2021 IEEE 37th International Conference on Data Engineering (ICDE) | 978-1-7281-9184-3/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICDE51399.2021.00172

TLBtree: A Read/Write-Optimized Tree Index for Non-Volatile Memory

Yongping Luo[1], Peiquan Jin[1*], Qinglin Zhang[2], Bin Cheng[2]
[1] *University of Science and Technology of China, Hefei 230027, China*
[2] *Tencent, Shenzhen 518057, China*
jpq@ustc.edu.cn

***Abstract*—With the rapid advance of Non-Volatile Memory (NVM), it has been a hot topic to improve traditional tree indices like B+-tree for NVM. However, due to the high cost of the writing operations on NVM, few existing tree indices can offer high performance for both read and write operations. For example, the WB-tree with unsorted leaf nodes is write-optimized but has poor search performance. To address this problem, in this paper, we propose a read/write-optimized tree index called TLBtree (Two-Layer B+-tree) for NVM. TLBtree consists of a read-optimized top layer and a write-optimized bottom layer. We notice that the top levels of a B+-tree are read frequently, while the bottom levels are written frequently. Motivated by such an observation, we propose to design a read-optimized top layer and a write-optimized layer for the TLBtree index. We offer several read optimizations to implement the top layer and employ write-optimized structures to organize the bottom layer. With this mechanism, we can alleviate the read and write tradeoff of the index on NVM. We conduct extensive experiments on a server with Intel Optane DC Persistent Memory and compare TLBtree with state-of-the-art NVM-based tree indices, including WB-tree, Fast&fair, and FPtree. The results show that TLBtree outperforms other indices in write-intensive workloads by up to 1.7x throughput and achieves comparable read-only performance with read-optimized indices.**

Keywords—Hybrid index, Read/write optimization, B+-tree, Non-volatile memory

I. Introduction

B+-tree has been a famous index structure in database systems [1, 2], which attracted much research attention to improve its read/write performance on modern hardware [2-7]. However, most previous work either focused on optimizing read performance [2-4] or improving write performance [5-7]. Few studies were toward optimizing both the read and write performance of the B+-tree.

The emerging of Non-Volatile Memory (NVM) technologies makes it more challenging to optimize the read/write performance of in-memory indices. We attribute it to two main reasons [8]: (1) Available NVM products like the Intel Optane DC Persistent Memory exhibit a slightly higher read/write latency than DRAM. (2) Programmers have to issue instructions like cacheline flush (`clflush`) and memory fence (`mfence`) to guarantee the durability and failure-atomicity of writes, which will deteriorate the effect of cache locality. As the write operations on NVM are costly, it is necessary to reduce NVM writes when designing NVM-friendly indices. Following this rule, Chen et al. proposed the WB-tree [9] with unsorted nodes to reduce NVM writes. However, WB-tree is not read-optimized, resulting in worse search performance than the ordinary B+-tree. Bztree [10] uses the costly `PMwCAS` [11] instruction and the shadowing techniques to enable the persistence of node splitting/merging operations. Fast&fair [12] proposed node-level detection algorithms to avoid accessing inconsistent states, but it needs to shift records when doing insert and delete operations, bringing additional persistent cost. WB-tree and Fast&fair sacrifice write performance for achieving good read performance. Some other researchers [13-16] proposed only to persist leaf nodes on NVM and let interior nodes reside in DRAM. Such a design needs to rebuild the tree structure first, which is not time-efficient. Besides, the internal nodes will occupy a large amount of DRAM space. As a result, most of the previous B+-tree-like indices designed for NVM fail to provide high performance for both read and write operations.

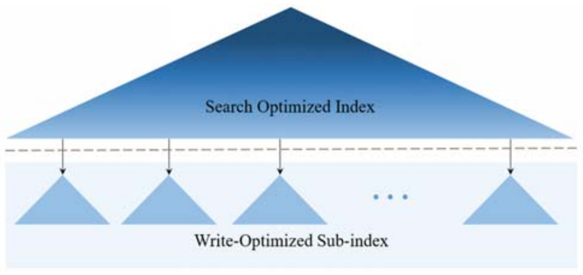

Figure 1. Two-Layer Persistent Index (TLPI) architecture composed of a search-optimized top layer and a NVM-friendly write-optimized bottom layer.

