Comparison of Modern Indexing Approaches on Persistent Memory

Kumar Biplav Rocil Machado Nisarg Shah Shaurya Shekhar
University of Wisconsin - Madison

{biplav, rmachado, ndshah4, sshekhar4}@wisc.edu

ABSTRACT

With the commercial availability of Persistent Memory and the performance benefits it brings over traditional block storage devices, many modern indexes have been developed to take advantage of low latencies and failure-atomicity of PMem devices in case of power failures. We compare the performance of three modern indexes - BzTree and LB+-tree, which are B-tree based indexes, and Dash, a hash-based index, on emulated PMem. LB+-tree and Dash are very recent and have not been compared by previous comparison studies. In general, we find that Dash performs the best, while BzTree performs the worst. We also lay out some design considerations to exploit unique characteristics of Persistent Memory while developing persistent indexes.

1. INTRODUCTION

Persistent Memory (PMem) is a new storage-class memory which provides non-volatile storage at a performance comparable to DRAM. It is also byte-addressable, which means it can be accessed via normal load-store instructions. Commercially available Intel Optane Persistent Memory [2] modules can hold upto 512 GB of capacity per module.

Naturally, there is significant interest in storing database indexes in PMem [3, 4, 9, 10, 11, 19] or a hybrid combination of DRAM and PMem [9, 14] for quick recovery on power failures. A previous study [8] has compared different B-tree based indexes on PMem, but does not provide any insights about the performance of hash-based indexing structures. In our work we compare BzTree [3] and LB+-Tree [9] (both B-tree based indexes) with Dash [11], a hash based index.

BzTree is a B+-tree specifically optimized for non-volatile memory (NVM) which is implemented in a latch-free manner. To become latch-free, B+ trees usually require multi-word updates to handle operations like node splits and merges. BzTree uses a persistent multi-word compare-and-swap operation (PMwCaS) [15] to enable the compare-and-swap operation to persist updates in PMem without giving up atomicity. The PMwCAS operation is implemented in software and requires no special hardware support other than a compare-and-swap (or equivalent) instruction.

LB+-tree [9] is a persistent B+-tree optimized for 3DX-Point [1] (Intel Optane persistent memory) that minimizes the number of NVM line writes and performs logless insertions. In 3DXPoint, the number of modified words written in a cache line does not affect the NVM performance. The granularity of persist operations are cache lines, and persist operations are more costly than NVM writes. Therefore, in order to reduce the number of NVM line writes, LB+-tree is designed to perform more NVM word writes per line. The node size of LB+-tree is set to 256B to best utilize the internal data access bandwidth of 3DXPoint memory. Optimization techniques like entry moving, logless node split, and distributed headers are used to improve LB+-tree's insertion performance.

Dash [11] is a comprehensive approach to build dynamic and scalable hashing on real PMem. The limited bandwidth of Persistent Memory is often saturated by excessive PMem reads in the form of existence checks in hash tables based indexes. Dash aims to reduce both Persistent Memory writes and reads for higher performance. Dash uses fingerprinting technique to avoid accessing PMem during reads. It avoids PMem writes during search operation by using optimistic verification approach to detect conflicts. Dash also employs bucket load balancing techniques to achieve high load factor while maintaining high performance.

2. RELATED WORK AND MOTIVATION

There have been a growing number of indexes for persistent memory in recent years, but only one work so far has attempted a comparison of the indexes. Experiments by Lersch et. al. [8] have evaluated BzTree with other B+-tree based indexes wBTree [5], NV-Tree [17] and FPTree [14]. Our experiments compare BzTree with LB+-Tree and Dash, which are more recent indexes. In addition to having indexes with different storage architecture (PMem-only vs hybrid DRAM + PMem), locking and persistence mechanisms, we also compare indexes based on different data structures. BzTree and LB+-trees are B+-tree based indexes and Dash is a hash table based index. Our goal is to benchmark these different types of indexes, summarize any limitations of the indexes based on our experiments and provide suggestions for improvements based on the results. At the end, we also lay out design considerations for persistent indexes based on our observations.

