NOBtree: A NUMA-Optimized Tree Index for Nonvolatile Memory

Zhaole Chu, Peiquan Jin, *Member, IEEE*, Yongping Luo, Xiaoliang Wang, *Member, IEEE*, and Shouhong Wan

***Abstract*—Nonvolatile memory (NVM) suffers from more serious nonuniform memory access (NUMA) effects than DRAM because of the lower bandwidth and higher latency. While numerous works have aimed at optimizing NVM indexes, only a few of them tried to address the NUMA impact. Existing approaches mainly rely on local NVM write buffers or DRAM-based read buffers to mitigate the cost of remote NVM access, which introduces memory overhead and causes performance degradation for lookup and scan operations. In this article, we present *NOBtree*, a new NUMA-optimized persistent tree index. The novelty of NOBtree is two-fold. First, NOBtree presents per-NUMA replication and an efficient node-migration mechanism to reduce remote NVM access. Second, NOBtree proposes a NUMA-aware NVM allocator to improve the insert performance and scalability. We conducted experiments on six workloads to evaluate the performance of NOBtree. The results show that NOBtree can effectively reduce the number of remote NVM accesses. Moreover, NOBtree outperforms existing persistent indexes, including TLBtree, Fast&Fair, ROART, and PACtree, by up to 3.23× in throughput and 4.07× in latency.**

***Index Terms*—Nonuniform memory access (NUMA) effect, nonvolatile memory (NVM), tree index.**

I. Introduction

The non-uniform memory access (NUMA) architecture is a prevalent design in modern multicore systems [1], [2]. In a NUMA system, CPU cores and memory DIMMs are organized into clusters known as NUMA nodes, connected by internode links like the Intel Ultra Path Interconnect (UPI). Each processor can access either the memory on its own NUMA node or that of another, resulting in local/remote memory access. Local memory access is inherently faster than remote memory access. Such an asymmetry in accessing cost is known as the NUMA effect [1], [2], which impacts application performance substantially.

Nonvolatile memory (NVM) [3] is an emerging memory technology characterized by byte-addressability and persistent storage capabilities. It challenges the traditional storage hierarchy by bridging the gap between DRAM and SSD, prompting a fundamental reevaluation of storage system design principles. In response to this paradigm shift, significant efforts have been devoted to optimizing conventional data management techniques for NVM, including index structures, file systems, and key-value stores. Current NVM devices, e.g., Intel Optane Persistent Memory ("Optane" for short in this article) [4], are integrated into the system much like DRAM, thus also experiencing the NUMA effect. Due to the higher latency and lower bandwidth than DRAM, NVM is more susceptible to the NUMA effect. With NUMA becoming ubiquitous, mitigating its impact is crucial when devising NVM indexes.

So far, many NVM-aware indexes have been proposed [5], [6], [7]. These indexes have introduced various techniques to optimize performance on NVM, but the consideration of the NUMA effect is notably absent in many of these designs. Several works have focused on mitigating the NUMA effect of NVM indexes, such as Nap [1] and PACtree [8]. However, Nap needs additional DRAM buffers and is tailored for specific workloads, and PACtree has huge extra space overhead caused by the log. These overheads will become more influential in machines with more NUMA nodes. In addition, Nap can not work well under uniform workloads and always undergoes a degraded scan performance because of the existence of DRAM buffers.

In this article, we introduce *NOBtree*, a novel NUMA-optimized tree index tailored for NVM environments. NOBtree adopts a decoupled tree structure comprising a static read-optimized upper layer and a write-optimized bottom layer to mitigate the NUMA effect. Briefly, this article makes the following contributions.

1) We present NOBtree, a new NUMA-optimized NVM-aware tree index with a decoupled structure. NOBtree proposes two new designs, including per-NUMA replication and an efficient node-migration mechanism, to reduce remote NVM access.
2) We propose a dedicated NUMA-aware NVM allocator that supports round-robin, local, and specific NUMA node allocations. It incorporates a post-crash garbage collection (GC) mechanism to reduce the overhead of persistence and adopts a two-layer architecture to reduce the contention of threads, thus improving the performance of NVM allocation.
3) We conduct comprehensive experiments on a two-socket server equipped with two CPUs and real NVM devices to compare NOBtree with state-of-the-art NVM indexes. The results show that NOBtree can effectively reduce the number of remote NVM accesses. Moreover,

Manuscript received 30 July 2024; accepted 30 July 2024. This work was supported by the National Science Foundation of China under Grant 62072419. This article was presented at the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES) 2024 and appeared as part of the ESWEEK-TCAD Special Issue. This article was recommended by Associate Editor S. Dailey. *(Corresponding author: Peiquan Jin.)*

The authors are with the School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China (e-mail: czle@mail.ustc.edu.cn; jpq@ustc.edu.cn; ypluo@mail.ustc.edu.cn; wxl147@mail.ustc.edu.cn; wansh@ustc.edu.cn).

Digital Object Identifier 10.1109/TCAD.2024.3438111

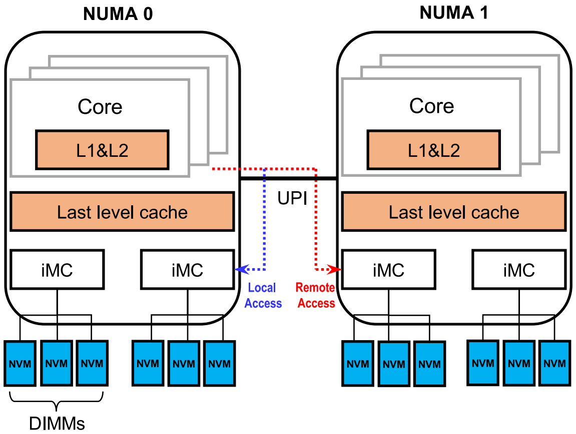

Fig. 1. Architecture of a typical NUMA system.

NOBtree outperforms existing persistent indexes, including TLBtree, Fast&Fair, ROART, and PACtree, by up to 3.23× in throughput and 4.07× in latency.

The remainder of this article is structured as follows. Section II summarizes related work. Section III details the design of NOBtree. Section V reports the experimental results. Section VI discusses the impact of Optane's discontinuation. Finally, in Section VII, we conclude this article.

II. Related Work

In this section, we first discuss the NUMA effect on NVM. Then, we summarize the recent advances in NVM-aware indexes and NUMA optimizations. Finally, we introduce techniques for NVM management.

A. NUMA Effect on NVM

NVM integrates into a system much like DRAM. Fig. 1 depicts a typical NUMA system. Given its higher latency and lower bandwidth than DRAM, NVM experiences more pronounced NUMA effects [9] than DRAM. Prior research [1], [8], [9] has underscored a noteworthy decrease in remote bandwidth for NVM. This effect is primarily attributed to the cache coherency protocol, particularly evident when threads on different NUMA nodes access the same NVM address. More specifically, today's Intel processor architectures rely on the directory coherence protocol among NUMA domains. For NVM devices, the directory information is also stored in the 3D-XPoint media. When accessing the NVM address on a remote NUMA node, it will incur the coherence state change, causing a write operation to the directory. That means even a read operation will result in small write operations to NVM, causing the performance drop of remote NVM accesses. Consequently, it is necessary to avoid or reduce the NUMA impact in designing data management systems or storage systems tailored for NVM.

B. NVM-Aware Indexes

There have been a lot of indexes designed for NVM, either for NVM-only or DRAM-NVM hybrid architecture [5], [6], [10]. The main challenge of the NVM index is how to ensure crash consistency without too much performance loss. The NVM index's operation must guarantee crash consistency because of NVM's feature of persistent storage. A commonly used approach is to use cacheline flush and fence instructions (e.g., *CLWB* and *SFENCE*) to guarantee the memory write order. However, using flush and fence instructions will lead to performance degradation. Previous works have introduced various techniques to optimize the performance of NVM indexes, such as fingerprints, indirect order arrays, selective persistence, and log-free node splitting mechanisms. However, the consideration of the NUMA impact is notably absent in many of these designs.

