PHAST: Hierarchical Concurrent Log-Free Skip List for Persistent Memory

Zhenxin Li, Bing Jiao, Shuibing He, *Member, IEEE*, and Weikuan Yu

Abstract—Skip list (skiplist) is a competitive index structure that offers superior concurrency and excellent performance but with high memory overhead and low access locality. Emerging persistent memory (PM) technologies present an opportunity to mitigate the capacity constraint of DRAM. However, data consistency on PM typically results in excessive write overhead. In addition, fast concurrent access to an index is critical to the throughput on high-end contemporary computer systems. In this article, we propose a Partitioned HierArchical SkiplisT called PHAST, which can simultaneously reduce the skiplist height and improve its access locality, through its hierarchy of component structures, while enabling fast parallel recovery in case of failure. To ensure high concurrency and fast data consistency, we also have developed writelock-free concurrent insert and log-free atomic split. Furthermore, we have developed a durable lock-free concurrent search that can discern transient structural inconsistencies and deliver highly concurrent read operations. We have conducted an extensive evaluation of PHAST compared to state-of-the-art studies such as NV-Skiplist, wB+-Tree, FPTree, and FAST-FAIR. Our evaluation results show PHAST outperforms other indexing structures by up to 4.05× and 2.87× in single-threaded inserts and searches, and 1.56× and 2.62× in concurrent inserts and searches.

Index Terms—Skiplist, persistent memory, concurrency

1 Introduction

Efficient indexing technologies are essential to the organization and operation of large-scale storage systems for them to meet the critical needs of search engines, social networks, and online media stores [1], [2]. For example, log-structured merge trees [3] are popularly leveraged for key-value stores because of their benefits to block-oriented disk devices. In addition, B+ tree and its derivatives [4] have been very popular for disk-based file and database systems [5], [6] because of their strengths in achieving high fan-out, balanced tree height, and efficient performance with minimized random disk seeks. On the other hand, indexing structures that can cause more random accesses such as hash tables [7] and skiplists [8] are typically employed in the volatile cache and memory systems that are smaller in volume but support fast random accesses.

Emerging persistent memory (PM) technologies such as Phase Change Memory (PCM) [9], Memristor [10], and 3D XPoint [11] have greatly changed the landscape of memory and storage systems. They provide not only high performance close to DRAM, but also data persistence with high-density and low power consumption [12]. As a result, PM is popularly used in various real-world applications, such as Redis [13], Memcached [14], and AI-based decision systems [15]. Many research studies have been undertaken to take advantage of persistent memory technologies in data-intensive indexing structures, such as B+ trees [16], [17], [18], [19], [20], hash tables [19], [21].

Compared to B+ trees and hash tables, the skiplist is a strong contender for fast indexing. As a probabilistic linked-list based indexing structure, skiplist achieves its excellent performance and superior concurrency through additional layers of pointers without incurring frequent balancing operations. It is widely used in large-scale databases [22] and key-value stores [1], [23], [24]. However, the additional layers of pointers required by skiplists lead to high memory overhead and low access locality.

PM technologies present an opportunity for skiplist with their large capacity and DRAM-like performance. However, due to the hardware characteristics of the PM device and the structural features of the skiplist, directly integrating the skiplist and the PM device would be inefficient. There are several challenges in designing a PM-based skiplist.

A major challenge is that the skiplist may suffer from high random access overhead in PM. The traditional skiplist stores only one item in each node. This design yields a low data access locality and causes a large number of small-sized random data accesses during the indexing of a given item. However, the physical media access granularity in PM is 256 bytes, implying small-sized random accesses can lead to poor performance because of the read or write amplification [25]. To address this issue, this paper proposes a Partitioned HierArchical skiplisT, PHAST, for fast indexing of skiplist in PM (Section 3). The idea of PHAST is to place the

- *Zhenxin Li and Shuibing He are with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China, and also with Zhejiang Laboratory, Hangzhou 311100, China. E-mail: {zhenxin, heshuibing}@zju.edu.cn.*
- *Bing Jiao and Weikuan Yu are with the Department of Computer Science, Florida State University, Tallahassee, FL 32306 USA. E-mail: {jiao, yuw}@cs.fsu.edu.*

Manuscript received 22 Oct. 2021; revised 20 Apr. 2022; accepted 4 May 2022. Date of publication 10 May 2022; date of current version 23 Aug. 2022.
This work was supported in part by the National Key Research and Development Program of China under Grant 2021ZD0110700, in part by the National Science Foundation of China under Grant 62172361, in part by Zhejiang Lab Research Project under Grant 2020KC0AC01, and the Alibaba Innovative Research Project.
(Corresponding authors: Shuibing He and Weikuan Yu.)
Recommended for acceptance by H. Huang.
Digital Object Identifier no. 10.1109/TPDS.2022.3173707

1045-9219 © 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on September 23,2022 at 09:03:34 UTC from IEEE Xplore. Restrictions apply.

inner nodes in DRAM and the leaf node in PM to reduce the number of random accesses to PM. Besides, PHAST augments the access locality by grouping KV pairs into one node to further reduce random PM accesses.

While the grouping of items can improve access locality, it brings an additional concurrency control overhead compared to traditional skiplists because multiple threads will concurrently access the same node. One classic approach is to use a per-node lock to serialize concurrent operations, like in FAST&FAIR [26] and FPTree [18]. However, the coarse-grained lock mechanism introduces extra waiting overhead when concurrent threads update the same nodes. Multi-Version Concurrency Control (MVCC) is another strategy for thread-safe concurrent indexing operations, but it is not practical for PM due to the high space consumption and garbage collection overhead. To address this challenge, PHAST introduces a relaxed RWLock-based concurrency control mechanism (Section 4), which supports writelock-free insertion and lock-free search to access the same nodes simultaneously without causing data inconsistency.

Furthermore, the skiplist suffers from high write overhead for data consistency in PM. Out-of-order writes to PM can cause data inconsistency when system crashes. To ensure data consistency, the persistent skiplist needs to use cache line flush (e.g., CLFLUSH, CLFLUSHOPT, CLWB) and memory fence (e.g., MFENCE, SFENCE) instructions to persist data to the PM. These instructions introduce additional write overhead. To tackle this issue, PHAST adopts writelock-free concurrent insert (Section 4.1) and log-free atomic in-place split (Section 4.2), which support write operations without resorting to costly data logging and copy-on-write (COW) strategies.

We have compared PHAST to recent indexing structures including NV-Skiplist [27], wB+-Tree [16], FPTree [18], and FAST-FAIR [26]. Our results show that PHAST has a better performance in both single-threaded and multi-threaded tests. PHAST outperforms other indexing structures by 1.33× to 4.05× in single-threaded inserts and 1.22× to 2.87× in single-threaded searches. For concurrency operations, PHAST outperforms FAST-FAIR by up to 1.56×, 2.62×, and 1.59× in inserts, searches, and mixed inserts/searches workloads.

2 Background and Motivation

2.1 Persistent Memory

Persistent memory is non-volatile and byte-addressable while providing the DRAM-like performance. With the release of Optane DCPMM [28], PM device is popularly used in various fields, such as file system [29], [30], transaction processing [31], [32], and indexing structures [20], [33]. Different from traditional disks, PM communicates with the memory controller in the cache line granularity (i.e., 64 bytes), while the physical media access granularity in PM is 256 bytes. Thus, the data transmission between CPU cache and PM will result in I/O amplification due to the mismatched sizes in different hardware layers. Besides, out-of-order writes to PM can cause data inconsistency when the system crashes. To ensure data consistency, memory write operations are typically followed by a cache line flush instruction to explicitly flush dirty cache lines and a memory fence instruction to avoid the instruction reordering.

TABLE 1
Bandwidth and Latency of DRAM and PM With 32-Thread Tests

	Bandwidth (GB/s)				Latency (ns)			
	Read		Write		Read		Write	
	Seq	Rnd	Seq	Rnd	Seq	Rnd	Seq	Rnd
DRAM	63.9	11.1	18.5	13.4	30	172	103	142
PM	11.0	2.9	2.3	0.8	172	635	847	2559

To show the performance difference between PM and DRAM, we evaluate their real performance with a micro benchmark using 32 threads and a 64-byte access size in our experimental platform (Section 5.1). As shown in Table 1, the maximal read bandwidth of PM is 11 GB/s and the maximal write bandwidth is 2.3 GB/s, which are 5× and 8× lower than that of DRAM. In addition, different from DRAM, PM suffers from about 3× higher read/write latency for random accesses than sequential accesses. As a result, it is critical for PM-based indexes to carefully consider the characteristics of PM.

2.2 Skiplist Overview

As shown in Fig. 1, a skiplist is a hierarchical data structure composed of multiple layers of linked lists. While the bottom layer is the largest and holds all items in the list, other layers are probabilistically constructed by adding additional forward pointers to an item. A node from one layer has a probability of p ($\frac{1}{2}$ in our design) to be chosen to be part of the adjacent higher layer. The height of a node is determined by the number of its forward pointers. Thus a skiplist of n items will consist of $log(n)$ layers, i.e., a height of $log(n)$, with an extra storage for $n/(1-p)$ forward pointers. Through these layers of forward pointers, a skiplist supports fast search performance while the bottom-level linked list allows fast concurrent insertion and modification without inflicting complex rebalancing operations like B-trees. Most common operations will have an average complexity of $\frac{1}{p}log(n)$, i.e., $O(log(n))$. A thorough treatment on skiplist is available in the literature [8].