3. BACKGROUND

In this section, we provide some background on Persistent Memory and challenges in designing indexes for Persistent Memory.

3.1 Persistent Memory

Persistent Memory (PMem) [2] is used today in database, storage, virtualization, big data, cloud computing, and artificial intelligence applications. PMem is a solid-state high-performance byte-addressable memory device that resides on the memory bus. Being on the memory bus allows PMem to have DRAM-like access to data, which means that it bandwidth and access latencies comparable to DRAM and the nonvolatility of NAND flash. It also has a lower cost per bytes than DRAM and can also contain higher capacities than DRAM. Intel Optane Persistent Memory modules can contain upto 512 GB per DIMM.

There are two main configuration modes in Intel Optane PMem. In the memory mode, DRAM is managed as a cache for PMem by the memory controller and the PMem itself acts as the main memory. This mode enables users to exploit large capacities of PMem without modifying applications. However, this mode does not support persistent data structures. This is because contents in the DRAM cache are lost upon power failure. Therefore, it can only be used as large volatile main memory. Second, it takes longer time to load data from PMem because a load consists of a DRAM cache visit (miss) followed by a PMem visit. Hence, it may be sub-optimal to run data-intensive applications with working sets larger than the DRAM cache capacity in the memory mode.

In the app-direct mode, PMem and DRAM are both directly accessed by the CPU. PMem modules are recognized as special devices by the OS. We can install file systems on PMem modules and use Intel's PMDK (Persistent Memory Development Kit) to map a file from PMem into the virtual memory space of an application. The OS runs DAX device drivers for PMem so that accesses to PMem get around the OS page cache. In this way, applications can directly access PMem using load and store instructions, and implement persistent data structures in PMem. We focus on the app-direct mode of PMem NVDIMMs in this paper.

3.2 Designing Persistent Indexes

With the numerous advantages of PMem described in previous section, storing indexes on it enables both high performance and fast recovery. However, these benefits bring their own challenges. First, care needs to be taken while developing the indexes to make sure the data is persisted in a consistent manner. Using the atomic flush and fence instructions may result in needless overheads due to stalls as the CPU waits for the data to be flushed from the cache. This requires the index designs to issue as little flushes as possible while ensuring consistency.

Another challenge is that PMem has a lower bandwidth than DRAM, and therefore excessive accesses to PMem can saturate the bandwidth and limit scalability. Thus indexes need to be designed in a way that minimizes read/write accesses to PMem. This is done by improving locality of accesses and by utilizing internal data access bandwidth of PMem. For instance, recent indexes read/write 256 bytes of data since internally PMem reads/writes data in blocks of 256 bytes.

4. BENCHMARKED INDEXES

In this section we describe the persistent indexes we have analyzed in our experiments in more detail. BzTree and LB+-tree are B-tree based indexes whereas Dash is a hash table based index. All three indexes expose a Get/Set interface to insert and lookup key-value pairs.

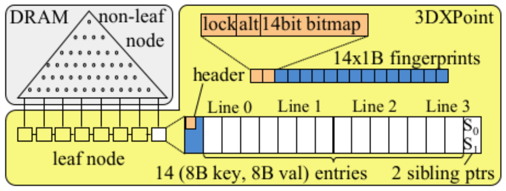


Figure 1: LB+-tree node structure

4.1 BzTree

BzTree [3] is a lock-free B-tree index designed for Non-Volatile Memories. It uses **P**ersistent **M**ulti-word **C**ompare-**A**nd-**S**wap (PMwCAS) [16] operations as a core building block. These operations are used to atomically handle operations like node splits and merges which usually require multi-word updates. It provides concurrency control as well as persistency guarantees.