Numerous works have endeavored to optimize indexes for NVM, such as Fast&Fair [5], TLBtree [6], and ROART [11]. Fast&Fair introduced a novel mechanism to maintain the order of records inside a node by shifting slots and splitting/merging nodes in a failure-tolerable manner. TLBtree proposed a decoupled two-layer structure to accelerate index performance, which also inspires the structure of NOBtree. ROART presented a series of techniques to optimize ART on NVM, including entry compression and selective persistence. Overall, these indexes represent the recent advances in NVM-aware indexes.

However, within the extensive body of work dedicated to optimizing indexes for NVM, only a small portion has specifically focused on mitigating the NUMA effect. Nap, a black-box approach, transforms any NVM index into a NUMA-aware counterpart [1]. This transformation involves introducing an in-DRAM NUMA-aware layer (NAL) to expedite hot lookups and utilizing a local NVM buffer to absorb insertions, updates, and deletions. While Nap is a versatile method that demonstrates commendable performance on skewed workloads, the presence of read/write buffers complicates the operation process, leading to a decline in scan performance. Additionally, Nap is tailored for skewed workloads and may not handle uniform workloads as efficiently.

Node replication (NR) [2] is a widely recognized technique for addressing the NUMA effect in DRAM. It involves replicating data structures across NUMA nodes and utilizing a NUMA-aware shared log for synchronization. However, full replication consumes substantial memory, and the synchronization overhead is considerable, making it unsuitable for NVM-oriented indexes. Therefore, PACtree [8] divides the tree index into two parts: 1) the search layer and 2) the data layer. PACtree selectively replicates the search layer and employs per-NUMA logs to synchronize the replications. The per-NUMA log records the structural modifications of the data layer, and background updating threads will replay the log to update the search layer asynchronously. However, a drawback of the logging method is the additional NVM consumption for the log, as PACtree maintains a log file in every NUMA node. In Section V, where we assess each index's NVM usage, we observe that PACtree requires 500 MB of NVM space for the per-NUMA log. This space overhead becomes significant in machines with multiple NUMA nodes.

C. NVM Allocator

Dynamic NVM allocation is crucial to building high-performance and scalable index structures. The memory allocators are well-defined for DRAM to achieve high scalability and low latency. However, NVM allocators must also consider crash consistency, necessitating a reevaluation of design principles and implementations. Several NVM allocators have been developed, primarily falling into two categories based on crash consistency mechanisms: 1) log-based and 2) GC-based. Log-based allocators leverage logging to record all changes in metadata and memory addresses. Upon a crash, replaying the log rebuilds the allocator's correct metadata, ensuring crash consistency. PAllocator [12], Poseidon [13], and NVAlloc [14] adopt the log approach.

Another mechanism for ensuring crash consistency is the GC mechanism, which provides an alternative to the write-ahead log approach. Unlike log-based allocators, GC-based allocators do not rely on persistent logs for maintaining crash consistency. Instead, they rebuild metadata by traversing the heap from a predefined persistent root pointer. GC-based allocators offer faster allocation and deallocation than log-based allocators because they avoid the overhead of persisting logs for every allocation or deallocation operation. However, they may incur longer recovery times after a crash, as they need to traverse the entire memory space to reconstruct metadata. Ralloc [15] and DCMM [11] employ the GC mechanism.

Previous studies have emphasized the importance of NUMA-aware NVM allocation to minimize costs [1], [8]. However, existing approaches typically rely on PMDK to build customized NVM managers, which may suffer from performance and scalability limitations. To address the shortcoming, we propose to develop a dedicated high-performance NVM allocator with NUMA-awareness. This allocator aims to optimize NVM allocation while considering the NUMA effect, thereby enhancing the performance and scalability of NVM-based indexes.

III. Design of NOBTree

In this section, we first introduce the motivation and overall structure of NOBtree. Then, we describe the optimizations of NOBtree for the NUMA architecture. Finally, we present the architecture and operation of the NVM allocator.

A. Motivation of NOBtree

To explore the impact of the NUMA effect on NVM indexes, we first perform a comparative experiment involving TLBtree [6] and Fast&Fair [5], along with a variant of TLBtree named TLBtree-NR. In TLBtree-NR, the tree undergoes replication across all NUMA nodes, ensuring that all operations exclusively interact with the tree within the local NVM, eliminating all the remote NVM access. Note that TLBtree-NR does not incorporate the NR algorithm [2]; thus, it does not support crash consistency. We utilize TLBtree-NR solely to gauge the highest-achievable read performance by TLBtree without leveraging DRAM.

Fig. 2 depicts the performance gap between TLBtree and TLBtree-NR. The experiment took place on a two-socket server equipped with two Intel Xeon Gold 6242R CPUs, each having 20 physical cores. Thread assignment follows a configuration where if the number of threads is below 20, they are allocated to a single node. However, if the thread count surpasses 20, threads are distributed across two nodes, and hyper-threading is enabled when exceeding 40. We employ Yahoo! cloud serving benchmark (YCSB)-like workloads to assess the lookup and scan performance, with keys following a Zipfian distribution parameterized at 0.99.

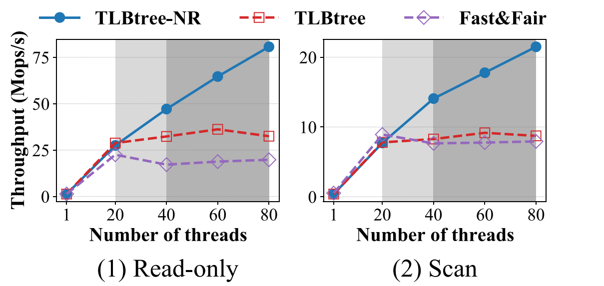

Fig. 2. Performance gain of reducing the NUMA effect: TLBtree-NR uses replications across NUMA nodes to reduce the NUMA effect, showing much higher performance than TLBtree and Fast&Fair, which incur NUMA effects with the increase of threads.

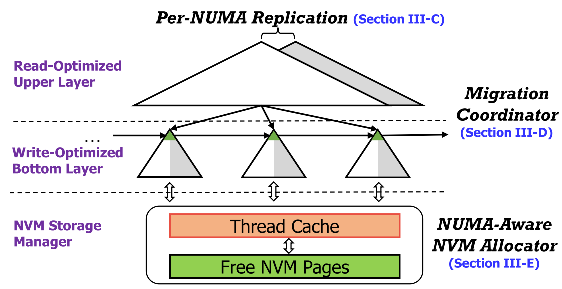

Fig. 3. Structure of NOBtree.

As shown in Fig. 2, TLBtree and Fast&Fair struggle to scale beyond a single node as thread count increases beyond 20. Conversely, TLBtree-NR consistently demonstrates robust scalability even though threads are distributed across the two sockets. The main reason for the poor scalability of TLBtree and Fast&Fair is the limited bandwidth of UPI and the cache coherency mapping cost across sockets. These results underscore the substantial NUMA impact on NVM index performance, revealing significant potential for enhancement. Therefore, there is a need to design a NUMA-optimized NVM index.