2.3 Challenges for in-PM Skiplist

PM has its characteristics such as the requirement of persistent instructions, the access size of 256 bytes, and the favor of sequential accesses. However, the idiosyncrasy of skiplist, such as layers of links, individual nodes, and point chasing, can not fully match the features of PM. Therefore, directly integrating a skiplist and the PM faces the following challenges.

High Random Access Overhead. Recent works [25], [34] have revealed that PM has poor performance for accesses with low locality, i.e., random accesses. To verify this, we test the latency of PM with 32 threads under sequential and random access patterns at the request size of 64 bytes in our experimental platform. Fig. 2a shows that PM delivers about 3× higher latency for random accesses than sequential accesses for both reads and writes. As each node in traditional skiplists only has one item, a large skiplist requires multiple layers to hold all the items. The read and write operations have to travel many layers of forwarding pointers, resulting

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on September 23,2022 at 09:03:34 UTC from IEEE Xplore. Restrictions apply.

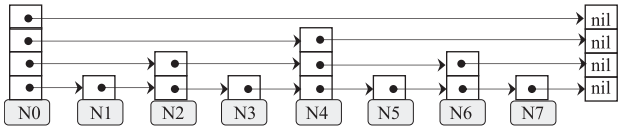


Fig. 1. Diagram for a generic skiplist with 8 nodes.

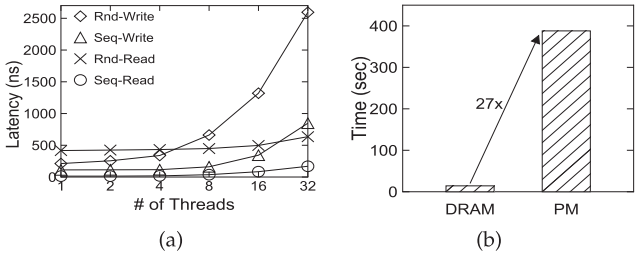


Fig. 2. (a) Latency with different access patterns in PM. (b) Write overhead for skiplist in DRAM and PM.

in low data access locality. Consequently, the PM-resident skiplist would suffer from a longer access latency than the DRAM-resident one [20], [35]. To address this issue, a common approach is grouping multiple items to an internal node [27], [36], similar to the height reduction method in various B+ trees. Inspired by this, PHAST proposes the partitioned hierarchical skiplist design (Section 3) and further solves the concurrency problem caused by grouping.

High Concurrency Control Overhead. While item grouping can improve locality, it leads to concurrency access contention because multiple threads may access the same node. Concurrency is important with ever-increasing core counts in contemporary server systems. However, most of the existing PM indexing study does not fully demonstrate the concurrency because of the complex programming for data consistency and persistence. For example, NV-Tree [17], wB+-Tree [16], and WORT [37] do not support concurrent accesses, while FAST&FAIR [26] and FPTree [18] deploy a per-node lock to serialize concurrent operations. However, the coarse-grained lock mechanism introduces extra waiting overhead when concurrent threads update the same node simultaneously. BzTree [38] proposes a latch-free indexing using a Multi-Words Compare And Swap (MWCAS) instruction [39], but it needs additional resources to allocate and reclaim MWCAS descriptors. Multi-Version Concurrency Control (MVCC) is another strategy for thread-safe concurrent indexing operations. However, MVCC is not practical for PM due to the high space consumption and garbage collection overhead. To keep the superior concurrency of traditional skiplists, PHAST proposes a relaxed rwlock-based concurrency control mechanism (Section 4).

High Write Overhead for Data Consistency in PM. PM has a lower write bandwidth than DRAM and requires the explicitly calling of flush and fence instructions to keep data persistency and consistency. This introduces additional high write overhead [16], [17]. Many B+ tree based indexing structures [16], [17], [18], [26], [35] have strived to improve the throughput by reducing the number of PM writes and flushes. For skiplist, however, the insertion of a node will trigger the update of multiple nodes across several layers. These updates introduce a large number of writes and flushes, which degrade the performance. To verify this, we insert 100M KVs with a single thread to an in-DRAM skiplist and an in-PM skiplist with flush and fence instructions. After warming up using 50M KVs, we insert the remaining 50M KVs and record the write time (i.e., create a new node and install it to layers of linked lists) excluding the locating time. As shown in Fig. 2b, the in-PM skiplist delivers a 27× higher write time than the in-DRAM skiplist, indicating the urgency to reduce the write cost of in-PM skiplist. Besides, the commonly used approaches to maintain data consistency, such as data logging or copy-on-write (COW), also bring redundant writes for PM-based indexes, leading to high write overhead. To reduce the write overhead, PHAST proposes writelock-free concurrent insert (Section 4.1) and log-free atomic in-place split (Section 4.2) techniques.

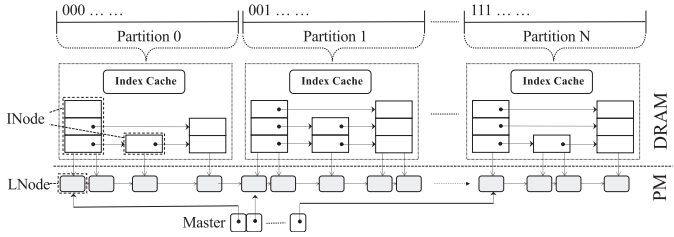


Fig. 3. PHAST: Partitioned hierarchical skiplist with selective persistence.

3 Partitioned Hierarchical Skiplist for Low Height and High Locality

We design PHAST to exploit the capacity advantage of PM while mitigating the drawbacks of traditional skiplists for highly concurrent indexing operations. We advocate two principles: (1) lowering the height of skiplists for fewer random accesses when travelling the layers and fewer pointers to be updated when installing a new node. (2) augmenting the locality by caching frequent keys and grouping KV pairs wherever possible in both DRAM and PM.

Fig. 3 provides an overview of PHAST organization. A PHAST is composed of a hierarchy of three-level components. At the top level, a PHAST is divided into a set of partitions based on the prefix bits of keys. Next, each partition is an augmented skiplist of internal nodes (INodes). Third, each INode in turn contains a set of pointers to the leaf nodes (LNodes) in PM. A LNode is the basic unit of KV pairs. Since DRAM has lower latency and higher throughput than PM, we adopt the selective persistence strategy [17], [18] to improve the throughput of PHAST: all LNodes reside in PM while partitions and INodes are located in DRAM. Partitions and INodes are volatile and can be reconstructed from the LNodes in PM upon a failure recovery.

3.1 Prefix-Based Partitioning

PHAST adopts a prefix-based partitioning strategy to divide the total range of keys. Fig. 3 shows an example of partitioning the keys into N ($N = 8$) partitions based on the first three bits. This allows a quick decision on the targeted partition for a key without key comparisons and skiplist traversals. With partitioning, the number of nodes in each partition is only a part of the total nodes, so the height of partitions is usually lower than that of the non-partitioned skiplist. Assuming a uniform distribution of keys and a probability of $\frac{1}{2}$ to increase the height of a skiplist node,

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on September 23,2022 at 09:03:34 UTC from IEEE Xplore. Restrictions apply.

using k prefix bits for partitioning can reduce the height of skiplist by k. This simple prefix-based partitioning embraces our first principle to achieve lower height, and helps avoid frequent traversal of many layers of pointers in a monolithic skiplist. However, for a workload with a skewed key distribution, it can result in uneven partitions, and diminished benefit when some partitions are much higher than others (Section 5.4). Thus, the actual number of partitions can be a tradeoff depending on the actual workload. Also note that we are not limited to k prefix bits for PHAST partitioning. These partitioning bits can be a set of consecutive bits anywhere based on the key space and the workload.

3.2 Index Cache

Common indexing data structures, such as B+ trees and linked lists contain internal nodes that are discontiguous in memory. Traversing such structures renders the CPU cache working to be ineffective. Particularly, for skiplists, layers of pointers and a large number of low-level nodes form a working set much bigger than the cache size. They can cause frequent eviction of the recently-accessed search-path pointers and the much-needed high-level nodes out of the memory cache. As a reflection of our second design principle for high locality, we introduce an in-DRAM auxiliary cache in each partition called *Index Cache*, which maps maximal keys (mIKeys in Section 3.3) of a specified skiplist layer to their corresponding INodes. This cache helps to mitigate the frequent misses for keys from high-level nodes and improve memory access performance. All keys in *Index Cache* are organized in a contiguous array and sorted in ascending order to allow fast $O(log(n))$ lookup. Considering an oversized cache could cause large DRAM consumption and diminished performance benefit, we use a threshold to control the size of Index Cache. The threshold can be determined based on the platform parameters (i.e., the available DRAM capacity) and the workload. In our current design, we empirically set the threshold as 1MB to achieve a moderate balance between the performance benefit and its space cost. Besides, PHAST can determine the cached layer based on the threshold at runtime. If the cache size of the current layer exceeds the threshold, PHAST will select the above layer to be cached and replace the old one.