PMwCAS works in two phases: In the first phase, it uses a descriptor to collect the expected and new values for each target word, persists the descriptor and automatically installs a pointer to the descriptor on each word. If the first phase succeeds, the second phase involves installing the new values, else the changes are rolled back. It makes use of a dirty bit to ensure no stale reads take place. An epoch based recycling scheme is used to enable the garbage collector to prevent threads from dereferencing pointers after its memory has been reclaimed.

Inner nodes are treated as immutable, expect for the case of updates to child pointers, while the leaf nodes accommodate inserts and updates. Thus, insertions to internal nodes sometimes result in replacement of the nodes as a whole with a new node which contains the new key. Inner nodes are thus, said to support copy-on-write replacement. Splits in a similar manner propagate up the tree. Leaf nodes have some free space into which insertions are made serially. Periodically, the leaf nodes are consolidated, which implies that the new unsorted additions are sorted. Thus, the inner nodes are search-optimized while the leaf nodes are write optimized.

4.2 LB+-Tree

LB+-tree [9] is a persistent B+ tree index optimized for 3D XPoint [1], a non-volatile memory technology developed jointly by Intel and Micron Technology. It has a hybrid storage architecture where leaf nodes are stored on Persistent Memory and inner nodes are stored on DRAM to improve searches for inner nodes, since DRAM reads are faster than Persistent Memory reads. The non-leaf nodes can be rebuilt from the leaf nodes upon power failure. Each node of the tree is 256 bytes or a multiple of 256 bytes (as shown in Figure 1), as 256 bytes is the internal data access size in Intel's Persistent Memory technology. For concurrency control, LB+-tree combines Hardware Transactional Memory (HTM) and a lock bit per leaf node.

LB+-tree introduces three optimizations to improve insertion performance. First is entry moving during insertions. In the best case, an insertion can update the header and insert the new entry if they are in the same line using only one NVM line write. Else, the insertion finds an empty slot in another line in the leaf node and incurs two NVM line writes. Here empty slots are actively created in the first line

Table 1: Comparison of key design choices of the indexes

Tree	Architecture	Node Structure	Concurrency	Persistence
BzTree	PMem-Only	Sorted Inner Nodes Partially Sorted Leaf Nodes	PMwCAS	PMwCAS
LB+-Tree	Hybrid (DRAM + PMem)	Sorted Inner Nodes Unsorted Leaf Nodes	HTM + Locking	NVM Atomic Write
Dash	PMem-Only	NA	Optimistic Locking	NVM Atomic Write

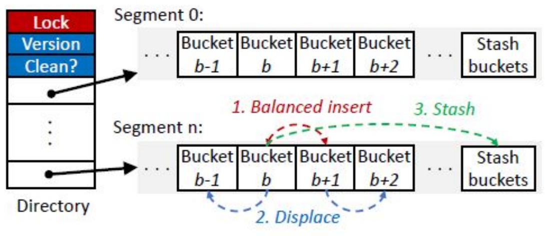


Figure 2: Architecture of Dash

(where the header is) by moving as many entries from the first line to the line being written to. Then, future insertions will more likely to find an empty slot in the first line, achieving the best case.

Second is logless node splits, where idea is to have two alternative sibling pointers and use the *alt* bit in the leaf header to indicate which pointer is in use. Finally it uses an 8-byte atomic write to set the *alt* bit to switch the sibling pointer to make the new node visible. This avoids logging for leaf node splits. Third, headers in a leaf nodes with size a multiple of 256 bytes are distributed (one header every 256 bytes) to make the above two optimizations works for larger node sizes.

4.3 Dash

Dash [11] is an approach to build **D**ynamic **a**nd **S**calable **H**ash tables on Persistent Memory. Prior work on building hash tables for Persistent Memory focused on reducing cacheline flushes and persistent memory writes. These hash tables were not scalable due to the limited bandwidth of the Persistent Memory. Excessive PMem accesses can easily saturate the system and prevent the system from scaling. The two main reasons for excessive PMem accesses in hash tables are excessive PMem reads that occur during existence checks, bucket probing and heavy-weight concurrency control.

