A Performance Study of Optane Persistent Memory: From Indexing Data Structures' Perspective

Abdullah Al Raqibul Islam*, Anirudh Narayanan†, Christopher York‡ and Dong Dai§
Department of Computer Science, University of North Carolina at Charlotte, NC, USA
Email: *aislam6@uncc.edu, †anaraya7@uncc.edu, ‡cyork5@uncc.edu, §dong.dai@uncc.edu

***Abstract*—In this paper, we study the performance of new Intel Optane DC Persistent Memory (Optane DC) from indexing data structures' perspective. Different from existing Optane DC benchmark studies, which focus on either low-level memory accesses or high-level holistic system evaluations, we work on the data structures level, benchmarking commonly seen indexing data structures such as *Linkedlist*, *Hashtable*, *Skiplist*, and *Trees* under various running modes and settings. We believe that indexing data structures are basic and necessary building blocks of various real-world applications. Hence, the accurate knowledge about their performance characteristics on Optane DC will directly help developers design and implement their persistent applications. To conduct these performance evaluations, we implemented `pmemids_bench`, a benchmark suite that includes *seven* commonly used indexing data structures implemented in *four* persistent modes and *four* parallel modes. Through extensive evaluations on real Optane DC-based platform under different workloads, we identify seven observations that cover various aspects of Optane DC programming. These observations contani some unique results on how different data structures will be affected by Optane DC, providing useful reference for developers to design their persistent applications.**

I. Introduction

Non-volatile memory (NVM) or persistent memory (PMEM) is a new kind of memory device that provides both near-DRAM data access latency and data persistence capability [1]. Different from block-based devices, PMEM can be directly accessed in bytes through the memory bus using CPU `load` and `store` instruction without using block-based interfaces. Due to its high density, low cost, and near-zero standby power cost [2], [3], [4], PMEM devices have been considered as a promising part of the next generation memory hierarchy. Among all the persistent memory solutions, the Intel Optane DC Persistent Memory (or Optane DC for short) has become the first commercially available PMEM device on the market in 2019 [5].

As the NVM device (e.g., Optane DC) becomes available, software developers start to consider porting their applications to persistent memory. However, to make it work efficiently, they need to have an accurate expectation of their applications performance on PMEM, as well as know how to re-design their applications to achieve the best performance.

Recently, one of the authors plans to port an application that operates graph-structured datasets to Optane DC. The author knows that there are multiple data structures that can be used to store graph-structured datasets, such as *Edge List*, *Adjacency List*, *Compressed Sparse Row*, or *Trees*; and for each data structure, there are multiple implementation choices [6], [7], [8]. Without knowing how each of these solutions will perform on Optane DC, it is hard for him to make the design decisions. In addition, the Optane DC/DRAM hybrid memory system provides even more choices: he may store part of the datasets in Optane DC for persistence and part in DRAM for performance. These options further complicate the programming and require developers to have a deep understanding about how Optane DC and how their applications with different designs will perform.

There are already several timely benchmark studies on Optane DC recently, aiming to provide such information [9], [10], [11], [12], [13]. However, their results are hard for developers to use mostly because of the granularity they worked on. The low-level device performance benchmarks [10], [11], [13], which focus on evaluating the memory access latency and bandwidth of Optane DC, are hard to be linked back to the performance of users' applications, which are typically built using data structures instead of individual Optane DC read/write access. On the other hand, the high-level performance benchmarks [9], [12], which essentially evaluate the holistic storage systems on Optane DC, do include too many system components and factors in their results, making it hard for application developers to reason the performance of their own implementations.

The challenge here is that, the developers need in-depth insights about the potential performance of their applications on Optane DC under different design choices, but existing benchmarks are either too high-level or too low-level, providing limited help. To bridge this gap, in this study, we propose to benchmark the Optane DC performance in the middle level, i.e., the data structures level, to help developers program their applications on the new PMEM devices.

We further limit the data structures to be indexing data structures, which are widely used in various applications for locating and accessing data elements. Typical examples are *Arraylist*, *Linkedlist*, *Hashtable*, *Skiplist*, and *Trees*, etc. We benchmark the indexing data structures for two reasons: 1) they are the basic and common units of most data-intensive applications, not only for storing the data but also for processing the data; 2) their accesses usually take a majority portion of the applications' memory accesses. Hence, knowing the performance of these commonly seen indexing data structures on Optane DC could help developers have a better picture of their applications performance and potentially lead to better system designs.

Specifically, in this study, we implement a new PMEM

Indexing Data Structure Benchmark (`pmemids_bench`) to benchmark commonly seen indexing data structures on Optane DC under different design choices. Apart from the existing benchmarks, `pmemids_bench` aims to help developers who are not necessarily experts in nor interested in building basal storage systems on Optane DC, but want to leverage Optane DC to improve the performance of their applications.

This goal leads to specific designs in `pmemids_bench`. First, we limit our focus to the commonly seen indexing data structures, such as *Hashtable*, *Linkedlist*, and *Trees*. We argue that if the developers need to use specific, advanced indexing data structures (e.g., LSM-tree [14]), they might prefer to use the holistic PMEM-aware storage system built based on that data structure (e.g., SLM-DB [15]) instead of implementing one manually. For such cases, the benchmark and experimental results from existing literature will be helpful for them to make the design decisions.

Second, we emphasize more on benchmarking the indexing data structures under different settings, such as: a) storing on Optane DC or DRAM; b) forcing persistence or not; c) supporting transactions or not; d) using multi-threads or not. We understand that in the real-world, developers might be constrained on how to program the persistent memory, for example, the data accessing concurrency might be limited by other parts of the application. So they will need to know the performance of the data structures in widely different settings and hence make more accurate design decisions based on their needs.

Third, instead of expecting developers to program Optane DC using low-level APIs such as cache line flushing, memory fence, or their own transaction mechanism, we focus on the case where developers use high-level programming libraries, such as the Intel PMDK library [16], to develop their applications. The high-level libraries are much easier to use and provide a better guarantee on the code correctness and data consistency. We expect they will be the de facto choice for application developers in the near future, hence benchmarking their performance would be useful to application developers.

In summary, our goal is to benchmark common indexing data structures in various settings on Optane DC to provide application developers useful insights about the potential performance of their applications on Optane DC. Besides, we also try to make the benchmark extensible by introducing standard interfaces and workload generators similar to YCSB (Yahoo! Cloud Serving Benchmark) [17]. This allows others to reuse our framework and implement their specific data structures for performance comparison. We summarize our contributions as follow:

- To the best of our knowledge, we conduct the first Optane DC performance study from the basic and commonly used *indexing data structure's perspective*. The collected insights aim to help application developers design and implement their applications.
- Through extensive evaluations, we concluded seven observations about the performance characteristics of indexing data structures on Optane DC, covering read/write performance of various persistent modes, transaction overheads, lock overheads, and non-local memory access performance, etc. We also discussed how these observations can be leveraged in application development.
- We collected our implementation and presented them as a complete benchmark suite (i.e., `pmemids_bench` [18]) to enable more standardized indexing data structures evaluations in the future.

The remainder of this paper is organized as follows: In Section II we introduce the basis of Optane persistent memory and its programming models. In Section III, we discuss a motivation example. In Section IV, we present the detailed design and implementation of `pmemids_bench` and the rationale of the designs. We discuss the evaluation results and the key insights in Section V, related work in Section VI, and the conclusions and future work in Section VII.

II. Persistent Memory and PMDK

In this section, we introduce the background of Intel Optane DC Persistent Memory technologies [9] and how to program Optane DC using the Intel PMDK library [16].

A. Optane Persistent Memory

Non-volatile memory (NVM) is a board concept. It consists of a set of technologies, such as PCM [19], [20], ReRAM [21], and STT-RAM [22], which use resistive memory to store data in a persistent way. Among them, the byte-addressable ones have been considered as promising complements of DRAM for the next generation memory systems. Intel Optane DC Persistent Memory is the first commercially available byte-addressable NVM device. Working on Intel Cascade Lake platforms, Optane DC scales up to 6TB capacity in a single machine [10].

Although the exact storage media of Optane DC has not been public yet, we do know that it uses a non-standard DDR-T protocol and a small internal buffer to enable out-of-order commands and data transfer to address the long latency to Optane media [9]. Also, similar to DRAM, the data transfer between CPU and Optane DC also go through the cache line (in 64 bytes). And internally, the PMEM buffer communicates with Optane media in 256 bytes. This does mean consecutive requests to the same 256 bytes could have less latency. More details about Optane DC can be seen at [23].