3.3 INode and LNode

In a traditional skiplist, each node stores only one KV pair. In view of the aforementioned drawbacks of skiplist for large-scale indexing, we deem that, besides partitioning the skiplist, it is necessary to exploit more potential from the node structure to further reduce the height of the skiplist. On the other hand, we need to retain the advantage of the skiplist for concurrent accesses and avoid inflicting additional writing overhead for failure-atomic data consistency in PM. Thus, we organize each partition into two more levels of units: the individual INodes and the LNodes, as shown in Fig. 4. Each LNode contains up to 56 KV pairs to reduce the skiplist height and save the data persistency and consistency overhead (Section 3.3.2). Each INode manages a large number of LNodes. A smaller INode reduces the number of key comparisons and multi-threaded conflicts within an INode, but it increases the height of PHAST and the

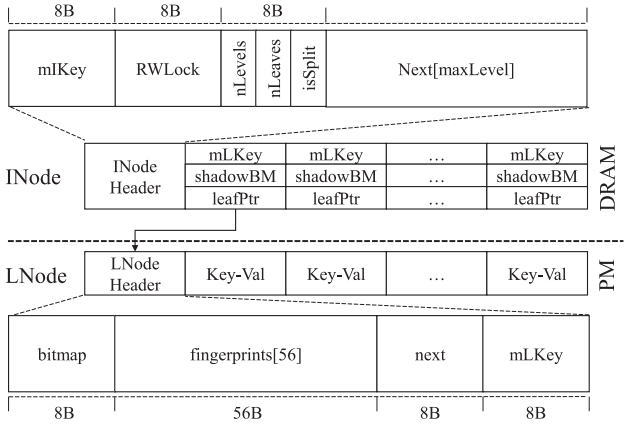


Fig. 4. INode and LNode in PHAST. For clarity, components are not drawn to the scale of their sizes.

number of INode splits. Based on our experimental analysis (Section 5.4), we currently empirically configure each INode to contain 128 LNodes. Together with these components, each INode manages up to 7168 KV pairs, reducing the height of the skiplist by nearly 13 ($\log_2 7168$) levels. Compared to NV-Skiplist [27] that groups multiple (64) key-value pairs into a single node and PSL [36] that uses B-Tree like nodes to manage internal nodes, these components can further reduce the skiplist height. These components work in concert to support highly concurrent operations at low write overhead, as will be elaborated in Sections 4.1 and 4.3.

3.3.1 INode

Each INode consists of a header, one array of keys, one array of shadow bitmap (shadowBM) and another array of pointers to its constituent LNodes on PM. The header consists of six main fields: the maximal key within the INode (mIKey), a RWlock (read/write lock) for concurrency control, the number of levels (nLevels), i.e., the height of the node, the number of LNodes (nLeaves), the flag indicating whether the node (INode or LNode) is splitting (isSplit), and an array of pointers to the next node for each level. The array of keys contains the maximal key from each LNode (mLKey). The shadowBM is used to indicate the empty positions in a LNode for concurrent inserts (Section 4.1). The beginning location of each LNode is recorded as a leaf pointer (leafPtr) in the third array. The header and the three arrays are all 64B aligned for efficient cache line accesses.

3.3.2 LNode

As shown by the bottom half of Fig. 4, each LNode contains a header followed by a group of KV pairs. The LNode header consists of a bitmap indicating the valid KV pairs in the LNode, an array of fingerprints (each 1B), a pointer that points to the next LNode on PM, and the maximal key within the LNode (mLKey). The LNode organization is similar to the leaf node in FPTree [18]. We make each LNode contain 56 KV pairs so that its bitmap (8 bytes) and fingerprints (56 bytes) can be persisted with one cacheline (64 bytes) instruction, reducing the data persistence overhead. Furthermore, the bitmap can be updated with a single 8-byte failure-atomic write instruction, saving the data consistency cost. The next pointer and the mLKey are 8B each.

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on September 23,2022 at 09:03:34 UTC from IEEE Xplore. Restrictions apply.

The Key and Value are 8B each, forming a 16B pair. Through these fields, the LNode header manages the critical information to support efficient data persistence and highly concurrent access to its internal KV pairs.

4 Relaxed RWLock-Based Concurrency Control

With the proposed hierarchical skiplist structure, multiple threads may concurrently access the INodes or LNodes, leading to current write, split, and read contention within the same node. To keep data consistency and high performance, we need efficient concurrency control mechanisms.

We have explored various concurrency strategies in PHAST and realized that we either need to introduce more pointers across the hierarchy of INode so that transient changes can be made atomic through CAS operations, or we need to protect the splitting INode or LNode within a transaction or an exclusive lock. The former happens to be very costly in performance due to many complex changes to our PHAST design. The transaction approach may be hardware specific. In our implementation, PHAST opt to use the read-write lock mechanism (RWLock) but in a more relaxed approach, such that more operations can be executed concurrently.

This relaxed approach is reflected in two aspects. First, we require each insert thread to acquire a read lock when there is no splitting, allowing concurrent insert operations, whereas a typical approach requires each thread to acquire a write lock that prevents them from being executed concurrently. Second, we configure each read thread as lock-free, allowing concurrent insert-search or split-search operations, whereas a typical approach requires each thread to hold a read lock that prevents them from being executed concurrently.

Algorithm 1 shows the relaxed RWLock-based concurrency control mechanism. All operations get the target node first (Line 1). If the thread is going to insert a KV pair, it acquires its read lock and check the boundary to get the right INode if the original target INode has been split (Line 2-8). Then it identifies the LNode by a binary search (Line 9). If the LNode is full, it will release the read lock and acquire the write lock. To avoid deadlock from the competition of the write lock, PHAST controls the behavior by *isSplit* flag. Only the thread that has successfully performed CAS is allowed to acquire the write lock, other threads will release the read lock and go to reinsert (Lines 11-17). The thread further checks if the INode is full. If so, it splits the INode to make room for an incoming LNode, otherwise, it will split the LNode (Line 18-22). After that, the thread will reset the *isSplit* flag, and release the write lock. Finally, the thread can insert the LNode according to the WriteLock-Free Concurrent Insert (WFCI, Algorithm 2) and release the read lock (Lines 27-28). Other modification-related operations will be forwarded to different functions similar to insert, while searches will be processed by a lock-free search that we will discuss in Section 4.3.

Based on this RWLock-based concurrency policy, we will discuss the concurrent insert, split, and search designs together with consistency-related efforts in the following three subsections.

Algorithm 1. RWLock-Based Concurrency Control(key, val, op)

```
1:  INode ← FindINode(key);
2:  if op.IsInsert() then
3:    LOC_1: Acquire(INode.ReadLock);
4:    while INode.maxLKey < key do        ▷find right INode
5:      Release(INode.ReadLock);
6:      INode ← INode.next[0];
7:      Acquire(INode.ReadLock);
8:    end while
9:    (LNode, shadowBM) ← BinarySearch(INode, key);
10:   if LNode.IsFull() then
11:     if CAS(INode.isSplit, False, True) then
12:       Release(INode.ReadLock);
13:       Acquire(INode.WriteLock);
14:     else
15:       Release(INode.ReadLock);
16:       goto LOC_1;                     ▷re-insert
17:     end if
18:     if INode.IsFull() then
19:       INodeSplit();                   ▷split INode
20:     else
21:       LNodeSplit();                   ▷split LNode as Fig. 6
22:     end if
23:     INode.isSplit ← False;
24:     Release(INode.WriteLock);
25:     goto LOC_1;                       ▷re-insert
26:   end if
27:   WFCI(shadowBM, LNode,
      key, val);                          ▷insert as Algorithm 2
28:   Release(INode.ReadLock);
29: else
30:   Other Operation;                    ▷e.g., read as Algorithm 3
31: end if
```

4.1 Writelock-Free Concurrent Insert

We design a writelock-free concurrent insert (WFCI) in two phases to insert KV pairs in the same LNode. The relaxed RWLock-based mechanism guarantees the insert is performed in a writelock-free way to achieve high concurrency. The failure-atomic CAS instruction is used to guarantee data consistency. Algorithm 2 shows the step of the insert processing. In the first phase, a thread will try to get an open bit from shadowBM and set it by CAS (Line 2-5). As the CAS instruction is failure-atomic, only one thread can succeed and insert the open entry in the LNode. The KV pair is then inserted to the corresponding entry in the LNode and persisted to PM via a sequence of CLWB and SFENCE instructions (Line 6-7). There is no need to persist shadowBM to PM since it is only used for a thread to locate the open entry in the LNode. In the second phase, the insert thread sets the corresponding bit in the default bitmap via another CAS (Line 9-12). The insert operation is done when the CAS instruction is executed successfully.

Fig. 5 shows an example of two threads (T1 and T2) performing concurrent insert operations. They first compete via CAS for Bit 2 (the first available bit) from shadowBM (SBM). Suppose that T1 wins the race. It then inserts and persists the KV pair (30, v1), updates the fingerprint, etc. T2 tries CAS again on shadowBM and obtains Bit 3, the next

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on September 23,2022 at 09:03:34 UTC from IEEE Xplore. Restrictions apply.

BM	SBM	KV [ ]				T1 (30, v1)	T2 (40, v2)
1000	1000	10				CAS(SBM, 1000, 1100)	CAS(SBM, 1000, 1100)
1000	1100	10				Write (KV[1]) Write (fingerprints[1]) Persist (KV[1])	Retry
1000	1100	10					CAS(SBM, 1100, 1110)
1000	1110	10					Write (KV[2]) Write (fingerprints[2]) Persist (KV[2])
1000	1110	10	30	40			
1000	1110	10	30	40		CAS(BM, 1000, 1100)	CAS(BM, 1000, 1010)
1010	1110	10	30	40		Retry	Persist (BM) Finish
1010	1110	10	30	40		CAS(BM, 1010, 1110)	
1110	1110	10	30	40		Persist (BM) Finish	
1110	1110	10	30	40			

Fig. 5. An example of concurrent insert operations from two threads, T1 to insert (30, v1) and T2 (40, v2). BM: bitmap; SBM: shadowBM; KV[]: the array of KV pairs.

available bit. It proceeds with the same instructions to insert its KV pair (40, v2). Since threads may progress at different paces, chances of T1 and T2 can arrive at the same time to update the default bitmap (BM). Suppose T2 wins in the CAS race on the bitmap. It then sets Bit 3 as 1 on the default bitmap and marks the KV pair (40, v2) as present. T1 then tries CAS again on the bitmap and sets Bit 2 as 1 to mark the presence of (30, v1).

