Optimizing Structural Modification Operation for B^+-Tree on Byte-Addressable Devices

Dingze Hong*†,Jinlei Hu*†, Jianxi Chen†, Dan Feng†,Jian Liu‡
†Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System of MoE
School of Computer Science and Technology, Huazhong University of Science and Technology
‡The Fifth Electronics Research Institute of MIIT, China.
{hongdingze,hujinlei,chenjx,dfeng}@hust.edu.cn, {liujian}@ceprei.com
Corresponding author: Jianxi Chen

***Abstract*—Persistent Memory (PM) offers both byte-addressability and non-volatility, making it well-suited for accelerating B^+-Tree indexes. However, existing persistent B^+-Tree indexes face significant performance challenges due to high structural modification operation (SMO) overhead. SMOs often result in costly item migrations and increased tail latency, which severely degrade the overall performance. In this paper, we present SSTree, a high-performance B^+-Tree index specifically optimized to address SMO overhead. SSTree introduces three key innovations: (i) efficient leaf node expansion using a list of *subnodes* to postpone expensive node splits, (ii) delegated fingerprints to speed up search operations across subnodes, and (iii) proactive subnode compaction that employs out-of-place updates to optimize item organization. Our evaluation demonstrates that SSTree delivers up to 4.38× higher write throughput and up to 62× lower tail latency compared to state-of-the-art persistent B^+-Tree indexes.**

***Index Terms*—persistent memory, tree index, performance optimization**

I. Introduction

Persistent memory (PM) is a rapidly advancing byte-addressable storage technology, exemplified by devices such as phase-change memory [1], Intel Optane DCPMM [2], and emerging CXL-based (Compute Express Link) memory devices like Samsung's Memory-Semantic SSD [3]. PM offers a larger capacity than DRAM while maintaining both byte-addressability and persistence [4], [5]. These characteristics make PM particularly well-suited for accelerating storage systems, especially the wildly used B^+-Tree due to their low tail latency and robust crash consistency [6]–[8].

However, the performance of persistent B^+-Tree indexes is constrained by the limited bandwidth and relatively high latency of PM [2], making structural modification operations (SMOs) significantly more expensive compared to DRAM-based systems [9]. Most persistent B^+-Tree implementations rely on the traditional node split policy, where a leaf node with no available free slot is split, moving half of the items to a new node. This process leads to read/write amplification and under-utilized PM bandwidth, ultimately reducing insertion performance. Additionally, the leaf node is typically locked during the split, blocking other threads and decreasing concurrency across the B^+-Tree index. Our evaluations in Section II-B demonstrate that SMOs can degrade insertion throughput by up to 60% and increase tail latency by up to 7×.

Existing persistent B^+-Tree indexes [8], [10]–[13] employ various methods to reduce the SMO overhead on PM, but these solutions remain sub-optimal. PACTree [14] introduces asynchronous batch processing to move SMOs off the critical path. However, this approach increases tail latency, as batching a large number of operations can quickly saturate the restricted write bandwidth. Additionally, the background merging threads can lag behind, eventually becoming a bottleneck. Some persistent B^+-Tree indexes [15], [16] attempt to reduce SMO concurrency issues by using hardware support such as Hardware Transactional Memory (HTM) [17]. However, HTM suffers from high abort rates when working with large datasets or mixed read/write workloads, making the performance of HTM-based B^+-Tree highly unstable [14]. Our observations show that many of these indexes experience infinite HTM aborts when datasets exceed 16 million keys. Additionally, techniques like shadow pointers, used in systems such as LB^+-Tree [8] and TLBTree [13], reduce the number of flushes needed for persistence but offer limited improvements in addressing SMO overhead.

Inspired by the concept of *separate chaining* in hash table [18], we propose extending a leaf node with a subnode list to reduce node split overhead. However, designing a chained structure for leaf nodes while maintaining high performance in persistent B^+-Tree introduces several challenges: *1) High overhead by pointer-chasing among subnodes*. The subnode list inherently suffers from a lower locality, which leads to more severe performance degradation, especially given the higher access latency of PM. *2) Reduced bandwidth utilization from subnodes fragmentation*. To enhance write performance, persistent B^+-Tree often uses unsorted leaf nodes [15]. However, this approach causes valid keys to be scattered across slots, resulting in inefficient search performance.

To address these challenges, this paper introduces SSTree,

*The first two authors contributed equally to this work. This work is funded by the National Key Research and Development Program (No.2022YFB4501300), National Natural Science Foundation of China (NSFC No.61821003, No.U22A2027), and the Open Project (HK202201317) of Key Laboratory of MIIT for Basic Software and Hardware Performance and Reliability Measurement.

an optimized persistent B$^+$-Tree index by utilizing the chained leaf node structure to minimize the SMO overhead. Instead of immediately splitting a leaf node when it runs out of slots, SSTree allocates a subnode and inserts it at the head of the list, thereby delaying node splits. This design improves throughput and reduces read/write amplification. Additionally, SSTree incorporates several key mechanisms to further enhance the overall performance: *1) Improved Space Utilization*. SSTree employs a subnode reclamation, which utilizes out-of-place updates for item movement. The updated items are moved to the head subnode as much as possible, compacting the data and improving locality. When the last key in an overflow subnode is moved, the subnode is reclaimed simultaneously. *2) Faster Search Performance*. SSTree introduces delegated fingerprints across subnodes within a logical leaf node. Each subnode stores the fingerprint of its successor, allowing SSTree to quickly check the fingerprint before accessing the subnode while maintaining ample space for item slots. Our evaluation demonstrates that SSTree delivers up to 4.38× higher write throughput and up to 62× lower tail latency compared to state-of-the-art persistent B$^+$-Tree indexes.