In hash tables, to find out if a key exists, we have to scan one or more buckets, which can incur many cache misses and PMem reads. Several prior works used bucket-level locking for concurrency but it incurs additional PMem writes to acquire/release read locks, further causing a bottleneck in the PMem bandwidth. Therefore, it is important to not just reduce PMem writes but also PMem reads.

Figure 2 shows the architecture of Dash. Dash introduces optimizations to mitigate these problems. First, to avoid unnecessary PMem reads during record probing, Dash adopts fingerprinting. The idea here is to generate fingerprints (which are 1-byte hashes) of keys and store them compactly. Fingerprints summarize the existence of keys. Scanning finger prints is much faster than scanning the actual keys.

Second, Dash adopts lightweight, optimistic locking. The lock consists of a single bit which acts as a lock and a version number for detecting conflicts. Insert operations will lock the affected buckets but the search operation proceeds without holding any lock and verifies the version number at the end of the read operation. This lock-free read design reduces the number of PMem writes incurred while using read locks.

Third, Dash uses several techniques during insert operation like balanced insert, displacement and bucket stashing (as shown in figure 2) to balance the loads across buckets while limiting the PMem reads needed. The bucket size is set to 256 bytes for better locality. Dash also supports near-instantaneous recovery, variable-length keys and achieves high space utilization. Dash applies to both extendible and linear hash tables.

4.4 Discussion

We provide a quick comparison of the three indexes in Table 1. While BzTree and Dash both store data in PMem-only, LB+-tree takes advantage of a hybrid storage mechanism where inner nodes are stored in DRAM to improve search performance. On power failure, they can be reconstructed from the leaf nodes which are stored in PMem and are thus non-volatile. Other indexes such as FPTree [14] have also adopted this design. Inner nodes are usually sorted since they are updated less frequently and this also improves the search performance. Leaf nodes are unsorted to quickly absorb the updates. BzTree will occasionally reorganize a leaf node and sort keys to improve search performance. While both Dash and LB+-tree use some form of locking, BzTree uses software-based PMwCAS operation for a lock-free approach. PMwCAS also ensures that data is persisted in PMem. On the other hand, LB+-tree and Dash use an NVM Atomic Write (NAW) for persistence. A NAW is an 8 byte word write followed by a `clwb` and an `sfence` to persist data in PMem.

5. EVALUATION

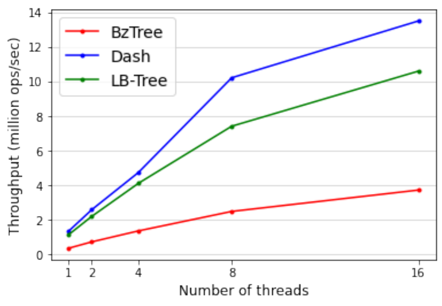
In this section we present our experimental setup, framework used and results.

5.1 Experimental Setup

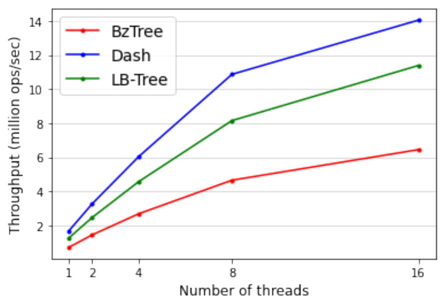
We perform our experiments on a Cloudlab [6] xl170 node with Intel E5-2640 v4 CPU at 2.4 GHz with 10 cores (20 logical threads), 25 MB L3 cache and 64 GB DRAM (4 x 16 GB). Persistent Memory is emulated in AppDirect mode using Linux kernel's `memmap` option and DAX (Direct Access) device drivers, as mentioned in Section 3.