B. Optane Programming Modes

Optane DC can be configured in either *Memory* mode or *App Direct* mode[16]. In *Memory* mode, the Optane devices are exposed as normal DRAM, while the DRAM becomes a transparent 'L4' cache to accelerate data accesses. This model does not guarantee data persistence nor allows direct access to PMEM. In *App Direct* mode, the Optane DC devices are directly exposed to users' applications, in parallel with DRAM. The existing file system data path is re-used to access the persisted data. Specifically, data accesses on Optane DC are through the *DAX-aware* file system, which transparently converts file operations to memory load/store operations. Since *App Direct*

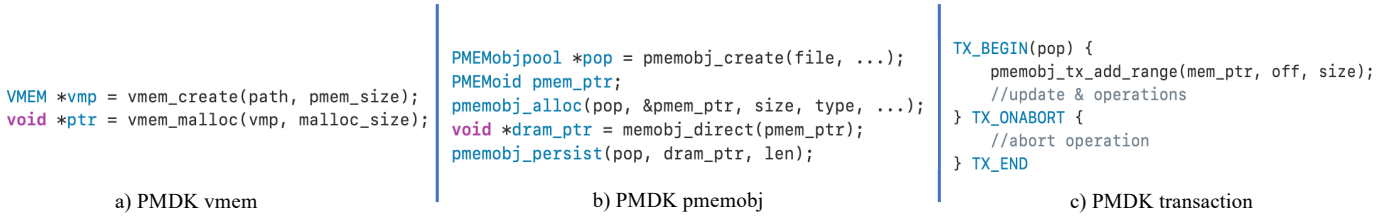

Fig. 1. Three different sets of PMDK programming APIs and typical usage patterns.

mode allows users to simultaneously access both Optane DC and DRAM and allows users to persist data as they need, it is then the more flexible way to program persistent memory. In this study, we focus on only this mode.

C. Program Optane DC Persistently

Optane DC devices are physically persistent, which means any write arrives at the memory subsystem of Optane DC will be guaranteed persisted. However, this does not mean calling `store` to a PMEM address will automatically guarantee data persistence. The reason is, the CPU caches, which are in the middle between CPU and Optane DC, are still volatile. So, the stored data might still be in the CPU caches. To make sure data is persisted, we need to call `CLFLUSHOPT` after `store` to flush the cache line to Optane DC. Also, modern CPUs reorder memory accesses for better performance. So, for two consecutive persistent memory writes, we may have the later one persisted before the first one, potentially breaking data consistency if failures happen in the middle. To avoid this, we need to call instructions such as `SFENCE` to enforce the order of multiple memory operations [24].

However, even with cache line flushed and memory fenced, writing a large chunk of data to Optane DC may still result in partially persisted data. This is because the atomic write unit of Optane DC is small (i.e., 8 bytes). So large writes will actually involve multiple atomic writes. If there are failures in the middle, data will be partially persisted. To guarantee data consistency for critical data structures larger than 8 bytes, we have to deploy transaction mechanisms such as logging (undo log and redo log) or copy-on-write (CoW) to protect these writes [25].

These extra measures for guaranteeing data persistence and safety complicate programming applications on Optane DC, but at the same time, also offer developers a wide range of flexibility to design their applications. For example, some auxiliary data structures on PMEM might do not need strong persistence or consistency, so that cache line flushing and transactions can be skipped to gain better performance. These different design choices lead to different performance characteristics of the Optane DC and its applications. Understanding their actual performance and the key factors under different workloads and system setting becomes critical. In this study, we developed `pmemids_bench` to help developers understand these characteristics.

D. Program Optane DC Using PMDK

The Persistent Memory Development Kit (PMDK) [16], formerly known as NVML, is a collection of libraries and tools, tuned and validated on both Linux and Windows for programming Optane DC. It builds on the DAX features of the operating systems, which essentially allows user applications to access persistent memory as memory-mapped files (Optane DC is operated in *App Direct* mode). PMDK provides different level of supports through its multiple libraries, shown in Fig 1.

Program Optane DC as DRAM. To gain the best performance out of Optane DC, developers can skip the persistence operations, such as cache line flushing and memory fencing when writing into Optane DC. In this case, the Optane DC is used as volatile memory and no data persistence is guaranteed. PMDK provides `libvmem` library for this use case. The basic APIs are shown in Fig. 1(a). Here, function `vmem_create` maps the Optane DC into the application's memory heap; then functions `vmem_malloc` and `vmem_free` are for the memory management, which internally leverages the `jemalloc` library [26]. Although there is no data persistence guarantee, we believe developers may still be interested in this model when they 1) hope to leverage the large capacity of Optane DC (up to 512GB per dimm); 2) hope to leverage the low standby energy consumption; or 3) have the data persistence be taken care of elsewhere.

Program Optane DC as Persistent Memory. This should be the most commonly used case for Optane DC. PMDK actually provides two set of APIs to developers for this purpose. The first one is `libpmem`. It provides low-level data persistent APIs, which are essentially an optimized wrapper of instructions such as `CLFLUSHOPT` and `CLWB`. The main limitation of this library, however, is it does not support dynamic memory management (such as malloc or free). The persistent memory management itself is challenging and will introduce extra overheads that the `libpmem` library wants to avoid. For example, each time the 'malloc-ed' memory must be persistently recorded by the library, so that if failure happens after the 'malloc', the library has a chance to free the unused memory to avoid memory leaking.

Due to the complexity of implementing a correct persistent memory management library, we expect developers who need dynamic memory management will not use `libpmem` directly. Hence, our focus in this study is on another PMDK persistent library: `libpmemobj`, which provides rich APIs to support various needs of application development.

We show example APIs in Fig. 1(b). The `libpmemobj` library provides a transactional object store, which includes useful facilities such as dynamic memory allocation, transactions support (via undo log and redo log), and persistent locks, etc. There are several important concepts worth discussing in the `libpmemobj` library. First, there are two kinds of pointers needed to operate an Optane DC address. One is `PMEMoid` and another is a normal pointer, such as `void *`. Any persistent pointer should be stored as `PMEMoid` and translated to a normal pointer (via `pmemobj_direct` function) before being used. So, there will be a translation overhead for any pointer access. Second, it is necessary to call `pmemobj_persist` to make sure a write to Optane DC is flushed successfully to the persistent domain. Such persistence operation will result in performance overheads as we mentioned earlier.

Program Optane with Transactions. The `libpmemobj` library provides transaction support to developers as shown in Fig. 1(c). The memory operations happened in the same transaction block (between `TX_BEGIN` and `TX_END`) will succeed or fail together. In users' applications, there are several places where transactions might be necessary. First, if users want to update continuous data that is larger than 8 bytes, such as updating a B-tree node in place, they should protect this update using a transaction, otherwise part of the update might be lost if failure happens. Second, if users need to update several distinct data in memory, for instance, updating several child pointers to re-balance a RB-tree, they also should protect these updates using a transaction, otherwise a failure in the middle may break the whole data structure.

Internally, PMDK uses undo/redo logs to implement transactions, which introduces overheads for copying data. So, developers might choose not to use PMDK transactions to get better performance. This choice might be acceptable if they can fix the partial persistence caused by system failures elsewhere. For instance, if all the data is already logged persistently, then incomplete operations can be fixed after application reboot using the logs. In this case, developers might skip PMDK transactions. For the same reason, in this study, we actually benchmark indexing data structures both with transactions and without transactions. When we do not use transaction, we simply assume developers themselves will take care of the data consistency.

III. Motivation Example

Recently, one of the authors was developing an application to process a graph-structured dataset. In this application, the dataset contains multi-millions of entities, each of which has a set of attributes. Entities have relationships, which also have attributes attached. The dataset is expected to be persistent, and will be continuously updated and queried by the application. Previously, the data was stored in SSD, and each time loaded and rebuilt into DRAM when application starts. Using Optane DC, we expect to persist the dataset and achieve high speed updates and queries at the same time. But, when we port the application to Optane DC, we noticed there are multiple data structures available, and it is hard to make the decision without knowing how each of them would perform on the Optane DC device.

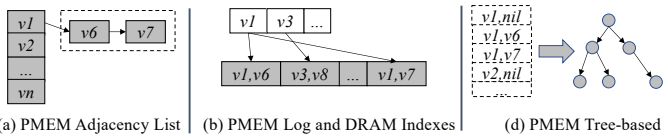

Fig. 2. Three exemplar design choices for storing graphs using Optane DC and DRAM. Here, we use gray color to indicate data is stored on Optane DC. We do not plot attributes of vertices and edges for simplicity.

In Fig. 2, we demonstrated three design choices for storing the graph-structured dataset. These designs are just examples out of large number of possible solutions.

