Closing the Performance Gap between DRAM and PM for In-Memory Index Structures

Technical Report

UCSC-CRSS-20-01
UCSC-SSRC-20-01

May 29th, 2020

Yuanjiang Ni*
yni6@ucsc.edu

Shuo Chen
s.chen@alibaba-inc.com

Qingda Lu
qingda.lu@alibaba-inc.com

Heiner Litz
hlitz@ucsc.edu

Zhu Pang
zhu.pang@alibaba-inc.com

Ethan L. Miller
elm@ucsc.edu

Jiesheng Wu
jiesheng.wu@alibaba-inc.com

Center for Research in Storage Systems
Storage Systems Research Center

Baskin School of Engineering
University of California, Santa Cruz
Santa Cruz, CA 95064

`http://crss.ucsc.edu/`
`http://ssrc.ucsc.edu/`

*Work performed while at Alibaba USA.

Abstract

Emerging byte-addressable persistent memories such as Intel's 3D XPoint enable new opportunities and challenges for in-memory indexing structures. While prior work has covered various aspects of persistent indexing structures, it has also been limited to performance studies using simulated memories ignoring the intricate details of persistent memory devices. Our work presents one of the first, in-depth performance studies on the interplay of real persistent memory hardware and indexing data structures. We conduct comprehensive evaluations of six variants of B+Trees leveraging diverse workloads and configurations. We obtain important findings via thorough investigation of the experimental results and detailed micro-architectural profiling. Based on our findings, we propose two novel techniques for improving the indexing data structure performance on persistent memories. *Group flushing* inserts timely flush operations improving the insertion performance of conventional B+-Trees by up to 24% while *Persistency Optimized Log-structuring* revisits log-structuring for persistent memories improving the performance of B+-Trees by up to 41%.

1 introduction

Persistent Memory (PM) devices combine memory and storage characteristics by increasing cost-efficiency over DRAM, providing non-volatility and byte addressability (via memory load/store instructions) as well as by supporting sub- s-scale latencies. The eminent availability of PM opens up new opportunities for many IO intensive applications. For instance, it improves the viability of in-memory databases by providing higher storage capacity for the same cost while providing native durability without incurring the high write overhead of flash SSDs. Due to the limited availability of PM, prior work resorted to DRAM-based emulation [58] or hardware simulation [39]. Our work is one of the first utilizing real PM hardware evaluating the actual performance benefits and trade-offs of PM. In contrast to existing studies of Intel's Optane memory [21, 57] that only evaluate raw performance i.e. latency and bandwidth, our work provides a thorough analysis of the interplay of PM and the indexing data structures commonly used in databases.

While there exists a large body of work on indexing structures as well as on PM, the interplay between the two remains largely unexplored. For instance, prior work on in-memory indexing has investigated scalability [5, 18, 30, 32, 34, 35], cache efficiency [6, 9, 35, 41, 42] and CPU efficiency [25, 47], however, only for conventional DRAM-based systems.

Prior research on PM has focused on topics including consistency and the read-write asymmetry of PM without focusing on the integration of PM technology and indexing data structures. For instance, there exists a large body of work on providing consistency and durability via write-ahead logging (WAL) [7, 13, 50] or native persistence [11, 15, 20, 28, 29, 37, 58]. Prior work on addressing the asymmetric read-write performance of PM proposed different techniques for trading additional reads for a reduction in writes [10, 60, 61], for instance by maintaining the keys of an index node in their insertion order [10] to trade-off writes for higher query overheads.

This paper looks at indexing structures from a new PM-based perspective. By studying a system with Intel Optane DC Persistent Memory Module (DCPMM), the first commercially available 3D XPoint-based PM product that offers desired characteristics of high density, byte addressability and low latency, we conduct a comprehensive evaluation of in-PM indexes to answer the following questions: (1) What are the implications of 3D XPoint memory [21, 57] on in-memory index performance? (2) What is the real-world performance impact of various PM indexing techniques? (3) Are there device or platform-specific techniques that efficiently optimize indexes in 3D XPoint memory? Our study targets sorted index structures instead of hash tables as the support of efficient range queries is critical for many cloud applications. Among sorted indexing structures, we focus on the B+-tree and its variants, as they are widely used both as standalone software components and as basic building blocks of advanced indexing structures including Wormhole [54] and the MassTree [35]. In comparison, as demonstrated in prior work [51, 55], index structures such as skip lists and tries provide inferior performance in many scenarios and are often enhanced by incorporating B-tree into their designs.

Our evaluation results demonstrate that the behavioral characteristics of PM are more intricate as shown by prior work that leveraged only simulation techniques. Despite sharing the same interface with DRAM in load/store instructions, PM should not be be treated as slow RAM. We first conduct a comprehensive evaluation on the PM indexing techniques with diverse workloads and configurations. Several important findings are made and analyzed with detailed profiling. Leveraging our findings we

have developed three optimizations to explore more potential techniques to improve the index performance for PM. The main findings and results of our optimizations are summarized below.

Main Findings: 1) PM bandwidth represents a greater performance challenge than PM latency for indexing structures with write-intensive workloads, increasing the performance gap between DRAM and PM from 2.2× to 4.4×. PM manufacturers should focus on improving bandwidth by increasing intra-module parallelism to address the most significant performance issue of PM. 2) The benefits of write-optimized indexing structures are limited in real-world applications. Keeping data unsorted at tree leaves only improves random insertion performance by 6% in a single threaded workload. 3) The granularity disparity between CPU caches and DCPMM (64 vs 256 bytes) contributes to sub-optimal bandwidth utilization in PMs. 4) The overhead of providing durability and consistency for PMs is small compared to the raw performance degradation introduced by PM over DRAM. 5) The state of the art consistency mechanism for B-Trees—FAST/FAIR—benefits from continuously flushing cache-lines to PM. With this optimization, enforcing consistency only introduces a latency penalty of 1.27× over conventional B-Trees without consistency guarantees. 6) The idiosyncrasies of DCPMM affect B-Tree parameter tuning such as node size.

Optimizations: 1) Interleaving that leverages software prefetching and fast context switching can be used to hide the longer memory access latency of PMs. We show that in-PM indexing structures benefit more from such technique than in-DRAM indexes. 2) We introduce *Group flushing*, a technique that prevents data reordering by flushing modified data in groups. An unmodified B-tree with software-directed group flushing achieves write throughput comparable to its write-efficient counterpart while having better search and range-query performance. 3) *Log-structuring* translates random writes to sequential writes at the cost of additional reads and garbage collection, effectively addressing the larger line size deployed by PM (265 Byte vs. 64 Byte). We show that the performance can be improved by up to 41% via log structuring, compared to the state-of-art PM-aware B+-Tree.

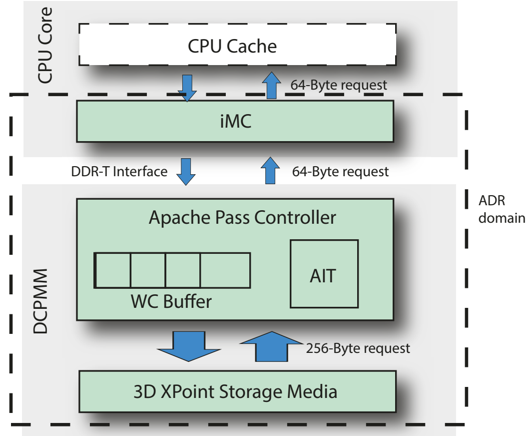


Figure 1: The internal details of the Intel Optane DC Persistent Memory Module

2 Background

2.1 3D XPoint Persistent Memory

While many PM technologies such as PCM (Phase Change Memory) [43], ReRAM [46] and MRAM [48] have been under development for the last three decades, Intel recently released 3D XPoint memory as one of the first commercially available PM products.

Figure 1 outlines the architecture of Intel's 3D XPoint platform [21]. DCPMM modules are connected to the integrated memory controller (iMC) on a recent Intel server CPU such as Cascade Lake via DDR-T, a proprietary memory protocol. While built on DDR4's electrical and mechanical interface, DDR-T provides an asynchronous command and data interface to communicate with the host iMC. It utilizes a request/grant scheme for initializing requests by sending a command packet from the host iMC to the DCPMM controller. Similar to DDR4, the iMC accesses DIMMS at the cache-line (64-byte) granularity. The iMC and the DCPMM modules connected to it form the Asynchronous DRAM Refresh (ADR) domain. Every cache line that reaches the ADR is guaranteed to be persisted, even in the event of a power loss. Note that the CPU caches as well as other buffers in the CPU are outside the ADR domain and hence suffer from data loss in the case of a power failure.