5.2 Evaluation Framework

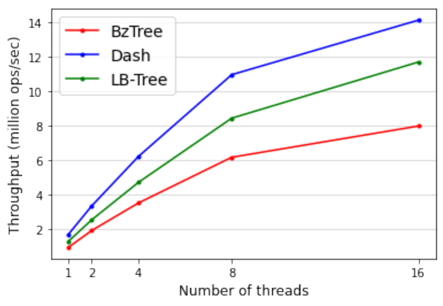
We used PiBench [8] as our evaluation framework. It is a persistent index benchmark tool targeted at data structures



(a) 50% writes

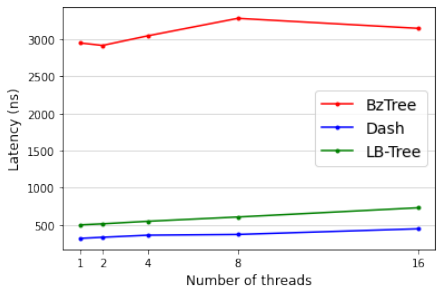


(b) 10% writes

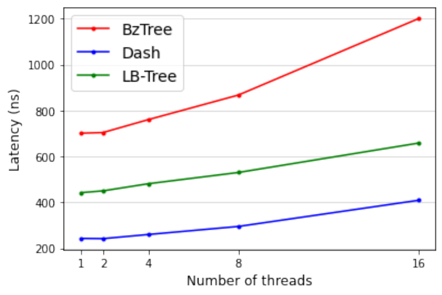


(c) Read-only

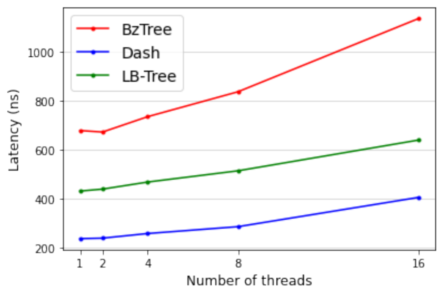
Figure 3: Throughput results



(a) 50% writes



(b) 10% writes



(c) Read-only

Figure 4: Median latency results

running on top of Intel Optane Persistent Memory. We created a shared library wrapper for each index. These shared libraries implement the interface for search, insert, delete and update operations. PiBench also supports executing scan operations but since not all indexes had the operation implemented we only run our experiments for inserts and lookups. We evaluated the indexes for three variations of workload - 50% writes, 10% writes and read-only. For each of these workloads, we run experiments on 1, 2, 4, 8 and 16 client threads. The keys are uniformly distributed with key size of 10 bytes and value size 32 bytes. Results with respect to the throughput, median and tail latencies of the indexes are presented in the following sections.

5.3 Throughput

Throughput across different threads for all three read/write workloads in shown in Figure 3. It is clear that Dash performs the best in all scenarios, followed by LB+-tree and BzTree. Dash scales up nearly linearly till 8 threads in all three plots. This could be due to superior insert/lookup performance of hash-based indexes compared to B-trees. We can also observe that speedup starts diminishing a bit after 8 threads. For 16-threaded Dash workloads, throughput only increases by 4.2% when we move from a 50% write workload to a 10% workload, and further 0.5% from 10% to read-only workload, suggesting that writes are almost as fast as reads in Dash. In contrast, BzTree throughput for 16 threads increases by 73.5% from 50% writes to 10% writes, and 23.8% from 10% to read-only workload.

5.4 Median Latency

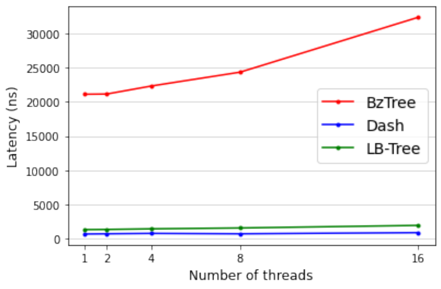
In the median latency results shown in Figure 4, we again find that Dash has the lowest median latencies, followed by LB+-tree and BzTree. For a single-threaded read-only workload in Figure 4c, it is interesting to see that Dash operates at a latency of about 235 ns, which is very close to DRAM access latencies on which we are emulating PMem. We also observed that latencies for all indexes decrease significantly from 50% writes to 10% writes and stay nearly the same for read-only workload.