Specifically, we can use *Adjacency List (on Optane DC)* to store the vertices in a list (typically a *hashtable* indexed by vertex Id) and the edges in multiple *linkedlists* (or a blocked linkedlist for better cache performance), attached to each source vertex. This design is demonstrated in Fig. 2(a).

We can also use log structure (such as a *Linkedlist (Optane DC)*) to persistently store all the edges and leverage an in-DRAM index to enable dynamic queries, as shown in Fig. 2(b). The logs are append-only, hence very efficient for updates. The DRAM index has multiple choices, such as *Hashtable*, *Skiplist*, *B-tree*, and *RB-tree*, etc. But, they should be consistent with data updates on Optane DC, and need to be rebuilt each time applications crash or reboot. Hence, both the performance of Optane DC log and DRAM index are important.

The third design we show here is to use tree data structures to directly store the edges on Optane DC, demonstrated in Fig. 2(c). We can consider an edge key as $[v_i, v_j]$ and the vertex (v) as a special case of edge ($[v,nil]$). We can leverage tree structures (i.e., *B+-tree*) that support `scan` operations to store the graph. Each tree node will store an edge. In this way, the common query such as finding neighbors can be well supported as scanning the tree. In this design, the performance of the Optane DC B+-tree structure is critically important.

Among these options, there are various indexing data structures (*Arraylist*, *Hashtable*, *Skiplist*, *B-tree*, *RB-tree*) on different memory devices (Optane DC and DRAM) involved. In fact, besides these high-level design choices, detailed implementation choices such as how many concurrent threads or other optimizations could also affect the performance significantly. Picking which solution would require developers to well understand the performance characteristics of Optane DC and their applications together. For instance, if PMEM Adjacency List will perform much worse with higher concurrency, then we may move to other data structures if our application does have high concurrency. However, this knowledge is still not widely available to application developers yet. And we want to solve this in `pmemids_bench`.

IV. Benchmark Design and Implementation

In Table. I, we first give a quick summary of what are included in `pmemids_bench`: the indexing data structures, their key implementation details, different persistent modes,

TABLE I
SUMMARY OF INDEXING DATA STRUCTURES, THEIR DIFFERENT RUNNING MODES UNDER VARIOUS WORKLOADS INCLUDED IN THE BENCHMARK SUITE.

Data Structure	Description	Persistent Modes	Parallel Modes	Workloads
Arraylist	Fixed items/size	DRAM	Single thread	A (100% Read) B (100% Write) (90% Update, 10% Insert)
Linkedlist	Appended only	PMEM-Volatile	Parallel, Saturated	
Hashtable	Chained Linkedlist	PMEM-Persist	Concurrent, Contention	
Skiplist	Fixed height, fair probability	PMEM-Trans	NUMA	
B-Tree	Values referenced/embedded			
B+-Tree	Values in the leaf nodes only			
RB-Tree	Instant rebalancing			

parallel settings, and various workloads. We will discuss them in more details in the following subsections.

A. Indexing Data Structures

In this study, we do not intend to identify the best persistent indexing data structures nor benchmark all possible indexing data structures. Instead, we focus on showing developers the performance they should expect from Optane DC when use commonly seen indexing data structures in their applications under various system settings.

As shown in Table I, in `pmemids_bench`, we implement seven commonly seen structures: *Arraylist*, *Linkedlist*, *Hashtable*, *Skiplist*, *B-tree*, *B+-tree*, and *RBTree*, and discuss their benchmarking results running in various settings. If needed, developers can easily extend `pmemids_bench` to include their new indexing data structures for benchmarking, which will be detailed in the later section.

Since each of these seven selected indexing data structures still has many variants, here we first discuss their key implementation aspects in `pmemids_bench`. More details can be found in our Github Repo [18].

First, we implement *Arraylist* as an unsorted array of fixed-size items. The array is allocated once at the beginning and will not dynamically grow. Since all items have fixed size, the array supports random access using item Id. Each time, a new item is inserted into a specific index of the array based on its Id. Finding an item is also a direct memory access. Each element in our arraylist is 128 bytes for storing the value. Such a persistent array is useful if developers know the size of their data ahead.

We implement *Linkedlist* as an unsorted list of elements linked using pointers. Each element is dynamically allocated when needed. Each time, the new element is appended at the end of the list. To locate an item from the unsorted linkedlist, we also implement an in-DRAM index (using *Hashtable* as described later) to fast locate the linkedlist element through an Id. Each element in our linkedlist is 156 bytes. The linkedlist is fit for implementing logs, especially if the logs need to grow continuously.

There are two ways of implementing *Hashtable*: open hashing or close hashing. The close hashing can not have more elements than the table slots, hence is limited in certain cases. In `pmemids_bench`, we implement open hashing, specifically the chained hashtable [27]. We allocate a fixed number of hash buckets at the beginning. The bucket number is determined by the *load factor* and *workload size* (total number of elements in the hashtable), both given in the benchmark configuration. The default load factor of our hashtable is 0.75, which offers a good trade-off between performance and space [28]. Each chain is implemented as an unsorted *Linkedlist*. The new element is inserted at the beginning of the chain each time. And checking whether an element is exist needs to scan the whole chain. Each element in hashtable is around 156 bytes.

We implement *Skiplist* using the fair probability (i.e. 0.5) for an element in the *i*-th layer to be shown in *(i+1)*-th layer [29]. The default maximal height is 16 and also configurable. Although there are other biased, deterministic and randomized *skiplist* [30], [31], we focus on the fair one as it is widely used in real systems, such as the in-memory key-value store Redis [32]. Each element in the skiplist takes around 256 bytes.

We implement *B-tree* and *B+-tree* in the similar way, except B-tree has both the keys and values stored in the internal nodes; while B+-tree has keys stored in the internal nodes and values stored in the leaf nodes. Previous research showed the size of tree nodes affects the cache behaviors and leads to different performance characteristics [33], [34], [35]. So, we implement two versions of B-tree: 1) the values are part of the tree nodes (*value embedded*); and 2) the values are stored separately and linked via a pointer in the tree node (*value referenced*). These two implementations lead to largely different tree node sizes. More specifically, the first version generates tree nodes around 2K bytes while the second version leads to tree nodes 256 bytes (calculated based on the default B-tree branching factor 18, value size 128 bytes, and pointer size 8 bytes). B+-tree does not store values in the internal nodes, hence only has one implementation. Its tree node size is similar to the second version of B-tree (256 bytes). For both trees, looking up a key inside a tree node is done sequentially as binary search will not introduce observable better performance with small branching factor.

We implement *RB-Tree* following its initial design [36]. We instantly re-balance the tree after each insertion. Although RB-Tree can have better performance with relaxed balancing [37], [38], we do not consider them in this study. The RB-Tree node is around 192 bytes.

B. Persistent Modes

Each of these selected indexing data structures has been studied thoroughly in DRAM setting. Our new contribution in `pmemids_bench` is to evaluate their performance under

different persistent modes on Optane DC and DRAM. Specifically, we implemented four different modes for each indexing data structure:

- *DRAM* mode is the original implementation of the indexing data structure in DRAM, without any persistence considered. This mode is for developers to compare what the performance they could get if they introduce DRAM indexing for their persisted data, for example the vertex index shown in Fig. 2(b).
- *PMEM-Volatile* mode uses the PMEM device as a volatile DRAM, ignoring the data persistence. In this mode, the indexing data structures are written into Optane DC, but without cache line flushing or memory fencing. Hence, it does not guarantee data will be persisted safely. This mode is implemented using PMDK `libvmem` library. It is useful if developers need large storage capacity and do not require data persistence.
- *PMEM-Persist* is the standard way of using Optane DC. As discussed in Section II-D, we leverage `libpmemobj` library to implement all the persisted indexing data structures. Specifically, we use its persistent, dynamic memory management APIs to allocate memory and use `_persist` APIs to properly flush the writes into Optane DC (in batch for better performance). But, we do not use the transaction mechanism for any of these memory updates. Instead, we expect the developers will have their own logic to solve the issues caused by potential partial writes (as discussed in Section II-D). Together with the *PMEM-Trans* mode, the results tell developers the performance with or without transaction in their applications, helping them make the trade-off.
- *PMEM-Trans* mode first guarantees all the data is persistent, similar to *PMEM-Persist* mode. But, it further leverages the transaction APIs from `libpmemobj` library to protect all operations that may contain multiple different memory accesses or one memory update larger than 8 bytes. For instance, appending an element to a linkedlist will include two operations: 1) allocate the new element and initialize it; 2) modify the pointer to point to the new element. We protect these two operations using a transaction to avoid memory leaking. This mode tells developers the overheads they will have if they use the default transaction mechanism provided by PMDK, as well as how they could optimize their data structures to reduce the overheads.