To address this problem, in this paper, we first propose a *Two-Layer Persistent Index* (*TLPI*) architecture (as shown in Fig. 1) to improve both the read and write performance of NVM-based indices. The TLPI architecture divides an index into two layers, including a top layer and a bottom layer. The top layer is read-optimized for fast retrieval, which is based on the fact that all operations in the tree index have to go through the top layer before reaching the bottom layer. Thus, the top layer is read frequently and should be search-friendly. On the other hand, the bottom layer in the TLPI architecture needs to be write-optimized because a tree index has to write data into the bottom layer, e.g., into the leaf nodes of the B+-tree. Also, as different bottom nodes may have different write frequencies, the bottom layer can contain different write-optimized sub-indices.

Figure 2 shows the read/write statistics for each layer in an in-memory B+-tree that runs on a randomly-accessing workload. We can see that the last three bottom layers, including the leaf layer, absorb about 99% of the writes. This observation motivates the design of a write-optimized bottom layer in the TLPI architecture. As a result, the TLPI architecture decouples read optimizations and write optimizations of an NVM-oriented index by the two-layer structure. Therefore, it is feasible to

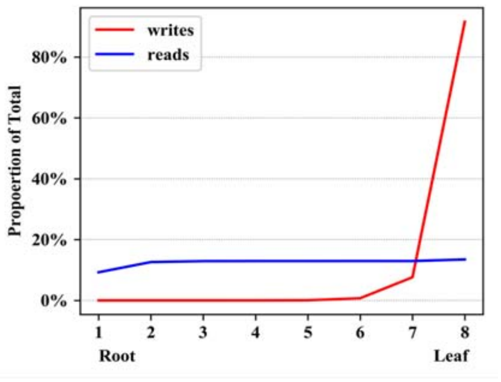

Figure 2. Reads and writes distribution in each level of an in-memory B-tree (16 fan-outs, 8 levels) running a randomly-accessing workload. Reads are even distributed in each level and writes are centered at the bottom layers

improve both the read and write performance of the index by devising appropriate index structures for the top and the bottom layer.

Based on the TLPI architecture, we propose a new NVM-friendly index called *Two-Layer B+-tree* (*TLBtree*). TLBtree follows the two-layer design of TLPI but presents an efficient implementation of each layer. Briefly, the top layer of TLBtree is cache-friendly and the bottom layer is write-atomic with efficient support to log-free node splitting. The bottom layer grows horizontally with new records inserted. Sub-indices in the bottom layer are linked together. When it reaches a threshold, the top layer will be rebuilt. We also present a gapped leaf-node structure in the top layer to reduce the rebuilding frequency. In summary, we make the following contributions in this paper:

(1) We noticed that most writes in a tree index are focused on a few bottom levels, based on which we present a two-layer persistent index architecture named TLPI for optimizing both the read and write performance of NVM-oriented indices.

(2) We propose a new read/write-optimized TLBtree following the TLPI architecture. TLBtree is composed of a read-optimized top layer and a write-optimized bottom layer. We devise efficient structures for each layer and invent a gapped leaf-node structure for the top layer to improve insertions and splits performance. TLBtree does rebuilding in a back ground manner and need not to block on-going and successive read and write operations. As a result, TLBtree can offer high read and write performance when running on pure NVM.

(3) We conduct extensive experiments on a real NVM-based environment with 1 TB of NVM and 384 GB of DRAM and compare TLBtree with three state-of-the-art NVM-based indices. The result shows that TLBtree outperforms its competitors in write-intensive workload and achieves comparable read-only performance, suggesting the two-layer persistent index architecture's efficiency.

The rest of the paper is structured as follows. Section II describes the background and related work. Section III gives an overview of TLBtree's design. In Section IV, we report the comparative experimental results on a real NVM environment, and finally, in Section V, we conclude the paper and discuss future research directions.