In summary, we make the following contributions:

- We analyze and demonstrate the significant impact of SMO overhead in existing persistent B$^+$-Tree.
- We propose SSTree, a new persistent B$^+$-Tree optimized to reduce SMO overheads based on our analysis. SSTree uses a subnode list to extend leaf nodes, providing a cost-efficient solution for leaf node expansion and reducing node splitting overhead. SSTree also introduce delegated fingerprints and proactive subnode compaction to accelerate the search performance.
- We validate the effectiveness of SSTree in reducing SMO overheads and improving overall performance through extensive evaluations across various workloads.

II. Background and Motivation

A. Byte-Addressable Devices

Persistent memory (PM) is a type of byte- addressable storage technology by combining disk-like durability with DRAM-like performance [2]. Ideally, PM will bridge the performance and capacity gap between main memory and secondary memory, making it suitable for high-performance persistent storage systems, especially accelerating the traditional tree indexes such as B$^+$-Tree.

Persistent memory has alternative product forms, including battery-backed up (BBU) DIMM [1] and 3DXPoint [2], already have commercially available products. Besides, the latest memory technology based on Compute Express Link (CXL) is expected to provide a new way to implement PM [3]. Recent works [4], [5] have empirically tested the characteristics of CXL devices. It is shown that CXL devices exhibit some performance trends similar to the Intel Optane DCPMM, the first commercial but discontinued. Their bandwidth also peaks with 8-16 threads and latency is 2×-4× higher compared with DRAM. Besides, commercial CXL products such as CMM-H [3] are announced to support persistent memory mode and target Optane DCPMM customers. Moreover, CXL is expected to support non-temporal instructions [5] and Persistent Memory Development Kit (PMDK) [4]. Therefore, we argue that current persistent B$^+$-Tree indexes designed for PM will still be functional and exhibit similar performance characteristics on the future CXL-based persistent memory.

However, PM comes with several shortcomings. The latency of PM is still several times that of DRAM. PM also exhibits limited bandwidth, especially for concurrent writes. Additionally, the granularity of data transfer between CPU and memory corresponds to a cache line (usually 64 bytes), which does not match the native 8-byte atomic access supported by the CPU. Such mismatch leads to read/write amplification at the hardware level. On the platforms that do not support Extended Asynchronous DRAM Refresh (eADR) [19], the CPU needs to execute specific instructions, such as flush (e.g. *clflush*, *clwb*) and fence (e.g. *mfence*), to prevent inconsistency caused by non-atomic PM write or out-of-order execution of modern CPU. The above shortcomings make it more challenging to implement high-performance persistent B$^+$-Tree.

B. Structural Modification Operation

Structural Modification Operation (SMO) causes significant overhead in B$^+$-Tree [9], [14], [16]. A *node split* occurs when a node is full and cannot be inserted. It moves part of the key-value pairs (which we refer to as *items* or *KV* for brevity) to a newly allocated node. Then, B$^+$-Tree recursively inserts the pointer of the new node into the parent node. During this process, the original node is protected by a lock. Thus, other threads must wait or retry frequently, degrading the concurrency. Moreover, with each leaf node split, half of the items are copied and re-inserted into the new node. Consequently, frequent node splits result in duplication of data copying, thereby incurring significant read/write amplification on PM. SMO also includes node rebalance and node merge. Memory-based B$^+$-Tree does not perform node rebalance or merge very often [6], [8], [9], [11]. Therefore, this paper focuses on the SMO overhead caused by node splits.

Existing persistent B$^+$-Tree indexes propose several methods to optimize SMO. However, we found that these designs still lack effectiveness:

HTM based SMO optimization. RNTree [15] and NBTree [16] mitigate the concurrent interference of SMO by dedicated hardware support such as Hardware Transactional Memory (HTM). However, HTM suffers a higher abort rate in large datasets and mixed read/write workloads. A transaction needs to cover more operations and becomes harder to execute atomically as the tree grows in height. Previous studies [14], [20] demonstrate that the abort rate of HTM increases considerably under 16 or more threads. This makes the performance of the B$^+$-Tree indexes based on HTM very unstable. Our test shows that many of them get stuck in infinite HTM abortion when the dataset contains more than 16 million keys. Besides, HTM is executed on the hardware level, making the program less portable and harder to profile.

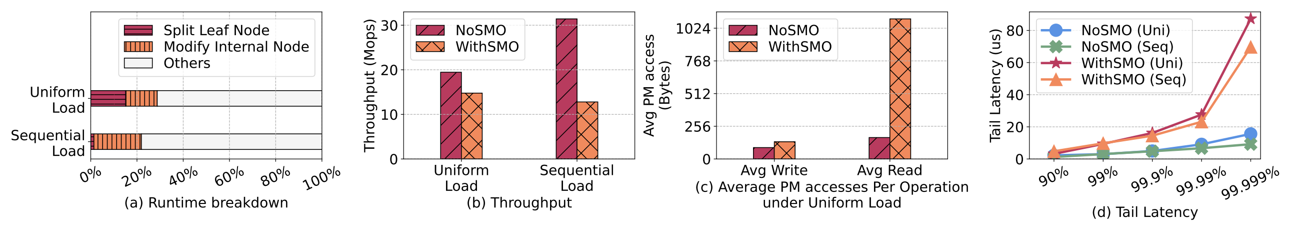

Fig. 1. SMO overhead analysis by evaluating LB$^+$-Tree. All the tests are performed under 32 threads.