C. Parallel Mode

It is known the Optane DC is much easier to be saturated comparing to DRAM [10]. So the performance would be vastly different when users' applications consist of multi-threads. So, we consider the performance of the indexing data structures in various parallel modes in `pmemids_bench`.

We list four parallel modes in Table. I. Among them, the *single thread* mode is a special case of parallel mode: only one thread is operating the indexing data structures. The *NUMA* mode is another special case where we benchmark the NUMA effects for all parallel modes. More details will be discussed later about the NUMA setting of our platforms. The keys modes here are actually *Parallel, Saturated* mode and *Concurrent, Contention* mode.

In *Parallel, Saturated* mode, we initialize multiple independent instances of each indexing data structure. Each data structure instance will be exclusively operated by an independent thread. In this way, all threads are running in parallel without any data race, hence do not need synchronization. The performance limitations will mainly come from the hardware bandwidth limitation. The results in this mode also suggest the ideal concurrent performance of each indexing data structure, where multiple threads are operating the data structures concurrently but no conflicts are generated.

In *Concurrent, Contention* mode, we initialize only one instance of each indexing data structure and have multiple threads operate it concurrently. There will be a massive data race. Then developers need to resolve these conflicts using synchronization methods, which could be blocking (e.g., mutex lock), non-blocking (e.g., spinning lock), lock-free, or wait-free. The actual implementations of synchronization strategy vary and can be quite different case by case. In `pmemids_bench`, we do not try to benchmark all of them. Instead, we focus on investigating the overheads of the commonly used *mutex locks* in both volatile and persistent implementations, and leave benchmarking specific synchronization methods to users.

To benchmark mutex locks, we use a single mutex lock to serialize all data accesses, including both reads and writes to the data structures. This will make the multiple threads run one by one to write to the data structures, similar to what happens when there is only one thread. But, running in multi-thread introduces extra overheads for initializing, competing, holding, and releasing the mutex lock. By comparing the performance with different number of concurrent threads, we can identify the overheads caused by the mutex lock and how they grow as the number of threads increases.

There are multiple choices provided to developers when they need to use mutex lock. And we want to understand their performance overheads. First, there are two mutex locks developers can pick: 1) the traditional `pthread_mutex_lock` offered by `pthread` library, and 2) the `pmemobj_mutex_lock` offered by the `libpmemobj` library. The semantic of these two mutex locks is similar. The main difference is the pmemobj mutex lock is stored on Optane DC persistently. This also means that if the applications crashed, the `libpmemobj` library will automatically re-initialize the lock to avoid deadlock after restart. Second, when developers pick the pthread mutex lock, they have the options to store it in DRAM or Optane DC. Although they might need to re-initialize the lock manually if store pthread mutex lock on Optane DC, this strategy is still useful when developers want to integrate locks into the PMEM-resident data structures. Third, when developers program in *PMEM-Persist* mode, they have the flexibility of using pthread mutex lock instead of pmemobj mutex lock, especially given the fact that DRAM-based pthread mutex lock will re-initialize itself after a crash. So, whether using different locks may lead

to a different performance of the indexing data structures is an important question to answer.

D. Workloads

The actual workloads running on real applications vary significantly and hard to predict. While at the same time, they do have a profound impact on the performance. Existing benchmarks like YCSB typically include some representative workloads, such as the five workloads (A to E) shown in the top part of Table II, to help benchmark the storage systems in a setting that is close to the real world [17]. However, in `pmemids_bench`, this is not feasible anymore, simply because unlike benchmarking storage systems, we do not know how developers will use the indexing data structures. So, in our implementation, we leverage the YCSB workload generator, but we do create different workloads for our evaluations.

TABLE II
WORKLOADS IN % OF DIFFERENT OPERATIONS.

YCSB Workload	A	B	C	D	E
Read	50	95	100	95	50
Update	50	5	-	-	-
Insert	-	-	-	5	-
Read & Update	-	-	-	-	50
pmemids_bench Workload	A (100% Read)		B (100% Write)		
Read	100		-		
Update	-		90		
Insert	-		10		

As the bottom part of Table II shows, there are only two workloads included in `pmemids_bench` by default. And they are the extreme workloads: 'A (100% Read)' and 'B (100% Write)'. In the 100% write workload, we actually include 10% new insertions and 90% updates on existing data. Also, the 100% read workload only contains point lookups as not all the data structures that we implement support range queries. For range queries, more details can be seen in Lucas et. al. [39].

By benchmarking these extreme workloads, we do not attempt to model the real-world workloads in `pmemids_bench`. Instead, we provide developers an expectation or boundary of the performance of their different design choices. Once developers have their PMEM-based applications implemented, they are free to generate the workloads they need to benchmark the accurate performance of their applications. Note that, all the workloads are generated using Zipfian distribution unless specified otherwise.

E. Implementation Details

We implement `pmemids_bench` based on a C++ version of YCSB [40]. Our code is in open source at Github [18].

We make several major changes to YCSB to enable our data structure-level benchmarking. Specifically, YCSB is designed for evaluating Cloud storage systems, whose typical architecture includes clients and servers connected via Ethernet with million-second latency. However, the latency of accessing DRAM and Optane DC is often less than micro-second (a thousand time less). So, the overheads introduced by the framework itself may become significant enough to change our results.

We carefully redesign the benchmark framework to remove these overheads. First, we no longer implement separate storage servers and clients. Instead, the client code has the indexing data structure implemented internally. This eliminates the overheads of network stacks. In addition, we move the data generation phase out of the critical path. Currently, YCSB generates a new key/value right before the operation, which will create noticeable overheads in the evaluations. In `pmemids_bench`, we pre-generate all key/value pairs before the actual operations and store them in DRAM. These pre-generated data are stored continuously and later accessed sequentially. They gain good cache behaviors and introduce little overheads to our results.

V. BENCHMARKING RESULTS AND ANALYSIS

In this section, we report benchmark results we collected in our evaluation platform and discuss our observations from these results. Due to the space limitation, we can not present all the results here. The complete results can be found in our Github Repo [18].

A. Evaluation Platform

We conducted all the evaluations on a Dell R740 rack server with two sockets. Each socket installs a 2nd generation Intel Xeon Scalable Processor (Gold 6254 @ 3.10G) with 18 physical (36 virtual) cores. The machine is running Ubuntu 18.04 with a Linux kernel version 4.15.0. To enable the NUMA evaluation, we put all the DRAM and Optane DC DIMMS into one socket (node 0). This socket has 6 DRAM DIMMS with 32GB each and 1 Optane DC DIMM with 128GB. In all the evaluations, except the NUMA ones, we bind all the threads to node 0 to enable local memory accesses. The total memory capacity is 192GB DRAM and 128GB Optane DC. We used PMDK 1.8 in the implementation.

B. Benchmarking Results and Observations

We collected extensive benchmark results from running seven indexing data structures implemented in four different persistent modes (*DRAM*, *PMEM-Volatile*, *PMEM-Persist*, *PMEM-Trans*) under four parallel modes (with thread number increases from 1 to 16) on two workloads (100% write and 100% read). All the evaluations were conducted 10 times to avoid bias; and the workloads are large enough (i.e., 1 million reads/writes in average) to place enough pressure on the data structures. In this section, we show some key results and summarize our key observations from these results.

Observation 1: PMEM-volatile mode performs better than other PMEM-based modes in Read operations.

It is easy to understand that *PMEM-Volatile* will lead to better *write* performance comparing with other PMEM modes, simply because *PMEM-Volatile* mode does not flush cache line nor fence memory operations to guarantee data persistence during writes. Results in Fig. 3(a) confirm this.