For wear-leveling and bad-block management DCPMM coordinates accesses to the underlying 3D XPoint storage media via an indirection layer. The Address Indirection Table (AIT) translates system addresses to device-internal addresses. The access

	Bandwidth (GB/s)				Idle Latency (ns)			
	Seq. Read	Rand. Read	Seq. Write	Rand. Write	Seq. Read	Rand. Read	NT Store	`clwb`
DRAM	105.9	70.4	52.3	52	81	101	86	57
DCPMM	38.9	10.3	11.5	2.8	169	305	90	62

Table 1: The basic performance characteristics of DCPMM. Non-temporal stores (NT) and cache line writebacks (`clwb`) are followed by a memory barrier to ensure that the store reaches the ADR domain.

granularity of the DCPMM media is 256 bytes [21], requiring the controller to translate 64-byte requests from the CPU into larger 256-byte requests. To avoid a 4× write amplification on every write, write combining buffers are employed to aggregate cache-line-sized data into 256-byte chunks. The number of write-combining buffers is limited and when all buffers are exhausted a buffer that is hopefully full is selected by the IO scheduling logic and flushed to the 3D XPoint media. Similarly, 256-byte buffers are allocated for incoming read requests.

DCPMM can operate in two different modes, the *Memory Mode* and the *App Direct Mode*. The Memory Mode transparently uses available DRAM as a cache and can only be used if durability is not a concern. In this paper, we employ the App Direct Mode as it provides the durability required by database systems as well as direct control of the memory device. In App Direct Mode, the DCPMM is exposed as a storage device hosting a DAX-capable filesystem [12, 50, 56] that can be mmap'ed and then accessed via load/store operations.

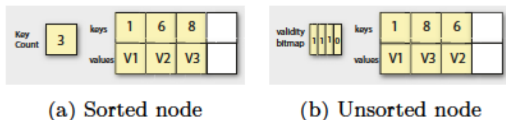
The basic performance characteristics exhibited by DCPMM on an Intel Cascade Lake server are summarized in Table 1. A more comprehensive study of the performance characteristics of the DCPMM can be found in a prior report [21]. Overall, DCPMM provides inferior performance compared to DRAM. Furthermore, the read throughput of DCPMM is about 4× higher than the write throughput (read/write asymmetry) and generally sequential workloads achieve 4× higher throughput than random workloads. The read/write asymmetry can be explained by the higher cost of writing data to the 3D-XPoint media. The performance gap between sequential and random workloads can be explained by the 4× IO amplification induced by the translation of 64-byte cache line accesses to 256-byte DCPMM accesses. In addition, the unloaded random read latency of DCPMM is about 3× higher than that of DRAMs and the random read latency is 2× higher than the sequential read latency. Finally, the unloaded latency of a non-temporal store or cache line writeback (`clwb`) is almost identical between DCPMM and DRAM.

2.2 Durability and Consistency in PM

A reliable storage system must be able to tolerate unexpected system failures by providing *durability*, guaranteeing a consistent system state at all times. As discussed in Section 2.1, CPU caches do not reside within the ADR domain and, therefore, cached data is lost during a power cycle. A solution to this problem is to explicitly flush modified data to PM by software. The cache-line writeback `clwb` and cache-line flush `clflushopt` instructions are supported by the X86 instruction set to force writeback of modified cache-lines to PM.

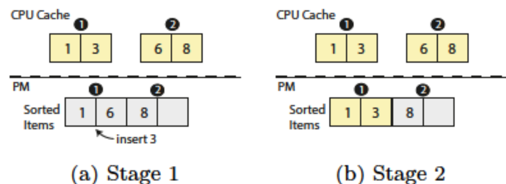
Besides ensuring that modified data reaches the persistence domain (durability), every write to PM needs to leave the system in a consistent state (atomicity). Modern architectures generally only support 8/16-byte atomic stores. However, for an update operation that modifies more than 16 bytes atomically, a more sophisticated failure-atomicity scheme needs to be utilized. Consider a simple example of inserting a node in a singly linked list. To insert a new node between the `prev` node and the `next` node, we need to first update the "next" pointer of the new node to `next` and second update the "next" pointer of the `prev` node to the new node. As these two updates (8-byte each) cannot be performed atomically, they may be written back from the CPU cache to the PM in any order. Consider a case where the second update is persisted first and then the system crashes before the first update has completed. In this case, the list will be in an inconsistent state with disconnected nodes.

Atomic In-place Updates. Data consistency can be guaranteed by enforcing the order of updates reaching the PM device. For instance, in case of the linked list example, ensuring that the first update is persisted before the second ensures consistency of the list under this failure scenario. To enforce the order between the first and second up-



(a) Sorted node (b) Unsorted node

Figure 2: Typical layouts of B+-Tree node.



(a) Stage 1 (b) Stage 2

Figure 3: Challenges in maintaining consistency for a sorted node: i) Element 3 is inserted into an array of sorted elements; ii) System crashes after cacheline ❶ is written back to the PM, resulting in an inconsistent state, *e.g.* element 6 is lost

date, they need to be performed in the correct order of the program, and then followed by a cacheline flush and a memory fence instruction such as `mfence` and `sfence` on X86. More complex data structures [15, 20, 28, 29] require breaking update operations into multiple atomic steps as both the reader and writer must be enabled to cope with inconsistent states.

Failure-Atomic Transactions. Native persistence is error-prone if more complex operations are involved. To remedy this, existing PM support libraries [7, 13, 50, 59] incorporate the failure-atomic transaction abstraction, which can be implemented with Write-Ahead Logging (WAL). For the singly linked list example, the programmer only needs to specify a failure-atomic transaction comprising the two updates. The transactional support ensures that either both updates are persisted or none of them take effect. Compared to native persistence, WAL entails the "double write" problem and involves non-trivial software overhead in managing the log space and performing the copying of the data.

2.3 In-PM B+-Tree Variants

The B-Tree [4] is a self-balancing multi-way tree structure. In a B-tree, each internal (non-leaf) node contains $k-1$ keys as separation values to divide its children into k subtrees, where k must be within the range of $\lceil \frac{b}{2} \rceil$ and b. When data is inserted or removed from a node, internal nodes may be joined or split to maintain the pre-defined number of children. A B+-tree [14] is a modified B-tree that stores all key-values in its leaves and maintains copies of keys in internal nodes. The leaves of a B+-tree are often connected via sibling pointers for fast traversal. While originally proposed to support efficient indexing on block storage, the B+-Tree is also widely used as an efficient in-memory index to alleviate the performance gap between the CPU cache and the main memory, with each node consisting of multiple cache lines [17]. Several studies investigated the design of in-DRAM B+-trees and its variants from different aspects such as cache consciousness [17, 42], scalability [5, 34, 47], the capability of memory latency hiding [9, 22, 40], and the opportunities of architecture-specific optimizations [25, 47]. With the advent of PM, while these techniques are still applicable, the durability and consistency requirements as well as different device characteristics require new techniques in the design of B+-trees.

Optimizing for Writes. In a conventional B+-Tree, each node maintains an array of items, i.e. keys, values or pointers; these items are sorted to reduce the search overhead. Upon an insertion, on average, half of the items have to be moved to maintain the key order, thus significantly increasing memory writes. As discussed in Section 2.1, unlike DRAM, PM technologies usually support lower write than read performance. Prior work [10, 11, 37, 58] addresses the problem of high write cost by leaving nodes unsorted. Figure 2b shows a possible layout of an unsorted node in such a B+-Tree. Different from the conventional design, an extra *validity bitmap* is added in the node header to encode the occupancy of the item slots. Each insertion consists of 3 steps: i) identifying an empty slot by consulting the *validity bitmap*, ii) putting the item into the empty slot, and iii) setting the corresponding *validity bit*. A search operation on a node involves checking every valid slot. One can apply the unsorted layout to selected write-heavy nodes such as leaves, or the entire tree. Chen et al. [10] show that the unsorted leaf scheme exhibits better search/update performance compared to a B+-Tree when all nodes are unsorted.

Ensuring Consistency. Ensuring consistency for B+-Tree updates is challenging as an insertion (or deletion) requires rewriting a large portion of a node (*e.g.* sorted node) that spans multiple cache lines. An example is shown in Figure 3. Inserting element number 3 into an array of elements modifies two cache lines. If the system crashes before cache line ❷ is persisted, the sorted array in the

PM is turned into an inconsistent state where element number 6 is lost. The most straightforward solution is to rely on WAL to ensure the atomicity of an update. However, WAL is often inefficient due to transactional overheads. Two other approaches have been proposed to maintain data consistency for B+-trees. First, unsorted B+-trees built on the *validity bitmap* [11, 37, 58] maintains the atomicity of node updates by ensuring the persistent order between writes to the new slot and the update to the *validity bitmap* (atomic in-place updates). If the system crashes before the update to the *validity bit* is persisted, the corresponding slot remains invalid and the insertion (the items might only be partially written) will be considered as failed. Second, Hwang et al [20] proposes two mechanisms—Failure-Atomic ShifT (FAST) and Failure-Atomic In-place Rebalance (FAIR)—which can be used to augment durability semantics to an sorted B+-Tree. The intuition is that if the modified cache-lines can be written back to the PM in order, the potential inconsistent states can be tolerated. Returning to the example in Figure 3 where cache line ❷ is persisted before the cache-line ❶. A failure might result in an array with a redundant 6, however, this case can be tolerated given the semantics of the B+-Tree.

