolga Soyata. 2010. Resistive Computation: Avoiding the Power Wall with Low-Leakage, STT-MRAM Based Computing. In *ISCA '10*. ACM Press, 371–382. https://doi.org/10.1145/1815961.1816012

[15] Yuliang He, Duo Lu, Kaisong Huang, and Tianzheng Wang. 2022. Evaluating Persistent Memory Range Indexes: Part Two. *arXiv:2201.13047 [cs]*. http://arxiv.org/abs/2201.13047

[16] Uwe Heinz. 2020. SAP HANA and Persistent Memory. https://blogs.sap.com/2020/01/30/sap-hana-and-persistent-memory

[17] Kaisong Huang, Tianzheng Wang, Darien Imai, and Dong Xie. 2022. SSDs Striking Back:The Storage Jungle and Its Implications on Persistent Indexes. In *CIDR '22*. 1–8.

[18] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. 187–200. https://www.usenix.org/conference/fast18/presentation/hwang

[19] Intel. 2019. Intel® Optane™ DC Persistent Memory Product Brief. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-dc-persistent-memory-brief.pdf

[20] Intel. 2020. Intel® Optane™ Persistent Memory 200 Series Brief. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-persistent-memory-200-series-brief.pdf

[21] Intel. 2020. Optimizing Write Ahead Logging with Intel® Optane™ Persistent Memory. https://software.intel.com/content/www/us/en/develop/articles/optimizing-write-ahead-logging-with-intel-optane-persistent-memory.html

[22] Intel. 2021. Intel Xeon Processors. https://www.intel.com/content/www/us/en/products/details/processors/xeon.html

[23] Intel. 2021. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture. https://software.intel.com/content/dam/develop/external/us/en/documents-tps/253665-sdm-vol-1.pdf

[24] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *arXiv:1903.05714 [cs]*. http://arxiv.org/abs/1903.05714

[25] JEDEC. 2020. Byte Addressable Energy Backed Interface. https://www.jedec.org/standards-documents/docs/jesd245a

[26] JEDEC. 2021. DDR4 NVDIMM-P Bus Protocol. https://www.jedec.org/standards-documents/docs/jesd304-401

[27] JEDEC. 2021. JEDEC Publishes DDR4 NVDIMM-P Bus Protocol Standard. https://www.jedec.org/news/pressreleases/jedec-publishes-ddr4-nvdimm-p-bus-protocol-standard

[28] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: reducing software overhead in file systems for persistent memory. In *SOSP '19*. ACM, 494–508. https://doi.org/10.1145/3341301.3359631

[29] Anuj Kalia, David Andersen, and Michael Kaminsky. 2020. Challenges and solutions for fast remote persistent memory access. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. ACM, 105–119. https://doi.org/10.1145/3419111.3421294

[30] Dimitrios Koutsoukos, Raghav Bhartia, Ana Klimovic, and Gustavo Alonso. 2021. How to use Persistent Memory in your Database. *arXiv:2112.00425 [cs]*. http://arxiv.org/abs/2112.00425

[31] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable dram alternative. In *ISCA '09*. ACM Press. https://doi.org/10.1145/1555754.1555758

[32] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: converting concurrent DRAM indexes to persistent-memory indexes. In *SOSP '19*. ACM, 462–477. https://doi.org/10.1145/3341301.3359635

[33] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. Association for Computing Machinery, 743–754. https://doi.org/10.1145/2588555.2610507

[34] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating persistent memory range indexes. *Proceedings of the VLDB Endowment* 13, 4, 574–587. https://doi.org/10.14778/3372716.3372728

[35] Lucas Lersch, Ivan Schreter, Ismail Oukid, and Wolfgang Lehner. 2020. Enabling low tail latency on multicore key-value stores. *Proceedings of the VLDB Endowment* 13, 7, 1091–1104. https://doi.org/10.14778/3384345.3384356

[36] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+Trees: optimizing persistent index performance on 3DXPoint memory. *Proceedings of the VLDB Endowment* 13, 7, 1078–1090. https://doi.org/10.14778/3384345.3384355