B. Overall Structure of NOBtree

Fig. 3 illustrates the overall structure of NOBtree. NOBtree includes two layers: 1) a read-optimized upper layer and 2) a write-optimized bottom layer. The upper layer is a read-optimized static structure used to rapidly locate the target subindex in the bottom layer during a read/write operation. The bottom layer is a series of write-optimized subindexes linked by pointers. The rationale behind this design is rooted in the observation that B+-tree nodes closer to the root are read-dominated, while write operations (insertion/deletion) primarily affect the bottom levels. The structure of these two layers has been designed to adapt to the features of the NVM. Moreover, both the read-optimized upper layer and the write-optimized bottom layer are designed to reduce the NUMA effect (the details will be described below).

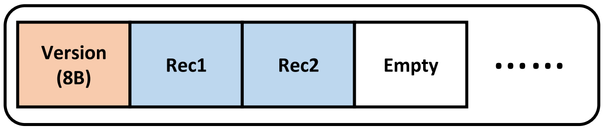

Fig. 4. Leaf-node structure in the upper layer.

1) Read-Optimized Upper Layer: The upper layer of NOBtree can take the form of any data structure capable of delivering high-search performance. In our current implementation, we opt for a customized k-ary tree-like index to serve as the upper layer. This layer is designed as a static structure with no structural modifications, utilizing a contiguous node array and eliminating all pointers. The main features of the upper layer are *static structure*, *read-optimized node layout*, and *per-NUMA replication*.

Static Structure: The static structure means there is no structural modification in the upper layer. We employ this design mainly for two reasons. First, the static structure is beneficial for read operations. Using the static structure means that we can neglect the write operations and employ a fully read-optimized index layout. Second, according to the previous work [6], write operations are rare in this layer. That is because the write operations in a B+-tree's inner node are triggered by the node split occurring in the bottom layer. The number of write operations decreases exponentially from bottom to top. Thus, it is reasonable to sacrifice the write performance for the read performance in the upper layer.

Insertions to the upper layer may fail if the target leaf node is full because there will not be any structural modifications. Such failures may lead to performance decline by lengthening the traversal path if the target subindex is not inserted into the upper layer. To address the performance issues stemming from a full and imbalanced upper layer, NOBtree will rebuild the upper layer once a predefined threshold is reached. The threshold will influence the frequency of reconstructions. It's crucial to strike a balance with this threshold because if rebuilding occurs too frequently, it can degrade NOBtree's performance due to the high cost of reconstruction. Therefore, we choose a well-tuned threshold based on the experimental result to achieve the highest performance. We will give a detailed description of the rebuilding process in Section IV-C.

Read-Optimized Node Layout: The upper layer of NOBtree plays a crucial role in swiftly locating the subindex, influencing the performance of both read and write operations. We utilize an array-based k-ary tree as the upper layer, categorizing nodes into immutable inner nodes and gapped leaf nodes. Inner nodes are ordered arrays and are always 100% full with no pointers. This design choice enables them to accommodate a maximum number of keys, thus lowering the tree height. The inner nodes are determined at the rebuilding time and will not change until the next reconstruction. Leaf nodes contain records with a key and a pointer to subindexes and employ a gapped structure. Fig. 4 shows the leaf-node structure. We reserve a few empty slots in each leaf node (preallocating some gaps when rebuilding the top layer). The empty slots in a leaf node can absorb new records generated by a subindex splitting of the bottom layer. The version is used for optimistic concurrency control like previous works [16]. In our implementation, the slot number is set to 15, and we reserve three empty slots when building the upper layer. Note that the size of inner nodes and leaf nodes is both 256B, aligning with Optane's internal page size, which can reduce the number of NVM accesses when traversing the tree. Moreover, all the nodes are allocated in contiguous memory space. Such a design can reduce the number of TLB misses during the operation process, leading to higher performance.

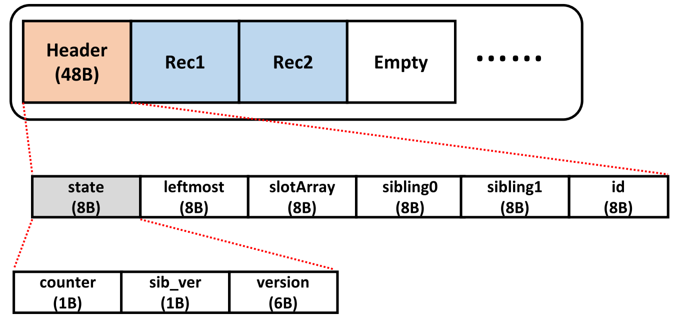

Fig. 5. Node structure in the bottom layer.

Per-NUMA Replication: While the read-optimized static structure of the upper layer can offer the ability to rapidly locate the subindex. Threads from other sockets still experience reduced search performance due to remote NVM access if the upper layer resides in only one NUMA node. To tackle this issue, we propose replicating the upper layer across all NUMA nodes. During an index operation, each thread accesses the upper layer on its local socket, effectively reducing the costly remote NVM access. We will give a detailed description in Section III-C.

2) Write-Optimized Bottom Layer: The bottom layer employs a group of subindexes, as illustrated in Fig. 3, which are indexed by the upper layer. The subindex is a write-optimized persistent B+-tree. All the roots of the subindexes are linked via pointers. This linked structure enables the asynchronous update of the upper layer while ensuring correctness simultaneously. The subindex incorporates various techniques to adapt to NVM, including a node size optimized for NVM, physically unsorted but logically sorted keys, and a log-free splitting/merging mechanism. We will describe the design of the bottom layer next.

NVM-Friendly Structure: The bottom layer of NOBtree is tailored to handle the majority of insert operations, necessitating the creation of an NVM-friendly structure. Fig. 5 illustrates the node structure in the bottom layer. Each node contains a header and several slots to store records. The header contains a *state* field, a *leftmost* child pointer, a *slotArray*, two sibling pointers, and a node *id*. The *state* field contains a *counter*(1 byte), a *sib_ver*(1 byte) and a node *version*(6 bytes). The *leftmost* pointer points to the child node whose keys are all less than the current node's minimal key. It is valid only for the inner node. The *slotArray* records the keys' order information in the node. The two sibling pointers and the *sib_ver* are used to implement the log-free splitting/merging mechanism. The *counter* counts the total number of records in this node. The node *id* is used for node migration. It is assigned when the node is created. The node *version* is used for concurrency control. We adopt optimistic concurrency control in our implementation. The header's size is less than 64 bytes, and the size of *state* is 8 bytes. NOBtree's bottom layer employs several techniques to improve the efficiency of write operations.

The node size is set to 256B, aligned with Optane's internal page size, ensuring at most one NVM access when traversing a node. In addition, the records within a node are physically unsorted but logically sorted. Such a design eliminates the need to shift records while inserting new keys. However, a completely unordered key arrangement harms searching and node-splitting operations. To cope with the problem, a *slotArray* is introduced to maintain the order information of the keys. For instance, $slotArray[3] = 4$ indicates that the third smallest key is stored in the fourth slot. In our implementation, each node comprises only 12 records, allowing us to represent the position information using just 4 bits. Therefore, we can embed the *slotArray* within an 8-byte field and update it atomically. This approach significantly improves search performance without incurring additional costly NVM writes.

Also, we implement a log-free splitting/merging mechanism using shadow sibling pointers, eliminating the logging overhead typically employed for ensuring crash consistency. This approach utilizes two pointers in one node to reference the sibling node. A parameter, denoted as *sib_ver*, indicates the functional pointer, i.e., the pointer pointing to the actual sibling node. During a node splitting, we initially allocate a new node and copy half of the records from the splitting node to this new node. Subsequently, we install the new node into the nonfunctional pointer. Finally, we visualize the new nodes by atomically updating the *sib_ver*. This strategy ensures crash consistency without resorting to the costly logging mechanism.