Selective Persistence. The idea of selective persistence is that the entire index can be reconstructed from a fraction of the tree. Performance can be improved by storing the recoverable part of the index in fast DRAM. Selective persistence can be applied to B+Tree [37, 58] as the leaf nodes of a B+-Tree are already linked in the sorted order and thus provides all the information to rebuild the internal nodes. The tradeoff of this approach is between performance and the failure-recovery time.

3 Experimental Setup

We now describe our methodology to evaluate the performance characteristics of different B+-Tree implementations on DCPMM.

3.1 Methodology

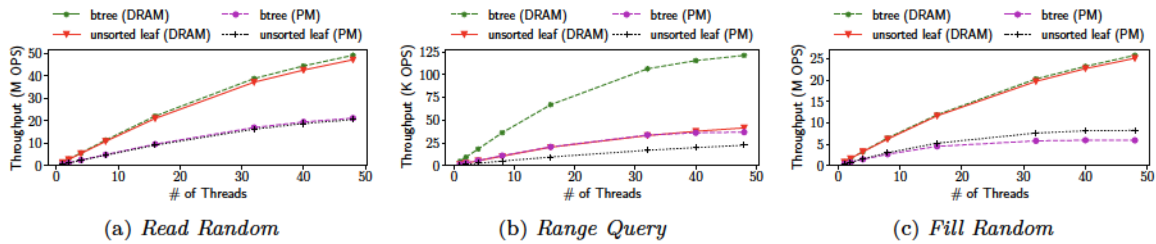
In this work, we compare five B+-Tree solutions: 1) *btree*: A conventional in-memory B+-Tree with sorted keys in each node; 2) *unsorted leaf*: A write-optimized B+-Tree with unsorted leaf nodes and sorted keys in internal nodes; 3) *btree-WAL:* A B+-Tree utilizing WAL for consistency; 4) *FAST/FAIR:* A B+-Tree with FAST/FAIR [20] for consistency; 5) *persistent unsorted:* A B+-Tree with *unsorted leafs* utilizing native atomic in-place updates to ensure the consistency of leaf updates and WAL for supporting the rare case of structural modifications; 6) *FAST/FAIR SP*: A FAST/FAIR B+-Tree that employs selective persistence. All of these five solutions employ the same fine-grained, optimistic concurrency control mechanism [6] to enable multi-core scalability. Note that we employ linear search for all our implementations as prior work [20] has shown that binary search performs worse when the node size is smaller than 4 KB due to branch mispredictions.

We profile the executions of the different B+-Tree implementations using multiple hardware performance counters, including the total number of instructions, instruction per cycles (IPC), cache miss stalls and resource related stalls, to interpret the results. The performance counters are obtained via perf [16]. The resource related stalls include stalls caused by the limited size of hardware resources such as the load/store queue as well as stalls induced by the execution of memory fences. We also collect the performance counters from the DCPMM, including the amount data that is read and written from and to the DCPMM controller as well as the amount of data that is read and written from and to the DCPMM storage media.

3.2 Experimental Environments

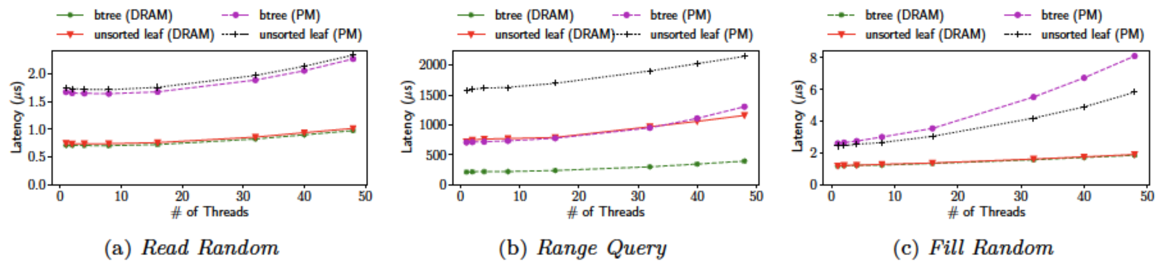
All of our experiments are conducted on a 2-socket, 56-core machine with 32KB/1024KB/38MB L1/L2/L3 Caches. The 12 memory channels (2 sockets $\times$ 6 channels/socket) are fully populated using DRAM and DCPMM modules. In particular, we deploy 96GB of DRAM (6 $\times$ 16 GB/DIMM) and 1.5TB of DCPMM (6 $\times$ 256 GB/DIMM). We use the ext4-DAX file system on the Fedora distribution (kernel 5.0.9). All our experiments are executed on a single socket by pinning threads and restricting memory allocation to the same NUMA node. Our code is written in C/C++ and compiled with clang 8.0.0 with -O3 flag.

We design four micro-benchmarks for evaluation. Unless otherwise stated, the system is warmed up by loading a tree with 160 million entries and then one of the four operations is performed: *Fill Random* randomly inserts 80 million additional key/value pairs; *Read Random* retrieves 320 million random keys; *Range Query* performs range query requests with a selection ratio of 0.001%; *Read/Write* simulates a mixed read-write workload. Both keys and values in these workloads are 16 bytes in size. We perform an exhaustive search of the B+-Tree node



(a) *Read Random* (b) *Range Query* (c) *Fill Random*

Figure 4: Throughput under three different workloads (higher is better)



(a) *Read Random* (b) *Range Query* (c) *Fill Random*

Figure 5: Latency under three different workloads (lower is better)

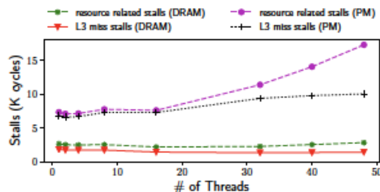


Figure 6: Profiling of *btree* under the *Fill Random*

size as one of the experiments. For all other experiments we utilize a node size of 512 bytes which provides good performance in all configurations (Section 5.2).

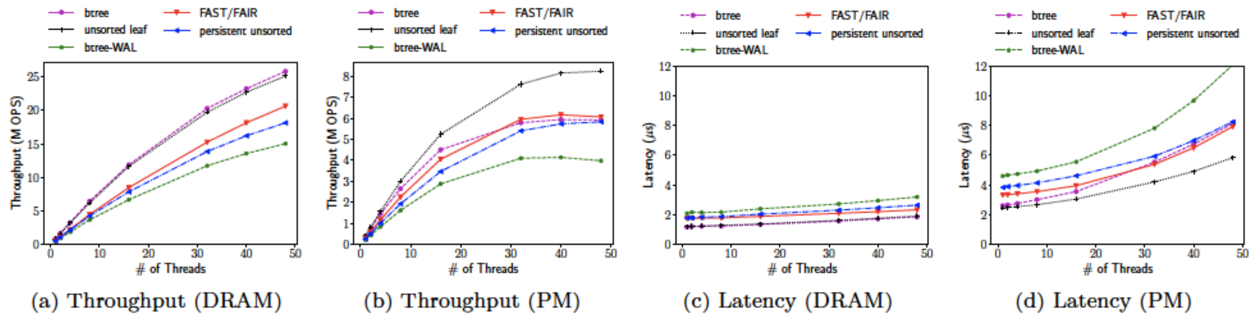
4 PM Indexing Techniques

In this section, we conduct experiments to measure the efficiency of PM techniques for addressing the read-write asymmetry and for providing storage consistency and durability. Furthermore, we analyze how B+-Trees can leverage a combination of DRAM and PM devices to optimize performance.

4.1 Write-optimized Indexing Structures

In the first experiment we analyze the benefit of write optimized data structures. As described in Section 2.3 the *unsorted leaf* index reduces writes while increasing computational complexity by maintaining the bitmaps. We compare *unsorted leaf* against a conventional *btree* baseline showing the achieved throughput in Figure 4 and latency in Figure 5 for three different workloads. We compare the in-DRAM Performance (prefixed by DRAM) and in-PM Performance (prefixed by PM).

Unsorted leaf reduces costly PM writes by avoiding sorting the entries, and thus can improve the update performance by up to 1.40× in the DCPMM (Figure 4c and Figure 5c). In DRAM, reads and writes have almost the same cost, so *unsorted leaf* only improves the update performance by 1.06×. The impact of searching values within unsorted nodes requires that all valid slots must be checked. In contrast, on average, only half of the entries need to be checked in a sorted node for an existing key. However, as *unsorted leaf* only suffers from additional overhead while accessing leaf nodes, the read performance of *unsorted leaf* is comparable to that of the *btree* (Figure 4a and Figure 5a). As the *unsorted leaf* incurs significant instruction overhead to sort the keys in a range query, it exhibits up to 3× performance degradation under the DRAM and 2× degradation under DCPMM (Figure 4a and Figure 5b). *Unsorted leaf* suffers from a smaller performance hit when performing a range query in PM as the software overhead becomes less significant compared to the cost of device accesses.