II. BACKGROUND AND RELATED WORK

A. *Non-Volatile Memory*

NVM is an industry-changing storage class memory technology, offering DRAM-level access latency and byte-addressability along with durability upon failure. NVM has a higher density and lower energy consumption compared to DRAM [17]. In other words, NVM is like a blend of two storage paradigms: byte-addressable DRAM and block-addressable storage (e.g., HDD/SSD). These properties make NVM highly promising for building a whole new database system that resides in persistent memory [18]. There are several distinct categories of NVM technology, among which Phase Change Memory [19] and 3DX-point [20] are the most promising candidates at delivering a huge capacity of storage class memory. Intel launched its commercial NVM products based on 3DX-point technology, namely Optane DC Persistent Memory [21]. Optane is so attractive that it has been used in many real commercial companies. According to some former experience with the Optane module [8], the Optane Memory module exhibits 3x read or write latency compared to the DRAM module. Its R/W bandwidth is much worse than the latter, primarily when tasked with write requests.

There are mainly two significant design challenges to apply NVM technology to existing in-memory data structures:

(1) **Durability.** Durability means data is durably stored in NVM when a write operation has finished. However, writes are not immediately persistent on NVM because of the CPU caching effect. Users should explicitly issue cacheline-flush instructions (e.g., `clflushopt`, `clwb`) and Memory fence instructions (e.g., `sfence`). Traditionally, we use the write-ahead log to guarantee durability, while this method is not economical on byte-addressable NVM as the data is redundantly stored. A log-free design, therefore, is more practical for NVM-based systems.

(2) **Consistency** (Failure Atomicity). Consistency means operations are atomic concerning other threads or failures. The atomicity unit in modern CPU is a word (e.g., 64 bits), while most writing operations touch multi-words in different cachelines. For example, a regular insert operation of B+-tree may trigger a node splitting that modifies several nodes. Modern CPUs exploit out-of-order-execution techniques to accelerate instruction execution, and CPU cache is essentially a black box to make things even harder. Thus, when a crash or power-loss occurs, a partial write operation may be persistent on the NVM devices, leading to an inconsistent state.

The literature [16] categorized software schemes to achieve data persistence in NVM memory, including logging, shadowing, `PMwCAS`, and NVM Atomic Writes (NAW). In the logging scheme, a log record is flushed into NVM first, and a consistent state can be recovered from the logs. Shadowing does not modify the original data in-place. It makes a new copy and utilizes a `CAS` operation that swaps the new and old data copy. The `PMwCAS` technique makes it possible to perform a `CAS` operation upon multi-words and guarantee persistence. It uses persistent records to save intermediate states. NAW is a scheme that ensures failure-atomicity manually by cacheline flush and memory fence instructions. Each NAW operation comprises several out-of-place updates and an 8 bytes in-place update to visualize all the updates. Note that there is another way to achieve failure-atomicity termed helping mechanism, which is adopted by Fast&fair [12]. In this scheme, one thread's inconsistent state can be detected, tolerated, or fixed by another thread. In general, NAW is the most simple, flexible, and light-weighted among all schemes. Therefore, many researchers have adopted the NAW scheme [9, 13-16]. We also adopt it in our design.

B. Persistent B+-tree Indices

A persistent B+-tree index is a B+-tree that resides in NVM-based memory and can recover from a normal reboot or failures to consistent states. It will not leave the index in an inconsistent state unless it can tolerate or fix it. The existing B+-tree on NVM are CDDS-tree [22], WB-tree [9], Fast&fair [12], Bztree [10], clfBtree [23], NV-tree [13], FPtree [14], DPTree [15], and LB$^+$-tree [16].

Some of the persistent B+-tree indices reside on pure NVM memory to guarantee instant recovery. The CDDS-tree [22], to the best of our knowledge, is the first persistent B+-tree under NVM. CDDS-tree uses a multi-version scheme to provide consistency without the additional overhead of logging or shadowing when doing node splitting or node merging. To maintain a sorted node failure-atomically, it needs to shift slots carefully with `clwb` and `sfence` instructions. The WB-tree [9] proposes to use an indirect vector to maintain an ordered node. At the same time, slots can be physically unsorted, which helps to simplify insert operation and reduce persisting costs as well. It also uses a WAL log to guarantee the persistence of structure modified operations. Thus, it can recover to a consistent state under the guard of minor log records. The Fast&fair [12] is a novel persistent B+-tree that maintains a sorted node by shifting slots and splits/merges nodes in a failure-tolerable manner. Other threads can detect the inconsistency and function correctly. To make things easier, The Bztree [10] uses the `PMwCAS` technique when performing multi-word atomic updates, making it the first lock-free persistent B+-tree index structure. But the `PMwCAS` and shadowing technique bring significant overhead.