Algorithm 2. WFCI(shadowBM, LNode, key, val): Write-Lock-Free Concurrent Insert

```
1:  while !LNode.IsFull() do
2:    (oldSBM, newSBM) ← shadowBM;
3:    pos ← FindFirstZero(oldSBM);
4:    newSBM[pos] ← 1;
5:    if CAS(shadowBM, oldSBM, newSBM) then
6:      LNode.entries[pos] ← (key, val);
7:      Persist(LNode.entries[pos]);
8:      while True do
9:        (oldBM, newBM) ← LNode.bitmap;
10:       LNode.fingerprints[pos] ← Hash(key);
11:       newBM[pos] ← 1;
12:       if CAS(LNode.bitmap, oldBM, newBM) then
13:         Persist(LNode.bitmap);
14:         Break;
15:       end if
16:     end while
17:     Break;
18:   end if
19: end while
```

4.2 Log-Free Atomic In-Place Split

Both INode and LNode have limited room and shall be split to accommodate more KV pairs. With the relaxed RWLock-based concurrency control in Section 4, PHAST ensures that LNodes are split in an exclusive manner, i.e., without interference by concurrent threads. Since PHAST is a linked list based data structure, it does not need to balance very often and deal with the wandering tree problem [40], as commonly required for tree based data structures. FAST-FAIR [26] supports log-free split of its nodes, but it still has to cope with the wandering tree problem when a new node triggers the split of parent nodes. In contrast, the simplicity of skiplist allows us to use log-free splitting in PHAST to support failure-atomic consistency. Furthermore, we keep the original full node and generate only one new node in the split process. Compared to the approaches with two new nodes [27], [38], such an in-place split manner can increase memory space utilization and reduce data copy overhead in the split.

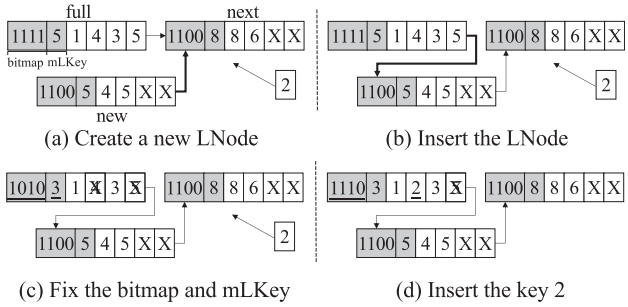


Fig. 6. Log-free LNode split for the insertion of key 2.

The log-free atomic in-place split is shown in Fig. 6. First, it allocates a new LNode and copies the larger half KV pairs and the next pointer in the full LNode to the new LNode. And then, it changes the next pointer of the full LNode to the new LNode using an 8-byte atomic write operation. If a failure occurred previously, the entire system remains consistent because the full LNode is not modified. After that, it clears the bitmap in the full LNode for the migrated KV pairs. Finally, it changes the mLKey in the LNode and the split process is successfully completed. If a failure occurs after the pointer changes, the system will be in an inconsistent state. However, this inconsistency can be detected and fixed when two adjacent LNodes are found to have the same mLKey. The recovery process can complete the split operation for the two LNodes. The consistency of the system is then guaranteed.

The split of an INode is a memory-based operation and has little effect on the data consistency of KV pairs on PM. First, it will create a new INode and copy the larger half mLKeys, corresponding leafPtrs and shadowBMs from the full INode to the new one. Then it sets some metadata information, e.g., nLevels, nLeaves, in the new INode. After installing the new node to the in-memory skiplist, the old INode's nLeaves and mIKey will be reset. As the header information is copied and reset in memory, the processing is fast.

4.3 Durable Lock-Free Concurrent Search

INode and LNode keep track of the maximal key inside them as mIKey and mLKey. These two keys need to be updated when a LNode or INode is split. Without the protection of an exclusive lock, it is possible that a read thread observes the two keys not yet properly updated in the middle of splitting, leading to incorrect search. Instead of using exclusive locks to prevent a read operation from observing the transient states, we design durable lock-free concurrent search without any restriction. The basic idea is to equip the read threads with the ability to validate the INode or the LNode twice to ensure that the pertinent INode and LNode have not changed during the search process.

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on September 23,2022 at 09:03:34 UTC from IEEE Xplore. Restrictions apply.

Algorithm 3. LockFreeSearch(key): Lock-Free Search the Target Value From PHAST

```
1:  INode ← FindINode(key);
2:  LOC_1:
3:  while INode.mIKey < key do
4:    INode ← INode.next[0];
5:  end while
6:  LNode ← SequentialSearch(INode, key);
7:  while LNode.mLKey < key do
8:    LNode ← LNode.next;
9:  end while
10: bitmap ← LNode.bitmap; mLKey ← LNode.mLKey;
11: for each occupied position i in bitmap do
12:   if LNode.fingerprints[i] == Hash(key) &&
        LNode.entries[i].key == key then
13:     value ← LNode.entries[i].value;
14:     Break;
15:   end if
16: end for
17: if INode.isSplit or mLKey! = LNode.mLKey then
18:   goto LOC_1;
19: end if
20: Return value;
```

Algorithm 3 shows the design of the lock-free concurrent search scheme. To search a key from PHAST, a thread first gets an INode from the index cache. Because of potential transient inconsistencies between INode and the index cache, the thread will try to validate if its mIKey is still larger than the requested key. If mIKey is no longer larger than the requested key, the thread is going to probe the next INode until it finds the right INode (Line 1-5). It then obtains the right LNode by searching the mLKey array (Line 6). Since the mLkey array is updated by shifting the keys, the thread sequentially searches the mLKey array to obtain the LNode. It then validates the boundary key (mLKey) to detect whether a split has updated the mLKey array and results in a stale view to the search thread. If this LNode is stale, this thread will probe the next one to attain the correct LNode (Line 7-9). To prevent the interference from inserts, this thread makes a copy of the bitmap and then traverses the occupied position within it (Line 10-16). If this LNode is not splitting (the isSplit flag is false) and no transient changes (the mLKey has not changed) are detected, the retrieved KV pair is valid and can be correctly returned. Otherwise, this search thread has to restart the process from LOC_1 (Line 17-19).

4.4 Other Operations

Update and Delete. Other write operations (i.e., update and delete) are simple variations to insert. For an update, PHAST only needs to persist the targeted value in the LNode using an 8-byte atomic write. For a delete, PHAST first checks whether the KV pair can be found in a LNode. If found, PHAST sets the corresponding key as invalid (MAX_UINT64 in our implementation). We do not clear the bit in the bitmap because this may result in the transient change of the bitmap, which cannot be observed by the lock-free search. The entries marked for deletion will be reclaimed during splitting.

Scan. The scan operation is much complex with the lock-free method since the split of LNodes will migrate some entries to the new LNode. The KV pairs retrieved from the new LNode could be redundant with the old one which will cause inconsistent views to the scan thread. Besides, the requested key range may cross several LNodes, the compelled rescan operation will bring high overhead when observing the transient states. Therefore, PHAST still adopts the traditional read lock in scan operations.

Index Cache Update. The index cache is a small cache of keys in DRAM designed to expedite the lookup of INodes. The addition of INodes at the lower levels will create stale entries in the index cache, resulting in more cache misses. While such misses can get resolved through the durable lock-free concurrent search scheme, it is still important to synchronize the index cache with the latest additions to better exploit its caching strength. However, concurrent read/write threads can access the index cache at the same time. Considering the lower frequency of adding INodes, PHAST adopts MVCC (Multi-Version Concurrency Control) to synchronize the index cache for concurrent threads. When several threads with additional INodes need to update the index cache, each will copy and modify the original index cache. Then they compete to install their copy as the latest index cache via an atomic CAS operation. One of them will succeed and the others have to retry.

Parallel Recovery. Partitions and INodes reside in DRAM and will be lost upon a power or system failure. While the physical KV pairs can be retrieved from the LNodes, a flat linked list of LNodes is insufficient to reconstruct this hierarchy efficiently, because it can force serialization of reconstruction due to the sequential traversal of LNode pointers. To support orderly and expedient construction of PHAST, we initialize an array of partition pointers on PM when PHAST is first created, shown as "Master" in Fig. 3. Each pointer is used to locate the beginning LNode of a partition. With the Master, each thread can proceed in parallel to reconstruct the INodes based on its constituent LNodes. Our reconstruction also performs a sanity validation on the recovered hierarchical structures, such as recalculation of fingerprints for all KV pairs, and the metadata consistency of LNode headers, INode headers and Index caches.

To avoid persistent memory leakage, we leverage existing PM atomic allocators from PMDK [41] to reclaim unused objects. PMDK creates a pool when the program starts running and allocates objects from the pool. During the recovery, we can get the allocated objects by scanning the whole pool and release the objects that are not added to linked lists of LNodes.