(a) Throughput (DRAM) (b) Throughput (PM) (c) Latency (DRAM) (d) Latency (PM)

Figure 7: The cost of persistence

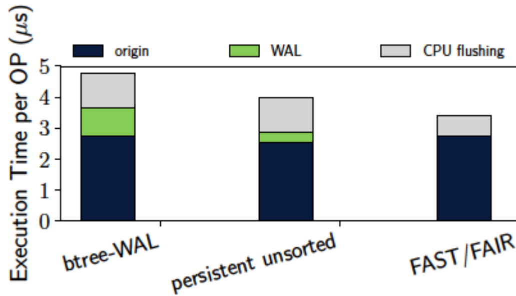


Figure 8: Persistence cost decomposition (4 threads)

The read performance of in-PM indexes is lower than that of the in-DRAM indexes by a constant ratio (Figure 4a and Figure 5a). For write workloads, the slowdown for *btree* ranges from 2.2× to 4.4× whereas the slowdown for *unsorted leaf* only ranges from 2.0× to 3.0× (Figure 4c and Figure 5c) as the number of threads increases from 1 to 48. This observation is in line with 3D XPoint's asymmetric read/write performance. When the load of the system is low (1 thread), the performance gap between PM and DRAM is between 2–3× as the node traversal dominates the total runtime with memory writes being mostly performed out of the critical path. As the number of execution threads increases, the DCPMM approaches its maximum I/O capacity. The impact on performance is twofold. First, increasing the PM operations creates back pressure, eventually filling up the hardware resources (reorder buffer, store queue) and stalling the processor pipeline. Second, the amortized penalty of L3 cache misses increases as it takes more time to process the read requests in a fully loaded DCPMM. Figure 6 shows the performance profile for *btree* on both DRAM and PM. Compared to the in-DRAM *btree*, the in-PM one experiences a drastic increase in L3 miss stalls and, furthermore, resource related stalls increase as the number of execution threads scales beyond 16, indicating that the bandwidth of PM is indeed becoming a bottleneck.

Finding 1: The gap between the insertion performance of in-DRAM indexes and that of the in-PM indexes widens as the number of execution threads increases.

Implication: The PM bandwidth can be a limiting factor for indexing structures to handle requests at large scale. First, the indexing structures should be designed to carefully avoid wasting PM bandwidth. Second, more hardware resources should be added to the system to mitigate the back pressure from the PM.

Our evaluation shows that the actual improvement of *unsorted leaf* on insertion performance is relatively small (e.g. 6%) in a single-thread workload, which is significantly less than what prior work [10], utilizing a simulation based methodology, suggested. Actual PM devices are better at hiding high cost of writes than expected. In a real-world system, PM writes can be queued at multiple layers (store buffer, cache, iMC, DCPMM controller) and slowly drained to PM. Finding 1 indicates that the DCPMM becomes bandwidth-bound as the number of execution threads increases beyond 16. As we can see, the impact of write reduction in the *unsorted leaf* on performance increases with I/O pressure on the DCPMM.

Finding 2: The real-world efficiency of write-optimized indexing structures varies significantly based on the intensity of the workloads.

Implication: To enable write-optimized indexing structures B+-trees have to be restructured, increasing the search overhead. As a result, for workloads with a low insertion rate conventional B+-trees are preferable over write-optimized implementations in PMs.

4.2 Storage Consistency

Figure 7 compares the insertion performance of persistent B-Trees with various consistency mechanisms. The persistent trees (*btree-WAL*, *FAST/FAIR* and *persistent unsorted*) pay extra costs to ensure storage consistency. To evaluate the persistence overhead, we compare them with their volatile counterparts (*e.g. btree* and *unsorted*).

Several general observations are made. First, as expected, the consistency mechanisms introduce runtime overheads, in particular, the insertion latencies of *btree-WAL* and *persistent unsorted* are up to 1.77× and 1.56× higher than the latency of the volatile B+-Tree implementations (Figure 7d). The persistence overhead includes i) execution of additional instructions to implement WAL and ii) introduction of execution stalls caused by memory barriers. Table 2 shows a detailed performance profile. It confirms that *btree-WAL*, *FAST/FAIR* and *persistent unsorted* indeed increase the number of instructions and experience more resource related stalls. Second, *btree-WAL* and *persistent unsorted* cannot achieve comparable throughput to their volatile counterparts (*e.g. btree* and *unsorted*) in the PM (Figure 7b). This is because the *persistent unsorted* and *btree-WAL* rely on write-ahead logging and, therefore, incur extra PM write traffic. In contrast, the WAL-free *FAST/FAIR* achieves comparable throughput as *btree*. Third, *btree-WAL* exhibits the lowest performance among the three persistent trees as employing write-ahead logging for the sorted node entails considerably higher overheads.

Compared to the latency in DRAM (Figure 7c), the latency of the persistent trees in PM increases drastically as the number of the execution threads increases (Figure 7d). In particular, the latency of *FAST/FAIR* increases by 2.40× in PM but only 1.35× when residing in DRAM. As we can see in Table 2, the DCPMM-level write-amplification introduced by the persistent trees remains high, indicating the full I/O capacity is still underutilized. Note that the write-amplification of the *btree-WAL* is relatively low. This is because it generates redundant writes written to the in-PM log sequentially.

Finding 3: The restricted I/O bandwidth of DCPMM limits the insertion rate of the persistent trees while due to random access patterns the PM bandwidth cannot be not fully utilized.

Implication: In conventional B+-Trees, random updates are unavoidable increasing the write-amplification for PMs. Given the constrained I/O capacity of PMs, the insertion performance can be further improved by reshaping the I/O pattern of the B+-Tree via log-structuring.

Despite the common belief that persistence introduces performance overheads Figure 7d shows an interesting counterexample. For the *Fill Random* workload, *FAST/FAIR* delivers 3% higher throughput than the volatile *btree* on DCPMM, when the number of execution threads is 48. This performance increase is provided by the explicit cache-line flushing mechanism required for ensuring consistency which, as a side-effect, forces cache-lines to be written back to the DCPMM controller sequentially. In contrast, the timing of cache-lines being written back in *btree* is implicitly controlled by the CPU cache. This observation has inspired us to develop a new optimization described in Section 6.2.

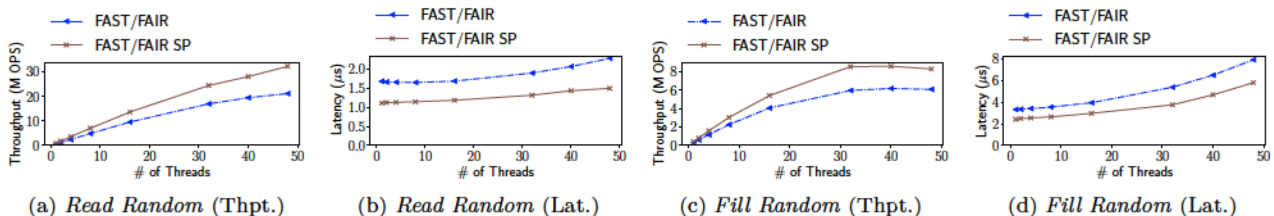
Finding 4: With conventional B-Trees, sequentially modified cache lines are often written back out of order by the cache controller, resulting in sub-optimal bandwidth utilization in PM.

Implication: We can improve the performance of the conventional indexes by preserving the spatial locality of updates for the DCPMM controller.

Figure 8 breaks down the runtime overhead: For each implementation, we list the execution time induced by the *btree* implementation itself as well as the time induced by WAL, memory fences and cache-line flushes. We observe the CPU flushing overhead of *btree-WAL* is actually low (Figure 8) and is close to that of *persistent unsorted* although it requires 2× more cache-line flush instructions (Table 2). This is because the frequency of memory barriers is lower in WAL (1 memory barrier for every 3 cache-line flushes) which allows multiple cache flush operations to proceed in parallel. As a result, the software overhead of WAL of 0.9 s almost outweighs the overhead for persisting data of 1.1 s in *btree-WAL*. In contrast, the software overhead of WAL in the case of *persistent unsorted* is minimal (0.3 s) as it is only required to implement rarely occurring structural modifications of the tree.

	CPU				Memory			
	Instruction	IPC	Resource Stalls	Per. Instr.	Read	Write	RA	WA
btree	2107	0.22	7740	0/0	1626	400	1.6	3.2
unsorted	2381	0.26	6532	0/0	1651	236	1.5	3.5
btree-WAL	4620	0.26	11240	12/4	2001	937	1.6	1.8
FAST/FAIR	2548	0.19	9982	6/10	2119	474	1.5	2.2
persistent unsorted	3137	0.22	9244	5/5	1973	411	1.4	2.6

Table 2: Profiling of the evaluated designs under the *FillRandom* workload (PM, 1 thread). It shows Instructions Per Cycles (IPC), the number of instructions, resource related stalls and persistence instructions (`clwb/sfence`), read traffic to PM (bytes), write traffic to PM (bytes), Read Amplification and Write Amplification per operation



(a) *Read Random* (Thpt.) (b) *Read Random* (Lat.) (c) *Fill Random* (Thpt.) (d) *Fill Random* (Lat.)