Other persistent B+-tree indices choose a selectively persistent strategy, given that using leaf nodes can reconstruct the inner search tree. They only guarantee leaf node's persistence, and all the interior nodes either store in DRAM or ignore failure-atomicity and durability constraints. The NV-tree [13] is the first NVM B+-tree index that adopts this strategy. NV-tree is composed of a nearly fixed inner search tree and a persistent link-list of leaf nodes. All the leaf nodes are unsorted. They accept writing requests in an append-only strategy, releasing the complexity of failure-atomicity. NV-tree needs to frequently rebuild the inner search tree, which is time-consuming on a real NVM system. FPtree [14] proposed to store fingerprints of the keys in the node header to accelerate searching in an unsorted leaf node. A node searching first probes the fingerprint array, then checks the slots with matching fingerprints. Note that FPtree uses the *Hardware Transaction Memory* (*HTM*) for concurrent coordination of inner nodes. DPTree [15] adopted a dual-stage index architecture and appended each insert atomically to a write-optimized adaptive log. When the log reaches a size threshold, it is merged to a giant base tree atomically. LB$^+$-tree [16] is similar to FPtree. It also uses a fingerprint array to achieve better leaf node search performance. They also put forward an entry-moving scheme and a log-free node split strategy to optimize insert performance. As reported in [16], LB$^+$-tree performs better than FPtree, making itself prominent among this selective persistent B+-tree index.

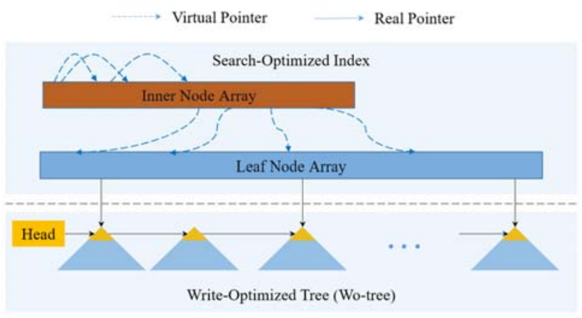

Figure 3. Two-layer structure of TLBtree. The top layer is less-mutable, cache-friendly and the bottom layer is a link-list of write-optimized trees.

III. DESIGN OF TLBTREE

The main advantage of the TLPI architecture is that its top layer and the bottom layer are decoupled. Therefore, the top layer and the bottom layer can adopt different index designs. It enables us to put together specific optimizations which may be too complicated to coexist in an individual index. Following the idea of TLPI, we present a new structure called TLBtree in this section. TLBtree consists of two layers, namely a search-optimized top layer and a write-optimized bottom layer. Figure 3 shows the structure of TLBtree. The search-optimized top layer aims for fast retrieval of sub-indices, while the write-optimized bottom layer aims to absorb insert and delete operations efficiently. As a result, TLBtree is promising to deliver both good read and write performance.

To make the top layer search-optimized, we linearize the top-layer index into a contiguous array. Also, we need not store the child pointers in inner nodes as we can calculate a child node's position in the node array. That means each inner node is 100% full and pointer-less. The leaf nodes in the top-layer index point to the bottom layer, including many write-optimized sub-indices linked together horizontally.

To make the bottom layer NVM-friendly and write-efficient, we made the following designs:

(1) We choose an Optane-friendly node size to maximize the total I/O bandwidth.

(2) Records are unsorted physically, but their order information is stored indirectly inside the header. This design enables append-only insertions.

(3) Insertions take two cacheline flushes, and deletions take only one cacheline flush, significantly reducing the persistent overhead.

(4) We propose a log-free splitting/merging mechanism to avoid logging overhead when doing structure modifications.

The bottom layer has a height limit. When the size of a sub-index in the bottom layer increases, we split the sub-index into two sub-indices. The records in the new sub-index can be accessed by the following steps:

(1) Locate the old sub-index.

(2) Traverse horizontally by sibling pointer till the correct sub-index.

(3) Access the corresponding record in the current sub-index.

The splitting of sub-indices may degrade read/write performance. Therefore, we make several optimizations to improve performance further. Firstly, we leave empty slots in the top layer leaf nodes to absorb moderate sub-index splitting. Secondly, when we find that traversal in the sub-indices link-list is beyond a configurable threshold, we rebuild the top layer using the link-list. Due to the decoupled structure of TLBtree, the rebuilding procedure can be done in a background manner without blocking any read and write operations.