To get an idea about the low performance of BzTree compared to other indexes, we plot L3 misses incurred by BzTree in the Figure 7. We found that BzTree incurs around 6x more L3 misses than Dash, pointing to the fact that it does not exploit the caches efficiently. On further investigation, we also find that for the single-threaded 50% write workload in Fig. 4a, around 85% of the time is spent in leaf node insertion, which involves a PMwCAS operation. Majority of the rest 15% of the time is spent traversing the tree. This traversal time itself is around 400 ns, greater than Dash's median latency for this workload.

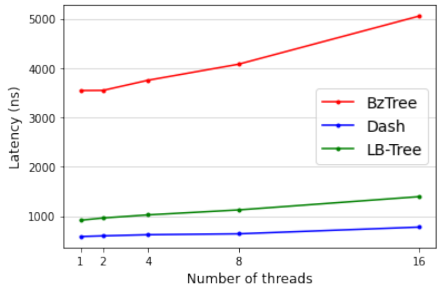
BzTree also pays the additional penalty of storing the non-leaf nodes in PMem, which results in increased time to traverse to the leaf node. Both BzTree and LB+-tree keep the inner nodes sorted and locate the appropriate node at the next level by performing a binary search on inner node's keys.

5.5 Tail Latency

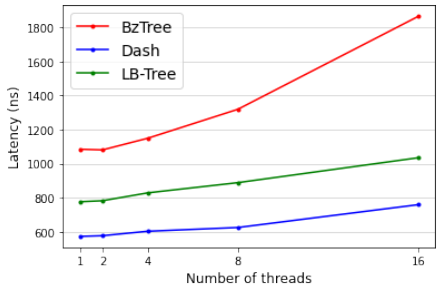
Tail (99% tile) latency results in Figure 5 show the same trend as median latencies, with Dash the lowest, followed by LB+-tree and BzTree. In Figure 5a, BzTree tail latency for the 50% write workload is 30 times that of Dash. For



(a) 50% writes

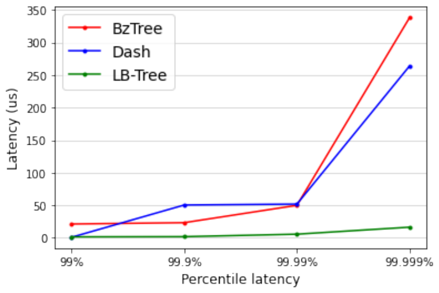


(b) 10% writes

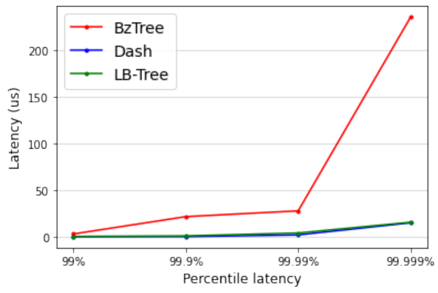


(c) Read-only

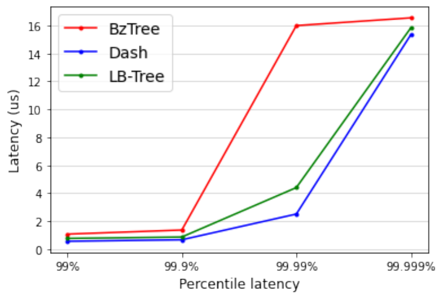
Figure 5: Tail (99 %tile) latency results



(a) 50% writes



(b) 10% writes



(c) Read-only

Figure 6: Latency predictability results

a single-thread, around 75% of the time is spent in the occasional node split operation in the B+-tree. In comparison, node split operation in LB+-tree takes much less time (around 800 ns) than a node split in BzTree, which takes around 16000 ns. This difference may be partly due to the way in which both indexes perform node splits. While Bztree allocates three new nodes - two new sibling nodes and their immediate parent, LB+-tree only allocates one new node (sibling to the node that is split).