Asynchronous SMO optimization. DPTree [21] and PACTree [14] propose an asynchronous mechanism for SMO. Such designs modify internal nodes asynchronously and periodically, moving SMOs out of the critical path. However, asynchronous designs necessitate additional shared message queues and complicate the concurrency control. Besides, the scalability is suboptimal because the background threads contend for limited CPU and memory resources against the foreground threads [22].

Shadow Pointer based SMO optimization. LB$^+$-Tree [8] and TLBTree [13] optimize the node split by leveraging shadow sibling pointers in the leaf node and atomically flipping the metadata. Such design only reduces the number of flush and fence instructions and mitigates the overhead of maintaining durability and recovery. It does not affect the number of total splits.

We analyze the impact of SMO by evaluating LB$^+$-Tree, a state-of-the-art persistent B$^+$-Tree index, using the same setup as described in Section IV-A. We first load the tree with 100 million keys. Figure 1a shows the runtime breakdown. Then we disable the node merge of LB$^+$-Tree, delete all keys, and reinsert them. The reinsertion of the keys will not trigger any SMO. We compare the performance difference between the two load phases. The results are shown in Figure 1b-d. We observe that, although SMO consumes 20%-25% of time, the throughput degrades to 0.76× and 0.4× respectively under uniform and sequential load. This means SMO causes considerable interference on concurrency that slows down the search performance of internal nodes. Besides, SMO not only incurs 1.5× higher writes but also results in 6.5× reads to PM due to sorting keys. Moreover, SMO is the main source of the tail latency. As shown in Figure 1d, SMO contributes up to 4× higher p90 latency and 7× higher p99.999 latency. In summary, SMO causes a critical bottleneck of persistent B$^+$-Tree indexes.

C. *Motivation and Design Challenges*

To reduce the SMO overhead, we propose a core idea to extend the B$^+$-Tree leaf node with a linked list of *subnodes*. It allocates extra space to store items instead of immediately resizing the index. Therefore, it provides an effective way for persistent B$^+$-Tree to delay SMO. There are still several challenges in designing such a chained structure within the leaf node while maintaining high performance:

Challenge 1. Search performance degrades as more keys are inserted into the subnode. Compared with the traditional leaf node, a linked sub-node list exhibits inherently lower locality due to pointer chasing. As a result, a search operation will incur more PM reads because it has to probe multiple subnodes to get the target key. Previous works [9]–[11] exploit *selective persistence* that stores part of the index in DRAM, reducing the high access overheads on PM at the costs of higher DRAM consumption. Therefore, it is a challenge to maximize the search performance while avoiding the overuse of DRAM.

Challenge 2. Low space utilization within the subnodes wastes PM bandwidth. The space utilization of leaf nodes in B$^+$-Tree is ususally limited to about 60%. Our evaluation in Section IV-C shows that the space utilization further decreases to 43% after deleting part of the keys. Persistent B$^+$-Tree indexes prefer unsorted leaf nodes for higher write performance [15]. The items will become scattered over multiple subnodes after several insertions and deletions. As a result, the index incurs more cache misses and underutilizes PM bandwidth. This motivates us to improve the data compactness inside a leaf node while maintaining high performance for insertion and deletion.

III. The Design of SSTree

A. *Overview*

To reduce the SMO overhead, the basic idea of SSTree is to extend the leaf node with a subnode list. This allows SSTree to flexibly expand the leaf node and reduce the frequency of node splits by linking more subnodes. The first subnode in the list is referred to as the *head subnode*, while subsequent ones are termed *overflow subnodes*. Each subnode starts with a *next pointer* to its successor or *NULL*. The size of the subnode is configured based on the internal access size of the PM device. For example, we use 256B subnode to match the internal XPLine of Optane DCPMM in our evaluated PM product [2].

The overall architecture of SSTree is shown in Figure 2, SSTree decouples the B$^+$-Tree index into a volatile search layer and a persistent data layer. The search layer includes all internal nodes and volatile metadata of leaf nodes. The data layer includes the remaining metadata and all the items of leaf nodes. The search layer is located on the critical path and supposed to be accessed frequently, so it is placed in DRAM for performance. The data layer occupies the majority space of the entire tree and requires surviving system crashes, so it is placed in PM for persistence. The volatile metadata of a leaf node consists of the version, the parent pointer, the sibling pointer, and a *PM Ptr* that points to the head subnode. There are also multiple bitmaps and fingerprint arrays in the leaf

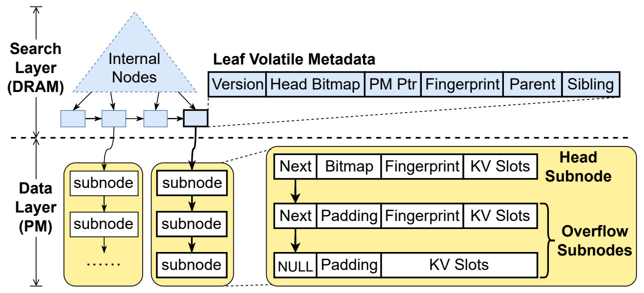

Fig. 2. The overall architecture of SSTree.

node, which we present in Section III-B. The remaining space is occupied by KV slots or necessary padding.

B. *Cost-efficient Leaf Node Expansion*

SSTree proposes a *cost-efficient leaf node expansion scheme* that allocates subnodes to delay the node split.