IV. EVALUATION

A. Experiment Setting

Real NVM Environment. We ran our experiments on a server with real NVM. The server contains 384 GB DRAM and 1 TB Optane DC memory, distributed upon two sockets. To avoid the impact of NUMA effect on the experimental results, we ran all programs using only one socket, which means that we only use the CPU and NVM memory within the same socket. The operating system on the server is CentOS with a kernel version 5.8.7. A DAX-aware ext4 file system is created after configuring all the Optane modules into *app-direct* mode. Then, we mount the file system using DAX option. We utilize PMDK 1.8 to map files on Optane into virtual memory space and handle basic memory allocation tasks.

Persistent Indices Compared. We compare six NVM-based persistent indices that reside in NVM. They are WB-tree, Fast&fair, FPtree, our write-optimized bottom layer index (abbreviate as Wo-tree), and two variants of TLBtree structures, including TLBtree_FO and TLBtree.

- *WB-tree.* The key-value pairs inside a WB-tree node are not sorted. That reduces the cacheline-flush number of insert operations but sacrifices search performance to some extent.
- *Fast&fair.* The Fast&fair leverages a novel detecting mechanism to tolerate inconsistency. All keys are sorted, but it has to shift slots to maintain the order.
- *FPtree.* The FPtree adopts a selective persistent strategy to guarantee the persistence of leaf nodes and puts all inner nodes into DRAM.
- *Wo-tree.* The Wo-tree is a log-free, write-optimized persistent B+-tree index. Similar to WB-tree, keys inside a node are physically unsorted. However, Wo-tree supports splitting and merge nodes without logging. It further reduces the cacheline-flush number.
- *TLBtree_FO and TLBtree.* They both adopt Wo-tree as the bottom layer, but they choose different top layer indices. TLBtree_FO chooses Fast&fair as its top layer, while TLBtree chooses our tailored read-optimized tree. Both TLBtree_FO and TLBtree limit its bottom layer to 2, as it renders a subtle tradeoff between read performance and write performance.

WB-tree, Fast&fair, and FPtree are three state-of-the-art persistent indices for NVM-based memory, in which WB-tree and Fast&fair are specially designed for NVM-only systems. Our evaluation focuses on performance under an NVM-only system, so we store all the nodes and auxiliary structures of each index in Optane DC memory, including the inner nodes of FPtree. Since WB-tree and FPtree are not open-source, we implemented them delicately in the Optane environment. Fast&fair is open-sourced, and we use its source codes in our evaluation.

To make the comparison fair enough, we implement six indices as close as possible. First, all the indices have the same node size in the same comparison. Second, we use the same interface to do memory allocation. We implemented a *Persistent Memory Allocator* to allocate memory from the pool file on Optane. It ensures that each node is 256 B aligned. It supports address transformation from physical memory address to offset inside the pool and vice versa, facilitating storing persistent addresses into Optane. Finally, all the indices reside on pure Optane memory and use `clwb` and `sfence` instructions to guarantee persistence. Also, we reduce the WB-tree's additional `clwb` instructions according to the advice in the literature [25].

Dataset. Without losing generality, we use an 8-byte key and an 8-byte payload as the record content, which is compatible with the configuration in Fast&fair. Since the B+-tree structure is self-balanced and not sensitive to the data distribution, we choose a random dataset as the initial dataset and build a persistent B+-tree index upon it.

B. Overall Performance

In this section, we evaluate the single thread throughput of six indices under different workloads. Each index is built upon 256MB non-duplicated random key-value pairs by calling the inserting function iteratively. All six indices have a node size of 256B, making the comparison fair.

The workload consists of look-up queries and inserting operations. According to the distribution of query keys, we divide the workload into two categories, a random workload and a skewed workload. In the random workload, the query keys are under random distribution. In the skewed workload, the query keys are under the *Zipfan* distribution, and we set the skewness value to be 0.7 (high skew). In each category, we have *read-only queries* (RO), *read-write queries* (RW, half of the query are insert queries), and *write-only queries* (WO). The keys in a look-up query are from the initial dataset, and we ensure that the keys of insert operations are not in the index.