NUMA Optimization: The NVM-friendly structure enhances the insert performance of NOBtree. However, a notable performance degradation will occur when all subindexes are placed in only one NUMA node while access threads are distributed across multiple nodes. This is attributed to two main factors. First, half of the threads consistently undergo remote NVM access, experiencing a costly operation process. Second, previous work [9] reveals that reading the same address from multiple sockets (denoted as the *near-far* access pattern) achieves very low-NVM bandwidth due to cache coherency protocol. A naive solution to this problem is replicating the entire bottom layer across all NUMA nodes [2], enabling each thread to access the local bottom layer and thus reducing remote NVM access. However, this approach will introduce huge extra NVM consumption and synchronization costs between the replications.

To cope with the problem without introducing extra overhead, we propose randomly distributing the nodes in the subindex across all NUMA nodes, as depicted in Fig. 3. This approach helps avoid *near-far* accesses, thereby improving index performance. We also introduce a node migration mechanism for the bottom layer. NOBtree migrates nodes within a subindex to the socket that accesses them most frequently, thereby reducing remote NVM access. We will give a detailed description in Section III-D. It's worth noting that these designs necessitate specialized NVM allocation methods, e.g., round-robin allocation and allocation to specific NUMA nodes. However, existing NVM allocators do not support such allocation patterns. Therefore, we propose a new NVM storage manager to fulfil these requirements.

3) NVM Storage Manager: The NVM storage manager (NSM) is responsible for allocating NVM. NOBtree relies on a new NUMA-aware NVM allocator, as shown in Fig. 3. To optimize allocation performance, we adopt a two-layer structure to organize NVM storage: 1) a thread-cache layer and an 2) NVM-free-page layer. The thread-cache layer is used to minimize contention among concurrent NVM allocations, and the NVM-free-page layer manages all free NVM pages. The details of NSM will be discussed in Section III-E.

C. Per-NUMA Replication

The NUMA optimization in the upper layer is per-NUMA replication, where the upper layer of NOBtree is replicated in each NUMA node. Leveraging NOBtree's decoupled structure, it can tolerate inconsistency between the upper and bottom layers, allowing it to replicate the upper layer in each NUMA node. This design enables threads to access the upper layer on the local socket, thus mitigating the costly remote NVM access.

While the idea of NUMA NR is straightforward, it is challenging to synchronize the replicas across NUMA nodes. A common approach is to use a shared log for synchronization [2], [8]. However, relying on a log to replay operations to the upper layer introduces vulnerabilities to NOBtree. Since we use only one thread to process the log entry, the insertion of a newly split node into the upper layer might be delayed, especially during periods of high-splitting activity. While delayed updates to the upper layer do not compromise the correctness of subsequent operations, they may affect performance stability. If subsequent operations need to access the split node before inserting it into the upper layer, they must traverse the subindex list to locate the target subindex. Also, the log-based method requires extra NVM space for logs; for example, PACTree needs 500 MB of NVM space to store the log on each node. Considering the potential drawbacks, it is more practical to adopt a synchronous approach for insertions into the upper layer.

The second reason that we do not delay the insertion to the upper layer is that structural modifications in the upper layer are less costly than anticipated, making immediate insertion feasible. Hence, there's no need to employ the logging approach for delayed insertion. In summary, we utilize a synchronous approach to update the upper layer for the aforementioned reasons. Upon a subindex root node split, the new root node is inserted into the upper layers immediately.

D. Migration Coordinator

The NUMA optimization in the bottom layer mainly relies on the node migration mechanism, which can reduce the remote NVM access in the bottom layer. We propose a migration coordinator for the bottom layer to handle the node migration. This optimization is based on an observation: *each thread has different data-accessing localities*. Traditional evaluations often distribute the workload equally among all threads executing the task, assuming the same hotspot data across them. However, systems like MySQL and Microsoft SQL Server typically assign a single core to service requests from a single session, resulting in different threads having distinct hotspots. This distribution characteristic presents an opportunity to design a node migration mechanism for the bottom layer in NOBtree. Leveraging the diverse hotspots of each thread, we can migrate nodes in the bottom layer to the socket that accesses them most frequently, thereby reducing remote NVM access. The node migration in the bottom layer involves two primary steps: 1) identifying the node to be migrated and putting it into a global queue and 2) the migration coordinator periodically retrieves node information from the global queue to execute migrations in the background. Inspired by the producer-consumer model, the process of node migration utilizes a global queue to record information about nodes slated for migration. When an operation thread accesses a specific node, it checks whether the node requires migration. If so, its information will be put into the global queue. A dedicated thread (migration coordinator) periodically polls the global queue and executes node migrations. Note that the migration coordinator runs in the background, minimizing disruptions to normal operations. To implement the node migration mechanism correctly and efficiently, we have to address the following challenges.

Identifying the Node to Be Migrated: NOBtree utilizes historical access statistics to determine nodes eligible for migration in the bottom layer. It monitors access to all leaf nodes, identifying nodes meeting two criteria: 1) they are considered hot nodes and 2) most accesses are remote NVM accesses. A node qualifies as a hot node if its total accesses surpass a threshold within a monitoring window. Subsequently, we calculate the percentage of accesses from remote nodes. If remote access constitutes the majority, we add the node's information to the global queue. This information includes the node's address, its parent and previous sibling addresses, and the target migrating NUMA node. The migration coordinator fetches this information periodically from the global queue and performs the migration accordingly. Other methods of identifying hot data can also be employed, maintaining the core concept of node migration.

However, maintaining access statistics can pose a scalability bottleneck due to the need for frequent modifications, which incur additional writes [17]. Given the expensive cost of NVM writes, we utilize an in-DRAM hash table to record the access information rather than maintaining statistics within the nodes themselves. The key in the hash table corresponds to the node *id*, as discussed in Section III-B2. To further alleviate the impact on the performance, we implement a sampling method, recording access statistics every ten operations. These statistics include the total number of local and remote accesses, enabling us to calculate the node's 'hotness' and determine whether migration is necessary.

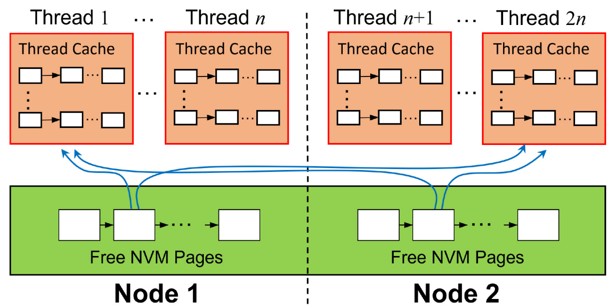

Fig. 6. Structure of the NSM.

Coordinating Migration With Other Operations: Coordinating migration with other operations is a key responsibility of the migration coordinator. Since node migration occurs in the background, structural modifications may occur in the migrating node, its parent, or its previous sibling node. If any structural modifications occur during migration, the migration process is aborted to maintain data integrity. During a migration, the node is first locked to block subsequent insertions. Then, we copy the node to the target NUMA node. Next, we update the node's parent and previous node to point to the new node. Note that we need to guarantee crash consistency during the two update operations. Because the two operations can not be done atomically, we use logs to ensure crash consistency. Finally, we mark the original node as obsolete and free it once no more readers access it. The migration does not block any reader during this process. We adopt an epoch-based reclamation strategy [18] to reclaim the obsolete node. Note that the migration relies on the dedicated NUMA-aware NVM allocator, which allows NVM allocation on specific NUMA nodes. In the subsequent section, we will present the details of the allocator.

E. NUMA-Aware NVM Allocator

NVM allocation profoundly affects the insert performance of indexes. Prior works [8], [11], [14] have demonstrated that different NVM allocators can result in a performance gap of 25% $\sim$ 50% for indexes. Thus, NOBtree presents a NUMA-aware NVM allocator to enhance the insert performance of the index.