But, interestingly, our results show *PMEM-Volatile* mode also leads to noticeably better *read* performance than other

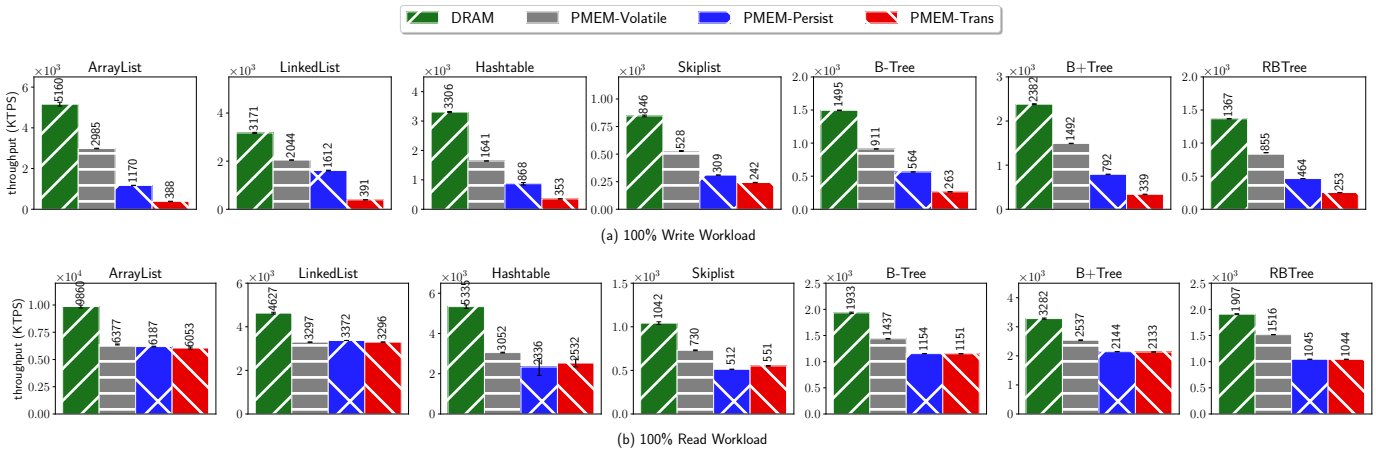

Fig. 3. The benchmark results of seven indexing data structures in a single thread case. The x-axis lists all the persistent modes and y-axis shows the throughput (KTPS or k transactions per second) after running the workloads. We tested all five workloads, and showed two of them (100% read and 100% write) here. Note that, all the experiments were repeated 10 times. We plot both the average and the variations in the figure.

PMEM modes. This is interesting because PMEM *reads* should be operated the same in all PMEM-based modes. But, our evaluations show different results in most of the indexing data structures (except *Arraylist* and *Linkedlist*) for 100% *read* workload.

We consider two potential reasons here. First, in *PMEM-Volatile*, all the memory pointers are direct pointers and do not need a translation. While in *PMEM-Persist* and *PMEM-trans* modes, we need to call function `pmemobj_direct` to convert a persistent pointer (`PMEMoid`) to normal pointer (`void *`) each time, as shown in Fig. 1(b). This memory translation is efficient, but still takes time and introduces overheads. Second, although all three PMEM-based modes use the same memory management library (`jemalloc`) internally, the `libpmemobj` used by *PMEM-Persist* and *PMEM-trans* modes does include necessary extra steps and memory space to guarantee transactional allocations. This will change the layout during dynamic memory allocations and potentially affect the later *reads* performance.

TABLE III
THE LAST LEVEL CACHE MISS RATIO OF DIFFERENT INDEXING DATA STRUCTURES IN 100% READ WORKLOAD.

Data Structures	PMEM-Volatile	PMEM-Persist	PMEM-Trans
Hashtable	50.831%	66.081%	65.958%
Skiplist	44.002%	68.366%	72.643%
B-Tree	31.771%	74.940%	76.309%
B+Tree	37.333%	69.854%	73.858%
RBTree	37.205%	65.296%	67.832%

To validate our hypothesis, in Table III, we show the cache miss ratios of five indexing data structures (i.e., *Hashtable*, *Skiplist*, *B-Tree*, *B+-Tree*, *RBTree*) that have better read performance in *PMEM-Volatile* mode. The cache miss ratios are collected using `perf` [41] while running the same 100% read workload in three PMEM-based modes. The results clearly show, for these indexing data structures, *PMEM-Volatile* obtains the least cache miss ratios among all PMEM-based modes; the other two modes (*PMEM-Persist* and *PMEM-Trans*) have similar cache miss ratios. This explains why *PMEM-Volatile* leads to the best read performance in actual data structures.

Observation 2: The Read performance of indexing data structures on Optane DC diverges from what the low-level benchmarks suggested.

Previous low-level benchmarking studies showed Optane DC has lower read bandwidth and higher read latency than DRAM [10]. We conducted the same low-level benchmarking on our evaluation platform and presented the results in Fig. 4. Here, we can see our Optane DC read bandwidth (single thread) is 2.0X lower than DRAM and the read latency is 2.8X higher than DRAM, which are similar to the one reported previously.

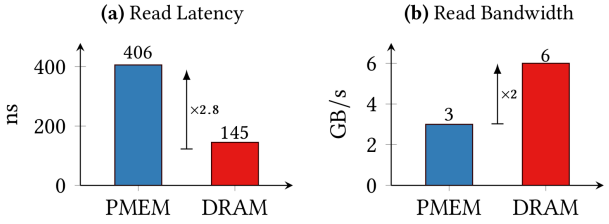

Fig. 4. The low-level benchmark results of random reads (from single thread) on our Optane DC platform. The left figure shows the read latency in nanoseconds; the right figure shows the bandwidth in GB/s. Note that, Both DRAM and PMEM are local memory accesses.

But, when it comes to the actual indexing data structures, the overheads of Optane DC are different across different data structures and may diverge a lot from the low-level benchmarking results, as shown in Fig. 3(b). For some indexing data structures, such as *Arraylist*, *Linkedlist*, and *B+-tree*, DRAM is less than 1.5x faster than PMEM-Persist (e.g., 3282 KTPS vs 2537 KTPS in *B+-Tree*). While, at the same time,

data structures such as *Skiplist*, *B-Tree*, and *RBTree*, do show 2x performance slowdown when moved from *DRAM* to *PMEM-Persist* mode.

We consider the divergent read performance mainly comes from the locality of different indexing data structures, which comes from two places. The first is spatial locality. The sizes of the elements in our data structures are larger than CPU cache line, for example, *Skiplist* element takes about 256 bytes and *B-tree* node also takes about 256 bytes. So reading elements can take advantages from both the CPU caches (64 bytes) and the Optane DC internal buffer (256 bytes) to gain better performance. The second source is temporal locality. The workload we generated in `pmemids_bench` follows the Zipfian distribution by default. This distribution has hot elements. So the generated workload may request the same data element repeatedly, in which case, later accesses can take advantage of the CPU caches and the internal buffers.

Observation 3: The persistent Writes on Optane DC have high overheads, while volatile Writes have lower overheads.

From Fig. 3, we can observe that *PMEM-Persist* introduces significant overheads in *writes* across all data structures. For most of the indexing data structures, the persistent writes introduce a 2-5x slowdown comparing to *DRAM* mode.

We also observed the *PMEM-Volatile* mode has lower *writes* overheads. This is expected since writes in *PMEM-Volatile* mode do not force cache line flushing nor memory fencing. But to what extent this will improve the performance is unknown.

We showed, for most of the indexing data structures, there is less than 2x slow down on *writes* in *PMEM-Volatile* mode comparing to DRAM. This is impressive given the PMEM media is much slower than DRAM [10]. Together with its high *read* performance as discussed earlier, *PMEM-Volatile* mode should be an attractive option for holding large datasets if data persistence is not a concern or handled elsewhere. The application could perform exceptionally well even comparing with DRAM, while at the same time be more cost-effective and energy-effective.

Observation 4: PMDK transaction mechanism introduces high overhead. The overhead increases with the size of the transaction.

From Fig. 3(a), it is easy to observe that *PMEM-Trans* mode leads to the worst *writes* performance across all indexing data structures. For *reads* performance, it is basically the same as *PMEM-Persist* as the transaction is not needed in reads.

PMDK implements transaction through redo/undo logs, which will copy the protected memory back and forth to ensure the transaction executed together. Here, the main overhead is memory copy, which increases when the size of the transaction increases. We observe the same from Fig. 3(a). For indexing data structures that have a small element size, such as *Hashtable*, *Skiplist*, and *RBTree*, *PMEM-Trans writes* perform only slightly worse than *PMEM-Persist*. While, on the other hand, data structures such as *B-tree* and *B+tree*, which have large tree node, will experience much higher overheads due to transaction operations, for example, *B-tree* with transaction achieves less than 1/2 performance comparing with *PMEM-Persist* mode.

To show how the size of protected memory changes the performance of PMDK transactions, we implemented two *B-tree*s in `pmemids_bench` for comparison. The main difference of them is whether it is the values (i.e., *value embedded*) or just the references pointing to the values (i.e., *value referenced*) that are stored in the non-leaf nodes. This creates *B-tree* with node size 2k bytes v.s. 256 bytes. We show the performance of both *B-tree* implementations in four persistent modes in Fig. 5. Here, we can see that, increasing the node size (value embedded) actually increases the *write* performance in DRAM, PMEM-Volatile, and PMEM-Persist modes; but decreases the performance of PMEM-Trans mode. The better performance of *value embedded* comes from the larger continuous writes (2k). But the transaction overheads out-weight the good cache behaviors and significantly reduce the write performance.