The overall result is shown in Fig. 4. WB-tree shows no advantage over other indices. It has the worst read performance and comparable write performance compared to other ones, which is mainly because the keys in WB-tree are not sorted inside a node, which incurs bad cacheline efficiency when searching a node. Wo-tree does not store keys physically sorted either, so it achieves a similar read performance as WB-tree. However, Wo-tree delivers a speedup of 1.4x over WB-tree and FPtree on write throughput. This advantage also holds for RW workload. We can conclude that Wo-tree is write-optimized for NVM memory, but it suffers from poor read performance.

For indices that contain sorted nodes, such as Fast&fair and FPtree, they achieve the best read performance. But when the query includes 50% insert operations, the query throughput drops by 20% and 60% for Fast&fair and FPtree, respectively. The write performance for FPtree is worse because it needs logging to guarantee failure-atomicity when doing node splitting. Moreover, FPtree is slightly slower than WB-tree

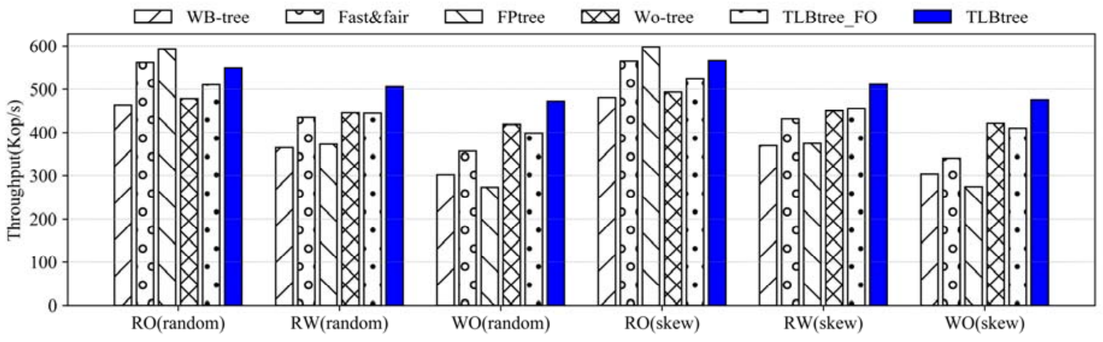

Figure 4: Overall throughput of six indices under different workloads.

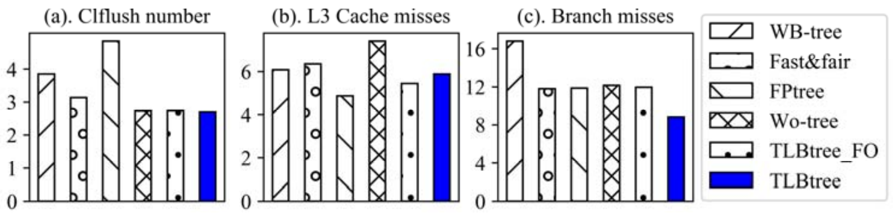

Figure 5: monitoring the hardware behavior of six indices under insert operations and look-up operations. (a) reports the average `clflush` number of one insert operation, and (b) and (c) report the average L3 cache missing number and Branch missing number of one look-up operation.

when doing node splitting as it needs to sort the keys when splitting a leaf node. We can conclude that FPtree is suitable for read-intensive workload but has poor performance for write-intensive workloads. Besides, FPtree does not persist inner nodes and takes more time to recover than other indices. In contrast, Fast&fair is slightly better than FPtree as it can achieve comparable throughput on RW workloads.

Our proposed TLBtree candidates perform well on both read and write workloads. They achieve comparable read-only performance with Fast&fair and FPtree. Besides, on the RW and WO workloads, they beat state-of-the-art indices. Combined with Fast&fair and Wo-tree, TLBtree_FO exhibits similar write-performance with Wo-tree and better read performance than Wo-tree. That shows the advantage of TLPI on combining read-optimized indices like Fast&fair and write-optimized indices like Wo-tree elegantly. TLBtree further improves read performance by using a tailored read-optimized top layer. The result shows that TLBtree achieves similar read performance with Fast&fair and up to 1.7x speedup on the write-only workload compared to read-optimized indices. TLBtree beats its competitors when query workload contains insert operations.

