LB+-Trees: Optimizing Persistent Index Performance on 3DXPoint Memory

Jihang Liu[1,2] Shimin Chen[1,2*] Lujun Wang[3]

[1]SKL of Computer Architecture, ICT, CAS
[2]University of Chinese Academy of Sciences
[3]Alibaba Group
{*liujihang,chensm*}*@ict.ac.cn, lujun.wlj@alibaba-inc.com*

ABSTRACT

3DXPoint memory is the first commercially available NVM solution targeting mainstream computer systems. While 3DXPoint conforms to many assumptions about NVM in previous studies, we observe a number of distinctive features of 3DXPoint. For example, the number of modified words in a cache line does not affect the performance of 3DXPoint writes. This enables a new type of optimization: performing more NVM word writes per line in order to reduce the number of NVM line writes. We propose LB$^+$-Tree, a persistent B$^+$-Tree index optimized for 3DXPoint memory. LB$^+$-Tree nodes are 256B or a multiple of 256B, as 256B is the internal data access size in 3DXPoint memory. We propose three techniques to improve LB$^+$-Tree's insertion performance: (i) Entry moving, which reduces the number of NVM line writes for insertions by creating empty slots in the first line of a leaf node; (ii) Logless node split, which uses NAW (NVM Atomic Write) to reduce logging overhead; and (iii) Distributed headers, which makes (i) and (ii) effective for multi-256B nodes. Theoretical analysis shows that entry moving reduces the number of NVM line writes per insertion of the traditional design by at least 1.35x in a stable tree. Our micro-benchmark experiments on a real machine equipped with 3DXPoint memory shows that LB$^+$-Tree achieves up to 1.12–2.92x speedups over state-of-the-art NVM optimized B$^+$-Trees for insertions while obtaining similar search and deletion performance. Moreover, we study the benefits of LB$^+$-Tree in two real-world systems: X-Engine, a commercial OLTP storage engine, and Memcached, an open source key-value store. X-Engine and Memcached results confirm our findings in the micro-benchmarks.

PVLDB Reference Format:
Jihang Liu, Shimin Chen, Lujun Wang. LB+-Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *PVLDB*, 13(7): 1078-1090, 2020.
DOI: https://doi.org/10.14778/3384345.3384355

1. INTRODUCTION

NVM technologies, including PCM [20], STT-RAM [2], and Memristor [23], have been studied for over a decade as a promising solution to address the DRAM scaling problem. In April 2019, Intel started to ship 3DXPoint based Intel Optane DC Persistent Memory DIMM products[1]. This is the first commercially available NVM solution targeting mainstream computer systems [10, 1].

Based on publicly available information [1, 13] and our experimental micro-benchmark study to understand the characteristics of 3DXPoint [15], we find that 3DXPoint conforms to many assumptions about NVM in previous studies: (i) Like DRAM, the data access granularity from CPU to 3DXPoint is 64-byte cache lines; (ii) 3DXPoint is modestly (i.e. 2–3x) slower than DRAM, but orders of magnitude faster than flash and HDDs; (iii) 3DXPoint writes are slower than reads; and (iv) Using special instructions (`clwb` and `sfence`) to persist data from CPU cache to 3DXPoint can drastically slow down write performance by up to an order of magnitude.

However, we also observe a number of distinctive features of 3DXPoint: (i) The number of modified words in a cache line does not affect the write performance; (ii) The internal data access granularity in 3DXPoint memory is 256B; and (iii) 3DXPoint performance degrades as applications access more data. These observations guide software design considerations and enable new optimization opportunities. One important new design principle is to reduce the number of NVM line writes rather than NVM word writes as in previous studies [5, 6] because the write content does not impact the NVM write performance. Therefore, we can perform *more* NVM word writes when a line has to be written to NVM in order to reduce the number of future NVM line writes.

Based on these observations, we propose LB$^+$-Tree, a persistent B$^+$-Tree index optimized for 3DXPoint memory. We set the tree node size to be 256B or a multiple of 256B because this choice best utilizes the internal data access bandwidth in 3DXPoint modules.

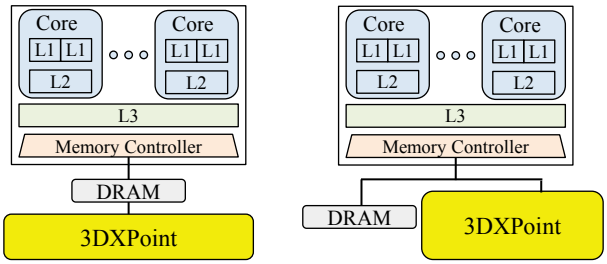
We propose the following three techniques to improve LB$^+$-Tree's insertion performance:

- *Entry moving:* An insertion inserts a new index entry and updates the header in a leaf node. If the insertion finds an empty slot in the same line (i.e. the first line) as the header, then it can use one NVM line write to both insert the new entry and update the header. This is the best case. On the other hand, if the first line is full, the insertion finds an empty slot in another line in the leaf node and incurs two NVM line writes, one for updating the header, one for writing the new entry. We take this opportunity to *actively* create empty slots in the first line 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.
- *Logless node split:* Previous NVM-optimized B$^+$-Trees rely on logging to support node splits. However, logging incurs

*Corresponding author

Proceedings of the VLDB Endowment, Vol. 13, No. 7
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3384345.3384355

[1]We use 3DXPoint and Intel Optane DC Persistent Memory interchangeably in this paper.



(a) Memory mode (b) App-direct mode

Figure 1: 3DXPoint NVDIMM modes.

extra NVM write and persist cost. Instead, we use NAW (NVM Atomic Write) to switch between two alternative sibling pointers for node splits.

- *Distributed headers:* For multi-256B nodes, traditional designs put all the meta-information in a centralized header in the beginning of the node. Examples of meta-information includes bitmap to identify valid and empty index entry slots and fingerprints to facilitate key search. However, the number of entry slots in the first line will decrease as the node size increases. This makes the entry moving optimization less effective. With the distributed headers technique, we distribute a piece of the header to every 256B block in the node. We also design a set of alternative headers. In this way, both entry moving and logless node split can remain effective for multi-256B nodes.

We prove the correctness of our proposed algorithms. That is, the insertion and deletion algorithms of LB$^+$-Trees are guaranteed to keep the LB$^+$-Tree leaf nodes in a consistent state in NVM memory in light of power failures and process crashes. We analyze the cost of insertions and show that entry moving reduces the number of NVM line writes per insertion of the traditional design by at least 1.35x in a stable tree.

We compare LB$^+$-Tree and state-of-the-art NVM optimized index structures on a real machine equipped with 3DXPoint memory. Our micro-benchmark experiments show that LB$^+$-Tree achieves up to 1.12–2.92x speedups over state-of-the-art NVM optimized B$^+$-Trees, WB$^+$-Tree [6] and FP-Tree [18], for insertions while obtaining similar search and deletion performance. We study the benefits of LB$^+$-Tree in two real-world systems: X-Engine, a commercial OLTP storage engine, and Memcached, an open source key-value store. X-Engine experiments show that LB$^+$-Trees achieve better performance than skip lists, which are used in X-Engine. Memcached experiments confirm our micro-benchmark findings.

Contributions. The contributions of this paper are fourfold. First, we present three design principles based on our observation of 3DXPoint characteristics and the analysis of existing NVM-optimized B$^+$-Tree studies. Second, we propose LB$^+$-Tree, which exploits entry moving, logless node split, and distributed headers for better insertion performance. Third, we formally prove the correctness of the proposed algorithms using the concept of NAW sections, and we present a theoretical analysis of the insertion cost. Finally, we preform micro-benchmark experiments and experiments with two real-world systems to show the benefits of LB$^+$-Tree.

Organization. The rest of the paper is organized as follows. Section 2 describes 3DXPoint characteristics, discusses related work, and presents the three design principles. Section 3 describes the design of LB$^+$-Tree. Section 4 evaluates the proposed solution using a real machine equipped with 3DXPoint memory. Finally, Section 5 concludes the paper.

2. BACKGROUND AND MOTIVATION

We begin by describing 3DXPoint memory in Section 2.1. We pay special attention to the similarities and differences between the characteristics of 3DXPoint and the NVM assumptions in previous simulation or emulation based studies. Then, we examine the main schemes to support persistent data structures in NVM memory in Section 2.2, and delve into the NAW scheme in Section 2.3. After that, we discuss existing NVM optimized B$^+$-Tree design choices in Section 2.4. Finally, we rethink the design principles for optimizing B$^+$-Trees on 3DXPoint memory in Section 2.5.