Figure 9: The efficiency of selective persistence

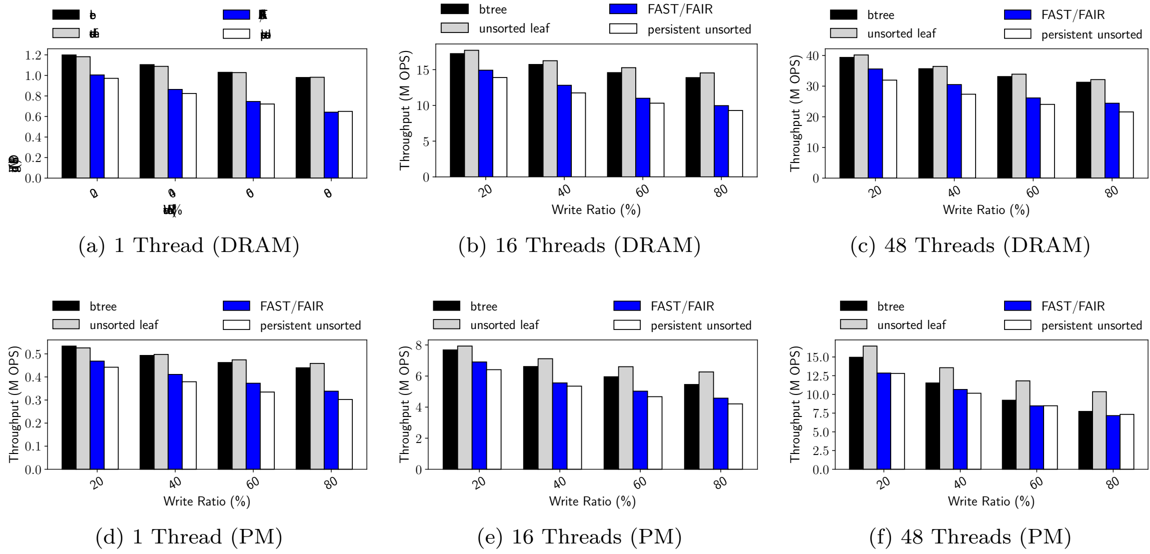
Finding 5: WAL provides a straightforward approach to ensure the consistency for complicated operations, however, incurs significant software overhead.

As shown in Table 2, *FAST/FAIR* requires more flush operations than the *persistent unsorted* B+-Tree. Intuitively, *FAST/FAIR* incurs more CPU flushing overheads and exhibits higher latency than the *persistent unsorted* implementation in the latency-bound workloads when the number of the execution threads is small. Figure 8, however, shows the opposite trend: The CPU flushing overhead of *FAST/FAIR* is 1.7× lower than that of the *persistent unsorted* tree. In contrast, the latency of *FAST/FAIR* and that of *persistent unsorted* is comparable when placed in DRAM under the same workload. This is because *FAST/FAIR* flushes cache lines continuously inducing smaller overheads for DCPMM.

Finding 6: *FAST/FAIR* in the DCPMM benefits from flushing cache-line continuously, reducing latency when the load of the system is low. When utilizing PM, the latency is only to 1.27× higher than that of the *btree*.

4.3 Selective Persistence

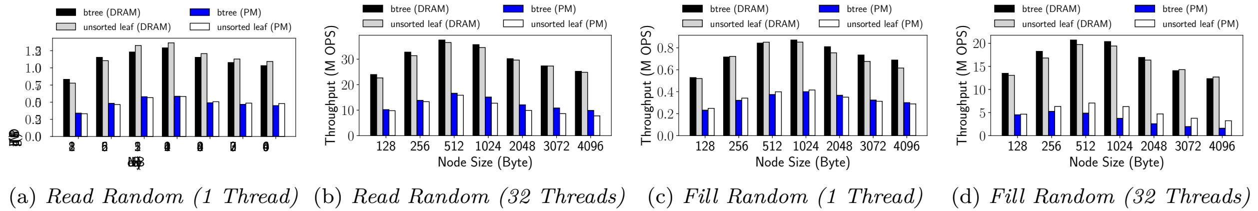
We examine the efficiency of the selective persistence technique described in Section 2.3. We store the internal nodes in fast DRAM as they can be rebuilt from the linked leaf nodes. First, as frequently accessed index nodes are placed in DRAM, the performance of leaf search benefits from selective persistence. As shown in Figure 9a and Figure 9b, storing index nodes in DRAM indeed improves the read performance by about 1.5× closing the gap between in-DRAM indexes and in-PM indexes. Second, the selective persistence can improve the insertion performance because it i) boosts the process of tree traversal and ii) avoids the PM writes to update the internal nodes stored in DRAM. As shown in Figure 9c and Figure 9d, the insertion performance is improved by up to 1.4× using the selective persistence technique. One drawback of the selective persistence approach is that it increases the recovery time. We have measured the time that it takes to rebuild the index nodes: as we increase the number of inserted entries from 10^4 to 10^8, the recovery time increases from 10 milliseconds to 10 seconds almost linearly.



(a) 1 Thread (DRAM) (b) 16 Threads (DRAM) (c) 48 Threads (DRAM)

(d) 1 Thread (PM) (e) 16 Threads (PM) (f) 48 Threads (PM)

Figure 10: The performance of mixed read-write workloads



(a) *Read Random (1 Thread)* (b) *Read Random (32 Threads)* (c) *Fill Random (1 Thread)* (d) *Fill Random (32 Threads)*

Figure 11: Performance sensitivity to the size of the node

5 Workload Performance

In this section, we extend our experiments to more diverse workloads and configurations.

5.1 Mixed Workloads

Figure 10 shows the index performance under workloads with varying ratios of PUT and GET operations and different number of execution threads. The first row in Figure 10 shows the performance of different index structures in DRAM and the second row shows the performance in PM. Two observations are made. First, The ratio of PUT/GET operations has a greater impact on the performance of the evaluated index structures in the DCPMM platform. In particular, the throughput of the *btree* degrades by 2.3× in the PM and only by 1.3× in the DRAM as the write ratio increases from 20% to 80% with 48 execution threads. This observation further demonstrates the read/write asymmetry issue of the DCPMM. Second, in-memory indexes do not benefit much from the improved write efficiency of unsorted B-Trees when dealing with read-intensive workloads or when the load of the system is low. For instance, in a workload with 16 execution threads and 40% PUT ratio, the *unsorted leaf* implementation only achieves a 2% higher throughput.

5.2 Sensitivity to the Node Size

Figure 11 shows the effect of the node size on performance. We make two observations. First, the performance of all evaluated B-Tree indexes peaks at the node size of around 512 Bytes in both *Fill Random* and *Read Random* benchmarks. Figure 12 shows the profiling of *btree* under the *Read Random* benchmark. As we can see, when the size of the node increases from 128 bytes to 512 bytes, the L3 miss penalty (L3-miss stall cycles) is significantly reduced, thus increasing overall performance.

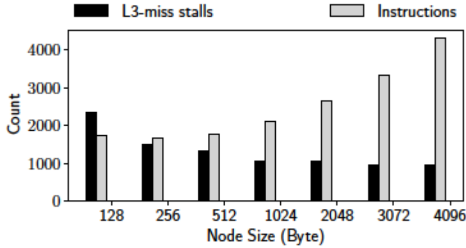


Figure 12: Profiling for *btree* under the *Read Random* application

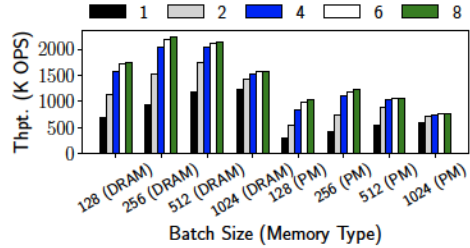
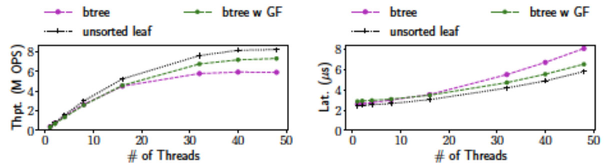


Figure 13: Interleaving efficiency



(a) Throughput (b) Latenc

Figure 14: Benefits of preserving sequentiality in software.

This can be explained by the improved efficiency of the hardware prefetcher for large nodes spanning consecutive cache lines. Note that the total amount of data that needs to be read from DRAM on a traversal is similar as trees with smaller nodes contain more levels. When the node size increases beyond 512 bytes, the L3 miss penalty no longer decreases, however, the key-scanning overhead continues to increase, as larger nodes need to be searched, leading to performance degradation. Second, the performance gap between in-PM *btree* and in-PM *unsorted leaf* increases as the size of the nodes increases to 32 threads for the *Fill Random* workload (Figure 11d). In this case, the sorted B+-Tree experiences an increasing number of PM writes as more entries need to be moved during an insertion. We also observe that the insertion performance of the *btree* and *unsorted leaf* is comparable to the single-thread workload (Figure 11c). It shows that the I/O bandwidth limitation of the DCPMM in handling write requests does not represent a bottleneck in this case.