Allocate Subnode with a Single Flush. All insertions to the leaf node initially fill the free slots in the head subnode. After all the slots in the head subnode are occupied, SSTree does not perform the traditional node split. Instead, the target keys will be inserted into a newly allocated subnode. After that, *head replacement* minimize the extra flushes for inserting into an overflow leaf, that's, replacing the original head subnode with the new subnode. Figure 3 shows an example of head replacement in SSTree. We denote the original head subnode as *Node A*. The thread is inserting an item with key 9, but *Node A* has no empty slot. SSTree first allocates a new subnode, which we denote as *Node B*. Then, SSTree writes the key into *Node B* and updates its bitmap. After that, SSTree saves the address of *Node A* into the next pointer of *Node B*. Finally, SSTree updates the DRAM meta. The PM pointer now points to *Node B*. The head bitmap indicates the free slots in *Node B*. At this time, the leaf node contains *Node B* as the head subnode and *Node A* as the overflow subnode. The head replacement only requires a single flush because all the changes are located in the same cache line. The traditional node split will not be performed until the number of overflow subnodes reaches a certain threshold (which we discuss in Section IV-D). Thus, the subnode allocation and head replacement are more lightweight than a node split. It enables SSTree to reduce the total number of SMOs and improves the overall insertion throughput.

Detect Invalid Subnodes with Dedicated Free Tag. The search layer will be completely lost after a system crash. All the head subnodes on the PM must be located before rebuilding the search layer. The sibling pointers in SSTree are not persisted for better performance. Thus, SSTree uses the next pointer with all bits as 1 as a *free tag*. A subnode that starts with a free tag will not be considered allocated. During the head replacement, SSTree executes a fence before changing the next pointer from the free tag to the valid address. If this step fails, the new subnode will still not be considered valid. During the system restart, SSTree checks every subnode in the available PM region and discards those with free tags.

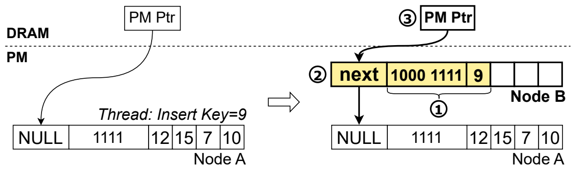

Fig. 3. Procedure of leaf node expansion with a single flush.

Among the remaining valid subnodes, the head subnode will not be pointed to by any next pointer. Starting from the head subnode, SSTree chases the next pointer to obtain the overflow subnodes for each leaf node. When initializing the available PM space, SSTree fills all the next pointers of each subnode with the free tag. Such initialization is executed only once, so it does not incur runtime overhead.

Accelerate the search with the delegated fingerprints. Unsorted leaf nodes can no longer use the traditional binary search algorithm and thus have to be searched linearly. As more overflow subnodes are added, the search spends more time traversing each overflow subnode. This may hurt the reading performance. Thus, SSTree proposes delegated fingerprints. Each subnode stores the fingerprint array of its successor. There are several reasons: *1)* Before accessing a subnode, SSTree must acquire its pointer located in the previous subnode, so the first cache line of each subnode is often fetched mandatory. *2)* Dispersing the fingerprints over the subnodes instead of gathering them in one makes room for the KV slots. *3)* Putting all the fingerprints in DRAM causes significant DRAM space consumption. The size of the fingerprint for each item is 1 byte. The head subnode is expected to be accessed most frequently, so its fingerprints and a volatile copy of its bitmap are placed in DRAM. SSTree uses the volatile metadata to filter the negative operation as much as possible. The highest bit of the bitmap copy indicates if the current leaf has at least 2 subnodes. If it is unset and the fingerprint of the target key does not match any of the used slots, then the search of the current leaf can be skipped.

C. *Proactive Subnode Compaction*

For unsorted B$^+$-Tree leaf, the valid keys will gradually scatter over the slots, resulting in poor locality and low bandwidth utilization. Besides, if subsequent insertions do not quickly fill the empty slots in the head subnode, search performance will be severely compromised. Therefore, SSTree introduces an *out-of-place item movement policy* to compact the keys. SSTree also proposes a *hitchhiked subnode reclamation scheme* to further reduce space consumption.

Out-of-place Item Movement. As shown in Figure 4, when updating the target key in an overflow subnode, SSTree first checks whether there is an empty slot in the head subnode. If so, SSTree will write the new item to the head subnode. Then, SSTree flips the bitmap in an atomic write to indicate that the slot in the head subnode is occupied and that the original slot is freed. In this way, SSTree gradually moves the items to the head subnode. Such item movement enhances data locality and thus reduces the execution time of subsequent operations.

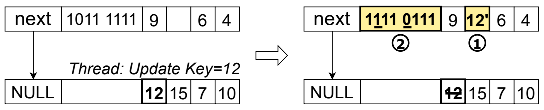

Fig. 4. Procedure of proactive subnode compaction by item movement.

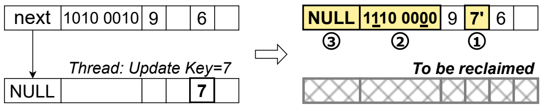

Fig. 5. Procedure of hitchhiked reclamation of overflow subnode.

An update with item movement behaves similarly to insertion. So it usually requires 2 flushes to persist data, one for the item and the other for the bitmap. But traditional persistent B^+-Tree only needs one flush to modify the value. Therefore, the cost of some updates is potentially higher. However, as the keys become more compact, subsequent operations are more likely to hit the head subnode. Thus the impact of item movement can be amortized. In addition, SSTree further improves the efficiency by moving multiple keys at once. If multiple items are in the same cache line as the target key, SSTree will try to move all of them.

Hitchhiked Subnode Reclamation. As the items move to the head subnode, the overflow subnodes become underutilized, and the last few ones will be empty at some point. If the current update removes the last key in an overflow subnode, then SSTree will perform space reclamation during the item movement, as shown in Figure 5. SSTree changes the next pointer in the previous subnode to *NULL*. At this point, the empty subnode can be reclaimed by the PM allocator. Similarly, hitchhiked subnode reclamation can be triggered by deleting the last key in an overflow subnode. The reclamation improves the space utilization of the leaf node.