2.1 3DXPoint Memory

3DXPoint based Intel Optane DC Persistent Memory is the first NVM main memory solution available for mainstream computer systems [10, 1]. The product is packaged in the NVDIMM format and plugs into standard DDR4 DIMM sockets. It uses the proprietary DDR-T protocol, which revises the DDR4 protocol to support asynchronous operations. The capacity of an NVDIMM is 128GB — 512GB. A dual-socket machine can be equipped with up to 12 NVDIMMs or up to 6TB of NVM memory.

There are two main NVDIMM configuration modes, as shown in Figure 1. In the memory mode, DRAM is managed as a cache for 3DXPoint memory by the memory controller. The main memory capacity is equal to the total size of the 3DXPoint memory. This mode enables un-modified applications to exploit the large main memory capacity provided by 3DXPoint. However, there are two drawbacks. First, the memory 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 3DXPoint because a load consists of a DRAM cache visit (miss) followed by a 3DXPoint 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, 3DXPoint and DRAM are both directly accessed by the CPU. 3DXPoint modules are recognized as special devices by the OS. We can install file systems on 3DXPoint modules and use PMDK (Persistent Memory Development Kit)[2] to map a file from 3DXPoint into the virtual memory space of an application. The OS runs DAX device drivers for 3DXPoint so that accesses to 3DXPoint get around the OS page cache. In this way, applications can directly access 3DXPoint using load and store instructions, and implement persistent data structures in 3DXPoint. In order to support persistent B$^+$-Trees, we focus on the app-direct mode of 3DXPoint NVDIMMs in this paper.

We have performed an experimental micro-benchmark study to understand the characteristics of 3DXPoint [15]. Based on our study and publicly available information [1, 13], we find that 3DXPoint conforms to many assumptions about NVM in previous studies: (i) Like DRAM, the data access granularity from CPU to 3DXPoint is 64-byte cache lines. (ii) 3DXPoint is modestly (i.e. 2-3x) slower than DRAM, but orders of magnitude faster than flash and HDDs. (iii) The read and write performance of 3DXPoint is asymmetric. 3DXPoint writes are slower than reads. (iv) Using special instructions (`clwb` and `sfence`) to persist data from CPU cache to 3DXPoint can drastically slow down write performance by up to an order of magnitude.

We also observe a number of distinctive features of 3DXPoint:

- *Observation 1: The number of modified words in a cache line does not affect the write performance.* Previous studies proposed various techniques (e.g., data comparison write, flip-N-

[2] http://pmem.io/pmdk/libpmem/

write) to improve NVM write performance by writing a subset of bits instead of the full cache line [22, 7, 9, 11]. For example, in data comparison write, the hardware reads the original line, compares the modified line with the original line to find the modified bits, and writes only the modified bits to the underlying NVM memory. The write latency is lower if fewer words are modified in the line. Since NVM reads are faster than NVM writes, data comparison write can significantly improve write performance. Since the effect of such techniques contradicts with Observation 1, we believe that optimizations such as data comparison write are not implemented in 3DXPoint. An explanation is that Intel Optane DC Persistent Memory implements hardware encryption to ensure data security from physical intrusion. A bit change in the original data can lead to many bit changes in the encrypted result (a.k.a. the avalanche effect [21]). Thus, it would be difficult to support write content based optimizations [24]. Note that hardware encryption is a desirable feature for NVM memory to protect sensitive information. Therefore, we argue that Observation 1 will be applicable beyond 3DXPoint to future NVM memory techniques with hardware encryption support.

- *Observation 2: The internal data access granularity in 3DXPoint memory is 256B.* To serve a 64B cache line read, 3DXPoint reads 256B internally and returns the 64B line. To serve a 64B cache line write, 3DXPoint reads 256B, modifies 64B, and writes the modified 256B internally. Therefore, applications can achieve better memory bandwidth with 256B reads. 256B writes can also be beneficial under the condition that there are no data persist operations in between writes.
- *Observation 3: 3DXPoint performance degrades as the working set size of an application increases.* This may be due to techniques such as caching in 3DXPoint memory modules. We find that the memory performance becomes stable when the size of the data accessed by applications is 1/8 or more of the total capacity of a 3DXPoint memory module.

Finally, NVM technologies, such as PCM, have endurance issues. The number of writes to a PCM cell before it wears out is about $10^8 - 10^9$. Therefore, wear-leveling and improving NVM endurance have been important topics in the literature [25, 7]. However, there has been no official information from Intel about 3DXPoint's endurance. Nevertheless, our techniques in this paper aim to reduce NVM writes for index insertions, which will also improve NVM endurance.

2.2 Schemes to Achieve Data Persistence

It is challenging to design persistent data structures in NVM memory. While NVM is non-volatile, CPU cache is still volatile, and its contents get lost upon power failure. Normal data store instructions complete when the data are written to CPU cache. However, the hardware controls when and in what order dirty cache lines are written back to main memory. Therefore, it is necessary to use special cache line flush instructions (e.g., `clwb`) followed by a memory fence instruction (e.g., `sfence`) to guarantee modified data are written back to NVM. We call `clwb` instructions followed by a `sfence` instruction a *persist* operation. As persist operations incur drastic overhead as discussed in Section 2.1, it is important to reduce not only NVM writes but also persist operations in the design of persistent data structures in NVM memory.

There are four software schemes to achieve data persistence in NVM memory: *logging* [17], *shadowing* [8], *PMwCAS* [3], and *NVM atomic writes (NAW)* [6]. Here, atomicity means that the write to NVM either succeeds or fails in its entirety in light of power failure and process crashes. An NAW is an 8B write followed by a persist operation. Note that data persistence and concurrency control are orthogonal issues. NAWs should not be confused with atomic instructions (e.g., compare-and-swap), which are atomic in light of instructions executed by other processor cores.

In write-ahead *logging*, we protect an NVM write by logging the write intention into a log in the NVM memory. We first write and persist a log record containing (address, old value, new value) to the log, then perform the actual NVM write without persisting it. During failure recovery, we check the log. If the log record does not exist, then the NVM write must not have occurred. If the log record exists, we compare the value at the address with the logged values, and can perform either undo or redo to achieve a consistent state for the persistent data structure.

In *shadowing*, we do not directly modify the persistent data structure. Instead, we create a shadow copy of the portion of the data structure to be modified in a newly allocated space in NVM, modify and persist the shadow copy, and then use a single NAW to switch the old copy and the new copy in the persistent data structure. Note that this requires that there is a single pointer to the portion of data structure to be modified (e.g., the pointer to a B^+-Tree node in its parent node). During failure recovery, the pointer points to either the old copy or the newly written and persisted copy. Therefore, the data structure is in a consistent state.

The *PMwCAS* is a mechanism to support persistent multi-word compare-and-swap operations. First, it records the address, the old value, and the new value for every target word in a descriptor buffer. Second, it persists the descriptor buffer. Third, it writes and persists the pointer to the descriptor buffer in every target word. Finally, it performs the actual writes to the target words. *PMwCAS* can be used to achieve both data persistence and latch free operations for data structures.

Logging and shadowing incur extra NVM write and persist cost for writing logs and creating new copy of data, respectively. *PMwCAS* incurs extra overhead for writing and persisting the descriptor buffer and the descriptor pointer to the target words. Such cost can be much higher than directly modifying the persistent data structure. Therefore, recent studies aim to use light-weight NAWs to make direct changes to the persistent data structure, thereby avoiding the overhead of logging and shadowing. We discuss NAW in detail in the following subsection.

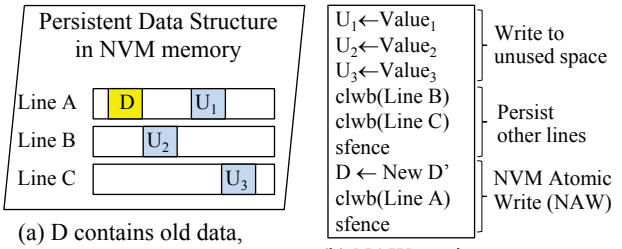
2.3 NVM Atomic Write (NAW)

An NVM atomic write (NAW) is an 8B word write followed by a `clwb` and a `sfence` to persist the data to NVM memory. It is guaranteed that the NVM stores the entire 8B word atomically. That is, upon failure, the NVM contains either the new 8B word or the original 8B word. The 8B data can never be partially written.

We often use an NAW in an *NAW section*. The basic idea is to prepare new changes in previously unused space so that the modifications do not impact the correctness of a data structure upon failure. Then we persist the new changes, and use a single NAW to switch the state of the data structure to include the new changes.