When query keys are under skew distributions, all indices get slightly higher throughput. We attribute it to the effect of CPU caches. Same as random workload, our Wo-tree, and TLBtree maintain high write throughput. To be accurate, TLBtree achieves about 1.6x/1.5x/1.8x/1.2x/1.2x speedup on write performance compared to WB-tree/Fast&fair/FPtree/Wo-tree.

To sum up, the TLPI architecture can alleviate read and write performance tradeoffs. Fast&fair and FPtree are read-optimized but not attractive under insert operations. Wo-tree is write-optimized but not suitable for read-intensive workloads. Our TLBtree puts the read-optimized and write-optimized ingredients together, and it shows the advantage of overall performance.

C. *Monitoring Hardware Behavior*

In this section, we measure six indices' hardware behavior under insert operations and look-up operations to reveal the optimization of Wo-tree and TLBtree. We mainly monitor the hardware behavior like cacheline flushes, L3 cache misses, and branch misses by a program counter and the Linux *Perf* tool [27]. The initial dataset is 256MB random key-value pairs, and all indices use a node size of 256B. We amortize the value to one operation and show results in Fig. 5.

Figure 5(a) is the cacheline-flush number incurred by one insert operation. cacheline-flush number is a representative metric for the persistent cost of NVM data structure. Most formal researches focus on reducing the cacheline-flush number [14-16]. As shown in Fig. 5(a), WB-tree and FPtree need four or five cacheline flushes for one insert operation. This number is higher than that of Fast&fair, which takes about three cacheline flushes. Wo-tree is write-optimized, and it needs 2.7 cacheline flushes to finish one insert operation. Compared to its competitors, it reduces the cacheline-flush number of WB-tree, Fast&fair, and FPtree by 40%, 16%, and 77%, respectively. The result is confirmed in Fig. 4: FPtree has the largest cacheline-flush number and then the worst write performance. TLBtree has the similar cacheline-flush number with Wo-tree. Besides, with the improvements of the search-optimized top layer, TLBtree achieves the best write performance among all indices.

Figure 5(b) shows the L3 cache misses for one look-up query. Because the memory access latency is orders of magnitude slower than the cache access and CPU cost, the look-up cost is highly impacted by L3 cache misses. FPtree exhibits the least

L3 cache misses than the other indices because its inner node is ordered and not failure-atomic. Therefore, FPtree achieves the best read performance. Wo-tree has a slightly larger value of L3 cache misses, followed by the other four indices with a similar value.

Figure 5(c) further shows the branch-miss number for one look-up query. Besides L3 cache misses, the branch misses cost is also an essential factor imfluencing the operation latency. WB-tree use a *slotArray* to store the order information of records, which leads to a higher branch misses number compared to the other indices. The top layer of TLBtree is a compact and pointer-less search tree, which is tailored for high search performance. That accounts for lower branch misses of TLBtree compared to Wo-tree and TLBtree_FO.

V. Conclusion

In this paper, we first propose a new kind of index architecture named TLPI to improve both the read and write performance for NVM-oriented indices. It divides the index into two layers, namely a top layer and a bottom layer. The top layer is read-optimized for fast retrieval because all searches have to go through the top layer before reaching the bottom layer. Thus, the top layer is read frequently and needs to offer high search performance. On the other hand, the bottom layer in the TLPI architecture needs to be write-optimized as 99% percent of the writes are biased to the last 2-3 level.

Based on the idea of TLPI, we further propose a new NVM-friendly index called TLBtree. TLBtree follows the two-layer design of TLPI but presents an efficient implementation of each layer. The top layer of TLBtree is cache-friendly, less mutable, and the bottom layer is write-atomic and supports log-free node splitting. The bottom layer grows horizontally with the insertions of new records. When it reaches a threshold, we rebuild the top layer. We also present a gapped leaf-node structure and a delayed rebuilding strategy to reduce the rebuilding cost. We conduct experiments on a real NVM environment with 1TB of Intel Optane persistent memory and 384GB of DRAM, and compare TLBtree with a number of existing indices. The results suggests the efficiency of the TLPI architecture.

In the future, we will evaluate TLBtree on other kinds of workloads, such as the OLTP and TPC-C workloads. We will also consider using various kinds of existing index structures to replace the top and bottom layers in the TLPI architecture. We will experimentally reveal how current indices perform in the TLPI architecture.