Log-free Crash Consistency. The B^+-Tree and the PM allocator are two separate modules. The modifications are non-atomic during the subnode reclamation. Without careful design, the index will lose the address of PM regions or save a pointer of an invalid subnode. Here we specify a detailed scheme to guarantee the crash consistency for hitchhiked subnode reclamation. SSTree also reclaims empty overflow subnodes during deletion, but the procedure is similar.

Suppose a leaf node with at least 2 overflow subnodes. The next pointer will be modified, and the metadata bitmap is not in the same cache line. In this case, the specific process of SSTree reclaiming an empty overflow subnode is ❶ Set the free tag for the overflow subnode to be reclaimed, and write the moved items. Then, 2 flushes were used to persist these data to PM and insert 1 fence. ❷ Flip the bitmap in the head subnode. Then, execute 1 flush and 1 fence to complete the item movement. ❸ Set the next pointer of the previous subnode to *NULL*. If a system crash occurs, the different scenarios and how SSTree deals with them are as follows: 1) If a system crash occurs before step ❷, but the free tag is not written, it is treated as no change has been performed. 2) If the system crash occurs before step ❷, and the free tag has been written, SSTree will find that the next pointer points to a subnode with the free tag, but the bitmap indicates there are valid keys inside. This indicates that the subnode's recycling operation was unexpectedly interrupted, and the bitmap indicates that the data in the subnode has not yet completed movement. In this case, SSTree will retry step ❶-❸. When retrying ❶, SSTree writes the unchanged item instead of the updated one. Therefore, the recovered state is equivalent to rolling back the update, but the leaf node becomes more compact. 3) If the system crash occurs after step ❷ but before step ❸ is completed, SSTree will still detect the inconsistent next pointer, but the bitmap indicates that the subnode is empty. So SSTree retries step ❸ to finish the reclamation.

In summary, SSTree requires at most 4 flushes and 2 fences to guarantee the crash consistency for subnode reclamation. When there is only 1 overflow subnode, the bitmap and next pointer are located in the same cache line of the head subnode. So step ❷-❸ can be combined into a single flush. In this case, the cost is reduced to 3 flushes and 2 fences. If the next pointer of the reclaimed subnode is not *NULL*, SSTree will lose the pointer of its successor once a system crash occurs after step ❶. Thus, SSTree only reclaims the final overflow subnode from the end of the leaf node.

IV. Evaluation

A. Experimental Setup

Our experiments are performed on a Ubuntu 20.04 server with Linux kernel 5.15, equipped with two Intel Xeon Gold 6240 CPUs clocked at 2.60GHz. Each CPU has 18 cores (36 threads via hyper-threading). The system has a shared 49.5MB L3 Cache, 192GB of 2666MHz DDR4 DRAM (6×32GB), and 512GB of Optane DCPMM (4×128GB) configured in the App Direct mode. To avoid NUMA effects, threads are pinned to the local NUMA (except for Section IV-E). All the code is compiled using GCC 9.4 with all optimizations enabled. We use PiBench [22] to evaluate all the indexes. The keys and values are all fixed 8-byte. We set the internal node to 512B and verified that it exhibits the optimal balance between DRAM consumption and performance. By default, the subnode is set to 256B containing 14/15 slots, and the maximum number of subnodes within a leaf node is set to 2. We discuss the parameter sensitivity in Section IV-D. We compare SSTree with several state-of-the-art B^+-Tree indexes including uTree [9], PACTree [14] and LB^+-Tree [8]. The original open-source version of LB^+-Tree causes infinite HTM abortions, so we fall back on the optimistic locking for concurrency control. We skipped the multi-threaded deletion of uTree because it causes deadlock.

B. Overall Performance

In this section, we evaluate the throughput and tail latency. In the load phase, we initialize an empty tree and insert 100 million keys. We perform 200 million test operations on update, search, and insert workloads and 40 million test operations on delete workloads. For the skewed workload, 80% operations target 20% of the keys.

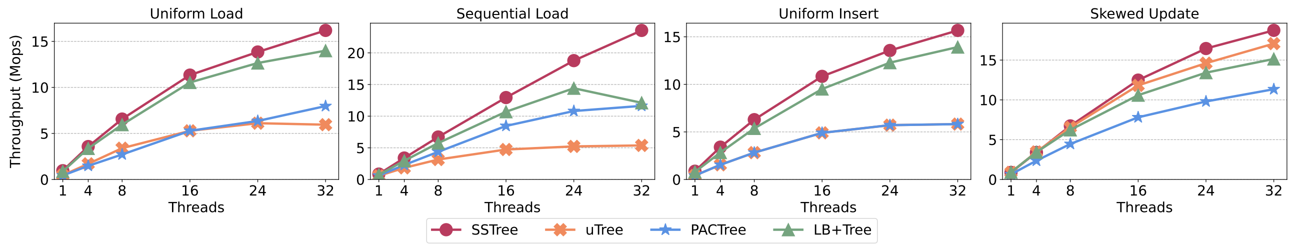

Fig. 6. Performance comparison of persistent B$^+$-Tree indexes under different write workloads.