NVM allocators are generally categorized into two types: 1) log-based [12], [13], [14] and 2) GC-based [11], [14], [15]. The former relies on logging to persist memory changes and heap metadata, ensuring the atomicity of operations in case of failure. However, this approach introduces additional persistence overhead. To mitigate this overhead, recent allocators utilize GC to rebuild heap metadata post-crash by scanning the entire memory space [15]. The GC-based allocators generally offer faster allocation speeds than the log-based allocators. Therefore, we adopt the GC-based NVM allocator.

Fig. 6 depicts the overall structure of the NSM, which consists of two main parts. The bottom part operates as an NVM space manager, responsible for managing the entire NVM space. Meanwhile, the upper part serves as the thread cache for NVM allocation and deallocation operations.

Algorithm 1: NVM Allocation

```
Function nvm_numa_malloc(size, node):
    if node < number_of_numa_node then
        // allocate to the given NUMA node
        return thread_cache.slab[node].slab_alloc(size);
    end
    // allocate NVM according to the pre-set mode
    return thread_cache.slab[mode].slab_alloc(size);
Function slab_alloc(size):
    size_class = get_size_class(size);
    address = freelist[size_class].pop();
    if !address then
        // get a NVM page according to the pre-set
            mode
        page = get_nvm_page(mode);
        fill the freelist with the requested NVM page;
        address = freelist[size_class].pop();
    end
    return address;
Function get_nvm_page(i):
    if i < number_of_numa_node then
        return NVM page on NUMA node i;
    else if i == LOCAL then
        numa_node = get_numa_node();
        return NVM page on NUMA node numa_node;
    else
        return NVM page in a round-robin way;
```

Algorithm 2: Lookup Operation

```
1  Function Lookup(key):
2      num_node = get_numa();
       // identify current thread's NUMA node
3      bottom_root = uptree[numa_node].Up_Lookup(key);
4      while bottom_root.max_key < key do
5          bottom_root = bottom_root.get_sib();
6      end
7      return Bottom_Lookup(bottom_root, key);
8  Function Bottom_Lookup(node, key):
9      cur = node;
10     while cur.leftmost != NULL do
11         cur = cur.get_child(key);
12     end
13     return cur.get_child(key);
14 Function get_child(key):
15     retry: old_version = node.version;
16     if key > node.get_sib().min_key then
17         return node.get_sib().get_child();
18     end
19     if leftmost != NULL then
           // search in the inner node
20         pos = get_pos(key);
21         if pos == 0 then
22             ret = leftmost;
23         else
24             ret = records[slotArray[pos − 1]].val;
25     else
           // search in the leaf node
26         pos = get_pos(key);
27         if records[pos].key != key then
28             ret = NULL;
29         else
30             ret = records[pos].val;
31     if node.version! = old_version then
32         goto retry;
33     end
34     return ret;
```

The bottom part manages the per-NUMA NVM pool, treating the NVM space as a free-page list with a default page size of 256KB. The thread cache comprises segregated lists, each containing blocks of the same size. Each block is used to serve a single allocation request. This two-layer structure helps alleviate contention for concurrent NVM allocations, resulting in high performance and scalability.

NSM offers three allocation modes: 1) *local* (allocating pages from the local node); 2) *round-robin* (allocating pages among NUMA nodes in a round-robin manner); and 3) *node-specific* (allocating pages from a specified node). Users initiate the mode during the creation of NSM. There are a certain number of slab managers inside a thread cache. Each contains segregated free lists and serves different modes of NVM allocation. Algorithm 1 shows the process of the NVM allocation. The allocation needs to provide two parameters, *size* and *node*. The slab manager responsible for the specified node is selected to allocate NVM. The slab manager first gets the size class of the request. Then, it retrieves an address from the corresponding free list based on the size class. If the free list returns an address successfully, the allocation is completed. Otherwise, the slab manager requests a free NVM page from the bottom NVM space manager, fills the free list with the page, and allocates memory from it.

In the NVM allocation process, thread contention arises primarily when threads request a page from the bottom NVM space manager. However, this contention is infrequent compared to the overall NVM allocations, allowing the NSM to deliver high-allocation performance. Additionally, only the page allocation requires persistence, while small allocations from the thread cache do not need to persist any metadata. These two features contribute to the high performance and scalability of NSM. Moreover, the three allocation modes provided by the NSM offer users greater flexibility in utilizing NVM.

IV. Operations of NOBtree

In this section, we describe NOBtree's operations, including lookup, insert, and the upper layer's rebuild. Due to the new structure and NUMA-aware designs of NOBtree, all these operations need to be redesigned.

A. Lookup

Algorithm 2 outlines the lookup process in NOBtree, which comprises two main steps: 1) searching in the upper layer to locate the subindex and 2) searching in the subindex to retrieve the desired result. We initiate the lookup process by selecting the local upper tree and then searching for the key within this upper tree to obtain the root of the subindex (lines 2 and 3). This search process in the upper layer is similar to the procedure employed in the traditional B+-tree. Starting from the root node, we recursively traverse the inner nodes until reaching the leaf node. Subsequently, we get the root of the target subindex. However, the obtained result may be incorrect due to subindex splits. To correct this, we traverse the linked list to get the true root of the target subindex (lines 4–6). Then we execute the lookup operation within this subindex and return the result (line 7).

The lookup process of the upper layer is similar to that in a traditional B+-tree. However, there are some distinctions due to the specific characteristics of the upper layer's structure. The inner node of NOBtree's upper layer is pointless to accommodate more keys. So when searching in the inner node, we will utilize a formula to calculate the position of the child node. This approach is feasible due to the static structure of the upper layer, where all nodes are allocated in a contiguous space. Consequently, the offset of each node remains fixed, and we can determine it by calculating the offset. Furthermore, the structure of the upper layer remains unchanged until we rebuild it, ensuring the reliability of this calculation method.

Similarly, the lookup process of the bottom layer, outlined in Algorithm 2 (lines 8–34), comprises two main steps: 1) locating the leaf node and 2) conducting a local search within the leaf. When probing a node, the process begins by snapshotting the version (line 15). Subsequently, we verify whether the node has undergone a split. If a split has occurred before accessing the node, we switch to its next sibling node (lines 16–18). If the node is an inner node, we find the first position where the key exceeds the target key. This search is facilitated using the *slotArray* (lines 19–24). Conversely, if the node is a leaf node, we identify the first position where the key is greater than or equal to the target key. We then check if the key matches the target key (lines 25–30). Finally, we verify the node's version. If it has changed during the search, the process is retried. Otherwise, the result is returned.

B. Insertion

The main steps of the insert operation are similar to those of the lookup operation. We begin by searching the local upper layer to locate the target subindex, followed by inserting the key into the subindex. Subsequently, we check whether we need to rebuild the upper layer. If reconstruction is necessary, it is executed in the background (lines 11–13). During insertion to the subindex, node splitting may occur in the subindex. If the root of the subindex splits and we need to insert the newly generated root into the upper layer, we will try to insert the new root into all upper layers (lines 14–17). If the insertion to the upper layer fails, we will store the new root in a temporary list (lines 18–20). We use the temporary list to accelerate the rebuilding process. Further details will be provided in Section IV-C.

Algorithm 4 offers a detailed overview of the insert operation within a node of subindexes. Initially, the node is locked to prevent subsequent insertions. If the node has been split just before the insertion attempt, we switch to the next node to perform the insertion (Algorithm 4, lines 3–6). If the node is full, a new node is created, and the keys are rearranged. Notably, the split operation is log-free due to the utilization of shadow sibling pointers. To ensure crash consistency, any updates to the node's header are followed by *CLWB* and *SFENCE* instructions. Node splits may propagate from the leaf to the root. If we find that the root of the subindex splits, we will check whether the height of the subindex exceeds the predefined threshold. If not, we will create a new root to accommodate the splitting node and update the upper layer to point to this new root (Algorithm 3, lines 25 and 26). Otherwise, we will insert the newly created node to the upper layer (Algorithm 3, lines 14–17).