Acknowledgments

This work is supported by the National Science Foundation of China (no. 62072419 and no. 61672479) and Tencent. Peiquan Jin is the corresponding author of this paper.

References

[1] D. Comer. The ubiquitous B-tree. ACM Computing Surveys. 11(2):121–137, 1979.

[2] P. Jin, C. Yang, C. S. Jensen, P. Yang, L. Yue. Read/write-optimized tree indexing for solid-state drives. The VLDB Journal, 25(5): 695-717, 2016

[3] C. Yang, P. Jin, L. Yue, P. Yang. Efficient buffer management for tree indexes on solid state drives. International Journal of Parallel Programming. 44(1): 5-25, 2016

[4] L. Li, P. Jin, C. Yang, Z. Wu, L. Yue. Optimizing B+-tree for PCM-based hybrid memory. In EDBT, 662-663, 2016

[5] P. Jin, P. Yang, L. Yue. Optimizing B+-tree for hybrid storage systems. Distributed Parallel Databases, 33(3): 449-475, 2015

[6] L. Li, P. Jin, C. Yang, S. Wan, L. Yue. XB+-Tree: A novel index for PCM/DRAM-based hybrid memory. In ADC, 357-368, 2016

[7] M. Bender, M. Farach-Colton, W. Jannen, R. Johnson, B. Kuszmaul, D. Porter, J. Yuan, and Y. Zhan. An Introduction to Bε-trees and Write-Optimization. USENIX Magazine, 40(5):22-28, 2015.

[8] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In FAST, 169-182, 2020.

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

[10] J. Arulraj, J. J. Levandoski, U. F. Minhas, and P. Larson. Bztree: A high-performance latch-free range index for non-volatile memory. PVLDB, 11(5):553–565, 2018.

[11] T. Wang, J. Levandoski, and P. Larson. Easy Lock-Free Indexing in Non-Volatile Memory. In ICDE, 461-472, 2018.

[12] D. Hwang, W. Kim, Y. Won, and B. Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. In FAST, 187–200, 2018.

[13] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. NV-Tree: Reducing consistency cost for NVM-based single level systems. In FAST, 167–181, 2015.

[14] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. FPtree: A hybrid SCM-DRAM persistent and concurrent b-tree for storage class memory. In SIGMOD, 371–386, 2016.

[15] X. Zhou, L. Shou, K. Chen, W. Hu, and G. Chen. DPtree: differential indexing for persistent memory. PVLDB, 13(4):421–434, 2019.

[16] J. Liu, S. Chen, and L. Wang. LB+Trees: optimizing persistent index performance on 3DXPoint memory. PVLDB, 13(7):1078–1090, 2020.

[17] Y. Luo, Z. Chu, P. Jin, S. Wan. Efficient sorting and join on NVM-based hybrid memory. In ICA3PP, 15-30, 2020

[18] A. Renen, V. Leis, A. Kemper, T. Neumann, T. Hashida, K.i Oe, Y. Doi, L. Harada, and M. Sato. Managing Non-Volatile Memory in Database Systems. In SIGMOD, 1541–1555, 2018.

[19] K. Chen, P. Jin, L. Yue. A novel page replacement algorithm for the hybrid memory architecture Involving PCM and DRAM. In NPC, 108-119, 2014

[20] 3D XpointTM: A breakthrough in non-volatile memory technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html , 2020.

[21] Intel Optane Memory, https://www.intel.com/content/www/us/en/architecture-and-technology/optane-memory.html, 2020.

[22] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In FAST, 61–75, 2011.

[23] W. H. Kim, J. Seo, J. Kim, and B. Nam. 2018. ClfB-tree: Cacheline Friendly Persistent B-tree for NVRAM. ACM Transactions on Storage. Article 5, 1-17, 2018.

[24] P. L. Lehman, and S. B. Yao.Efficient locking for concurrent operations on B-trees. ACM Transactions on Database Systems. 6(4):650–670, 1981.

[25] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. SIGARCH Computer Architecture News, 42(3):265–276, 2014.

[26] Intel Optane™ Persistent Memory 200 Series Brief https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-persistent-memory-200-series-brief.html , 2020.

[27] Linux Perf Wiki, https://perf.wiki.kernel.org/index.php/Main_Page, 2020