DEFINITION 2.1 (**NAW SECTION**). *An NAW section is a code section that consists of zero or more NVM writes to unused space, followed by a set of* `clwb` *and a single* `sfence` *to persist lines containing the modified unused space except the line containing D, and a single NAW to modify an old data item D.*

Figure 2 shows an example NAW section. The NAW section first writes new values to previously unused space U_1, U_2, and U_3. Then it persists lines B and C. Since U_1 is in the same line as D, Line A is not persisted. Finally, it uses an NAW to modify D to a new value D'. This NAW persists the entire Line A, which contains the newly modified U_1. D' typically indicates that the previous unused



(a) D contains old data, U1,U2,U3 are unused

(b) NAW section

Figure 2: NAW (NVM Atomic Write) section.

U_1, U_2, and U_3 are now taken by new data items. Upon failure, if the NAW has not yet occurred, D reflects the old state of the data structure, regarding U_1, U_2, and U_3 as unused space. If the NAW succeeds, then U_1, U_2, and U_3 must also be written correctly in NVM. D' reflects the correct new state.

We formally prove the correctness guarantee of an NAW section:

THEOREM 2.1 (**NAW SECTION PERSISTENCE**). *It is guaranteed that upon failure the data structure protected by the NAW section is either in the old state before the NAW section or in the new state after the NAW section.*

PROOF. When there is a failure, the NAW either succeeds or fails. If it fails, then the data structure is in the old state because the writes to unused space do not change the state of the data structure. If the NAW succeeds, then both the new data items and D' are correctly written to NVM. The data structure is in the new state after the NAW section. Therefore, the data structure is guaranteed to be either in the old state or in the new state. □

What if a state transition requires modifying multiple used locations? In general, NAW cannot support such complex state transitions. We have to fall back to more costly schemes, such as logging. For some data structures (e.g., B$^+$-Tree leaf nodes [6]), it is possible to design multiple *intermediate recoverable states* and use multiple NAW sections to achieve data persistence. As illustrated in Figure 3, the state transition requires to modify k words $D_1, \cdots, D_k$. We reserve 1 bit in D_1 to show if the state transition is ongoing. NAW section 1 sets the ongoing flag in D_1. The NAW section 2–k modify $D_2, \cdots, D_k$, respectively. $D_2, \cdots, D_k$ are all recoverable. The last NAW section clears the flag and updates D_1. Upon failure, the flag in D_1 can be checked to see if the transition is complete. If it is complete, then the data structure is in the new state. If the transition is ongoing, then a recovery function rolls back the data structure to the old state. In this way, multiple NAW sections can protect complex state transitions.

2.4 Existing NVM Optimized B+-Tree Designs

There are several considerations in designing NVM optimized persistent B$^+$-Trees in previous studies [5, 6, 18, 3].

First, CPU cache performance is important because the trees are in main memory. Therefore, NVM optimized B$^+$-Trees are often based on cache-optimized B$^+$-Trees [19, 4]. The nodes in cache-optimized B$^+$-Trees are aligned to cache line boundaries. Their size is one or a few cache lines large.

Second, NVM writes are to be reduced as much as possible. Chen et al. [5] proposed to make the leaf nodes unsorted with a bitmap to indicate the valid entries and empty slots. If a leaf node is not full, then an insertion writes to an empty slot. Compared to traditional sorted leaf nodes, unsorted leaf nodes reduce the cost of moving existing valid entries to ensure the key order.

Third, NAW is the preferred scheme to achieve data persistence. Chen et al. [6] proposed WB$^+$-Tree, whose leaf node structure contains a sorted indirection array to facilitate binary search in unsorted leaf nodes. However, both the bitmap and the indirection array need to be modified for an insertion. The solution is to use two NAW sections with an intermediate state, which is recoverable because the indirection array can be rebuilt by reading the valid index entries in the leaf node.

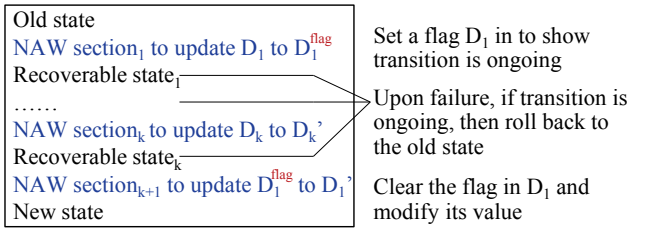


Figure 3: Using multiple NAW sections to protect a general state transition when the intermediate states are recoverable.

Fourth, non-leaf search can be improved by placing non-leaf nodes in DRAM as DRAM reads are faster than NVM reads. Oukid et al. [18] proposed the selective persistence technique to place non-leaf nodes of FP-Tree in DRAM while keeping leaf nodes in NVM[3]. The non-leaf nodes can be rebuilt from the leaf nodes upon power failure. In this way, the search in the non-leaf part of the tree can be significantly improved.

Other Considerations. Oukid et al. [18] proposed to compute and store a 1-byte fingerprint per key in a fingerprint array in every leaf node in FP-Tree. Then the fingerprint of the search key can be compared with multiple fingerprints using SIMD instructions to speed up search in leaf nodes. Moreover, previous studies also consider concurrency control for B$^+$-Trees in NVM. FP-Tree exploits hardware transactional memory and locking bits in leaf nodes for concurrency control purpose [18]. In comparison, Arulraj et al. [3] proposed to use `PMwCAS` to implement latch free and NVM persistent operations on BzTree.

2.5 Design Principles for 3DXPoint Memory

Our goal is to optimize B$^+$-Tree designs for 3DXPoint memory. As 3DXPoint conforms to many of the NVM assumptions in previous studies, we will follow the above design considerations for NVM optimized B$^+$-Trees as the starting point of our design.

We propose the following three design principles:

- *Design principle 1: reduce the number of NVM line writes rather than NVM word writes.* First, according to Observation 1, the number of words written in a cache line does not impact the NVM write performance. Second, the granularity of persist operations are cache lines, and persist operations are more costly than NVM writes. Hence, it is important to reduce the number of NVM line writes. This principle enables a new type of optimization: performing more NVM word writes per line in order to reduce the number of NVM line writes.
- *Design principle 2: set node size to be a multiple of 256B.* Based on Observation 2, a multiple of 256B best utilizes the internal data access bandwidth in 3DXPoint modules.
- *Design principle 3: completely remove logs and use NAW for node splits.* Existing studies perform NAWs to improve insertions to leaf nodes that are not full. However, they still rely on logging to support node splits when inserting into full leaf

[3]The capacity ratio R between NVM and DRAM must be considered. Note that if the average branching factor is B, then the non-leaf part of the tree in DRAM takes roughly B times smaller space than the leaf nodes in NVM. It is advisable to ensure B to be similar to or larger than R. The capacity of a single DDR4 DRAM DIMM is up to 32GB large. The capacity of a 3DXPoint NVDIMM module is up to 512GB. Hence, R= 16.

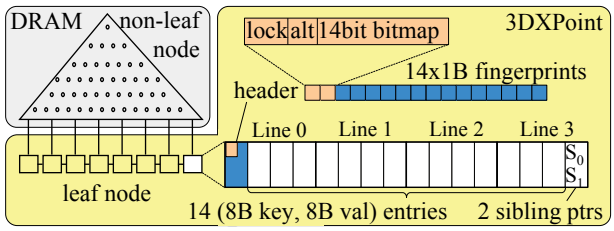


Figure 4: The LB$^+$-Tree structure with 256B nodes.

nodes. We would like also to improve node split performance by exploiting NAWs.

In addition, Observation 3 gives a constraint on experimental setup. We would like to model the situation where the application (e.g., DBMS) utilizes a large portion of the available NVM space. To achieve this, we must set the data set size in the experiments to be at least 1/8 of the 3DXPoint capacity. If the data size is too small, the measured performance may not be representative according to Observation 3.

3. LB+-TREE DESIGN

We propose the LB$^+$-Tree index, a persistent B$^+$-Tree optimized for 3DXPoint that minimizes the number of NVM **L**ine writes and performs **L**ogless insertions. Our LB$^+$-Tree design follows the proven design choices of previous NVM optimized B$^+$-Trees as a starting point, and applies the three design principles for 3DXPoint memory. We set the node size of LB$^+$-Trees to be 256B or a multiple of 256B (Design principle 2).

In the following, we first present our design for LB$^+$-Trees with 256B nodes in Section 3.1–3.3. We overview the tree structure in Section 3.1. We describe how the leaf insertion algorithms reduce the number of NVM line writes (Design principle 1) and achieve logless leaf node splits (Design principle 3) in Section 3.2 and Section 3.3, respectively. Then, we extend the design to support node sizes that are multiples of 256B in Section 3.4. Finally, we analyze the cost of LB$^+$-Trees in Section 3.5.