Algorithm 3: Insert Operation

```
1  Function Insert(key, val):
2      num_node = get_numa();
3      bottom_root = uptree[numa_node].Up_Lookup(key);
4      query_time = query_time + 1;
5      while bottom_root.max_key < key do
6          bottom_root = bottom_root.get_sib();
7          goes_step = goes_step + 1;
8      end
9      res = Bottom_Insert(bottom_root, key, val);
10     average_goes_step = goes_step/query_time;
11     if average_goes_step > threshold then
12         rebuild_upper();
13     end
14     if res.flag = TRUE then
           // the root of the subindex splits
15         for i = 0; i < num_numa; i++ do
16             succ = uptree[i].Up_Insert(res.key, res.val);
17         end
18         if succ = FALSE then
               // failed to insert to upper layer
19             mutable.append(res.key, res.val);
20         end
21     end
22 Function Bottom_Insert(node, key, val):
23     res = recursive_insert(node, key, val);
24     if res.is_split then
           // node split spreads to the root
25         if res.level < threshold then
26             create a new root and update the upper layer;
27         else
28             return {TRUE, res.split_k, res.split_node};
29     end
30     return {FALSE, 0, NULL};
31 Function recursive_insert(node, key, val):
32     if node.leftmost != NULL then
33         child = node.get_child(key);
34         res = recursive_insert(child, key, val);
35         if res.is_split then
36             return node.store(res.split_k, res.split_node)
37         return {node.level, FALSE, 0, NULL};
38     else
39         return node.store(key, val)
```

C. Rebuilding the Upper Layer

Rebuilding the upper layer starts with collecting all the roots of subindexes. Then, based on the number of records, we calculate the number of leaf nodes and the tree height. Subsequently, we fill the leaf nodes with the records and recursively construct inner nodes until the root is created.

To mitigate the overhead of rebuilding the upper layer, we implement several optimizations to reduce the frequency and associated costs of this process, thus enhancing the index's stability and efficiency.

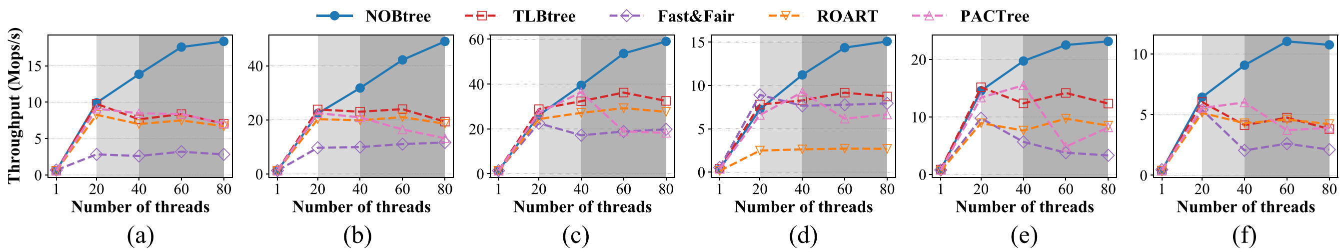

Fig. 7. Throughput under various workloads. (a) Write-heavy. (b) Read-heavy. (c) Read-only. (d) Scan. (e) Read–Modify–Write. (f) Write-only.

Algorithm 4: Node Insert Operation

```
Function store(key, val):
    Lock();
    if key > node.get_sib().min_key then
        Unlock();
        return node.get_sib().store(key, val);
    end
    if count == split_threshold then
        // node needs to split
        split_node ← create new node;
        split_k ← the middle key;
        rearrange the keys in the old node and insert key;
        update the header of split_node;
        clwb();sfence();
        update the header of current node;
        clwb();sfence();
        Unlock();
        return {level, TRUE, split_k, split_node};
    else
        insert the key;
        Unlock();
        return {level, FALSE, 0, NULL};
```

First, although the upper layer is static, its leaf node reserves some empty slots to absorb limited insertions. Such a design can effectively reduce the total number of rebuilds. Second, instead of initiating a rebuilding process immediately after a failed insertion, we defer this action until the average probing length in the linked list exceeds a predefined threshold. As shown in Algorithm 3 (lines 10–13), we use the parameter *average_goes_step* to record the average probing length along the linked list. Once this parameter exceeds the predefined threshold, indicating a significant performance degradation, the upper layer is rebuilt. This approach ensures that rebuilding is only triggered when failed insertions noticeably impact performance. As a result, the frequency of rebuilding occurrences is reduced. In our evaluation, we observed less than 20 reconstructions after inserting 200 million keys.

V. Performance Evaluation

We conduct experiments on a 2-socket server equipped with two Intel Xeon Gold 6242R CPUs, each featuring 20 cores. The server contains 256-GB DRAM and 2048-GB Intel Optane DC persistent memory, equally distributed over two sockets. We configure all Optane modules to `App-Direct` mode and create a DAX-aware ext4 file system. Then, we mount the file system using the DAX option.

Competitors: We conduct a comparative evaluation of NOBtree against four NVM-oriented indexes: 1) Fast&Fair [5]; 2) PACtree [8]; 3) ROART [11]; and 4) TLBtree [6]. We leverage their respective open-source codes for our assessment. It is noteworthy that PACtree also incorporates measures to mitigate the NUMA effect. Nap is not considered in the comparison as it is a black-box approach applicable to any index, which is orthogonal to our design, and its use of buffers may result in performance degradation for scan workloads.

Workloads: The dataset consists of 200M randomly generated integers (8 bytes long). There are mainly six kinds of YCSB-like workloads, including *read-only* (100% lookup), *read-heavy* (95% lookup and 5% insert), *write-heavy* (50% lookup and 50% insert), *write-only* (100% insert), *read–modify–write* (50% lookup and 50% read–modify–write), and *scan* (100% scan). The YCSB is an open-source benchmarking suite for evaluating the maintenance and retrieval capabilities of computer programs [19]. It is widely used to compare the performance of database systems and database indexes. We made slight modifications to its core workload generator by replacing all update operations with insert operations and then generated the experimental workloads. In these workloads, all the lookup and scan operations follow a Zipfian distribution with a parameter of 0.99. The experiments scale from 1 thread to 80 threads. When the number of threads is less than 20, they are assigned to a single node. However, if the thread number exceeds 20, threads are distributed across two nodes, and hyper-threading is enabled when exceeding 40.