For the single-threaded read-only scenario in Figure 5c, the BzTree tail latency is 1085 ns. This is high compared to its corresponding median latency of 679 ns (in Figure 4c). This is due to increase in linear search time within the leaf node. This may be because the leaf node has not yet reached the minimum threshold required for its consolidation, where it creates a new node and copies all records to it in a sorted manner to improve search performance. This points to the fact that BzTree may benefit from some sort of fingerprinting mechanism like in Dash and LB+-tree to speed up this linear search and reduce tail latencies.

5.6 Latency Predictability

We plotted latency at various percentiles for a single thread in Figure 6 to infer about the predictability of the three indexes. We found that for 50% writes in Figure 6a, 99.999% tile latency in Dash is comparable to BzTree, while LB+-tree maintains very predictable latencies in all scenarios. High 99.999% tile latency in BzTree and Dash is attributed to the occasional node and segment splits respectively. Dash does employ a load balancing strategy to delay segment splits and increase space utilization. In the 10% write workload, there would be even fewer splits and therefore the latency

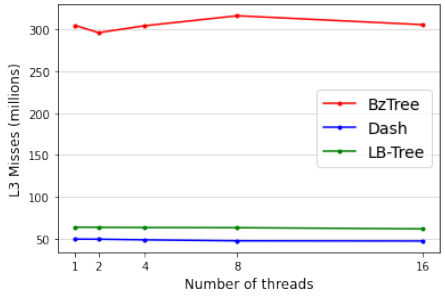


Figure 7: Read-only workload - L3 misses across number of threads

is not much affected. On the other hand, latency is more predictable for LB+-tree because of lower overhead of node splits. Even at 99.999% tile, a major chunk of time is spent in inserting the key in leaf node. The measurements for read-only workload in Figure 6c seem to fluctuate beyond 99.9 %tile but they are all below 20 us and are not as significant as seen for 50% and 10% writes.

6. DESIGN CONSIDERATIONS

In this section, we highlight some design considerations for building indexes for Persistent Memory. First, reduce PMem reads and writes as much as possible. Having 256 byte nodes (for B-trees) or buckets (for hash-based indexes) can maximize bandwidth utilization, since this is the internal data access bandwidth in Intel PMem. Second, fingerprinting is a common technique adopted in Dash, LB+-tree and other indexes [7, 13, 18] to improve searching on unsorted leaf nodes and buckets. Third, hybrid storage is also a popular design choice, especially for tree-based indexes where non leaf nodes are stored in DRAM to improve search performance.

Finally, another area to consider when designing PMem indexes is concurrency control, between hardware-based mechanisms like HTM and software-based approaches like PMwCAS. Certain limitations of HTM mean that it is suitable for small key/value sizes (as in our case) whereas PMwCAS works better for larger sizes [12].

7. FUTURE WORK

In future, it needs to be seen whether Copy-on-Write (CoW) is a bad fit for persistent memory storage as it is known to amplify the number of expensive persistent memory accesses and consume additional persistent memory bandwidth. Also, due to time constraints we have used different implementations of the indexes available publicly on Github instead of implementing them from scratch. This could potentially introduce noise in the results.

Since Intel Optane Persistent Memory is still a relatively new technology, actual hardware with Persistent Memory is not available readily and currently not publicly supported by any cloud providers. Benchmarking these indexes on real persistent memory hardware would help quantify the deviation in measurements when using hardware emulation techniques. Lastly, it remains to be seen how the overhead of various concurrency control mechanisms contribute to our results.