3.1 Overview of LB+-Trees with 256B Nodes

Tree Structure. The tree structure of the 256B-node LB$^+$-Tree is illustrated in Figure 4. The tree node size is 256B. All nodes are aligned at 256B boundaries. As 256B is a multiple of cache line size (64B), an LB$^+$-Tree is a cache-optimized B$^+$-Tree. The tree consists of two parts: non-leaf nodes and leaf nodes. As 3DXPoint is slower than DRAM, we follow the FP-Tree to place non-leaf nodes in DRAM and leaf nodes in 3DXPoint to improve the performance of the non-leaf part of the tree. Note that while non-leaf nodes are volatile in this design, they can be reconstructed from the persistent leaf nodes in 3DXPoint during failure recovery.

A 256B leaf node consists of four 64B lines, as depicted in Figure 4. A (8B key, 8B value) index entry takes 16B. There are four 16B units in every line and 16 units in total in the entire node. We use 14 of the 16 units to store index entries.

The first unit in Line 0 stores the 16B header. It consists of a 16-bit bit array and a 14B fingerprint array. The bit array contains a 14-bit bitmap, each bit of which corresponds to an index entry slot. "1" means that the slot is occupied by a valid entry. "0" means that the slot is empty. The other two bits are the $lock$ bit for concurrency control, and the alt bit, which specifies one of the two alternative sibling pointers. The fingerprint array improves search computation inside the leaf node. $hash(key)$ computes a 1B fingerprint for a given key. The 14 1B-fingerprints correspond to the 14 index entry slots in the node. Thus, a single 128-bit SIMD instruction can compare the fingerprint of a search key with the 14 fingerprints to quickly locate the slots that contain potential matches.

The last unit in Line 3 contains two 8B sibling pointers. The alt bit in the header chooses which pointer is presently in use. The effective sibling pointer points to the right sibling of the leaf node, or NULL if the node is the right-most leaf node in the tree.

Algorithm 1: LB$^+$-Tree search.

```
1  Function LBTreeSearch(Root r, Key k)
2      Again:
3      if _xbegin() != _XBEGIN_STARTED then goto Again;
4      leaf= SearchNonLeaf(r);
5      if leaf.lock == 1 then _xabort(); goto Again;
6      match_set= SimdSearch(hash(k), leaf.fgprt);
7      found= False; val= 0;
8      foreach slot ∈ match_set do
9          if (leaf.bitmap[slot]==1) and
             (leaf.entry[slot].key==k) then
10             found=True; val= leaf.entry[slot].val; break;
11     _xend();
12     return (found, val);
```

Concurrency Control. We follow the FP-Tree to combine hardware transactional memory (HTM) and a $lock$ bit per leaf node for concurrency control. Our implementation uses primitives of Intel Transactional Synchronization Extensions (TSX) for HTM. However, it is straightforward to port the code to other processor architectures (e.g., IBM Power8, ARM) that support HTM.

Cache line flush instructions (e.g., `clwb`) cannot be used with HTM because HTM requires to keep all modified cache lines of a transaction in CPU cache so that the modifications are hidden from other processor cores before the transaction commits. However, such special instructions are necessary to persist data to 3DXPoint memory. This problem is solved by introducing the $lock$ bit.

An index operation starts an HTM transaction. When it reaches the leaf level, it checks the $lock$ bit in the leaf node to see if any index write operation (i.e. insertion or deletion) has locked the node. It aborts the transaction if $lock$=1, and continues only if $lock$=0. An index read operation then performs the search in the transaction. In contrast, an index write operation sets $lock$=1, commits the transaction, and performs the actual writes outside the transaction.

This concurrency control design has the following desired properties. First, concurrent index read operations can proceed at the same time because the transactions are read-only without modifying any fields in tree nodes. Second, an index write operation has exclusive access to its target leaf node. This is because the write to $lock$ conflicts with the read of $lock$ in concurrent transactions visiting the same node, and new transactions that see $lock = 1$ will abort. Third, `clwb` can be used outside of HTM transactions.

Basic Operations. The LB$^+$-Tree ***search*** algorithm is listed in Algorithm 1. Code line 3, 5, and 11 implement the above concurrency control design. In Intel TSX, $\_xbegin()$, $\_xabort()$, and $\_xend()$ starts, aborts, and commits the transaction, respectively. Successful $\_xbegin()$ returns $\_XBEGIN\_STARTED$. When a transaction aborts because of $\_xabort()$ or because hardware detects conflicts, the execution jumps to $\_xbegin()$ as if it returns an error code, and the algorithm retries. The search operation first searches the non-leaf nodes to find a leaf node. Then, it uses SIMD (e.g., `_mm_cmpeq_epi8`) to search the fingerprint array to get a $match\_set$ of candidate matches. Finally, it checks each potential match to verify if the slot is valid and the key is equal to the search key.

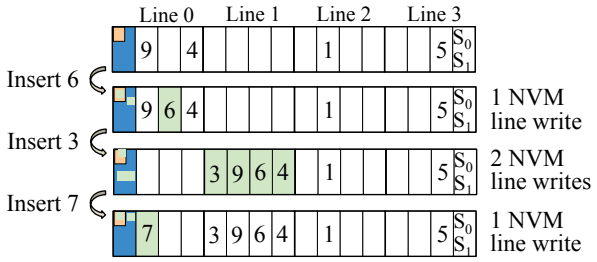


Figure 5: Inserting into a leaf node with empty slots.

The ***insertion*** operation is the focus in our study and will be described in detail in the next two sections. The ***deletion*** operation is simpler than the insertion operation. Given an index entry to delete, it performs a search to find the leaf node and the leaf slot where the entry is stored. Then it performs an NAW to update the bitmap, clearing the bit corresponding to the slot. Since only a single NAW is performed, it is guaranteed that the leaf node will be in a consistent state in NVM memory upon failure. Note that it is expected that data sets are growing and therefore empty slots resulting from deletions are likely to be quickly used by insertions. Consequently, we do not implement the node merge for deletions [14].

Since the deletion operation performs a single NAW, we can prove the following based on Theorem 2.1:

THEOREM 3.1. *The deletion algorithm for LB$^+$-Trees with 256-byte sized nodes is guaranteed to keep the leaf node in a consistent state in NVM memory upon failure.*

Crash Recovery. The index write operations (i.e. insertions and deletions) are guaranteed to maintain the linked list of leaf nodes in a consistent state in 3DXPoint memory, as discussed in the above and in Section 3.2–3.4. During crash recovery, we scan the linked list of leaf nodes to build the non-leaf nodes in DRAM.

The scan also checks the $lock$ bit in every leaf node and clears it if necessary. Note that $lock$ may be set in a leaf node as in the following scenario. An insertion to the leaf node sets $lock = 1$ and commits the transaction. The modified cache line is somehow evicted from CPU cache and written back to NVM. Then there is a power failure before the insertion finishes.

We measure the time to rebuild the non-leaf nodes during recovery in Section 4.2 and find that the rebuild time can be kept reasonably low (e.g., 0.1 second) for a tree that contains 2 billion index entries. One can further reduce the rebuild time by placing one or more levels of non-leaf nodes in 3DXPoint. This is an interesting tradeoff between index performance, complexity and rebuild time.

3.2 Reducing NVM Line Writes for Insertions

Given a new index entry (k, v), the insertion operation searches for the leaf node that k belongs to. Then, it finds an empty slot to store the entry and updates the fingerprint and the bitmap in the leaf header. The number of NVM line writes depends on whether the selected slot to store the new entry is in the same line (i.e. Line 0) as the header. The good case is when the slot is in Line 0. The insertion writes only one NVM line. When the slot is in a line different from Line 0, the insertion incurs two NVM line writes, doubling the cost. Therefore, we would like to apply Design principle 1 to increase the chance that there are empty slots in Line 0.

The basic idea of is depicted in Figure 5. After inserting 6, Line 0 is full. The insertion of 3 has to incur two NVM line writes. It writes Line 0 and Line 1. We take this opportunity to move the index entries in Line 0 to Line 1 to create empty slots in Line 0. As a result, the next insertion of 7 becomes the good case. It finds an empty slot in Line 0, and thus incurs only one NVM line write. Note that the number of word writes in a line does not impact the performance of NVM line write. This enables the entry moving optimization, which performs more NVM word writes in order to reduce the number of NVM line writes for future insertions.