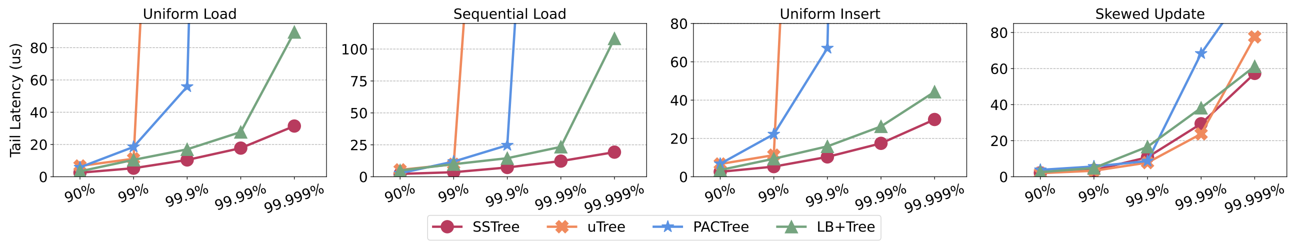

Fig. 7. Tail latency under different write workloads.

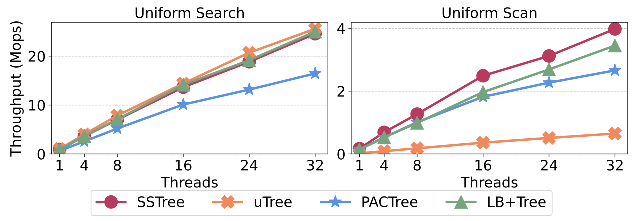

Fig. 8. Performance comparison of persistent B$^+$-Tree indexes under uniform read workloads.

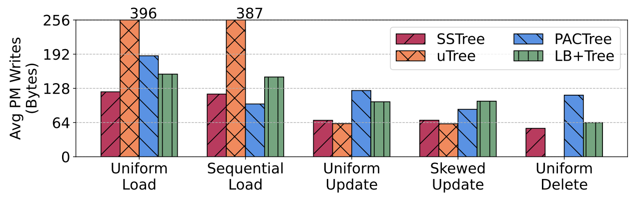

Fig. 9. Average PM writes per operation.

Write Performance. Figure 6 shows the throughput of uniform and skewed/sequential write operation. The throughput of SSTree outperforms other indexes by 1.2×-2.7× for uniform load and insert workload. SSTree also achieves 2×-4.4× performance gain for the sequential load. This is due to optimizing the leaf node, which replaces node splits with cost-efficient leaf node expansion. For skewed update, the throughput of SSTree is 1.14×/1.7×/1.6× higher than uTree/PACTree/LB$^+$-Tree. This is because SSTree introduces item movement to make the keys inside leaf nodes more compact, which increases the cache efficiency.

Figure 7 shows the tail latency distribution under different write workloads. SSTree exhibits the lowest tail latency for insertion. SSTree outperforms other indexes by 1.9×-3.5× in p99 latency and up to 62× in p99.999 latency. For the uniform insert workload, SSTree outperforms other indexes by up to 57× in p99.9 tail latency. The tail latency improvement compared with LB$^+$-Tree becomes less obvious under the uniform insert workload. But SSTree still achieves 1.8× better performance. For skewed updates, the p99.9 latency of SSTree is slightly higher than uTree and LB$^+$-Tree because SSTree introduces item movement with reclamation. However, SSTree still exhibits the lowest p99.999 latency.

Read Performance. Figure 8 shows the throughput under uniform read workloads. SSTree exhibits comparable point search performance to uTree and LB$^+$-Tree, with less than 4% throughput loss. Our evaluation also shows that they exhibit the same tail latency. Thus, SSTree mitigates the search performance degradation of accessing multiple subnodes to find the target key. For range scan, SSTree outperforms other indexes by 1.1×-6×. uTree performs poorly because its item list suffers from poor locality and incurs lots of random accesses on PM.

C. Space Efficiency

In this section, we evaluate the optimization of SSTree for memory bandwidth and space consumption.

PM Bandwidth Consumption. We measure the average PM bandwidth used per operation. All the tests are performed under 32 threads. The workloads are configured as the same as Section IV-B. Figure 9 shows the average PM writes of each write operation. For insertion, SSTree flushes 2 cache lines on average. This proves the effectiveness of the overflow subnode in delaying SMO. uTree exhibits high PM writes due to contention of the key-value list. For other workloads, the average PM writes for each operation in SSTree are close to 1 cache line. Thus, the item movement overhead can be amortized even if it requires more than 1 flush. The average PM writes of deletion in SSTree are slightly lower than 1 cache line. This is because part of deletions are directly filtered by the DRAM metadata of SSTree due to attempts to delete non-existent keys.

Figure 10 shows the average PM reads per operation. SSTree generally reads 2 cache lines from PM, one for the

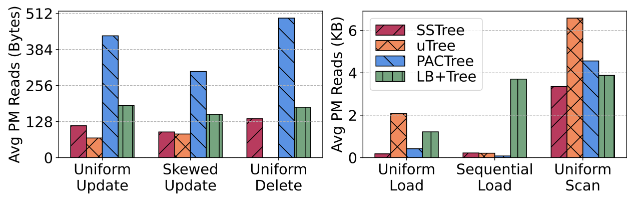

Fig. 10. Average PM reads per operation.

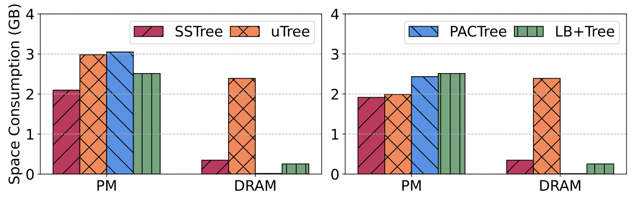

Fig. 11. Memory space consumption of PM and DRAM after load phase (Left) and mixed operations (Right).

item and another for the metadata. uTree maintains a list node that fits half a cache line for each item. It also filters many PM metadata accesses by aggressively consuming DRAM, exhibiting the lowest PM accesses for updates.