We find the insertion performance of in-PM *btree* and in-DRAM *btree* peaks at different node sizes as shown in Figure 11d. In particular, when the size of the node increases from 256 byte to 512 byte, the performance of the in-DRAM *btree* increases by 1.13× whereas that of the in-PM *btree* degrades by 1.08×. This is because in a write intensive workload, the overhead of writing more data to the PM outweighs the reduced L3 miss penalty of increasing the node size.

Finding 7: In order to maximize the performance, parameters such as node size for in-DRAM indexes need to be re-tuned for in-PM indexes.

6 Case Studies

In this section, we conduct three studies to show the potential optimizations enabled by our findings. We focus on showcasing the key ideas and their impact on performance.

6.1 Interleaving Operations

Major in-memory B+-tree operations such as query, insertion, and removal are implemented by traversing the tree from the root to leafs, resulting in the *pointer chasing* problem: a node cannot be accessed before the previous node's pointer is resolved. If the accessed node is not already in the CPU cache, the CPU stalls waiting for data from the main memory. As shown in Table 1, DCPMM's longer read lantecy compared to DRAM exacerbates the pointer chasing issue. There are several techniques proposed to hide memory latencies [9, 22, 40] of B+-tree variants. One effective approach is to group multiple operations against a data structure and then issue them as a batch, thus increasing memory level parallelism [22, 40]. In this section, we examine the efficiency of interleaving the execution of multiple GET requests. The multi-GET performance of the *btree* with varying node size and batch size is shown in Figure 13, where throughput is computed as batch size times the number of multi-GET operations / second. The efficiency of interleaving decreases as the node size grows. For instance, with DRAM, when the batch size is increased from 1 to 4, the GET throughput increases by 2.1× with 256-byte nodes but only by 1.2× with 1024-byte nodes. This is due to the fact that modern CPU cores have a

limited number (10 20) of line fill buffers (LSB) to support memory parallelism. With a smaller node size, fewer cache lines need to be prefetched at once for a single GET operation, hence more GET operations can be performed in parallel. Due to larger access latencies in-PM B+-tree benefits more from interleaving. For instance, with 128-byte nodes the multi-GET throughput of the in-PM *btree* increases by 3.4× compared to the conventional implementation, whereas with the same configuration the in-DRAM *btree* only increases performance by 2.5×.

Lesson 1: In-PM indexes benefit more from interleaving compared to its in-DRAM counterparts as long as adequate hardware resources (LSBs) are provided. System designers should consider increasing hardware resources on future CPUs to enable higher memory parallelism.

6.2 Group Flushing

As described in **Finding 4**, when PM is used as memory and no explicit cache-line flush is used to ensure persistence, the order of the modified cache lines being written back to the PM controller is implicitly decided by the cache replacement algorithm. Since the data in the CPU cache is managed at the cache line granularity (typically 64 bytes), sequential updates on the software level may be translated into small random accesses by the PM controller, resulting in sub-optimal bandwidth utilization. Intuitively, the leaf and internal node updates in the *b-tree* largely consist of sequential accesses and should induce low write amplification within the device. However, Table 2 shows that the device write amplification for the *btree* with the *Fill Random* workload is close to 4, indicating little sequentiality is preserved when accessing the PM.

The most efficient solution to this problem is to match the unit of the cache replacement policy to the access granularity of the PM controller. However, this would require significant modifications to the CPU architecture. We investigate the potential improvement of preserving sequentiality in PM via a simple software-level solution—*group flushing*. The key idea of *group flushing* is to explicitly flush modified cache lines in contiguous groups via cache-line flush instructions. We insert cache-line flushing instructions into the source code following updates to large blocks such as in the case of node updates and node initializations in the B+-Tree implementation. CPU flushing involves non-trivial run-time overhead, and is only beneficial if multiple adjacent cache-lines can be flushed together. Therefore, we only perform flushes if two or more cache-lines can be flushed in batch.

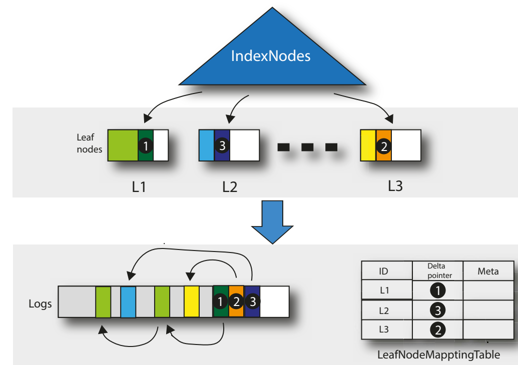


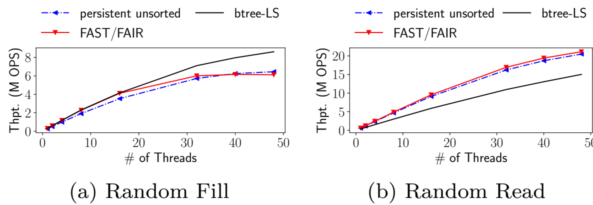
Figure 15: Design considerations.

We denote a B+Tree optimized with *group flushing* as *btree w GF*. Figure 14 compares the *btree w GF* to the original B+Tree (*btree*) as well as the unsorted B+Tree (*unsorted leaf*). As we can see, *btree w GF* achieves 24% higher insertion throughput compared to the *btree*. The *unsorted leaf* still achieves a slightly higher update performance, however, it sacrifices range search performance by up to 50% as shown in Figure 5b and Figure 4b. We also observe that the latency of *btree w GF* is higher than that of the *btree* when the thread number is less than 16. This is because, in the case of a low system load, writes to PM do not represent the main performance issue and the group flushing technique incurs extra instruction overheads due to additional `clwb` instructions. With **Finding 2** showing that *btree* and *unsorted leaf* have comparable performance under low load, in practice *btree* can be used by only enabling *group flushing* when the system load is high.

Lesson 2: Prior research focuses on addressing the write performance bottleneck by reducing writes at the software level, for instance, by leaving nodes unsorted. *Group flushing* provides a new direction by improving the I/O efficiency of PM by addressing the granularity disparity between the CPU and the PM. We envision *group flushing* to encourage hardware designers investigating PM-aware CPU cache replacement policies to further avoid the software overheads.

6.3 Log-structuring

Finding 3 and **Finding 6** motivate us to employ log-structuring for improving the update performance of persistent B+-Trees. As shown in Figure 15, random update operations on a conventional



(a) Random Fill (b) Random Read

Figure 16: Performance comparison between *btree-LS*, *persistent unsorted* and *FAST/FAIR*.

B+-Tree are translated to small random in-place writes in the PM address space. Due to the access disparity between CPU caches and the PM (64 vs 256 bytes), this random write pattern results in a significant write amplification of 3.2× as shown in Table 2. To confirm this issue, Table 2 shows that the write amplification of the *unsorted btree* is almost 3, indicating that two thirds of the write bandwidth is wasted, due to random in-place updates in the PM address space.

By incorporating a log-structured layout [44, 45], random writes can be batched into large sequential runs to reduce write amplification and thus better utilize the limited PM write bandwidth. However, this approach requires an efficient way to locate data in the log space. Virtualized B-Trees [34, 52] decouple the physical representation of the tree node from the logical representation. An indirection layer maps a logical identifier of a tree node to a chain of delta records in the log space, each record in the chain representing an update to the corresponding node. The effectiveness of log structuring, to a large degree, depends on the efficiency of garbage collection (GC) and failure recovery.

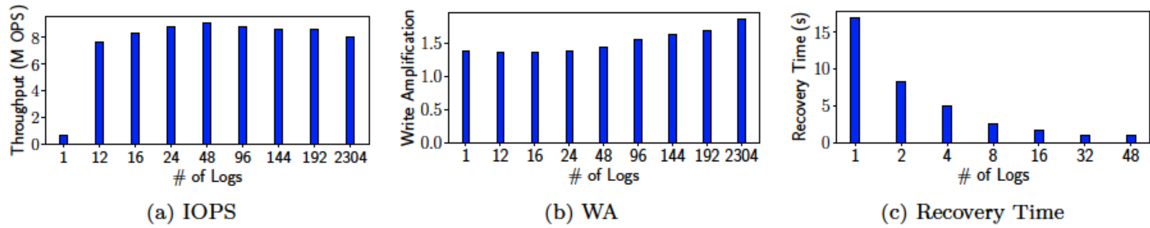
6.3.1 Design Considerations

Employ indirection only for leaf nodes. The virtualized B+-Tree in prior works [34, 52] fully separates the physical representation from the logical view. However, Wang et al. [51] demonstrates that the indirection overhead of the BwTree [34] can be as high as 18%. We propose a novel hybrid layout: indirection is only used when accessing leaves that are write-heavy. This approach represents a good trade-off between read and write performance. Figure 15 shows the organization of our proposed tree where each leaf node is identified by its leaf node ID (LNID) and a leaf node mapping table is used to translate the LNID to the physical pointer to the head of the delta (update) chain. For reads, a list traversal is performed on the delta chain to identify the search key. For an update, a delta record is prepared, the delta record is flushed to the DCPMM and the newly prepared delta is prepended to the delta chain. If the number of the items in the delta chain exceeds the pre-defined node size, a split operation is performed. The valid data on the chain is then distributed into two new nodes of equal size.