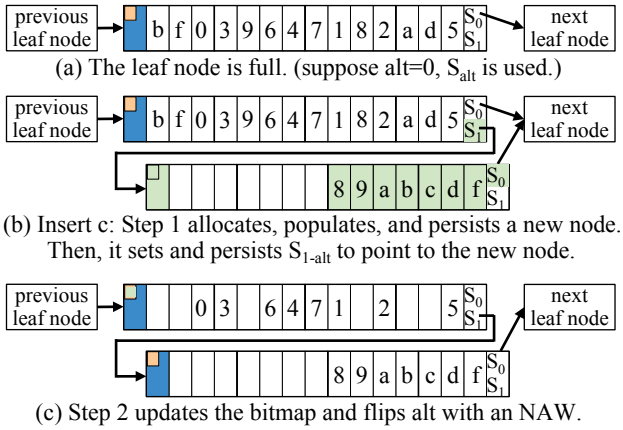


Figure 6: Logless leaf node split.

The LB$^+$-Tree insertion algorithm is listed in Algorithm 2. It distinguishes the good case (Code line 15–20) from the other cases (Code line 21–32), where it moves as many index entries as possible from Line 0 to the line that stores the new index entry. An interesting aspect of the algorithm is that it does not directly modify the leaf header. Instead, it copies the leaf header to a 16B $dword$ implemented as an array of two 8B words (i.e. 8B unsigned integers). It performs the fingerprint and bitmap modifications on $dword$. Finally, it writes $dword$ back to the leaf header using 8B word writes. As a result, the leaf header is modified only at judiciously selected places in the code, which makes it easy to reason about the correctness of the data persistence support.

THEOREM 3.2. *If the leaf node has empty slots, the insertion algorithm for 256B-node LB$^+$-Trees is guaranteed to keep the leaf node in a consistent state in NVM memory upon failure.*

PROOF. We consider the following two cases. Case 1: When the empty slot to insert the new entry is in Line 0, the algorithm performs an NAW section that writes the entry to an unused space, and performs a NAW to the first word in the header. Note that the first word contains the bitmap and 6 fingerprint slots, which cover all slots in Line 0. Case 2: When the empty slot is in Line $lineid = 1, 2,$ or 3, the code performs a number of writes to unused spaces in Line $lineid$. Then it writes to the second word in the header. This write may modify unused fingerprint slots. After that, it performs an NAW to the first word in the header. This forms an NAW section. From Theorem 2.1, we know that in both cases the leaf node will be in a consistent state in NVM memory. □

3.3 Logless Node Splits Using NAWs

The splitting performance of leaf nodes has been one of the major performance bottlenecks for NVM optimized B$^+$-Tree insertions. Existing designs rely on logging to support node splits, incurring extra NVM write and persist overhead.

We propose a logless node split design for LB$^+$-Trees, as illustrated in Figure 6. The basic idea is to have two alternative sibling pointers and use the alt bit in the leaf header to indicate which pointer is in use. The operation first allocates a new leaf node, copies half of the index entries from the existing leaf node, and inserts the new node into the linked list, using S_1 in the existing leaf node. Since S_1 is presently not in use, all these changes write to previously unused space. Then, we can use an NAW to write the leaf header, which both sets the alt bit to switch the sibling pointers

Algorithm 2: LB$^+$-Tree insert.

```
1  Function LBTreeInsert(Root r, Key k, Value v)
2      Again:
3      if _xbegin() != _XBEGIN_STARTED then goto Again;
4      leaf= SearchNonLeaf(r);
5      if leaf.lock == 1 then _xabort(); goto Again;
6      leaf.lock= 1;
7      _xend();
8      if leaf.isFull() then return LBTreeLeafSplit(leaf, k, v);
9      else
10         dword.word[0,1] = leaf.word8B[0,1];
11         return LBTreeLeafInsert(leaf, dword, k, v)
12 Function LBTreeLeafInsert(Leaf leaf, DWord dword, Key k,
     Value v)
13     slot= FindFirstEmptySlot(leaf.bitmap);
14     lineid= WhichLine(slot);
15     if lineid==0 then // slot is in Line 0
16         leaf.entry[slot]= (k, v);
17         word= dword.word[0];
18         word.fgprt[slot]= hash(k); word.bitmap[slot]= 1;
19         leaf.word8B[0]= word;
20         clwb (leaf.line[0]); sfence ();
21     else // slot is in Line 1–3
22         leaf.entry[slot]= (k, v);
23         dword.fgprt[slot]= hash(k); dword.bitmap[slot]=1;
24         line_empty_set=
             FindEmptySlotsInLine(dword.bitmap, lineid);
25         from= 0; // first entry in line 0
26         foreach to ∈ line_empty_set do // move entries
27             leaf.entry[to]= leaf.entry[from];
               dword.fgprt[to]= dword.fgprt[from];
               dword.bitmap[to]= 1; dword.bitmap[from]= 0;
28             from ++;
29         clwb (leaf.line[lineid]); sfence ();
30         leaf.word8B[1]= dword.word[1]
31         leaf.word8B[0]= dword.word[0];
32         clwb (leaf.line[0]); sfence ();
33     return DoneInsert;
```

in effect and clears bits in the bitmap for the empty slots. In this way, we avoid logging for leaf node splits.

The LB$^+$-Tree leaf node split algorithm is listed in Algorithm 3. At the beginning, the algorithm allocates a new node, and copies the largest 7 of the 14 index entries in the full leaf node to the new node. We store the copied entries in the last two lines so that future insertions to the new node can use empty slots in Line 0 for better performance. Then, sibling pointers are changed to insert the new node into the linked list without changing the current effective pointer. After that, the algorithm considers two cases. Code line 13–19 supports the case where the new entry to insert belongs to the new leaf. For the other case where the new entry belongs to the old leaf, Code line 20–23 persists the new leaf and the sibling pointer, then calls $LBTreeLeafInsert$ to insert the entry. Note that the $dword$ parameter to the call reflects the cleared bits in the bitmap and the flip of alt. Therefore, $LBTreeLeafInsert$ will find an empty slot using $dword$ and correctly update the leaf header.

THEOREM 3.3. *In the case where the leaf node splits, the insertion algorithm for 256B-node LB$^+$-Trees is guaranteed to keep the leaf nodes in a consistent state in NVM memory upon failure.*

Algorithm 3: LB$^+$-Tree leaf node split.

```
1  Function LBTreeLeafSplit(Leaf leaf, Key k, Value v)
2      newleaf= AllocLeafNode();
3      newleaf.word8B[0,1]= 0;
4      move_set= GetSlotsForTopKeys(leaf, 7);
5      dword.word[0,1] = leaf.word8B[0,1];
6      to= 7; // move to slot[7..13] in Line 3 and 4
7      foreach from ∈ move_set do // move entries
8          newleaf.entry[to]= leaf.entry[from];
           newleaf.fgprt[to]= leaf.fgprt[from];
           newleaf.bitmap[to]= 1; dword.bitmap[from]= 0;
9          to ++;
10     newleaf.sibling[0]= leaf.sibling[leaf.alt];
11     dword.alt= 1 - leaf.alt;
12     leaf.sibling[dword.alt]= newleaf;
13     if k > leaf.entry[move_set[0]].key then // key in new leaf
14         newleaf.entry[6]= (k, v); newleaf.fgprt[6]= hash(k);
           newleaf.bitmap[6]= 1;
15         clwb (newleaf.line[0..3]); clwb (leaf.line[3]);
16         sfence ();
17         leaf.word8B[1]= dword.word[1]
18         leaf.word8B[0]= dword.word[0];
19         clwb (leaf.line[0]); sfence ()
20     else // key in leaf
21         clwb (newleaf.line[0..3]); clwb (leaf.line[3]);
22         sfence ();
23         LBTreeLeafInsert(leaf, dword, k, v)
24     return DoneSplit;
```

PROOF. From the perspective of the LB$^+$-Tree, the new leaf node and the pointer $leaf.sibling[1 - leaf.alt]$ are unused space. Therefore, in the first case where the new entry is inserted into the new node, the code forms an NAW section. It writes to unused space, persists lines other than Line 0 of the old leaf node, and then performs a single NAW. In the second case where the new entry is inserted into the old node, the proof of Theorem 3.2 shows that $LBTreeLeafInsert$ performs an NAW section in both its code branches. The leaf split code simply adds more writes to unused space and persist operations to other lines not containing the NAW address. There is still a single NAW. Therefore, according to the Definition 2.1, the resulting code also forms an NAW section. Since both cases form an NAW section, from Theorem 2.1, we know that the leaf nodes will be in a consistent state in NVM memory. □

3.4 LB+-Trees with Multi-256B Nodes

In this section, we extend the design of LB$^+$-Trees with 256B sized nodes to support $m \times 256$B sized nodes, where $m \geq 2$. Larger node sizes can be beneficial because (i) the non-leaf part of the tree takes smaller DRAM space, and (ii) the best performing node size depends on CPU and memory configurations, such as memory bandwidth and the memory controller queue size, and can be larger than 256B. (In fact, our experiments find that 512B nodes can have better performance than 256B nodes in Section 4.2.)