Memory Space Consumption. We compare the DRAM and PM consumption of different persistent B$^+$-Tree indexes. We first load 100 million (1.6GB) uniform keys. Then we perform 200 million mixed operations, with 20% deletion and 80% update. Figure 11 shows the memory consumption of each B$^+$-Tree index after the load phase and mixed operations. After the load phase, the PM consumption of SSTree, uTree, PACTree, and LB$^+$-Tree is 2.1GB, 2.98GB, 3.05GB, and 2.51GB respectively. After the mixed operations, the PM consumption of SSTree, PACTree, and LB$^+$-Tree is 1.92GB, 2.43GB, and 2.51 GB, respectively. After the mixed operations, SSTree's PM consumption is reduced by 8.6%. This is because SSTree performs item movement and reclaims empty overflow subnodes. LB$^+$-Tree only reclaims empty leaf nodes, so the space consumption has almost no change. Therefore, compared with other B$^+$-Tree indexes, SSTree achieves lower PM consumption and higher leaf node space utilization in the long run. The DRAM consumed by SSTree, uTree, PACTree, and LB$^+$-Tree is 356MB, 2447MB, 0.02MB, and 259MB, respectively, after the load phase. After the mixing operations, the DRAM space consumption of all indexes except PACTree has almost no change because they haven't implemented node merge yet. SSTree also places some critical metadata on DRAM, increasing the DRAM space consumption. This is a trade-off between performance and space utilization. However, SSTree does not require any special concurrency control policy. Therefore, such designs can be easily integrated into traditional B$^+$-Tree with node rebalance and merge.

D. Techniques and Sensitivity Study

In this section, we evaluate the sensitivity of methods in SSTree. All the tests are performed under 32 threads.

Sensitivity to the Maximum Number of subnodes. Table I shows the performance impact of the maximum number of

TABLE I
THROUGHPUT (MOPS) VARYING THE MAXIMUM NUMBER OF SUBNODES.

	N=2	N=3	N=4
Sequential Load	22.99	25.21 (+9%)	25.54 (+11%)
Skewed Search	26.74	24.75 (-7%)	22.6 (-15%)

TABLE II
THROUGHPUT (MOPS) W/ AND W/O ITEM MOVEMENT.

	w/o Item Movement	w/ Item Movement
Skewed updates (1t)	0.74	0.84 (+13%)
Skewed updates (32t)	17.03	18.73 (+10%)

overflow subnodes in a leaf node (which we denote as N in the table). Sequential insertion gains up to 1.1× performance boost because more SMOs are delayed. Search performance gradually decreases as more overflow subnodes are added due to pointer chasing and the increasing false positive rate of the fingerprints. As the maximum number of subnodes increases from 2 to 4, the search performance reduces by 15%.

Effectiveness of Proactive Subnode Compaction. Table II shows the throughput comparison of SSTree with and without element movement enabled under the skewed update workload. SSTree gains 1.1×-1.13× performance by introducing out-of-place item movement.

Recovery Overhead. We load SSTree with 100 million keys and measure the total recovery time. As the number of threads increases from 1 to 32, the total recovery time reduces from 2.7 seconds to 0.44 seconds.

E. Portability

We evaluate the effect of SSTree designs by varying the characteristics of byte-addressable devices. We sequentially load 100 million keys under 32 threads. We force all threads to run on the local socket. We replace the original local PM pool with different memory types from the local or remote socket. All flushes and fences introduced for persistence are removed. PACTree does not natively support DRAM-only mode so we skipped it. The results are shown in Figure 12. SSTree exhibits the best performance among all indexes, with 1.1×-13× speed up on DRAM and 1.3×-3.9× speed up on remote PM. Thus, the designs of SSTree can reduce the SMO overhead on other byte-addressable platforms.

V. RELATED WORK

Persistent trie indexes. Persistent trie indexes [23]–[25] exhibit better performance under uniform workloads. They are also more suitable for variable-length keys. However, compared with B$^+$-Tree, radix trees typically incur significantly higher space consumption. They also exhibit poor locality and much lower performance than B$^+$-Tree under scan workload.

Persistent B$^+$-Tree indexes. There have been various designs to optimize B$^+$-Tree for PM. FAST&FAIR [12] proposes a log-free scheme that detects inconsistency by checking the duplication of adjacent items. TLBTree [13] stores all internal nodes in a contiguous array and replaces pointer-chasing with array index calculation. These indexes only use PM so their performance and scalability are still limited.

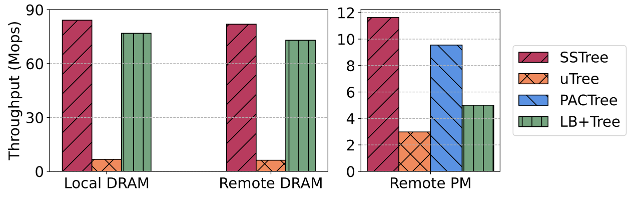

Fig. 12. Performance on other types of byte-addressable devices.

Hybrid indexes. Various B$^+$-Tree indexes exploits DRAM to compensate for the limitations of PM. NV-Tree [10] proposes selective persistence which places internal nodes in DRAM. FPTree [11] introduces fingerprints for leaf nodes. uTree [9] stores all the keys in DRAM to minimize the tail latency. NBTree [16] consumes space more aggressively by not reusing the deleted slots to support lock-free operations. Regarding SMO, these indexes lack effective optimization, rely heavily on HTM, or severely waste memory space. DPTree [21] employs bloom filters and multiple buffer trees on DRAM to filter out PM accesses. ROWE-Tree [26] uses a hash table to cache hot items and absorb reads. These indexes maintain extra indexes apart from the base tree to reduce the interference between read and write. However, they incur significantly higher space consumption. Furthermore, their designs are orthogonal to us.