Supporting multiple logs. A single log used by all threads is the most straightforward log space organization, however, this approach severely limits the concurrency of updates and recovery operations. At the other extreme, a per-node log in *persistent unsorted* trees avoids the scalability issues at the cost of random writes and higher write amplification. We choose the middle ground by adopting a multi-log layout where updates are dispatched to a circular buffer-backed log based on the hash of its LNID. When the garbage of a log exceeds a predetermined threshold, a GC task is initiated by one of the multiple GC threads to start at the head of each circular buffer copying the valid data to its tail. The validity of a delta record is determined by its availability in the mapping table. During GC, valid delta records of the same chain are consolidated by the garbage collector into a new record and the address is updated in the mapping table.

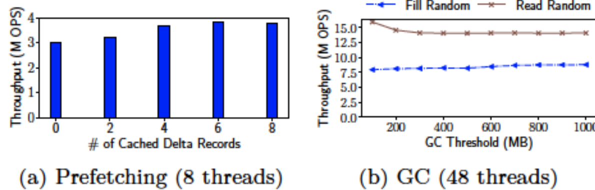
Avoiding random PM writes to the mapping table. Each node update requires modifying the corresponding chain head pointer in the leaf node mapping table, producing undesirable random writes. We recognize the fact that all the data required for the mapping table during recovery is always persisted in logs enabling us to keep the mapping table in DRAM and rebuild it upon recovery. When a failure is detected, multiple log replay threads are initiated each one assigned with a number of logs. The threads work in parallel by scanning each log in the chronological order rebuilding the delta chains in the mapping table.

Hiding access latencies with prefetching. The delta chain reduces the spatial locality within a leaf nod degrading search performance. For instance, two consecutive items in a conventional B-Tree may be separated in the log space of a virtualized B+-Tree, resulting in an additional cache miss when accessed together. Prefetching is used to mitigate the performance degradation for reads. In particular, for each leaf node, we cache the pointers to the most recent delta records, and prefetch them whenever a chain traversal is initiated. The number of delta records cached, represents a trade-off between read performance and the space consumption of the mapping table.



(a) IOPS (b) WA (c) Recovery Time

Figure 17: The impact of the number of logs.



(a) Prefetching (8 threads) (b) GC (48 threads)

Figure 18: The impact of GC and delta prefetching.

6.3.2 Evaluation

The results of the log-structured B+-tree implementation (denoted as *btree-LS*) are compared with those of *persistent unsorted* and *FAST/FAIR* in Figure 16, with GC disabled. One can see in Figure 16a that *btree-LS* achieves 41% higher insertion performance as compared to *FAST/FAIR*, from efficiently reducing the PM-level write amplification with a high system load. A 37% degradation in read performance is also observed (Figure 16b), mainly because of the reduced spatial locality in cache.

Figure 17 demonstrates the impact of log count selection by running *Fill Random* with 48 threads. Figure 17a shows that the update performance first increases drastically with the number of logs, as the contention for log updates is removed. After the number of logs goes beyond 48, the performance gradually drops as write sequentiality diminishes. Figure 17b confirms that write amplification inside the DCPMM increases from 1.4 to 1.7 as the number of logs increases. Figure 17c shows the recovery time of 160M entries by varying log count, with the replayer count equal to the log count and GC threshold set at 50MB. By effectively exploiting parallelism of the recovery process, with 48 replayer threads (and 48 logs) the whole index can be recovered in 0.89 secs and the speedup over the single-thread implementation is 21×.

We show the impact of garbage collection and delta prefetching in Figure 18. Figure 18b shows the impact of garbage collection. The background GC workers attempt to keep the amount of garbage data within a tunable threshold. The update performance degrades by only up to 7% as the GC threshold decreases from 1000MB to 100MB. Meanwhile the read performance increases by 20%. This is because GC is also responsible for consolidating small delta records and thus a more aggressive GC setting better preserves spatial locality. Figure 18a shows the efficiency of delta prefetching. As the number of cached delta entries is increased from 1 to 8, the read performance is improved by 26%. Delta prefetching can barely improve performance when the number cached delta entries goes beyond 4. We suspect this is because the maximum level of memory parallelism supported by the hardware is reached.

Lesson 3: Log-structuring efficiently utilizes the limited write bandwidth by significantly reducing device write amplification, at the cost of search performance and additional implementation complexities. By placing the indirection layer in DRAM, exploiting parallelism within the log space and judiciously selecting software parameters such as the garbage threshold, we can make log-structuring practical for PM indexes.

7 related work

Several studies investigate persistent B+-Tree designs for PM. Chen et al. [10] highlight the importance of reducing write overheads by proposing leaving nodes in the B+-tree unsorted. NV-Tree [58], FPTree [37], and wB^{+}-Tree [11] adopts the unsorted node organization to reduce the consistency overhead. NV-Tree [58] investigates the effectivenss of selective-persistence. To facilitate the search on an unsorted node, FPTree [37] extracts the fingerprint of the key into a metadata area while keeping the node virtually sorted with a small slotted array. FAST/FAIR [20] proposes a log-free, failure-atomic shift and rebalance algorithm to guarantee the failure-atomicity of write operations.

WORT [28] shifts its focus to radix tree, arguing that radix tree has more straightforward rebalancing operations as compared to B+-tree. Recent work investigates hash-based indexing [36, 61] for PM. Recipe [29] provides a principled approach for converting concurrent DRAM indexes into failure-atomic indexes for persistent memory. These works utilize a simulation based methodology ignoring the intricate details of real PM devices. Our work examines the effect of PM on B+-Tree indexing structures, presenting important findings that are missing from prior research.

Complementary to prior research on in-PM indexes, our study evaluates the real-world performance of in-PM B+-tree variants and explores the corresponding design space constraints by the characteristics of DCPMM. Xie et al. [55] conduct a comprehensive performance evaluation of several advanced in-DRAM index structures. Wu et al. [53] study the basic performance and formalize an "unwritten contract" of of Intel Optane SSD. However, Intel Optane SSDs exhibit quite different characteristics than their memory-form counterparts. Izraelevitz et al. [21] and Yang et al. [57] uncover the basic general performance characteristics and internal detals of DCPMM, while our work focuses on in-PM index performance. Lersch et al. [33] study the performance of PM range indexes in 3D XPoint memory while our work complements it with new insights and optimizations.

Relational database engines [1, 2, 3, 26, 38, 49] have been optimized for persistent memory. Pelley et al. [38] reduce the software complexity of traditional ARIES-style WAL. Write-Behind Logging [3] is a new logging and recovery protocol tailored for enavbling fast random accesses in PMs. 3 Tier BM [49] includes PM as a new caching layer and re-designs in-memory caching to leverage byte-addressability. In comparision to these studies on database storage management, our work focuses on the design of in-PM indexing structures. PM has also been incorporated into key-value stores [19, 23, 24]. SLM-DB [23] utilizes a persistent B+Tree in PM for indexing and allows all SSTables to be stored in a single level. NoveLSM [24] utilizes a byte-addressable skip list in PM to reduce logging and (de)serialization overheads. Our study on ordered indexing structures is complementary to the above work.

In-memory indexing has been extensively researched to address the increasing processor-memory performance gap. Cache-conscious index structures [41, 42] have been proposed to improve caching efficiency, while prefetching [9] and interleaving [8, 22, 27, 40] can hide memory latency by increasing memory-level parallelism. Several new in-memory index structrures have been proposed in the past decade including BwTree [34], a latch-free implementation of the B+-Tree that utilizes indirection and compare-and-swap operations for high scalability. ART [31, 32] is a trie-based data structure with adaptive nodes to reduce space consumption. Masstree [35] combines a trie and a B+-Tree to efficiently handle arbitrary keys. Wormhole [54] is a fast ordered index that combines the strengths of three indexing structures (hash table, prefix tree, B+-Tree). All these studies were conducted on DRAM, ignoring PM specific performance issues. We leave the study of these data structure on PM for future work.

Log-structuring [44] is a classical storage technique and has been constantly revisited by modern storage system designs. NOVA [56] is a log-structured filesystem designed for PMs that employs a per-inode log to improve concurrency. The BFTL [52] layer transforms fine-grained B-Tree operations into SSD-friendly block-based accesses by virtualizing the disk-based B-Tree index. Bw-Tree [34] virtualizes the in-memory B+-Tree index to enable i) latch-free updates and ii) log-structuring for the flash memory storage. Our study draws inspiration from the above work, while focusing on the implications of utilizing log-structuring for DCPMM.