The first idea that comes to our mind is to employ the structure of the 256B node for a multi-256B node. That is, the node consists of a header (with a k-bit bitmap, the $lock$ bit and the alt bit, and a $k \times 1$B fingerprints), $k \times 16$B index entry slots, and two sibling pointers at the end, where $k = \lfloor \frac{256m - 16.25}{17.125} \rfloor$. Unfortunately, the header size increases to 32B, 49B, and 66B when the node size is 512B, 768B, and 1024B, respectively. Thus the number of index entry slots in the 64-byte Line 0 decreases to two slots, zero slot,

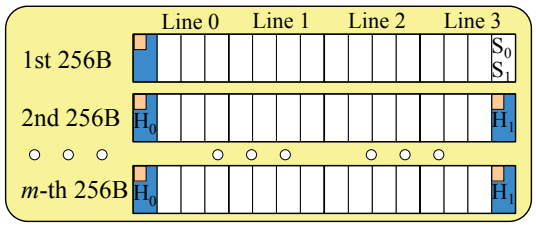


Figure 7: The $m \times 256$B node with distributed headers.

and zero slot, respectively. This naïve multi-256B node design is undesirable because our proposed technique of moving entries to reduce NVM line writes become less effective.

To address the problem of the naïve design, we propose a multi-256B node design with *distributed headers*, as illustrated in Figure 7. In this design, we divide the node into 256B sized blocks. The centralized header in the naïve design is distributed across the blocks. Every block has a 16B header in Line 0 and 14 index entry slots. The header manages the slots in the same block by maintaining a 14-bit bitmap and 14 1B-fingerprints. Like the 256B sized node, the first block has the $lock$ bit and the alt bit in the header, and two sibling pointers at the end.

One problem arises: multiple distributed headers may be updated at node splits. The multiple writes to the headers cannot be handled by a single NAW, and would require logging. We cope with this problem by introducing an alternative header at the end of block $2 \sim m$, as shown in Figure 7. We reuse the alt bit to indicate whether the front or the back headers are active. Then, we can write to the previously unused header in block $2 \sim m$ and perform a single NAW to the first header to switch both the sibling pointers and the headers in block $2 \sim m$, thereby achieving logless splits.

We extend the algorithms of 256B-node LB$^+$-Trees to support multi-256B nodes. The main difference resides in how to process multiple blocks in leaf nodes. The search algorithm performs the fingerprint comparison, bitmap check, and key comparison in every block. The deletion algorithm updates the header in the block where the deleted entry is found. The insertion algorithm finds the first non-full block, and performs insertion into this block. When all the blocks are full, it performs a leaf node split operation. It allocates a new node, copies half the entries to the new node, sets the alternative sibling pointer and the alternative headers, then uses a single NAW to update the header in Line 0 of the first block, switching the sibling pointer and the headers in use. Based on the discussion, it is straightforward to extend the previous proofs to show the following guarantee:

THEOREM 3.4. *The insertion and deletion algorithms of LB$^+$-Trees with multi-256B nodes are guaranteed to keep the LB$^+$-Tree leaf nodes in a consistent state in NVM memory upon failure.*

3.5 Cost Analysis

We compare the entry moving optimization for insertions proposed in Section 3.2 with the traditional design, which represents existing NVM optimized B$^+$-Trees.

We consider the general case of inserting into a newly split 256B node. Note that it is reasonable to assume that after a large number of insertions, every leaf node in a stable tree is the result of a leaf node split. We denote the newly allocated node $leaf_n$ and the previously full node $leaf_f$. As shown in Figure 8(a), there are 15 entries in the two nodes. Depending on which node contains the entry causing the split, one node has 8 entries, the other has 7 entries. We use the superscript to denote the number of entries after split. We consider $leaf_f^7$, $leaf_f^8$, $leaf_n^7$, and $leaf_n^8$. Suppose insertions are random. The four situations are equally likely.

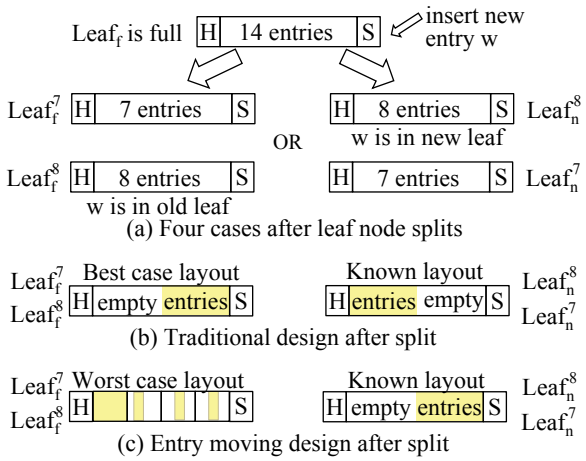


Figure 8: Consider four cases after leaf node splits (The new leaf layout is known. The layout of the old leaf node can be arbitrary. We compare the worst layout for our proposed scheme with the best layout for the traditional scheme.)

THEOREM 3.5. *In a stable tree with 256B nodes, the entry moving design reduces the number of NVM line writes per insertion of the traditional design by at least* 1.35*x*.

PROOF. In the traditional design, As shown in Figure 8(b), $leaf_n$ is always packed from the beginning such that slots in Line 0 are occupied. Therefore, any insertion to $leaf_n^7$ or $leaf_n^8$ incurs two NVM line writes. For $leaf_f$, we consider the best distribution of empty slots. That is, the empty slots are at the beginning of the node. Then the first three insertions will take empty slots in Line 0, incurring one NVM line write. The rest will incur two NVM line writes. Therefore, the average NVM line writes per insertion is $\frac{3\times1+4\times2}{7} = \frac{11}{7}$ for $leaf_f^7$, and $\frac{3\times1+3\times2}{6} = \frac{3}{2}$ for $leaf_f^8$. The overall best-case average NVM line writes per insertion is an average of the four situations: $(2+2+\frac{11}{7}+\frac{3}{2})/4 = 1.77$.

In the entry moving design, As shown in Figure 8(c), $leaf_n$ is always packed from the end, and the empty slots are in the beginning (i.e. in Line 0 and Line 1). Then all insertions except one incur one NVM line write. Therefore, the average NVM line writes per insertion is $\frac{6\times1+1\times2}{7} = \frac{8}{7}$ for $leaf_n^7$, and $\frac{5\times1+1\times2}{6} = \frac{7}{6}$ for $leaf_n^8$. For $leaf_f$, we consider the worst distribution of empty slots. That is, Line 0 are fully occupied and the other three lines all have empty slots. Therefore, three insertions will trigger entry moving to Line 1, 2, and 3, respectively. The average NVM line writes per insertion is $\frac{4\times1+3\times2}{7} = \frac{10}{7}$ for $leaf_f^7$, and $\frac{3\times1+3\times2}{6} = \frac{3}{2}$ for $leaf_f^8$. The overall worst-case average NVM line writes per insertion is an average of the four situations: $(\frac{8}{7}+\frac{7}{6}+\frac{10}{7}+\frac{3}{2})/4 = 1.31$.

Therefore, comparing the worst-case average of the entry moving design and the best-case average of the traditional design, we see that the entry moving design is at least $1.77/1.31 = 1.35$x better than the traditional design. □

We can extend this analysis to $m \times 256$B nodes. In the LB$^+$-Tree node, every 256B block contains a header and 14 slots. There are $14m$ slots. $3m$ slots are in Line 0. After a leaf node split, one of $leaf_f$ and $leaf_n$ contains $7m$ entries, the other contains $7m+1$ entries. It is easy to adapt the above analysis to compute the average NVM line writes per insertion. In the traditional design, the centralized header becomes larger as the node size increases, leaving fewer slots in Line 0. We can show that the overall best-case average NVM line writes per insertion is $(2+2+2-\frac{3}{7m}+2-\frac{3}{7m-1})/4 \geq 1.77$. In the entry moving design, the overall worst-case average NVM line writes per insertion is $(\frac{8}{7}+\frac{8m-1}{7m-1}+1+\frac{3}{7m}+1+\frac{3}{7m-1})/4 \leq 1.31$. Therefore, we have the following.

Table 1: Machine configuration.