VI. Conclusion

Existing persistent B$^+$-Tree indexes suffer from high structural modification operation (SMO) overhead. In this paper, we design SSTree, a high-performance B$^+$-Tree index optimized for SMO. SSTree extends the traditional leaf node with a list of *subnodes* and delays the high-overhead node split by cost-efficient leaf node expansion. SSTree introduces delegated fingerprints to accelerate the search. SSTree proposes out-of-place item movement with hitchhiked subnode reclamation to increase space utilization. SSTree also employs a detailed failure-atomic policy to minimize extra overhead for crash consistency guarantee. Our evaluation shows that SSTree outperforms state-of-the-art B$^+$-Tree indexes by up to 4.38× in write throughput and achieves up to 62× lower tail latency.

References

[1] D. Narayanan and O. Hodson, "Whole-system persistence," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, 2012, pp. 401–410.

[2] S. Gugnani, A. Kashyap, and X. Lu, "Understanding the idiosyncrasies of real persistent memory," *Proceedings of the VLDB Endowment*, vol. 14, no. 4, pp. 626–639, 2020.

[3] Samsung Semiconductor USA, "Samsung CXL Solutions – CMM-H," https://semiconductor.samsung.com/us/news-events/tech-blog/samsung-cxl-solutions-cmm-h, [Accessed 09-05-2024].

[4] Y. Fridman, S. Mutalik Desai, N. Singh, T. Willhalm, and G. Oren, "Cxl memory as persistent memory for disaggregated hpc: A practical approach," in *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, 2023, pp. 983–994.

[5] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, C. Song, J. Huang, H. Ji, S. Agarwal, J. Lou, I. Jeong *et al.*, "Demystifying cxl memory with genuine cxl-ready systems and devices," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 105–121.

[6] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *Proceedings of the 7th ACM european conference on Computer Systems*, 2012, pp. 183–196.

[7] B. Lepers, O. Balmau, K. Gupta, and W. Zwaenepoel, "Kvell: the design and implementation of a fast persistent key-value store," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 447–461.

[8] J. Liu, S. Chen, and L. Wang, "Lb+ trees: Optimizing persistent index performance on 3dxpoint memory," *Proceedings of the VLDB Endowment*, vol. 13, no. 7, pp. 1078–1090, 2020.

[9] Y. Chen, Y. Lu, K. Fang, Q. Wang, and J. Shu, "utree: a persistent b+-tree with low tail latency," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2634–2648, 2020.

[10] 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 *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015, pp. 167–181.

[11] 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 *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 371–386.

[12] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in {Byte-Addressable} persistent {B+-Tree}," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018, pp. 187–200.

[13] Y. Luo, P. Jin, Z. Zhang, J. Zhang, B. Cheng, and Q. Zhang, "Two birds with one stone: Boosting both search and write performance for tree indices on persistent memory," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 5s, pp. 1–25, 2021.

[14] W.-H. Kim, R. M. Krishnan, X. Fu, S. Kashyap, and C. Min, "Pactree: A high performance persistent range index using pac guidelines," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 424–439.

[15] M. Liu, J. Xing, K. Chen, and Y. Wu, "Building scalable nvm-based b+ tree with htm," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–10.

[16] B. Zhang, S. Zheng, Z. Qi, and L. Huang, "Nbtree: a lock-free pm-friendly persistent b+-tree for eadr-enabled pm systems," *Proceedings of the VLDB Endowment*, vol. 15, no. 6, pp. 1187–1200, 2022.

[17] Intel, "Transactional Synchronization with Intel® Core™ 4th Generation processor," https://www.intel.com/content/www/us/en/developer/articles/community/transactional-synchronization-in-haswell.html, [Accessed 21-05-2024].

[18] Wikipedia, "Hash table," https://en.wikipedia.org/wiki/Hash_table, [Accessed 09-05-2024].

[19] Intel, "eADR: New Opportunities for Persistent Memory Applications," https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html, [Accessed 21-05-2024].

[20] L. Lersch, X. Hao, I. Oukid, T. Wang, and T. Willhalm, "Evaluating persistent memory range indexes," *Proceedings of the VLDB Endowment*, vol. 13, no. 4, pp. 574–587, 2019.

[21] X. Zhou, L. Shou, K. Chen, W. Hu, and G. Chen, "Dptree: differential indexing for persistent memory," *Proceedings of the VLDB Endowment*, vol. 13, no. 4, pp. 421–434, 2019.

[22] Y. He, D. Lu, K. Huang, and T. Wang, "Evaluating persistent memory range indexes: part two," *Proceedings of the VLDB Endowment*, vol. 15, no. 11, pp. 2477–2490, 2022.

[23] R. Binna, E. Zangerle, M. Pichl, G. Specht, and V. Leis, "Hot: A height optimized trie index for main-memory database systems," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 521–534.

[24] S. Ma, K. Chen, S. Chen, M. Liu, J. Zhu, H. Kang, and Y. Wu, "{ROART}: Range-query optimized persistent {ART}," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021, pp. 1–16.

[25] J. Hu, Z. Wei, J. Chen, and D. Feng, "Rwort: A read and write optimized radix tree for persistent memory," in *2023 IEEE 41st International Conference on Computer Design (ICCD)*. IEEE, 2023, pp. 194–197.

[26] X. Zou, F. Wang, D. Feng, T. Guan, and N. Su, "Rowe-tree: A read-optimized and write-efficient b+-tree for persistent memory," in *Proceedings of the 51st International Conference on Parallel Processing*, 2022, pp. 1–11.