8. CONCLUSION

In this paper, we have studied the performance of three indexes on persistent memory: BzTree and LB+-Tree which are B-tree based indexes and Dash, which is a hash-based index. We made use of PiBench to benchmark these indexes on three different workloads. The results show that Dash performs the best, followed by LB+-Tree and BzTree. The poor performance of BzTree has been attributed to it not being optimized for cache locality. We also analyzed other design decisions such as the node structure and concurrency control mechanisms employed by these indexes and have provided some design considerations to be kept in mind when designing persistent memory based indexes.

9. MEMBER CONTRIBUTIONS

All members had equal contributions in the report and presentation. The remaining individual contributions are listed below:-

- **Kumar Biplav**: Worked on setting up the Pibench and collected benchmarking data for all the three indexes.
- **Rocil Machado**: Worked on setting up Dash on Cloudlab machine and wrote Pibench wrapper for Dash.
- **Nisarg Shah**: Worked on setting up LB+-tree on Cloudlab machine and wrote Pibench wrapper for LB+-tree. Also worked on evaluating performance bottlenecks in BzTree.
- **Shaurya Shekhar**: Worked on setting up BzTree on Cloudlab and the PiBench wrapper for BzTree.

10. REFERENCES

[1] 3D XPoint™: A Breakthrough in Non-Volatile Memory Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html, 2020.

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

[3] 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.*, 11(5):553–565, Jan. 2018.

[4] H. Cha, M. Nam, K. Jin, J. Seo, and B. Nam. B^3-tree: Byte-addressable binary b-tree for persistent memory. *ACM Trans. Storage*, 16(3), July 2020.

[5] S. Chen and Q. Jin. Persistent b¡sup¿+¡/sup¿-trees in non-volatile main memory. *Proc. VLDB Endow.*, 8(7):786–797, Feb. 2015.

[6] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[7] A. Kipf, D. Chromejko, A. Hall, P. Boncz, and D. G. Andersen. Cuckoo index: A lightweight secondary index structure. *Proc. VLDB Endow.*, 13(13):3559–3572, Sept. 2020.

[8] L. Lersch, X. Hao, I. Oukid, T. Wang, and T. Willhalm. Evaluating persistent memory range indexes. *Proc. VLDB Endow.*, 13(4):574–587, Dec. 2019.

[9] J. Liu, S. Chen, and L. Wang. Lb+trees: Optimizing persistent index performance on 3dxpoint memory. *Proc. VLDB Endow.*, 13(7):1078–1090, Mar. 2020.

[10] M. Liu, J. Xing, K. Chen, and Y. Wu. Building scalable nvm-based b+tree with htm. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, New York, NY, USA, 2019. Association for Computing Machinery.

[11] B. Lu, X. Hao, T. Wang, and E. Lo. Dash: Scalable hashing on persistent memory. *Proc. VLDB Endow.*, 13(10):1147–1161, Apr. 2020.

[12] D. Makreshanski, J. Levandoski, and R. Stutsman. To lock, swap, or elide: On the interplay of hardware transactional memory and lock-free indexing. *Proc. VLDB Endow.*, 8(11):1298–1309, July 2015.

[13] A. Mathew and C. Min. Hydralist: A scalable in-memory index using asynchronous updates and partial replication. *Proc. VLDB Endow.*, 13(9):1332–1345, May 2020.

[14] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 371–386, New York, NY, USA, 2016. Association for Computing Machinery.

[15] T. Wang, J. Levandoski, and P. Larson. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 461–472, 2018.

[16] T. Wang, J. Levandoski, and P. Larson. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 461–472, 2018.

[17] J. Yang, Q. Wei, C. Wang, C. Chen, K. L. Yong, and B. He. Nv-tree: A consistent and workload-adaptive tree structure for non-volatile memory. *IEEE Transactions on Computers*, 65(7):2169–2183, 2016.

[18] X. Zhou, L. Shou, K. Chen, W. Hu, and G. Chen. Dptree: Differential indexing for persistent memory. *Proc. VLDB Endow.*, 13(4):421–434, Dec. 2019.

[19] P. Zuo, Y. Hua, and J. Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 461–476, USA, 2018. USENIX Association.