CPU	Intel Cascade Lake-SP, Dual-socket, 28 cores at 2.5 GHz (Turbo Boost at 3.8GHZ)
L1 Cache	32 KB iCache & 32 KB dCache (per-core)
L2 Cache	1 MB (per-core)
L3 Cache	39 MB (shared)
Total DRAM	394 GB
NVMM Spec	Intel Optane DC 2666 MHz QS (000006A)
Total NVMM	512 GB [2 (socket) x 2 (channel) x 128 GB]
Linux Kernel	4.9.135
CPUFreq Governor	Performance
Hyper-Threading	Disabled
NVDIMM	Firmware 01.01.00.5253, App direct mode
Power Budget	Avg. 15W, Peak 20W

THEOREM 3.6. *In a stable tree with $m \times 256B$ nodes, the entry moving design reduces the number of NVM line writes per insertion of the traditional design by at least $1.35x$.*

Space Overhead. In a multi-256B node, the space overhead of all the metadata (including all headers and sibling pointers) is 32B per 256B sized block, i.e. 12.5%. Each 256B sized block contains a 16B alternative header except the first block. Thus, the space overhead of the alternative headers is less than 16B/256B = 6.25%.

4. EXPERIMENTAL EVALUATION

In this section, we compare LB$^+$-Tree with state-of-the-art NVM optimized index structures on a real machine equipped with 3DX-Point memory. We describe the experimental setup in Section 4.1. Then, we perform micro-benchmarks to measure the performance of the index structures in Section 4.2. After that, we investigate the benefits of our solution in two real-world systems in Section 4.3–4.4: (i) X-Engine [12], an optimized storage engine for large scale OLTP processing, and (ii) Memcached [16], a popular open source in-memory key-value store.

4.1 Experimental Setup

Machine Configuration. The machine is equipped with two Intel Cascade Lake-SP CPUs. Each CPU has 28 cores and a shared 39MB L3 cache. There are a 32KB L1I cache, a 32KB L1D cache, and a 1MB L2 cache per core. The system is equipped with 394GB DRAM and 512GB 3DXPoint memory. The 512GB 3DXPoint memory consists of four 128GB NVDIMMs. Each CPU has two NVDIMMs. Every pair of NVDIMMs attached to a CPU is configured to be a single 256GB module. From the perspective of a CPU, one of the 256GB module is local, the other is remote. There are NUMA effects for 3DXPoint memory accesses. In our experiments, we avoid the NUMA effects by setting the CPU affinity of our programs to run on CPU 0, and making sure that the programs only access DRAM and 3DXPoint memory local to the CPU.

The machine is running Linux with 4.9.135 kernel. The two 256GB 3DXPoint modules are shown as two special devices in Linux. File systems are installed on the 3DXPoint modules using the fsdax mode. Then we use PMDK to map files into the virtual address space of our programs and access 3DXPoint using load and store instructions. We use `clwb` and `sfence` to persist data to 3DXPoint memory.

B$^+$-Tree Structures to Compare. We compare three B$^+$-Tree structures in the micro-benchmarks and in Memcached experiments: (i) WB$^+$-Tree [6] ("wb-tree"), (ii) FP-Tree [18] ("fp-tree"), and (iii) our proposed solution LB$^+$-Tree ("lb-tree").

WB$^+$-Tree and FP-Tree are state-of-the-art NVM-optimized B$^+$-Tree structures. Our proposed LB$^+$-Tree builds on the ideas of WB$^+$-Tree and FP-Tree, and focuses on improving the insertion performance in light of the distinctive features of real NVM memory hardware (i.e. 3DXPoint). First, we propose entry moving in leaf nodes to reduce the number of NVM line writes. In contrast, both WB$^+$-Tree and FP-Tree insert a new entry into an empty slot in a leaf node without moving existing entries. Second, we achieve logless leaf node splits by using alternative sibling pointers and alternative headers, while WB$^+$-Tree and FP-Tree perform costly write-ahead logging for leaf node splits.

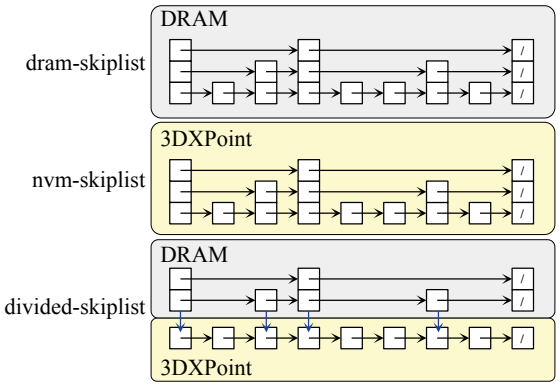


Figure 9: Three skip list designs.

Apart from the insertion algorithms, the three trees have different leaf node designs. First, both WB$^+$-Tree and FP-Tree's leaf nodes have centralized headers. In comparison, LB$^+$-Tree's multi-256B leaf nodes contain distributed headers. This design maximizes the number of entries that co-locate in the same lines as the headers. Hence, after entry moving, more insertions will enjoy a single NVM line write. Second, FP-Tree and LB$^+$-Tree's leaf nodes employ a fingerprint array to support SIMD comparison, while WB$^+$-Tree's leaf node contains an indirection array to support binary search. We quantify the impact of the two designs on leaf node search performance in Section 4.2.

We make the implementation details as close as possible for the three trees to make the comparison fair. First, the non-leaf nodes of all the trees are in DRAM and the leaf nodes are in 3DXPoint memory. Second, the trees all have the same node sizes. Third, they use the same memory allocation routines, which pre-allocate large memory buffers then serve tree node allocation requests from the pre-allocated buffers, thereby minimizing the overhead for calling allocation functions in libc and PMDK. Finally, we employ the same concurrency control mechanism for all three trees, which combines Intel TSX transactions and a lock bit per leaf node.

Incorporating LB$^+$-Tree into Real-World Systems. We incorporate LB$^+$-Tree into two real-world systems: X-Engine [12] and Memcached [16]. X-Engine uses the skip list as its main index structure. The core index structure in Memcached is a hash index. In both cases, the index structures expose a key-value interface to the systems. The keys and values are variable sized strings.

To support the two systems, we modify LB$^+$-Tree to store (8B key pointer, 8B value pointer) entries. We store the keys and values outside the index structure in NVM memory. Key comparison is performed with memory comparison instead of integer comparison. Then, we modify the interface functions to call the LB$^+$-Tree search, insertion, or deletion methods. We modified less than 150 lines of code in either of the two systems.

NVM Optimized Skip List. X-Engine uses skip lists as its in-memory index structure. Therefore, we compare LB$^+$-Trees and skip lists in the X-Engine experiments.

We consider three skip list designs as shown in Figure 9: (i) dram-skiplist, which is the original skiplist implementation in X-Engine, (ii) nvm-skiplist, which is the original skiplist placed in 3DXPoint memory, and (iii) divided-skiplist. For divided-skiplist, we generalize the selective persistence technique of B$^+$-Trees to

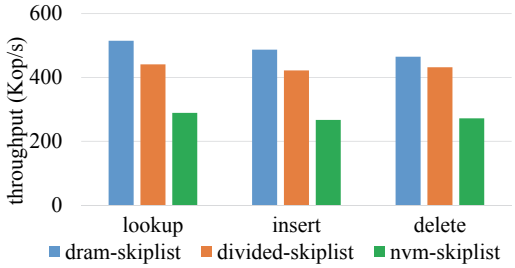


Figure 10: Performance comparison of three skiplist designs.

optimize skip lists. The key idea is that we can put the part of a data structure in DRAM for better performance if it can be rebuilt during recovery and is small enough. We store the bottom layer as an ordered linked list in 3DXPoint and the upper layers as a skip list in DRAM. Each element that is higher than one level in the original skip list is divided into a bottom element and an upper element with one fewer level. Then the upper element contains a pointer to point to the bottom element in the bottom layer in 3DXPoint memory. In this way, search in the upper layer enjoys fast DRAM reads. The upper layer can be rebuilt from the persistent bottom layer upon crash recovery.

Figure 10 compares the performance of the three skip list designs using micro-benchmarks. We bulkload the skip lists with 2 billion (8B key, 8B value) entries, then perform random lookups, insertions, and deletions using a single thread. The figure reports the throughput of the index operations without persisting the data. From the figure, we see that while it is slightly slower than dram-skiplist, divided-skiplist is about 1.5x (computed as the ratio of the throughput) faster than nvm-skiplist. This is mainly because it is faster to visit the upper levels in DRAM.

We choose divided-skiplist as the persistent skip list design in 3DXPoint memory. To achieve data persistence, we use NAWs to write pointer changes of the bottom layer back to 3DXPoint memory. During crash recovery, the upper skip list in DRAM can be rebuilt from the bottom elements.