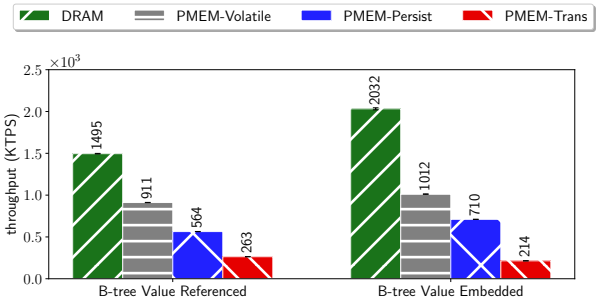

Fig. 5. The performance of two B-Tree implementations (value referenced vs. value embedded) in different persistent modes. The tree node size changes from 256 byts to 2K btyes.

This observation shows the transaction mechanism provided by the PMDK library introduces high overheads. To minimize them, developers should reduce the size of memory updated inside the transaction. In addition, for some cases, developers may further consider leverage after-failure fixing mechanisms instead of using transactions. For example, in *Linkedlist*, two operations: 'allocate a new node' and 'append the new node to the linkedlist' should both success or fail to avoid memory leaking. If developers protect these two operations using transaction, they will experience a similar overhead as shown in our benchmarking results. However, instead of using transaction, developers can actually fix the memory leaking after the failures. Specifically, the `libpmemobj` library already tracks all the memory allocation internally. So, if a failure happens in-between these two operations, after the crash and restart, developers can scan all the recorded memory allocations and free the ones that are not reachable through the *Linkedlist* to fix the potential memory leaking.

Observation 5: Both Optane DC writes and reads do not scale well in indexing data structure level.

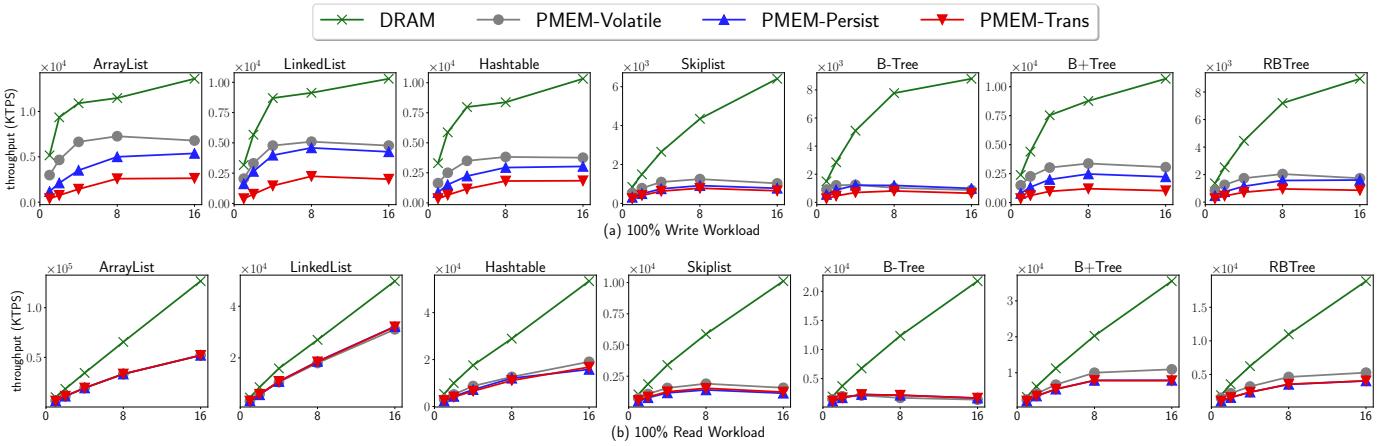

Fig. 6. The benchmark results of seven indexing data structures in the *Parallel, Saturated* mode. The x-axis is the thread number; the y-axis is the aggregated throughput (KTPS) across all threads. We used up to 16 threads to make sure each thread is running in a physical core. We show both 100% write and 100% read workloads here.

All the previous observations are based on single-thread benchmarking results shown in Fig. 3. We further investigated the parallel performance of Optane DC.

In this set of evaluations, we benchmarked the indexing data structures in the *Parallel, Saturated* mode. In this mode, we started a given number of threads to run the workloads. Each of these threads initialized a local instance of the indexing data structures and operated on its local instance only. This mode can be seen in concurrent applications where 1) multiple threads read/write to separate datasets; or 2) multiple threads read/write to the same dataset but happen do not conflict due to the workload pattern or the lock design. For both scenarios, the bottleneck will be on how fast Optane DC can be saturated by multiple independent threads. Note that, in all evaluations, we bind all the threads to node 0 to ensure local memory accesses.

In Fig. 6(a), we show 100% *write* performance of different indexing data structures running in this *Parallel, Saturated* mode. Here, we can observe that *writes* in Optane DC do not scale when thread number increases. Across all the indexing data structures, all three PMEM-based modes (*PMEM-Volatile*, *PMEM-Persist*, and *PMEME-Trans*) have flat or descendent curves when the thread number increases from 4 to 16. As this mode does not introduce any data race nor involve synchronization, we believe this comes from the limited scalability of Optane DC itself. Such a trend is distinct from the DRAM mode, where all data structures show good scalability on write performance with more threads.

We also plot the 100% *read* performance in the *Parallel, Saturated* mode. The results are shown in Fig. 6(b). Here, we can observe a similar pattern, i.e., limited scalability, as *writes* for most of the indexing data structures, except *Arraylist* and *Linkedlist*. This is interesting, because different from other data structures, only these two data structures do reads in one-hop, which means only one Optane DC access is needed to fulfill a read request. While, other indexing data structures, such as RB-Tree, will need to access Optane DC multiple times for searching the element before reading it. We believe it is such a read fanout that affects the read scalability.

The results on read scalability of *ArrayList* and *Linkedlist* actually agree with the conclusion made by previous Optane DC low-level benchmark results [11], quoted "*Optane DC reads scale with thread count; whereas writes do not.*" But, our indexing data structures level benchmarking further suggests that the data structure itself actually has a profound impact on the Optane DC read scalability. And the impacts are not equal to DRAM and Optane DC. More accurately, if reading from a data structure requires multiple data accesses to locate that element, the read performance will not scale on Optane DC.

The key insight for application developers is two-fold. First, if your application is highly concurrent, then picking the indexing data structures that have less read fanout is necessary to exploit the Optane DC concurrency. Second, if your core data structures do have high fanout, then increasing the parallelism will not improve the Optane DC read/write performance. Developers should look for DRAM-based solutions.

Observation 6: Storing mutex locks in PMEM or DRAM does not change the performance much.

We further investigated the overheads of mutex locks using the *Concurrent, Contention* mode. In this mode, we initialized one instance for each indexing data structure and had multiple threads operate it concurrently. To maximize the mutex lock overheads, we actually used a single mutex lock to protect the whole data structure for both *read* and *write* operations. In this way, we can compare the overheads of different mutex lock choices in an extreme scenario with their impacts amplified.

In Fig. 7, we plot the *write* performance of all the indexing data structures in two concurrent cases, i.e., single thread and 16 threads. This is to show the overheads caused by intensively competing the same lock for different data structures. Note that, we implement mutex lock differently in these persist modes: in *DRAM* and *PMEM-Volatile* mode, we directly use

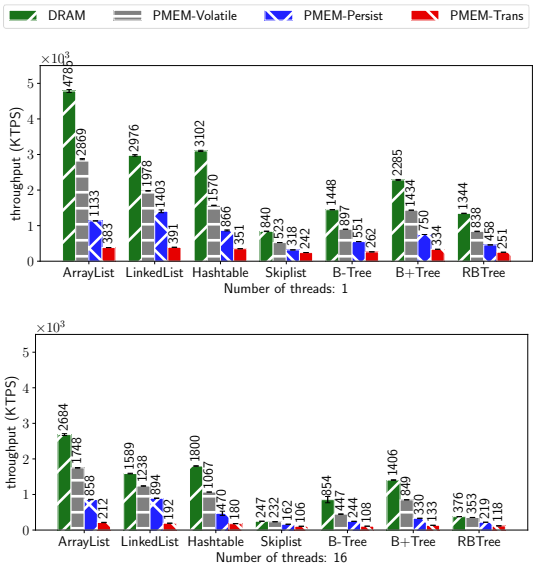

Fig. 7. The 100% *write* performance comparison of two cases running in the *Concurrent, Contention* mode. Here, x-axis groups indexing data structures and each group contains results of different persistent modes. The y-axis shows the throughput in KTPS.

the pthread mutex lock stored in DRAM; in *PMEM-Persist* and *PMEM-Trans* modes, we use the pmemobj mutex lock stored in PMEM.