8 Conclusion

Prior research on PM-aware data structures has leveraged simulation or emulation methodologies, ignoring the intricate details and performance pathologies of persistent memories such as Intel's 3D XPoint. This paper focuses on the B+-tree and its variants, exploring the common design issues of sorted index structures in a real system utilizing persistent memory. We demonstrate how to achieve high performance on a real-world persistent memory platform, combining several different techniques and providing an in-depth analysis of their micro architectural performance characteristics. Furthermore, we present two novel techniques *group flushing* and *log structuring for PM. Group flushing* improves performance by 24% by addressing the granularity disparity between CPU caches and the DCPMM controller. In addition, our study revisits the *log structuring* technique in the context of persistent memories further improving performance by 41%. We release the source code devel-

oped as part of this work as open-source to enable future research on 3D XPoint-based indexing structures.

References

[1] Mihnea Andrei, Christian Lemke, Günter Radestock, Robert Schulze, Carsten Thiel, Rolando Blanco, Akanksha Meghlan, Muhammad Sharique, Sebastian Seifert, Surendra Vishnoi, et al. SAP HANA adoption of non-volatile memory. *Proceedings of the VLDB Endowment*, 10(12):1754–1765, 2017.

[2] Joy Arulraj, Andrew Pavlo, and Subramanya R Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 707–722. ACM, 2015.

[3] Joy Arulraj, Matthew Perron, and Andrew Pavlo. Write-behind logging. *Proceedings of the VLDB Endowment*, 10(4):337–348, 2016.

[4] R. Bayer and E. M. Mccreight. Organization and maintenance of large ordered indexes. *Acta Inf.*, 1(3):173–189, September 1972.

[5] Anastasia Braginsky and Erez Petrank. A lock-free b+tree. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, page 58–67, New York, NY, USA, 2012. Association for Computing Machinery.

[6] Sang Kyun Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *Proceedings of the VLDB Endowment*, volume 1, pages 181–190, 2001.

[7] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 433–452, New York, NY, USA, 2014. ACM.

[8] Shimin Chen, Anastassia Ailamaki, Phillip B Gibbons, and Todd C Mowry. Improving hash join performance through prefetching. *ACM Transactions on Database Systems (TODS)*, 32(3):17–es, 2007.

[9] Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. Improving index performance through prefetching. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, pages 235–246, New York, NY, USA, 2001. ACM.

[10] Shimin Chen, Phillip B. Gibbons, and Suman Nath. Rethinking database algorithms for phase change memory. In *CIDR'11: 5th Biennial Conference on Innovative Data Systems Research*, January 2011.

[11] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment*, 8(7):786–797, 2015.

[12] Jungsik Choi, Jiwon Kim, and Hwansoo Han. Efficient memory mapped file i/o for in-memory file systems. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, 2017.

[13] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 105–118, New York, NY, USA, 2011. ACM.

[14] Douglas Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[15] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 373–386, 2018.

[16] Arnaldo Carvalho De Melo. The new linux'perf'tools. In *Slides from Linux Kongress*, volume 18, pages 1–42, 2010.

[17] Richard A. Hankins and Jignesh M. Patel. Effect of node size on the performance of cache-conscious b+-trees. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '03, pages 283–294, New York, NY, USA, 2003. ACM.

[18] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A simple optimistic skiplist algorithm. In *Proceedings of the 14th International Conference on Structural Information and Communication Complexity*, SIROCCO'07, pages 124–138, Berlin, Heidelberg, 2007. Springer-Verlag.

[19] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 967–979, Boston, MA, July 2018. USENIX Association.

[20] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 187–200, Oakland, CA, February 2018. USENIX Association.

[21] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[22] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. Exploiting coroutines to attack the “killer nanoseconds”. *Proc. VLDB Endow.*, 11(11):1702–1714, July 2018.

[23] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. SLM-DB: Single-level key-value store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 191–205, Boston, MA, February 2019. USENIX Association.

[24] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for nonvolatile memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 993–1005, Boston, MA, July 2018. USENIX Association.

[25] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D Nguyen, Tim Kaldewey, Victor W Lee, Scott A Brandt, and Pradeep Dubey. FAST: fast architecture sensitive tree search on modern cpus and gpus. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 339–350. ACM, 2010.

[26] Hideaki Kimura. FOEDUS: OLTP engine for a thousand cores and nvram. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 691–706. ACM, 2015.

[27] Onur Kocberber, Babak Falsafi, and Boris Grot. Asynchronous memory access chaining. *Proceedings of the VLDB Endowment*, 9(4):252–263, 2015.

[28] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: Write optimal radix tree for persistent memory storage systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 257–270, Santa Clara, CA, February 2017. USENIX Association.

[29] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: Converting concurrent dram indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP ’19, pages 462–477, New York, NY, USA, 2019. ACM.

[30] Philip L. Lehman and s. Bing Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4):650–670, December 1981.

[31] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *ICDE*, volume 13, pages 38–49, 2013.

[32] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, DaMoN ’16, pages 3:1–3:8, New York, NY, USA, 2016. ACM.

[33] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. Evaluating persistent memory range indexes. *Proceedings of the VLDB Endowment*, 13(4):574–587, 2019.

[34] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. The Bw-Tree: A b-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 302–313. IEEE, 2013.

[35] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys ’12, pages 183–196, New York, NY, USA, 2012. ACM.

[36] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. Write-optimized dynamic hashing for persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 31–44, 2019.

[37] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FP-Tree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 371–386, New York, NY, USA, 2016. ACM.

[38] Steven Pelley, Thomas F Wenisch, Brian T Gold, and Bill Bridge. Storage management in the nvram era. *Proceedings of the VLDB Endowment*, 7(2):121–132, 2013.

[39] M. Poremba, S. Mittal, D. Li, J. S. Vetter, and Y. Xie. Destiny: A tool for modeling emerging 3d nvm and edram caches. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2015.

[40] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. Interleaving with coroutines: A practical approach for robust index joins. *Proc. VLDB Endow.*, 11(2):230–242, October 2017.

[41] Jun Rao and Kenneth A. Ross. Cache conscious indexing for decision-support in main memory. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 78–89, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[42] Jun Rao and Kenneth A. Ross. Making b+-trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 475–486, New York, NY, USA, 2000. ACM.

[43] Simone Raoux, Geoffrey W Burr, Matthew J Breitwisch, Charles T Rettner, Y-C Chen, Robert M Shelby, Martin Salinga, Daniel Krebs, S-H Chen, H-L Lung, et al. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, 2008.

[44] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[45] Stephen M Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured memory for dram-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 1–16, 2014.

[46] Fujitsu Semiconductor. Fujitsu semiconductor releases world's largest density 8mbit reram product from september. `https://www.fujitsu.com/global/products/devices/semiconductor/memory/reram/spi-8m-mb85as8mt.html`, 2019.

[47] Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Satish, and Pradeep Dubey. PALM: Parallel architecture-friendly latch-free modifications to b+ trees on many-core processors. *Proc. VLDB Endowment*, 4(11):795–806, 2011.

[48] Luc Thomas. Basic principles, challenges and opportunities of stt-mram for embedded memory applications. In *33rd International Conference on Massive Storage Systems and Technology (MSST 2017)*, 2017.

[49] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. Managing non-volatile memory in database systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1541–1555. ACM, 2018.

[50] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM.

[51] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. Building a Bw-Tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 473–488, New York, NY, USA, 2018. ACM.

[52] Chin-Hsien Wu, Tei-Wei Kuo, and Li Ping Chang. An efficient b-tree layer implementation for flash-memory storage systems. *ACM Trans. Embed. Comput. Syst.*, 6(3), July 2007.

[53] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Towards an unwritten contract of intel optane SSD. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19). USENIX Associa-*

tion, Renton, WA, 2019.

[54] Xingbo Wu, Fan Ni, and Song Jiang. Wormhole: A fast ordered index for in-memory data management. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 18:1–18:16, New York, NY, USA, 2019. ACM.

[55] Zhongle Xie, Qingchao Cai, Gang Chen, Rui Mao, and Meihui Zhang. A comprehensive performance evaluation of modern in-memory indices. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 641–652. IEEE, 2018.

[56] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST '16, pages 323–338, Berkeley, CA, USA, 2016. USENIX Association.

[57] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. *arXiv preprint arXiv:1908.03583*, 2019.

[58] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: reducing consistency cost for nvm-based single level systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 167–181, 2015.

[59] Lu Zhang and Steven Swanson. Pangolin: A fault-tolerant persistent memory programming library. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, pages 897–911, Berkeley, CA, USA, 2019. USENIX Association.

[60] Pengfei Zuo and Yu Hua. A write-friendly and cache-optimized hashing scheme for non-volatile memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 29(5):985–998, 2017.

[61] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 461–476, 2018.