5 EVALUATION

5.1 Experimental Setup

System Configurations. Our experiments run on a server with two Intel Xeon Gold 6148 CPUs. Each CPU has 40 logical cores, 27.5 MB L3 cache, 64 GB DDR4 DRAM, and two Intel Optane DCPMMs (256GB in total). The OS is Ubuntu-18.04. Every pair of PMMs attached to a CPU is configured to be a single 256 GB namespace in App Direct mode, and is mounted with ext4-dax file system. To avoid NUMA effects, we perform all experiments on one CPU by pinning threads

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on September 23,2022 at 09:03:34 UTC from IEEE Xplore. Restrictions apply.

Fig. 7. Microbenchmark single-threaded performance.

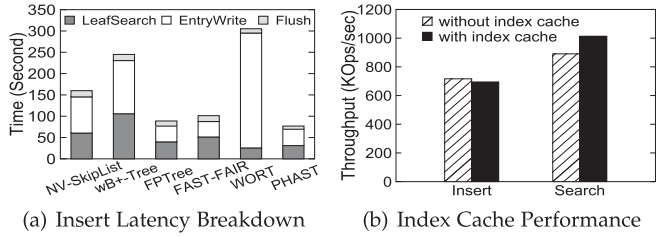


Fig. 8. Insert latency breakdown and the effects of index cache.

to a NUMA node accessing the local PMMs. All objects on PM are allocated by *libpmemobj* of PMDK [41] and persisted by the *pmemobj_persist* function. The real bandwidth and latency of the DRAM and PM devices in our hardware platform are listed in Table 1 of Section 2.1.

Comparisons and Configurations. We compare PHAST with the state-of-the-art indexing structures for PM: NV-Skiplist [27], wB+-Tree [16], FPTree [18], FAST-FAIR [26], and WORT [37]. These indexing structures use different strategies to trade-off among structure height, access locality, persistence overhead, and access concurrency. In the experiments, we use the following default parameters reported in the related paper to configure each indexing structure. For NV-Skiplist, the number of KV pairs in each leaf node is 64. For wB+-Tree, FPTree, and FAST-FAIR, their fanouts of the internal node are 7, 256, and 30, respectively. Accordingly, the number of KV pairs in their leaf nodes is 7, 64 and 30, respectively. For WORT, the fanouts of the internal and leaf node are 16 and 1, respectively. For PHAST, the number of partitions is 128 and the size threshold of Index Cache is 1MB. Each INode contains 128 LNodes and each LNode contains 56 KV pairs. For all tests, both the key and value sizes are 8B. To show the performance benefits from the partitioned hierarchical skiplist structure, we implement PHAST in a single-threaded version without concurrency-control fields including RWLock and shadowBM. To verify the effectiveness of the relaxed concurrency control mechanism, we also implement a concurrent version of PHAST with all structural fields enabled (Section 3.3.1).

5.2 Single-Threaded Performance

Microbenchmarks. We evaluate PHAST using microbenchmarks with keys and values in a uniform distribution. For insert tests, we first randomly insert 50M entries to warm up the structures and then measure the throughput for inserting the next 50M entries. For other tests, we measure the throughput for 50M random searches/updates/scans on a KV store with 100M entries. The scan tests are configured to retrieve 50 entries from a start key. Fig. 7 shows the single-threaded throughput.

For insert operations, PHAST has the best performance that outperforms NV-Skiplist, wB+-Tree, FPTree, FAST-FAIR, and WORT, by 2.56×, 3.76×, 1.33×, 1.61×, and 4.05×, respectively. PHAST delivers the best throughput because it has better search and data persistence performance owing to its lower height, higher locality, and log-free persistence technique. An insert operation incurs the leaf node searching time (LeafSearch), the entry writing time in the leaf node (EntryWrite), and the data persistence time (Flush). As shown in Fig. 8a, PHAST cuts off much of the LeafSearch time and the Flush time. Because of the different number of entries in leaf nodes, PHAST takes a longer EntryWrite time compared to FAST-FAIR but similar to FPTree. WORT has the longest EntryWrite time because it has the samllest leaf node (i.e., one KV pair) which results in more object allocations from PMDK and persistence overhead.

For search operations, PHAST outperforms FPTree, FAST-FAIR, NV-Skiplist and wB+-Tree by 1.22×, 1.30×, 2.42× and 2.87×, respectively. PHAST gets high throughput owing to the use of the index cache to improve locality. To verify this, we evaluate the Insert and Search throughputs of PHAST with and without the index cache. As Fig. 8b shows, although the index cache slightly degrades the insert throughput by 3.1% because of index synchronization (Section 4.4), it increases the search throughput by 13.8%. We also note that WORT has the best search throughput because it does not need comparisons of keys during searching. For update operations, they require additional KV write and CLWB operations compared to search operations. Thus the throughputs are generally lower than those of the search operations.

For scan operations, PHAST outperforms NV-Skiplist and wB+-Tree because of its fast search performance. PHAST is slower than FAST-FAIR because it has more unsorted keys in the LNode, while keys in the leaf node of FAST-FAIR are sorted. The performance of FPTree and PHAST is lower than FAST-FAIR by 29.0% and 33.0% respectively. However, keeping the ordering of entries will offset FAST-FAIR's insert performance, as previously shown. We also note that PHAST is comparable to FPTree because they both have a similar size of leaf nodes and need to sort KVs when returning to the client.

Note that we do not report the search and scan throughputs of WORT in Fig. 7 because the released code of WORT does not support such operations. In addition, WORT has shown much lower scan throughput compared to wB+ Tree and FPTree [37]. Due to these reasons, we exclude WORT from the rest of the evaluation.

YCSB-Benchmark. We then evaluate PHAST using YCSB [42] macrobenchmarks, including six default workloads as listed in Table 2. For the Load workload, we insert 100M entries as generated by YCSB in its *load* phase and measure the throughput. Then we measure the performance of all workloads using 50M entries in *run* phase.

Similar to previous microbenchmark tests, Fig. 9 shows PHAST delivers an overall better throughput compared to other index structures. For Load, PHAST outperforms NV-Skiplist and FPTree by 3.27× and 1.38×, respectively. Note that Workloads A, B, C, and F in YCSB have a skewed distribution of keys. PHAST also achieves the highest performance for these workloads. In the scan-dominated workload E, PHAST has a better performance than NV-Skiplist and wB+-Tree, but is slower than FPTree and FAST-FAIR.

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on September 23,2022 at 09:03:34 UTC from IEEE Xplore. Restrictions apply.

TABLE 2
YCSB Workloads

Workload	Operations	Distribution
LOAD	100% insert	uniform
A	50% read, 50% update	zipfian
B	95% read, 5% update	zipfian
C	100% read	zipfian
D	5% insert, 95% read	latest
E	5% insert, 95% scan	zipfian
F	50% read, 50% read-modify-write	zipfian

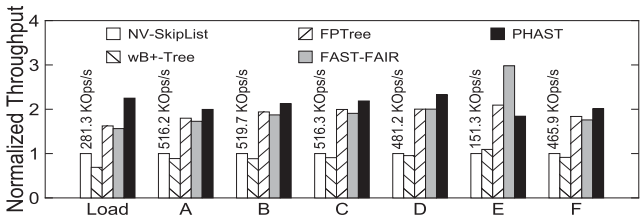


Fig. 9. YCSB single-threaded performance.

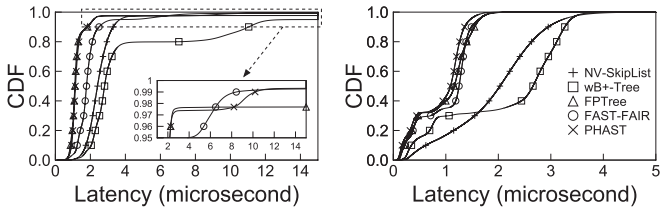
Fig. 10 shows the CDF of tail latencies of different indexing structures under different YCSB workloads. YCSB workloads have similar patterns in terms of tail latency. Thus we only show the results of Workload C and Load, which have 100% search and insert operations, respectively. As shown in Fig. 10a, FPTree delivers better tail latencies than PHAST, for up to 97.5% of the operations. Beyond that, its tail latency degrades rapidly, and becomes much worse than PHAST. This is because the log-free design of PHAST can largely reduce persistence overhead in slot splitting and thus helps tail latency and the overall throughput. For Workload C, Fig. 10b shows that PHAST achieves better tail latencies than all other schemes. This demonstrates that the design of PHAST with low height and high locality is effective to improve the access speed of skiplists.

5.3 Concurrency Performance

For concurrent tests, we compare PHAST only with FAST-FAIR because (1) NV-Skiplist does not focus on concurrency and its released code does not work for concurrent operations; (2) FAST-FAIR has shown its much higher performance than FPTree and wB+-Tree for concurrent operations in the evaluation results [26].

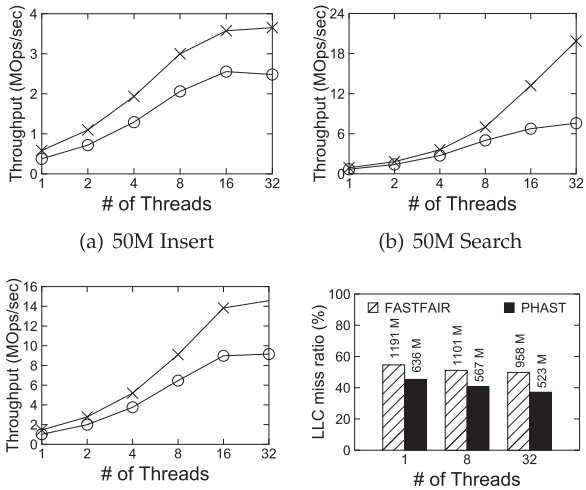
Microbenchmarks. We evaluate PHAST by expanding the benchmark to a concurrent version. We use three workloads: Insert, Search, and Mixed, similar to the test of FAST-FAIR. For the Insert workload, we first perform 50M inserts to warm up the indexing structures, then all threads evenly share 50M entries and insert them concurrently. We record the time until all threads finish their loads. In Search workload, each scheme performs 50M search operations in 100M entries. These 50M search operations are also evenly divided among threads. For the Mixed workload, we also warm up the store with 50M entries, then perform 50M inserts and 200M searches using concurrent threads. All KV pairs are generated via a uniform distribution.