From these results, we can see that competing the single mutex lock among multiple threads significantly decreases the *write* performance. This is expected because the single mutex lock creates significant contentions. More than this, we also see that the persistent mutex locks used in *PMEM-Persist* and *PMEM-Trans* modes actually have less effects on the performance, especially when compared with the pthread mutex locks in the other two modes. The results suggest that PMEM-based data structures are less affected by mutex locks than their DRAM counterparts. This is because DRAM-based data structures have higher throughput and hence higher contention.

The developers actually have multiple choices to use mutex locks in their applications. So, in addition to know that *mutex locks kill the performance*, it is also important to understand how different ways of using mutex locks would affect the performance. In this evaluation, we implemented and compared four different ways that developers could use to program mutex locks. Specifically, when program in *PMEM-Volatile* mode, developers can directly use the pthread mutex lock. And, they have the options to store the lock in DRAM (separated from the In-PMEM data structures) or in Optane DC (together with the In-PMEM data structures). When program in *PMEM-Persist* mode, developers can use either PMEM-based pmemobj mutex lock or DRAM-based pthread mutex lock, both of which will be re-initialized after applications reboot. These choices may have an impact on the performance.

To show how these ways will affect the performance, we plot the *write* performance of these four cases with increasing the number of threads in Fig. 8. Due to the limited space, we only use *RBTree* as an example. Other indexing data structures have the similar results. From Fig. 8, we observed that, for both *PMEM-Volatile* and *PMEM-Persist* modes, putting mutex locks in DRAM or Optane DC does not affect the performance of the data structure much. We believe these results come from the fact that the pmemobj mutex lock is internally also pthread mutex lock, and the frequent *acquire/release lock* operations are actually done in CPU cache level without being affected by the location.

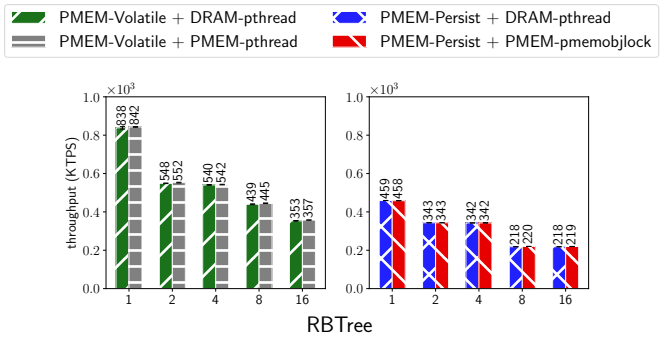

Fig. 8. The 100% *write* performance comparison of four different mutex lock usages in PMEM-based RBTree, with thread number increases from 1 to 16.

Observation 7: Non-local memory accesses affects Optane DC more than DRAM; affects writes more than reads.

We further evaluated how NUMA architecture could affect the performance of Optane DC. In the previous evaluations, we bind threads to node 0, which has all the Optane DC and DRAM DIMMS installed. In this evaluation, we ran the same evaluations but bind all threads to node 1, which does not have any memory installed. In this way, all memory accesses become non-local and will experience higher latency. In Fig. 9, we compared the performance of these two cases on all indexing data structures. It contains results on two workloads (100% read and 100% write) using a single thread.

We have several observations from the results in Fig. 9. First, the throughput of indexing data structures using non-local memory accesses is indeed lower for both DRAM and Optane DC, simply because of the higher latency in each non-local memory access. Second, non-local memory accesses affect the most on *PMEM-Trans* writes. For most of the indexing data structures, we see only half bandwidth if they wrote to non-local memory. We believe this is because transaction writes on Optane DC introduce extra writes on PMEM undo/redo logs. These writes are also non-local, hence further slow down the writes. Third, non-local memory affects *writes* more than *reads*, mostly because *writes* do not leverage cache as well as *reads*. For the same reason, the non-local memory also affects more in *PMEM-Persist* mode, in which the cache line needs to be flushed after each write. The insight for developers here is, if they need to read/write data stored in remote nodes anyway, they might consider loading or buffering the data from remote

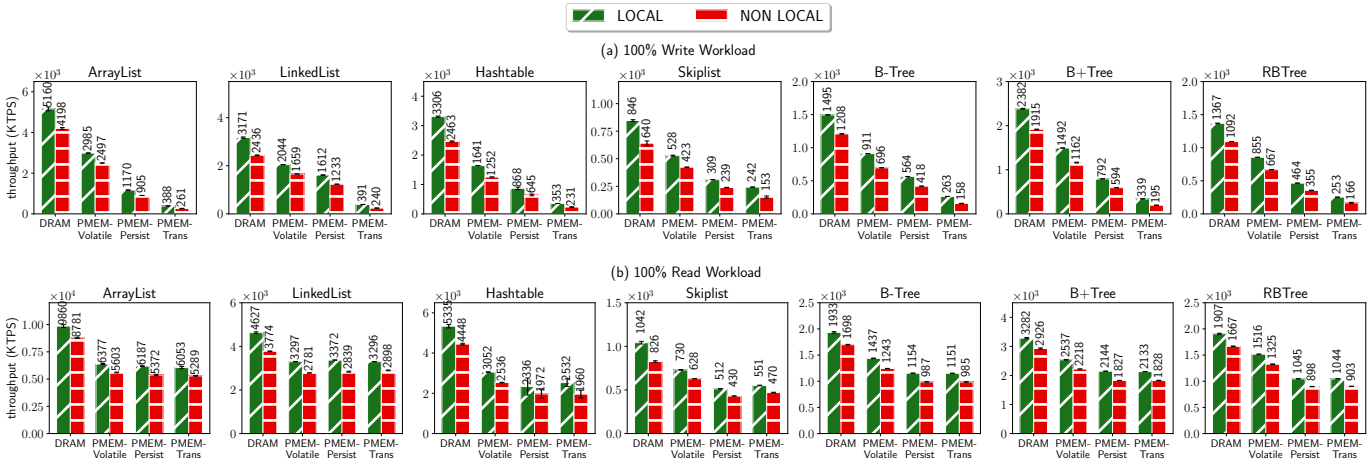

Fig. 9. The benchmark results of seven indexing data structures in different persistent modes accessing local memory and remote memory. All evaluations are based on single thread.

Optane DC to remote DRAM first, instead of directly accessing the remote persistent memory.

VI. Related Work

There have been several benchmark studies on Optane DC recently. Some focus on Optane DC low level device properties [10], [11], [13]; some focus on high level application on Optane DC [12], [42]. Our work is inherently different from them as we focus on indexing data structures, which are the build blocks of various applications, aiming to provide developers insights about the potential performance of their applications on Optane DC under different design choices.

Lately, there are studies starting to focus on indexing data structures on Optane DC as well, such as [39], [43]. Among them, Philipp et. al. [43] benchmarked low-level primitive operations of indexing data structures, such as splitting or merging tree nodes. Lucas et. al. [39], on the other hand, benchmarked B+-Tree-based indexing data structures that are designed specifically for persistent memory, such as BzTree [44], FPTree [45], NV-Tree [46], and wBTree [24]. Although focusing on different data structures and levels, we do believe their results are good complementary to our study.

Since we benchmark indexing data structures in this study, it is worth noting that there have been a significant amount of efforts conducted in designing new indexing data structures on persistent memory, such as persistent B+-Tree [47], [24], [45], [46], persistent Hashtable [48], [49], [50], persistent Skiplist [51], and persistent RB-tree [52]. Several works [53], [54] also proposed general guidelines for porting data structures to persistent memory. All these research works do include some performance evaluations. But these evaluations are specific and cannot directly serve as benchmarks for application developers. Also, most of these works were not evaluated on real Optane DC platforms.

VII. Conclusion and Future Work

In this paper, motivated by the trends of adopting the new Optane DC devices into various applications and the needs for developers to better understand the performance of Optane DC, we conducted the first performance study of the Intel Optane DC from indexing data structures' perspective. To do so, we implemented `pmemids_bench`, a benchmark suite that includes seven commonly used indexing data structures implemented in different persistent modes and parallel modes. Through extensive evaluations on real Optane DC platforms, we summarized seven observations covering various aspects of Optane DC programming, such as persistence overheads, transaction overheads, scalability, and lock overheads, etc. Some of our observations match the previous benchmarking results well, but some do not, which makes our results useful and interesting to end-users. We believe `pmemids_bench` has the potential to be a necessary reference for developers who are designing their applications to work with Optane DC. In the future, we plan to enrich the benchmark by adding more commonly seen data structures, such as stack and graph. We also plan to investigate more lock selections in the *Concurrent, Contention* mode, such as read-write lock, fine-grained locking, and lock-free concurrency, etc.