A. Throughput

In this experiment, we evaluate the throughput of each index under various workloads. Fig. 7 shows the throughput of NOBtree and other indexes under the six workloads. Across the six workloads, NOBtree consistently outperforms other indexes. Specifically, under 80 threads, NOBtree achieves 3.10×, 3.13×, 2.41×, 2.32×, 2.86×, and 3.06× higher throughput compared to other indexes on average under the six workloads, respectively. The high-read-throughput of NOBtree is attributed to the per-NUMA upper-layer replication and the node migration mechanism in the bottom layer, which can reduce the costly remote NVM accesses effectively. Additionally, the dedicated NUMA-aware NVM allocator contributes to NOBtree's superior insert performance. Although rebuilding the upper layer is time-consuming, the rebuilding frequency is rather low because we have reserved some empty

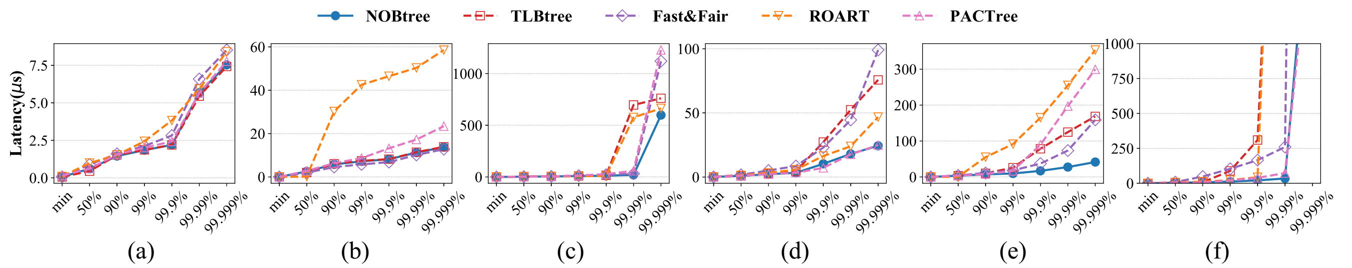

Fig. 8. Tail latency under skewed workloads (20t: 20 threads, 40t: 40 threads). (a) Lookup(20t). (b) Scan(20t). (c) Insert(20t). (d) Lookup(40t). (e) Scan(40t). (f) Insert(40t).

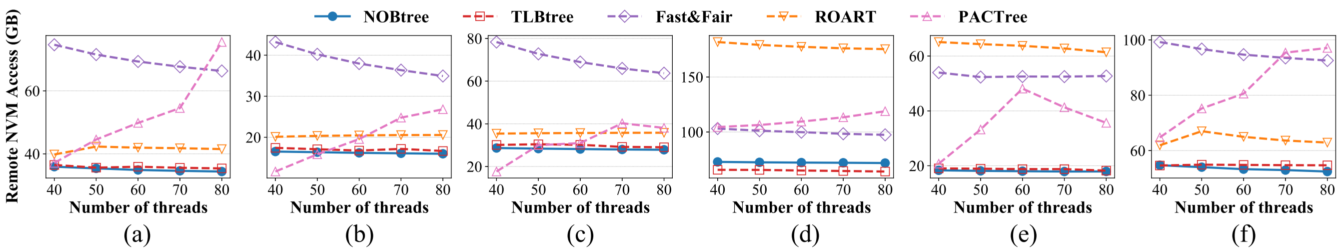

Fig. 9. Remote NVM access under various workloads. (a) Write-heavy. (b) Read-heavy. (c) Read-only. (d) Scan. (e) Read–Modify–Write. (f) Write-only.

slots to absorb insertions. We have tuned the threshold to reduce the rebuilding time, which improves the performance.

Despite PACtree's efforts to address the NUMA effect, its performance struggles to scale effectively, especially beyond 40 threads. In particular, PACtree's performance on the read-only workload experiences a sharp decline, ultimately ranking as the least efficient index. This decline could be attributed to the influence of hyper-threading and cache coherence protocols. We measure the data volume of the remote access in Section V-C and find that PACtree experiences an increasing volume of remote NVM access when the thread number increases from 40 to 80. This is the main reason for PACtree's degraded performance. PACtree recommends using snooping protocols for NVM, but most platforms use a directory-based protocol by default because snooping may not scale well to more CPU cores. The server in our experiments adopts the directory-based protocol and does not allow changes to coherence protocols, potentially contributing to the suboptimal performance of PACtree.

B. Tail Latency

In this section, we assess the tail latency of the index's lookup, scan, and insert operations under 20 and 40 threads. As shown in Fig. 8, NOBtree consistently exhibits the lowest-tail latency across most cases and achieves 1.46×, 2.59×, and 8.25× lower latency than other indexes on average under the three workloads, respectively. Notably, for the scan operation, Fast&Fair outperforms others under 20 threads, attributed to its larger leaf node size compared to NOBtree, which is particularly advantageous for scan operations. However, under 40 threads, the impact of the NUMA effect leads to increased tail latency for all indexes across all workloads. In this scenario, NOBtree maintains the lowest-tail latency across all three workloads, providing further evidence of its superior performance.

C. Remote NVM Access

In this section, we measure the data traffic caused by remote NVM accesses. We utilized the Intel Performance Counter Monitor (Intel PCM)[1] to quantify remote NVM access during runtime. Fig. 9 presents the results for six workloads, where we varied the thread number from 40 to 80 and distributed threads across two sockets. NOBtree consistently achieves less remote NVM access amount than TLBtree except the *scan* workload. This can be owing to the replication of the upper layer and the node migration. Under the *scan* workload, NOBtree's smaller leaf nodes incur more remote NVM access compared to TLBtree. Since multiple nodes need to be scanned to obtain results, smaller node sizes lead to scanning more nodes. Nevertheless, as shown in Fig. 7, NOBtree still outperforms TLBtree in throughput under the *scan* workload because TLBtree places data on a single socket, causing threads on the other socket to consistently access remote NVM. In addition, Fast&Fair experiences a significant amount of remote NVM access, which is primarily due to its large node size.

D. Throughput Under Uniform Workloads

In this section, we will evaluate all indexes under uniform workloads. We omit the *write-heavy* and *write-only* workloads as the insert operation is always randomly distributed. As shown in Fig. 10, the performance of all the indexes degrades compared with those under skewed workloads. However, NOBtree's performance still outperforms others, indicating the efficiency of NOBtree under uniform workloads, which is mainly owing to the per-NUMA upper-layer replication as it can effectively reduce costly remote NVM access.

[1]https://github.com/intel/pcm

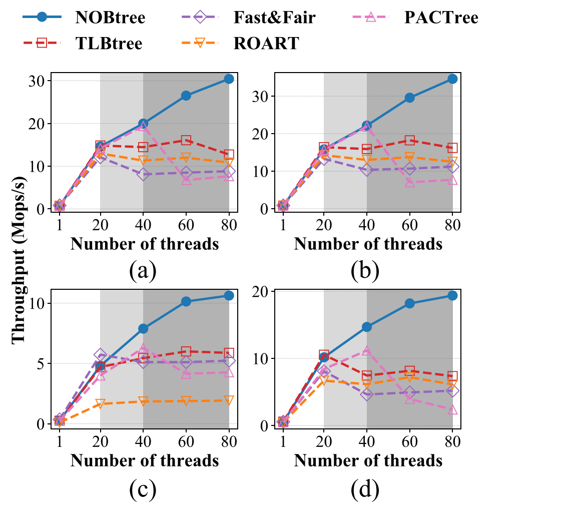

Fig. 10. Throughput under uniform workloads. (a) Read-heavy. (b) Read-only. (c) Scan. (d) Read–Modify–Write.

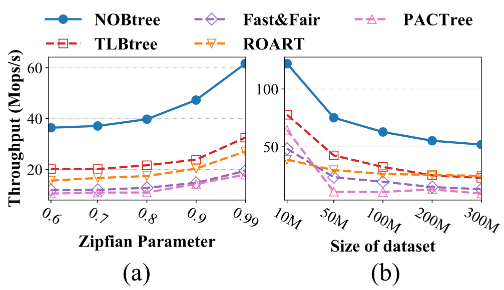

Fig. 11. Impact of workload skewness and data volume. (a) Impact of Skewness. (b) Impact of Data Volume.

E. Sensitivity Analysis

Impact of Workload Skewness: The skewness of the workloads reflects the deviation of query keys toward hotkeys, a characteristic often described by the Zipf parameter in the Zipfian distribution. In this experiment, we first load 100M keys and then perform 200M lookup operations. We vary the Zipf parameter from 0.6 to 0.99. Fig. 11(a) shows the throughput under 80 threads. One can see that all the indexes' performance improves with the increase of skewness. The performance of all indexes improves with increasing skewness. This improvement can be attributed mainly to the effect of the CPU cache. Higher skewness implies that most queries concentrate on a small set of keys, enabling the index to utilize the CPU cache more efficiently. NOBtree achieves the highest performance across all skewness, indicating its stable performance for different access patterns.