Fig. 11a shows PHAST outperforms FAST-FAIR by up to 1.56× in inserts, and both PHAST and FAST-FAIR gain the best throughput around 16 threads. PHAST's superiority comes



(a) Load (b) Workload C

Fig. 10. Tail latency for single-threaded YCSB workloads.



(a) 50M Insert (b) 50M Search

(c) 50M Insert/200M Search (d) LLC Misses

Fig. 11. Microbenchmark concurrency performance.

from our WFCI technique allowing concurrent insert operations and does not have the cost of balancing a tree, while FAST-FAIR uses an exclusive lock (write lock in its implementation) to synchronize multiple inserts of a leaf node.

Fig. 11b shows PHAST outperforms FAST-FAIR significantly for concurrent search, from 1.33× (1 thread) to 2.62× (32 threads). This is because PHAST benefits from the grouping keys strategy and the index cache to achieve better memory access locality, while FAST-FAIR has to dereference address pointers and incur more last level cache (LLC) misses. We have examined the LLC cache misses using the *perf* tool [43]. Fig. 11d shows LLC miss ratios with a varying number of threads. The actual number of misses in millions is provided on top of each bar. The result shows FAST-FAIR's miss ratio is higher and the miss count is almost the double that of PHAST's. This indicates FAST-FAIR incurs more LLC cache misses, degrading its average search performance.

Fig. 11c shows PHAST also performs better than FAST-FAIR for the read/write mixed workload by up to 1.59× with 32 threads. Besides, the comparison between the two is similar in pattern like that of the Insert workload. The reason is that both PHAST and FAST-FAIR support lock-free search and perform similarly for search operations. Thus, for the mixed workload, the performance difference from their insert operations dominates over the difference from the search operations.

YCSB-Benchmark. We also evaluate the concurrency performance using multi-threaded YCSB workloads, comparing PHAST with FAST-FAIR. Due to the limit of the space, we only show Load, Workload A, and Workload C here. The experimental configurations are the same as the single-

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on September 23,2022 at 09:03:34 UTC from IEEE Xplore. Restrictions apply.

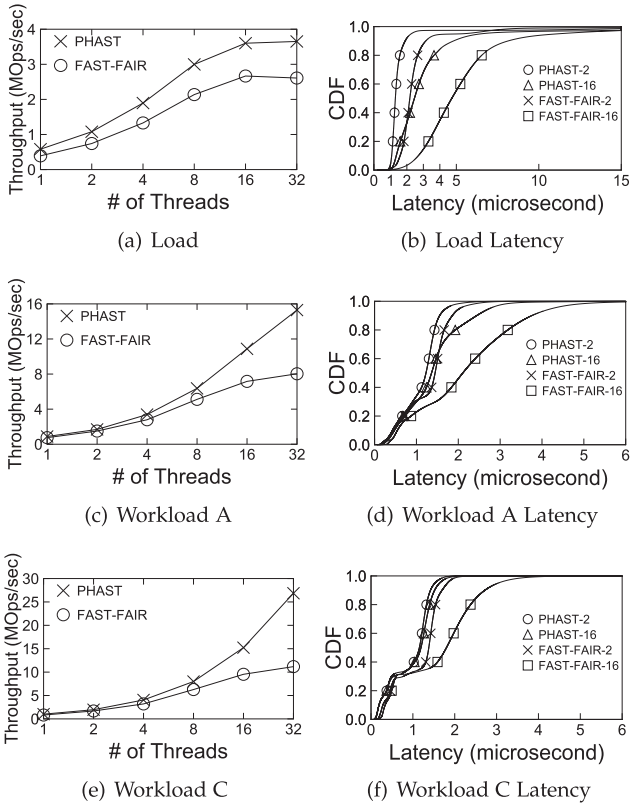


Fig. 12. YCSB concurrency performance.

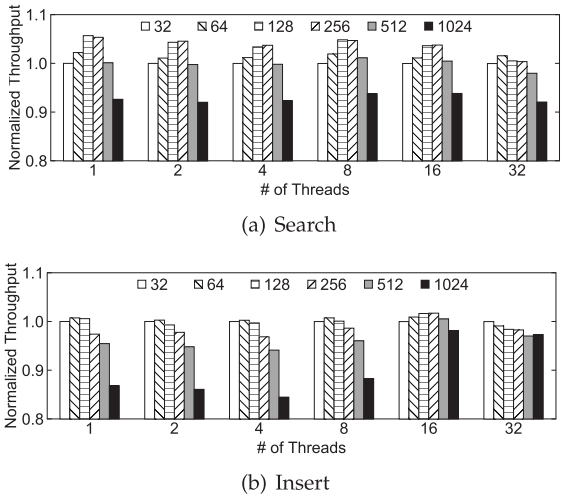


Fig. 13. Performance with different INode sizes.

threaded YCSB tests except that, for each workload, the total number of operations is evenly divided among all the threads.

As shown in Figs. 12a, 12c, and 12e, PHAST outperforms FAST-FAIR for Load, Workloads A and C by up to 1.49×, 1.90×, and 2.40×, respectively. The relative performance gains are higher than those from the single-threaded YCSB tests. As we mentioned before, PHAST gets benefits from WFCI in concurrent inserts and a lower height which results in a better insert performance. Workloads A and C both benefit from PHAST's efficient lock-free concurrent search. PHAST's performance in Workload A is not as good as it in Workload C, this is because updates require an additional persistent operation. Also, note that the throughput of PHAST scales well with an increasing number of threads while the throughput of FAST-FAIR is getting to be saturated around 16 threads in Workload A and C.

We have analyzed the distribution of tail latency from the multi-threaded YCSB tests. We use a suffix to denote different numbers of threads. As shown in Figs. 12b, 12d, and 12f, PHAST-2 represents the distribution of tail latency for PHAST with 2 threads. In Load, PHAST achieves better tail latencies than FAST-FAIR in 2 threads and 16 threads respectively. This is because PHAST benefits from our WFCI technique for concurrent inserts while threads in FAST-FAIR have to compete for the exclusive clock, resulting in longer tail latencies. With FAST-FAIR, the tail latency gets increased sharply when we increase the number of threads from 2 to 16. This is because of the intense competition from multiple threads on the exclusive lock.

For search-dominated Workload C, Fig. 12f shows that PHAST and FAST-FAIR have similar tail latencies for up to 30% operations. Beyond 30%, the latency quickly increases due to the cache misses. Note that the tail latency of PHAST scales gracefully with 16 threads, compared to the same for FAST-FAIR. This is because PHAST incurs much fewer number of LLC cache misses for search operations (see Fig. 11d) and its lock-free search scheme. For Workload A, due to 50% update operations, PHAST's superiority is not as much Workload C, but the latency is still better than FAST-FAIR in both 2 and 16 threads.

5.4 Other Tests

INode Size Impact. The INode size (i.e., the number of LNodes in an INode) impacts both the search and insert performance. Specifically, a smaller INode reduces the number of key comparisons and multi-threaded conflicts within an INode, while a larger INode can lower the height of PHAST and reduce the number of INode splits. Fig. 13 shows the performance of PHAST when executing concurrent tests with various INode sizes. The experimental setting is the same as the microbenchmark test in Section 5.3 except that we vary the number of LNodes among 32, 64, 128, 256, 512, and 1024. We can find both search and insert throughput decrease significantly when the number of LNodes is greater than 256. For searches, PHAST gets the highest throughput with 64 or 128 LNodes. For inserts, the highest throughput is achieved at 128 or 256 LNodes. To obtain moderate performance for both search and insert operations, we empirically set the number of LNodes to 128.

LNode Size Impact. The LNode size (i.e., the number of KVs in an LNode) is associated with the data persistence and consistency overhead. To study the impact of LNode sizes, we fix the INode size to 128 and vary the LNode size among 16, 32, 56, and 64. Fig. 14 shows the normalized throughput for the insert operation in the microbenchmark. We can find that, when the size is smaller than 56, the throughput increases linearly as the size increases. This is because a larger LNode triggers fewer splits, resulting in less write and flushing time. However, when the size is larger than 56, the bitmap and fingerprints can not be compressed in one cache line thus need extra cache line flushes, leading to performance degradation. Therefore, we set the LNode size to 56 to achieve the best performance.

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on September 23,2022 at 09:03:34 UTC from IEEE Xplore. Restrictions apply.

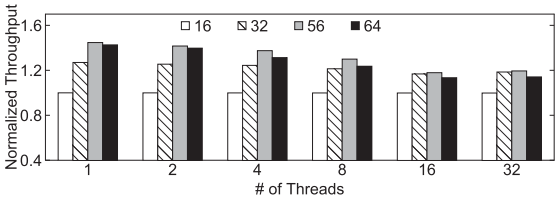


Fig. 14. Insert performance with different LNode sizes.