VIII. Acknowledgment

We sincerely thank the anonymous reviewers for their valuable feedback. This work is supported by NSF grant CNS-1852815 and its REU supplement.

References

[1] K. Suzuki and S. Swanson, "A survey of trends in non-volatile memory technologies: 2000-2014," in *2015 IEEE International Memory Workshop (IMW)*. IEEE, 2015, pp. 1–4.

[2] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling," in *2009 42nd Annual IEEE/ACM international symposium on microarchitecture (MICRO)*. IEEE, 2009, pp. 14–23.

[3] J. J. Yang, D. B. Strukov, and D. R. Stewart, "Memristive devices for computing," *Nature nanotechnology*, vol. 8, no. 1, p. 13, 2013.

[4] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the 36th annual international symposium on Computer architecture*, 2009, pp. 24–33.

[5] Optane, "Intel Optane Persistent Memory," https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html, 2019, accessed: 2019-11.

[6] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, 2009, pp. 233–244.

[7] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a {PC}," in *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, 2012, pp. 31–46.

[8] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer, "Llama: Efficient graph analytics using large multiversioned arrays," in *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 2015, pp. 363–374.

[9] I. B. Peng, M. B. Gokhale, and E. W. Green, "System evaluation of the intel optane byte-addressable nvm," in *Proceedings of the International Symposium on Memory Systems*, 2019, pp. 304–315.

[10] A. van Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper, "Persistent memory i/o primitives," in *Proceedings of the 15th International Workshop on Data Management on New Hardware*, 2019, pp. 1–7.

[11] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor *et al.*, "Basic performance measurements of the intel optane dc persistent memory module," *arXiv preprint arXiv:1903.05714*, 2019.

[12] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 135–148, 2017.

[13] H. Shu, H. Chen, H. Liu, Y. Lu, Q. Hu, and J. Shu, "Empirical study of transactional management for persistent memory," in *2018 IEEE 7th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 2018, pp. 61–66.

[14] 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.

[15] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y.-R. Choi, "Slm-db: single-level key-value store with persistent memory," in *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, 2019, pp. 191–205.

[16] pmem.io Persistent, "Persistent Memory Programming," https://pmem.io, 2019, accessed: 2019-11.

[17] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 143–154.

[18] "Pmem index data structure benchmark," https://github.com/DIR-LAB/ycsb-storedsbench, accessed Feb. 14, 2020.

[19] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-change technology and the future of main memory," *IEEE micro*, vol. 30, no. 1, pp. 143–143, 2010.

[20] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung *et al.*, "Phase-change random access memory: A scalable technology," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 465–479, 2008.

[21] H. Akinaga and H. Shima, "Resistive random access memory (reram) based on metal oxides," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2237–2251, 2010.

[22] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating stt-ram as an energy-efficient main memory alternative," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2013, pp. 256–267.

[23] L. Gwennap, *First Optane DIMMs Disappoint*. The LinleyGroup, 2019.

[24] S. Chen and Q. Jin, "Persistent b+-trees in non-volatile main memory," *Proceedings of the VLDB Endowment*, vol. 8, no. 7, pp. 786–797, 2015.

[25] J. Gray *et al.*, "The transaction concept: Virtues and limitations," in *VLDB*, vol. 81, 1981, pp. 144–154.

[26] J. Evans, "Scalable memory allocation using jemalloc," 2011.

[27] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*, 2nd ed. The MIT Press, 2009.

[28] Oracle, "Hashtable," https://docs.oracle.com/javase/8/docs/api/java/util/Hashtable.html, 2019, accessed: 2019-12.

[29] W. Pugh, "Skip lists: A probabilistic alternative to balanced trees," in *Algorithms and Data Structures*, F. Dehne, J. R. Sack, and N. Santoro, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1989, pp. 437–449.

[30] F. Ergun, C. Sahinalp, J. Sharp, and R. Sinha, "Biased skip lists for highly skewed access patterns," 09 2001, pp. 216–229.

[31] A. Bagchi, A. L. Buchsbaum, and M. T. Goodrich, "Biased skip lists," *Algorithmica*, vol. 42, no. 1, p. 31–48, May 2005. [Online]. Available: https://doi.org/10.1007/s00453-004-1138-6

[32] J. L. Carlson, *Redis in action*. Manning Publications Co., 2013.

[33] R. Hankins and J. Patel, "Effect of node size on the performance of cache-conscious b + -trees," *Sigmetrics Performance Evaluation Review - SIGMETRICS*, vol. 31, pp. 283–294, 06 2003.

[34] S. Chen, P. Gibbons, and T. Mowry, "Improving index performance through prefetching," *ACM SIGMOD Record*, vol. 30, 01 2002.

[35] J. Rao and K. A. Ross, "Making b+- trees cache conscious in main memory," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 475–486. [Online]. Available: https://doi.org/10.1145/342009.335449

[36] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," in *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, ser. SFCS '78. USA: IEEE Computer Society, 1978, p. 8–21. [Online]. Available: https://doi.org/10.1109/SFCS.1978.3

[37] S. Hanke, "The performance of concurrent red-black tree algorithms," in *Algorithm Engineering*, J. S. Vitter and C. D. Zaroliagis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 286–300.

[38] S. Hanke, T. Ottmann, and E. Soisalon-Soininen, *Relaxed Balanced Red-Black Trees*, 11 2006, pp. 193–204.

[39] L. Lersch, X. Hao, I. Oukid, T. Wang, and T. Willhalm, "Evaluating persistent memory range indexes," *Proc. VLDB Endow.*, vol. 13, no. 4, p. 574–587, Dec. 2019. [Online]. Available: https://doi.org/10.14778/3372716.3372728

[40] "Yahoo! cloud serving benchmark in c++," https://github.com/basicthinker/YCSB-C, accessed Jan. 1, 2020.

[41] "perf: Linux profiling with performance counters," https://perf.wiki.kernel.org/index.php/Main_Page, accessed Jan. 1, 2020.

[42] M. Weiland, H. Brunst, T. Quintino, N. Johnson, O. Iffrig, S. Smart, C. Herold, A. Bonanni, A. Jackson, and M. Parsons, "An early evaluation of intel's optane dc persistent memory module and its impact on high-performance scientific applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356159

[43] P. Götze, A. Kumar Tharanatha, and K.-U. Sattler, "Data structure primitives on persistent memory: An evaluation," 2020.

[44] 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 Endow.*, vol. 11, no. 5, p. 553–565, Jan. 2018. [Online]. Available: https://doi.org/10.1145/3164135.3164147

[45] 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," 06 2016, pp. 371–386.

[46] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "Nv-tree: Reducing consistency cost for nvm-based single level systems," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*. Santa Clara, CA: USENIX Association, Feb. 2015, pp. 167–181. [Online]. Available: https://www.usenix.org/conference/fast15/technical-sessions/presentation/yang

[47] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in byte-addressable persistent b+-tree," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*. Oakland, CA: USENIX Association, Feb. 2018, pp. 187–200. [Online]. Available: https://www.usenix.org/conference/fast18/presentation/hwang

[48] M. Nam, H. Cha, Y. ri Choi, S. H. Noh, and B. Nam, "Write-optimized dynamic hashing for persistent memory," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 31–44. [Online]. Available: https://www.usenix.org/conference/fast19/presentation/nam

[49] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 461–476. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/zuo

[50] B. Debnath, A. Haghdoost, A. Kadav, M. G. Khatib, and C. Ungureanu, "Revisiting hash table design for phase change memory," in *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, ser. INFLOW '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2819001.2819002

[51] Q. Chen and H. Y. Yeom, "Design of skiplist based key-value store on non-volatile memory," *2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pp. 44–50, 2018.

[52] C. Wang, Q. Wei, L. Wu, S. Wang, C. Chen, X. Xiao, J. Yang, M. Xue, and Y. Yang, "Persisting rb-tree into nvm in a consistency perspective," *ACM Trans. Storage*, vol. 14, no. 1, Feb. 2018. [Online]. Available: https://doi.org/10.1145/3177915

[53] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram, "RECIPE : Converting Concurrent DRAM Indexes to Persistent-Memory Indexes," 2019. [Online]. Available: http://arxiv.org/abs/1909.13670{%}0Ahttp://dx.doi.org/10.1145/3341301.3359635

[54] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, ser. FAST'11. USA: USENIX Association, 2011, p. 5.