Impact of Data Volume: In this experiment, we explore the impact of dataset volume on the index's performance. We load several keys to construct the index and then evaluate the lookup performance under 80 threads. The lookup operations follow the Zipfian distribution with a parameter of 0.99. The amount of keys varies from 10 M to 300 M. Fig. 11(b) illustrates the results. The performance of all indexes degrades as the data volume increases. This decline occurs because more keys will heighten the indexes, leading to longer traversal paths during operations. NOBtree consistently outperforms other indexes.

TABLE I
COMPARISON OF NVM CONSUMPTION (GB)

NOBtree	TLBtree	Fast&Fair	PACtree	ROART
6.29	5.78	5.12	6.32	25.03

F. Space Cost

The NVM consumption of each index is shown in Table I. In this experiment, we measure the NVM usage of each index after loading 200 million records with 8-byte keys and values. Fast&Fair demonstrates the lowest-NVM consumption, mainly attributed to its larger node size. NOBtree consumes more NVM spaces than TLBtree, primarily due to two reasons. First, we reduce the number of keys that a single node can store to allocate space for maintaining migration statistics, which leads to more nodes after the key loading. Second, NOBtree replicates the upper layer across NUMA nodes, introducing additional NVM consumption. The size of the NOBtree's upper layer is only about 34 MB, which is relatively small compared to the total index size. Thus, it is possible to put the upper layer in a faster DRAM to further accelerate the performance of NOBtree.

VI. FURTHER DISCUSSIONS

As of July 2022, Intel decided to discontinue its Optane product [3]. It is a major setback for NVM-related research. However, the NVM technology has been regarded as an efficient solution to partially address the storage wall issue, and NVM-oriented structures and algorithms are needed in the storage architecture involving NVM. So far, except for the 3D-XPoint technology used by the Optane series, there are other technologies for implementing NVM, such as phase change memory (PCM), spin-transfer torque magnetoresistive RAM (STT-MRAM), and resistive RAM (RRAM) [20]. A recent report showed that the total sales of STT-MRAM might reach 98.3 billion dollars by 2033 [21]. Thus, there is still an urgent demand to study efficient structures optimized for NVM devices. There are similarities between Optane and other NVM devices, such as nonvolatility, read/write asymmetry, and byte addressability. Therefore, current research related to Optane can still inspire future work.

In addition, the growing demand for larger memory capacity has led to the development of new memory technologies like compute eXpress link (CXL). CXL provides a cache-coherent interface for connecting CPUs, memory, and accelerators. CXL-attached memory exhibits characteristics similar to NVM, such as byte addressability and near-DRAM performance with higher capacities [3], [22]. CXL-attached memory can be viewed as remote NUMA memory, and most current research on CXL memory is based on emulating memory in a remote NUMA node [22]. The proposed approach in this study can also be applied to indexes for CXL-attached memory in the future, given the similarity between remote NUMA memory and NVM.

VII. Conclusion

In this article, we presented NOBtree, a new index structure designed to mitigate the NUMA effect in NVM indexes. NOBtree employs a decoupled tree structure, which consists of a read-optimized upper layer and a write-optimized bottom layer to enhance both read and write performance. To improve the read performance in the NUMA architecture, we proposed per-NUMA replication for the upper layer and a node migration mechanism for the bottom layer. Additionally, we devised a dedicated NUMA-aware NVM allocator to optimize the insertion performance of NOBtree. The experimental results across diverse workloads suggested the effectiveness and efficiency of NOBtree.

References

[1] Q. Wang, Y. Lu, J. Li, and J. Shu, "Nap: A black-box approach to NUMA-aware persistent memory indexes," in *Proc. OSDI*, 2021, pp. 93–111.

[2] I. Calciu, S. Sen, M. Balakrishnan, and M. K. Aguilera, "Black-box concurrent data structures for NUMA architectures," in *Proc. ASPLOS*, 2017, pp. 207–221.

[3] D. Koutsoukos, R. Bhartia, M. Friedman, A. Klimovic, and G. Alonso, "NVM: Is it not very meaningful for databases?" *Proc. VLDB Endow.*, vol. 16, no. 10, pp. 2444–2457, 2023.

[4] (Intel Corp., Santa Clara, CA, USA). *Intel Optane Technology*. 2020. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html

[5] D. Hwang, W. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in byte-addressable persistent B+-tree," in *Proc. FAST*, 2018, pp. 187–200.

[6] 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 Trans. Embed. Comput. Syst.*, vol. 20, no. 5s, pp. 1–25, 2021.

[7] B. Zhang, S. Zheng, Z. Qi, and L. Huang, "NBTree: A lock-free PM-friendly persistent B+-tree for eADR-enabled PM systems," *Proc. VLDB Endow.*, vol. 15, no. 6, pp. 1187–1200, 2022.

[8] W. Kim, M. K. Ramanathan, X. Fu, S. Kashyap, and C. Min, "PACTree: A high performance persistent range index using PAC guidelines," in *Proc. SOSP*, 2021, pp. 424–439.

[9] B. Daase, L. J. Bollmeier, L. Benson, and T. Rabl, "Maximizing persistent memory bandwidth utilization for OLAP workloads," in *Proc. SIGMOD*, 2021, pp. 339–351.

[10] Y. Luo, P. Jin, Q. Zhang, and B. Cheng, "TLBtree: A read/write-optimized tree index for non-volatile memory," in *Proc. ICDE*, 2021, pp. 1889–1894.

[11] S. Ma et al., "ROART: Range-query optimized persistent ART," in *Proc. FAST*, 2021, pp. 1–16.

[12] I. Oukid, D. Booss, A. Lespinasse, W. Lehner, T. Willhalm, and G. Gomes, "Memory management techniques for large-scale persistent-main-memory systems," *Proc. VLDB Endow.*, vol. 10, no. 11, pp. 1166–1177, 2017.

[13] A. Demeri, W. Kim, M. K. Ramanathan, J. Kim, M. Ismail, and C. Min, "Poseidon: Safe, fast and scalable persistent memory allocator," in *Proc. 21st Int. Middlew. Conf.*, 2020, pp. 207–220.

[14] Z. Dang et al., "NVAlloc: Rethinking heap metadata management in persistent memory allocators," in *Proc. ASPLOS*, 2022, pp. 115–127.

[15] W. Cai, H. Wen, H. A. Beadle, C. Kjellqvist, M. Hedayati, and M. L. Scott, "Understanding and optimizing persistent memory allocation," in *Proc. ISMM*, 2020, pp. 60–73.

[16] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 183–196.

[17] J. Ge et al., "SALI: A scalable adaptive learned index framework based on probability models," *Proc. ACM Manag. Data*, vol. 1, no. 4, pp. 1–25, 2023.

[18] R. Binna, E. Zangerle, M. Pichl, G. Specht, and V. Leis, "HOT: A height optimized Trie index for main-memory database systems," in *Proc. SIGMOD*, 2018, pp. 521–534.

[19] 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.

[20] A. Chen, "A review of emerging non-volatile memory (NVM) technologies and applications," *Solid-State Electron.*, vol. 125, pp. 25–38, Nov. 2016.

[21] C. Tom and H. Jim, *Emerging Memories Branch Out*, Santa Clara, CA, USA, Storage Netw. Ind. Assoc., 2023.

[22] Y. Sun et al., "Demystifying CXL memory with genuine CXL-ready systems and devices," in *Proc. MICRO*, 2023, pp. 105–121.