Partition Number Impact. To avoid the bias from the partitioning design and index cache, we design an extreme workload to populate the skiplist with a distribution that all keys are distributed to 1% key space out of 100M keys. This makes only one partition take effect in PHAST. The subsequent operations are randomly distributed upon the skewed keys, as the tests in Fig. 7. Fig. 15a shows that the throughput of all schemes declines, but PHAST still has the best performance in all operations. PHAST outperforms FPTree (the second best one) by 1.18×, 1.07×, and 1.07× in insert, search and update, respectively.

Large Dataset. Fig. 15b shows the single-threaded throughput with a dataset of one billion KVs. The experimental configurations are the same with Fig. 7 except that we populate the indexing structure with a larger dataset. Compared to Fig. 7, the absolute throughput is dropped for all the index structures, but the relative superiority is not changed. In insert workload, PHAST outperforms FPTree and FAST-FAIR by 1.15× and 1.56×. By leveraging index cache, PHAST still superiors to FPTree and FAST-FAIR in search and update workloads by 1.26× and 1.61×, 1.16× and 1.41×, respectively.

Variable-Size Values. Same to prior works [16], [17], [18], PHAST can use indirection pointers to support variable-size KVs. That is, the index structure only contains pointers, which refer to the locations of actual variable-size keys or values stored in extra data areas. Fig. 15c shows the results of the variable-size values with microbenchmarks under a single thread. The values are randomly generated with sizes between 8 to 256 bytes. We can find the performance degradation for all the indexes because such an approach brings additional pointer chasing overhead. In addition, more PM writes and flush instructions are needed to persist both the indirection pointers and actual KVs. However, it does not affect the advantages of PHAST over other indexes. PHAST still outperforms other indexes by up to 1.51× for insert operations and 2.65× for search operations.

Recovery Time. Fig. 15d shows the recovery time of PHAST with 100M entries. No comparisons have been provided because the released code packages for NV-Skiplist, wB +-Tree, FPTree, and FAST-FAIR do not support recovery.

TABLE 3
INode Footprint With M KV Pairs, N Denotes the Number of INodes

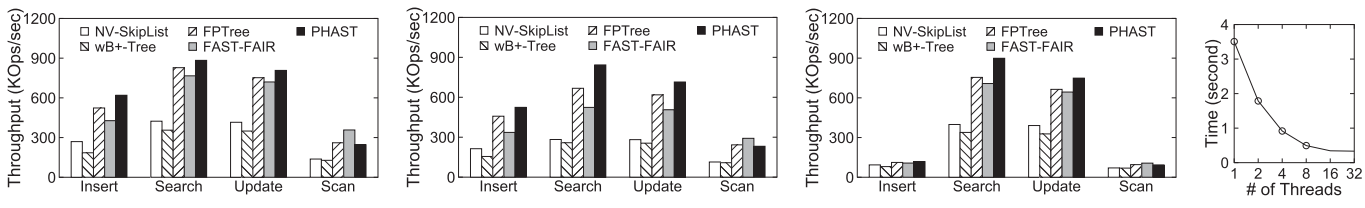
	# of INodes	INode Size (Bytes)
Skiplist	$\lceil \frac{M}{2} \rceil$	$16 + 8 * \lceil \log_2(N) \rceil$
NV-Skiplist	$\lceil \frac{M}{0.75*64} \rceil$	$16 + 8 * \lceil \log_2(N) \rceil$
PHAST	$\lceil \frac{M}{0.75*56*0.75*128} \rceil$	$3096 + 8 * \lceil \log_2(N) \rceil$

With an increasing number of threads, the recovery time goes down gradually until the reconstruction speed is nearly saturated from 16 threads and beyond. The recovery time is reduced by more than 8× from 1 thread to 16 threads, demonstrating that PHAST's structural hierarchy is well organized for expedient parallel recovery.

DRAM Consumption. With the hierarchical design, PHAST can significantly reduce the number of INodes, thus reducing the DRAM consumption. Table 3 shows the DRAM footprint of INodes for a generic skiplist (Skiplist, only bottom-layer nodes on PM), NV-Skiplist, and PHAST (according to the configuration in Section 5.1). The INode of the Skiplist and NV-Skiplist consists of a key (8B), a child pointer (8B), and the forward pointers (8*$\lceil \log_2(N) \rceil$, where N is the number of INodes). PHAST has the metadata information (3096B according to the Fig. 4) and the same forward pointers (8*$\lceil \log_2(N) \rceil$). For 100M KV pairs, the DRAM consumption is 10.4 GiB, 365.5 MiB, and 76.1MiB for Skiplist, NV-Skiplist, and PHAST, respectively. These results show that PHAST has the lowest DRAM overhead of storing INodes. PHAST also brings additional DRAM overhead in the index cache. Its size is linearly related to the number of INodes and can be calculated as $N * 16B$. For 100M KV pairs, the index cache size is 0.38 MiB, which is acceptable.

6 Related Work

Skiplists. Fomitchev *et al.* [44] add a marker and pointer in leaf nodes of singly-linked lists to improve deletion concurrency. But it focuses inter-node concurrency while PHAST on intra-node concurrency. Crain *et al.* [45] boost insert concurrency in internal nodes by delaying the raising of them and using a specified thread to promote them later. PSL [36] enhances skiplist locality by grouping multiple entries in index nodes and leverages single instruction multiple data to perform parallel traversal. S3 [46] introduces an additional indexing layer to boost search performance and stores two unordered keys in one leaf node to optimize write efficiency. All these works are designed for DRAM memory. NV-Skiplist [27] is the skiplist designed for PM. It groups



(a) One partition test. (b) Large dataset test. (c) Variable-size value test. (d) Recovery test.

Fig. 15. Performance for other tests. (a) One partition test using skewed distribution. (b) Large dataset test with 1 billion entries. (c) Varying the size of value from 8 bytes to 256 bytes. (d) Recovery time with 100M entries.

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on September 23,2022 at 09:03:34 UTC from IEEE Xplore. Restrictions apply.

multiple keys in one leaf node and uses log-free persistence approach. However, NV-Skiplist creates two new LNodes in splits whereas PHAST generates only one. Besides, PHAST uses new index structure and concurrency techniques to further improve performance.

Other Persistent Indexing Structures. wB+-Tree [16] stores both internal and leaf nodes on PM, but it relies on costly undo-redo logs when a split occurs. NV-Tree [17] stores its internal nodes in DRAM and the leaf nodes on PM, but all internal nodes are stored in consecutive memory blocks, leading to costly rebuilds when an internal node overflows. FPTree [18] also only stores leaf nodes on PM. WORT [37] is a radix tree and adopts costly copy-on-write for both merge and split of nodes. David *et al*. [47] propose three log-free techniques for concurrent data structures to reduce consistency overhead. RECIPE [48] explores the crash consistency principle to convert in-memory indexes to in-PM indexes. But RECIPE requires some conditions and can't apply to PHAST. FAST-FAIR [26] proposes failure-atomic shifts and failure-atomic in-place rebalancing to reduce cache line flushes in data persistence. Its KV pairs in the leaf nodes are ordered, leading to high write overhead. LB+-Tree [35] performs more word writes in a cache line to reduce cache line flushes and keeps the node size 256B-aligned to best utilizes 3D XPoint's internal bandwidth. These techniques are complementary to PHAST.

Concurrent Indexing Structures. There are three main strategies for concurrent indexing operations, lock-based solutions [26], [27], multi-version concurrency control [17], [49], and lock-free compare-and-swap [50] or LL/SC implementations. FPTree [18] employs hardware transnational memory to handle the concurrency of in-memory internal nodes and fine-grained locks for in-PM leaf nodes. Non-blocking implementations of indexing structures have been provided for various lists and trees [51], [52], [53]. PALM [50] performs batched index operations in an atomic manner on lock-free concurrent B+ trees. FAST-FAIR [26] uses lock-free read transactions to allow read operations to happen during concurrent writes. But it restricts read queries to proceed in a specified order. In addition, FAST-FAIR does not support lock-free write operations. In contrast, PHAST performs writelock-free concurrent inserts, lock-free concurrent searches, and log-free split operations.

7 Conclusion

In this paper, we propose the design and implementation of a high-performance skiplist, PHAST, which leverages persistent memory to tackle the memory overhead and boost indexing performance. PHAST consists of equipped techniques including writelock-free concurrent insert and log-free atomic split for fast write operations and durable lock-free concurrent search for highly concurrent read operations. Our results show that PHAST efficiently reduces indexing overhead and outperforms other indexing schemes for a wide range of operations with high concurrency.

Acknowledgments

Zhenxin Li and Bing Jiao have equal contribution. Shuibing He and Weikuan Yu are co-corresponding authors.