[37] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 2019. PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs. In *ASPLOS '19*. ACM, 411–425. https://doi.org/10.1145/3297858.3304015

[38] Zhiye Liu. 2018. Fujitsu Targets 2019 for NRAM Mass Production. https://www.tomshardware.com/news/fujitsu-nram-nantero-carbon-nanotube,37437.html

[39] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: scalable hashing on persistent memory. *Proceedings of the VLDB Endowment* 13, 8, 1147–1161. https://doi.org/10.14778/3389133.3389134

[40] Tobias Maltenberger, Till Lehmann, Lawrence Benson, and Tilmann Rabl. 2022. Evaluating In-Memory Hash Joins on Persistent Memory. In *EDBT '22*. OpenProceedings.org, 2:368–2:372. https://doi.org/10.48786/edbt.2022.23

[41] Moohyeon Nam, Hokeun Cha, Young-Ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-optimized dynamic hashing for persistent memory. In *FAST '19*. USENIX Association, 31–44.

[42] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. 2020. AGAMOTTO: How Persistent is your Persistent Memory Application?. In *OSDI '20*. 1047–1064.

[43] Ian Neal, Gefei Zuo, Eric Shiple, Tanvir Ahmed Khan, Youngjin Kwon, Simon Peter, and Baris Kasikci. 2021. Rethinking File Mapping for Persistent Memory. 97–111. https://www.usenix.org/conference/fast21/presentation/neal

[44] Matt Ogle, Trent Bates, Bruce Wagner, and Rene Franco. 2021. How to Balance Memory on 2nd Generation Intel Xeon Scalable Processors. https://downloads.dell.com/manuals/common/balancing_memory_xeon_2nd_gen.pdf

[45] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. 2014. SOFORT: a hybrid SCM-DRAM storage engine for fast data recovery. In *DaMoN '14*. ACM, 1–7. https://doi.org/10.1145/2619228.2619236

[46] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A hybrid SCM-DRAM persistent and concurrent B-Tree for Storage Class Memory. In *SIGMOD '16*. ACM, 371–386. https://doi.org/10.1145/2882903.2915251

[47] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *ISCA '09*. ACM Press. https://doi.org/10.1145/1555754.1555760

[48] SNIA. 2021. NVM Programming Model (NPM). https://www.snia.org/tech_activities/standards/curr_standards/npm

[49] Dell Technologies. 2022. Dell Rack Servers. https://www.dell.com/de-de/work/shop/deals/enterprise-deals/poweredge-rack-server-deals

[50] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing Non-Volatile Memory in Database Systems. In *SIGMOD '18*. ACM, 1541–1555. https://doi.org/10.1145/3183713.3196897

[51] Alexander Van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Persistent memory I/O primitives. In *DaMoN '19*. ACM, 12:1–12:7. https://doi.org/10.1145/3329785.3329930

[52] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2020. Building blocks for persistent memory. *The VLDB Journal*. https://doi.org/10.1007/s00778-020-00622-9

[53] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: lightweight persistent memory. *ACM SIGARCH Computer Architecture News* 39, 1, 91–104. https://doi.org/10.1145/1961295.1950379

[54] Yinjun Wu, Kwanghyun Park, Rathijit Sen, Brian Kroth, and Jaeyoung 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*. Association for Computing Machinery, 1–3. https://doi.org/10.1145/3399666.3399898

[55] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *FAST '16*. USENIX Association, 323–338. https://www.usenix.org/node/194455

[56] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *FAST '20*. USENIX Association, 169–182.

[57] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: reducing consistency cost for NVM-based single level systems. In *FAST '15*. USENIX Association, 167–181.

[58] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. 2009. A durable and energy efficient main memory using phase change memory technology. In *ISCA '09*. ACM Press. https://doi.org/10.1145/1555754.1555759

[59] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. 2019. DPTree: differential indexing for persistent memory. *Proceedings of the VLDB Endowment* 13, 4, 421–434. https://doi.org/10.14778/3372716.3372717