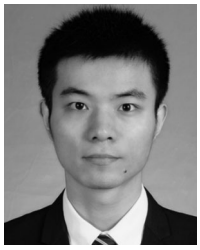
References

[1] Facebook, "RocksDB," 2019. [Online]. Available: https://rocksdb.org/
[2] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, 2010.
[3] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-Tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.
[4] D. Comer, "Ubiquitous B-Tree," *ACM Comput. Surv.*, vol. 11, no. 2, pp. 121–137, Jun. 1979.
[5] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new Ext4 filesystem: Current status and future plans," in *Proc. Linux Symp.*, 2007, pp. 21–33.
[6] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS file system," in *Proc. USENIX Annu. Tech. Conf.*, 1996, Art. no. 1.
[7] W. D. Maurer and T. G. Lewis, "Hash table methods," *ACM Comput. Surv.*, vol. 7, no. 1, pp. 5–19, Mar. 1975.
[8] W. Pugh, "Skip lists: A probabilistic alternative to balanced trees," *Commun. ACM*, vol. 33, no. 6, pp. 668–676, 1990.
[9] H.-S. P. Wong *et al.*, "Phase change memory," *Proc. IEEE*, vol. 98, no. 12, pp. 2201–2227, Dec. 2010.
[10] J. J. Yang and R. S. Williams, "Memristive devices in computing system: Promises and challenges," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 9, no. 2, pp. 1–20, 2013.
[11] Intel and micron produce breakthrough memory technology, 2020. [Online]. Available: https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/
[12] A. Badam, "How persistent memory will change software systems," *Computer*, vol. 46, no. 8, pp. 45–51, 2013.
[13] Intel, "Redis-PM," 2018. [Online]. Available: https://github.com/pmem/redis/tree/3.2-nvml
[14] Lenovo, "Memcached-PM," 2018. [Online]. Available: https://github.com/lenovo/memcached-pmem
[15] C. Chen *et al.*, "Optimizing in-memory database engine for AI-powered on-line decision augmentation using persistent memory," *Proc. VLDB Endowment*, vol. 14, no. 5, pp. 799–812, 2021.
[16] S. Chen and Q. Jin, "Persistent B+-Trees in non-volatile main memory," *Proc. VLDB Endowment*, vol. 8, no. 7, pp. 786–797, 2015.
[17] 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 *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 167–181.
[18] 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 *Proc. Int. Conf. Manage. Data*, 2016, pp. 371–386.
[19] F. Xia, D. Jiang, J. Xiong, and N. Sun, "HiKV: A hybrid index key-value store for DRAM-NVM memory systems," in *Proc. USENIX Annu. Tech. Conf.*, 2017, pp. 349–362.
[20] Y. Chen, Y. Lu, K. Fang, Q. Wang, and J. Shu, "uTree: A persistent B+-tree with low tail latency," *Proc. VLDB Endowment*, vol. 13, no. 12, pp. 2634–2648, 2020.
[21] M. Nam, H. Cha, Y.-R. Choi, S. H. Noh, and B. Nam, "Write-optimized dynamic hashing for persistent memory," in *Proc. 17th USENIX Conf. File Storage Technol.*, 2019, pp. 31–44.
[22] A. Prout, "The story behind MemSQL's skiplist indexes," 2017. [Online]. Available: https://laptrinhx.com/the-story-behind-memsql-s-skiplist-indexes-814186695/
[23] LevelDB, 2020. [Online]. Available: https://github.com/google/leveldb
[24] G. Huang *et al.*, "X-Engine: An optimized storage engine for large-scale E-commerce transaction processing," in *Proc. Int. Conf. Manage. Data*, 2019, pp. 651–665.
[25] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *Proc. 18th USENIX Conf. File Storage Technol.*, 2020, pp. 169–182.
[26] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in byte-addressable persistent B+-Tree," in *Proc. 16th USENIX Conf. File Storage Technol.*, 2018, pp. 187–200.
[27] Q. Chen, H. Lee, Y. Kim, H. Y. Yeom, and Y. Son, "Design and implementation of skiplist-based key-value store on non-volatile memory," *Cluster Comput.*, vol. 22, no. 2, pp. 361–371, 2019.
[28] Intel Optane persistent memory, 2019. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on September 23,2022 at 09:03:34 UTC from IEEE Xplore. Restrictions apply.

[29] R. Kadekodi *et al.*, "WineFS: A hugepage-aware file system for persistent memory that ages gracefully," in *Proc. ACM SIGOPS 28th Symp. Oper. Syst. Princ.*, 2021, pp. 804–818.
[30] Y. Chen, Y. Lu, B. Zhu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. Shu, "Scalable persistent memory file system with kernel-userspace collaboration," in *Proc. 19th USENIX Conf. File Storage Technol.*, 2021, pp. 81–95.
[31] D. Castro, A. Baldassin, J. Barreto, and P. Romano, "SPHT: Scalable persistent hardware transactions," in *Proc. 19th USENIX Conf. File Storage Technol.*, 2021, pp. 155–169.
[32] K. Wu, J. Ren, I. Peng, and D. Li, "ArchTM: Architecture-aware, high performance transaction for persistent memory," in *Proc. 19th USENIX Conf. File Storage Technol.*, 2021, pp. 141–153.
[33] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. Shu, "FlatStore: An efficient log-structured key-value storage engine for persistent memory," in *Proc. 25th Int. Conf. Architect. Support Program. Lang. Oper. Syst.*, 2020, pp. 1077–1091.
[34] S. Gugnani, A. Kashyap, and X. Lu, "Understanding the idiosyncrasies of real persistent memory," *Proc. VLDB Endowment*, vol. 14, no. 4, pp. 626–639, 2020.
[35] J. Liu, S. Chen, and L. Wang, "LB+Trees: Optimizing persistent index performance on 3DXPoint memory," *Proc. VLDB Endowment*, vol. 13, no. 7, pp. 1078–1090, 2020.
[36] Z. Xie, Q. Cai, H. V. Jagadish, B. C. Ooi, and W.-F. Wong, "Parallelizing skip lists for in-memory multi-core database systems," in *Proc. IEEE 33rd Int. Conf. Data Eng.*, 2017, pp. 119–122.
[37] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, "WORT: Write optimal radix tree for persistent memory storage systems," in *Proc. 15th USENIX Conf. File Storage Technol.*, 2017, pp. 257–270.
[38] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson, "BzTree: A high-performance latch-free range index for non-volatile memory," *Proc. VLDB Endowment*, vol. 11, no. 5, pp. 553–565, Jan. 2018.
[39] T. L. Harris, K. Fraser, and I. A. Pratt, "A practical multi-word compare-and-swap operation," in *Proc. 16th Int. Conf. Distrib. Comput.*, 2002, pp. 265–279.
[40] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, Feb. 1992.
[41] Persistent memory development kit, 2020. [Online]. Available: http://pmem.io/
[42] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 143–154.
[43] Perf Wiki, 2020. [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page
[44] M. Fomitchev and E. Ruppert, "Lock-free linked lists and skip lists," in *Proc. 23rd Annu. ACM Symp. Princ. Distrib. Comput.*, 2004, pp. 50–59.
[45] T. Crain, V. Gramoli, and M. Raynal, "No hot spot non-blocking skip list," in *Proc. IEEE 33rd Int. Conf. Distrib. Comput. Syst.*, 2013, pp. 196–205.
[46] J. Zhang *et al.*, "S3: A scalable in-memory skip-list index for key-value store," *Proc. VLDB Endowment*, vol. 12, no. 12, pp. 2183–2194, 2019.
[47] T. David, A. Dragojevic, R. Guerraoui, and I. Zablotchi, "Log-free concurrent data structures," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 373–386.
[48] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram, "RECIPE: Converting concurrent DRAM indexes to persistent-memory indexes," in *Proc. 27th ACM Symp. Oper. Syst. Princ.*, 2019, pp. 462–477.
[49] K. A. Bailey, P. Hornyack, L. Ceze, S. D. Gribble, and H. M. Levy, "Exploring storage class memory with key value stores," in *Proc. 1st Workshop Interact. NVM/FLASH Oper. Syst. Workloads*, 2013, Art. no. 4.
[50] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey, "PALM: Parallel architecture-friendly latch-free modifications to B+ trees on many-core processors," *Proc. VLDB Endowment*, vol. 4, no. 11, pp. 795–806, Aug. 2011.
[51] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel, "Non-blocking binary search trees," in *Proc. 29th ACM SIGACT-SIGOPS Symp. Princ. Distrib. Comput.*, 2010, pp. 131–140.
[52] A. Natarajan and N. Mittal, "Fast concurrent lock-free binary search trees," in *Proc. 19th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2014, pp. 317–328.
[53] J. J. Levandoski, D. B. Lomet, and S. Sengupta, "The Bw-Tree: A B-Tree for new hardware platforms," in *Proc. IEEE Int. Conf. Data Eng.*, 2013, pp. 302–313.

Zhenxin Li received the bachelor's degree from Sichuan University, in 2020. He is currently working toward the PhD degree with the College of Computer Science and Technology, Zhejiang University. His research interests include parallel I/O systems, file and storage systems, and key-value stores.



Bing Jiao received the bachelor's degree from Fuyang Normal University, in 2015, and the master's degree from Hunan University, in 2018. He is currently working toward the PhD degree with the Computer Architecture and SysTems Research Laboratory, Department of Computer Science, Florida State University. His research interests include file and storage systems, key-value stores, and high performance computing.

Shuibing He (Member, IEEE) received the PhD degree in computer science and technology from the Huazhong University of Science and Technology, in 2009. He is now a ZJU100 Young professor with the College of Computer Science and Technology, Zhejiang University. His research areas include parallel I/O systems, file and storage systems, key-value stores, high-performance, and distributed computing. He is a member of the ACM.

Weikuan Yu received the bachelor's degree in genetics from Wuhan University, Wuhan, China, and the master's degree in developmental biology from Ohio State University, Columbus, Ohio, where he also received the PhD degree in computer science, in 2006. He is currently a full professor with the Department of Computer Science, Florida State University, Tallahassee, Florida. His research areas include cloud computing, parallel file and storage systems, deep learning, and data analytics.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on September 23,2022 at 09:03:34 UTC from IEEE Xplore. Restrictions apply.