4.2 Micro-Benchmark Experiments

In this section, we run micro-benchmark experiments to compare the three B$^+$-Tree structures. In the experiments, we generate random keys for bulkloading the trees and for the index operations. We make sure that the total size of the bulkloaded index entries is 32GB unless otherwise noted. In this way, the tree size is at least 1/8 of the 256GB 3DXPoint memory module, satisfying the requirement of Observation 3 in Section 2.1. Moreover, we ensure that the deletion keys exist and the insertion keys do not exist in the bulkloading keys. Therefore, the insertion and the deletion operations will modify the trees.

Insertion Performance Varying Number of Operations. In this set of experiments, we bulkload a tree with 2 billion (8B key, 8B value) entries 70% or 100% full, then perform random insertions and dense insertions. Dense insertions model skewed index accesses. Index entries with similar keys are inserted (e.g., new tweets are indexed on the tweet send time), forming a hot spot in the index. For this experiment, the operation keys are monotonically increasing and are larger than the maximum key in the bulkloaded tree so that the operations all visit the right-most leaf node.

Figure 11 reports execution time in ms. The lower the better. From the figure, we see the following points:

- When leaf nodes are 70% full, insertions often find empty slots in leaf nodes. LB$^+$-Tree achieves 1.20–1.24x and 1.12–1.21x speedups over WB$^+$-Tree and FP-Tree, respectively. The improvement mainly comes from the entry moving optimization.
- When leaf nodes are 100% full, insertions often incur leaf node splits. LB$^+$-Tree performs logless leaf node split, while WB$^+$-Tree and FP-Tree relies on logging to ensure persistence. As a result, LB$^+$-Tree achieves 2.55–2.92x and 2.31–2.45x speedups over WB$^+$-Tree and FP-Tree, respectively.
- For dense insertions, the right-most leaf node sees a large number of insertions. Since multiple keys are inserted into the same node, more later insertions can benefit from the entry moving operations of previous insertions. Moreover, when the node is to split, LB$^+$-Tree exercises logless node splits. In this scenario, LB$^+$-Tree achieves 2.31–2.69x and 2.22–2.33x speedups over WB$^+$-Tree and FP-Tree, respectively.

Overall, LB$^+$-Tree achieves 1.20–2.92x and 1.12–2.45x speedups over WB$^+$-Tree and FP-Tree for insertions, respectively.

Insertion Performance Varying Number of Threads. Figure 12 reports the insertion throughput while increasing the number of threads from 1 to 16. We bulkload a tree with 2 billion (8B key, 8B value) entries 70% or 100% full. Then every thread performs 100K insertions. For dense insertions, every thread focuses on inserting to a different hot spot in the tree, modeling concurrent skewed index accesses. The Y-axis is throughput in million operations per second. The higher the better.

We see that all three trees show almost linear performance scaling as the number of threads increases from 1 to 16. This shows the effectiveness of the concurrency control mechanism combining HTM and lock bits.

Overall, the performance comparison shows similar trends as in Figure 11. When the leaf nodes are 70% full, LB$^+$-Tree achieves 1.17–1.30x and 1.22–1.38x speedups over WB$^+$-Tree and FP-Tree, respectively. When the leaf nodes are 100% full, LB$^+$-Tree achieves 2.35–2.69x and 2.10–2.53x speedups over WB$^+$-Tree and FP-Tree, respectively. For dense insertions, LB$^+$-Tree achieves 1.90–2.38x and 1.89–2.55x speedups over WB$^+$-Tree and FP-Tree, respectively. The entry moving and logless node split optimizations work effectively for multiple threads.

Insertion Performance Varying Node Size. Figure 13 reports insertion performance while varying tree node size from 256B to 1024B. We run the same experiments as in Figure 11. From the figure, we see that LB$^+$-Tree achieves the best throughput in all the cases of different node sizes and insertion workloads.

Comparing the different node sizes, we see that 512B is slightly better than the other sizes when nodes are 70% full, as shown in Figure 13(a). However, when nodes are 100% full or in the case of dense insertions, 256B is significantly better than the other node sizes, as shown in Figure 13(b) and (c). The larger the leaf node, the lower the insertion throughput. This is because the larger the leaf node, the more index entries are copied during leaf node split and the more NVM writes are performed.

Overall, 256B is a good choice of leaf node size for LB$^+$-Tree.

Search and Deletion Performance. As shown in Figure 14, we see that LB$^+$-Tree, FP-Tree, and WB$^+$-Tree have similar index search and deletion performance.

A search operation reads a node at every level from the tree root to the leaf level. It incurs DRAM reads for non-leaf nodes and NVM reads for a leaf node. While the leaf search operations of the three trees differ, the dominant cost is the memory read cost. As a result, the three trees have similar search performance.

We quantify the search time within a leaf node by running a large number of search operations in a fixed leaf node. WB$^+$-Tree employs a sorted indirection array to perform a binary search, while FP-Tree and LB$^+$-Tree perform a SIMD comparison using the fingerprint array to quickly filter out unmatched entries. On average,

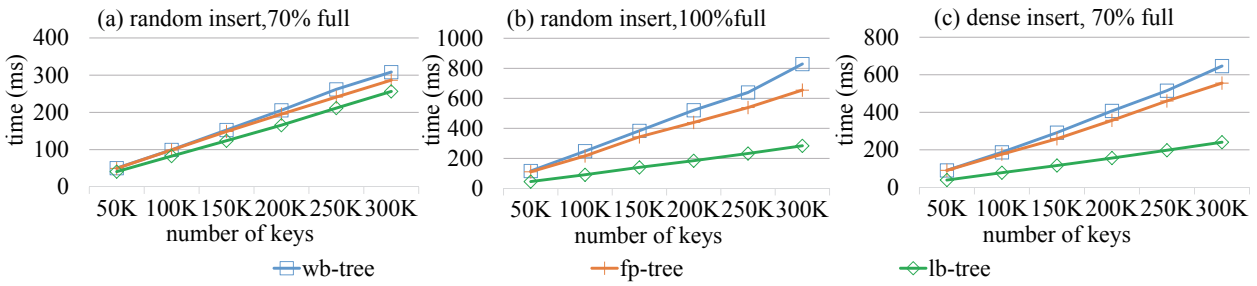


Figure 11: Insertion performance varying the number of operations. (We bulkload a tree with 2 billion (8B key, 8B value) entries 70% or 100% full, then perform 50K–300K random insertions, or dense insertions to the right-most leaf node in a single thread.)

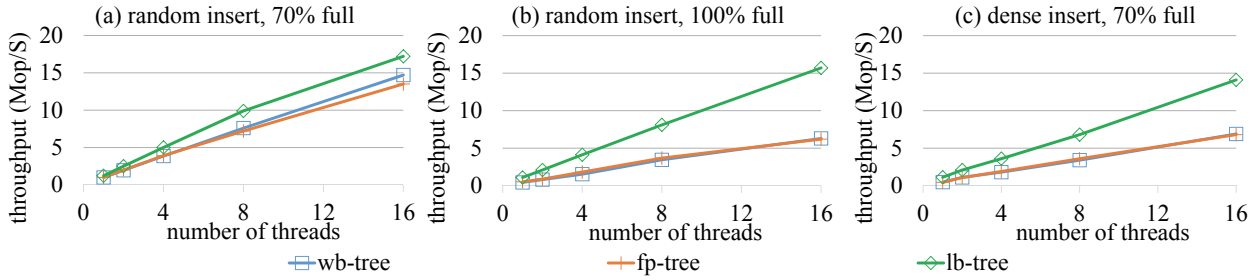


Figure 12: Insertion performance varying the number of threads from 1 to 16. (We bulkload a tree with 2 billion (8B key, 8B value) entries 70% or 100% full. Then every thread performs 100K random insertions, or a different group of 100K dense insertions.)

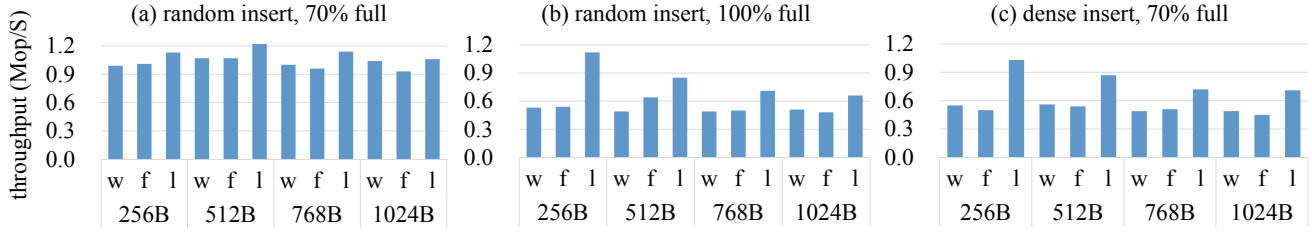


Figure 13: Insertion performance varying node size. (We bulkload a tree with 2 billion (8B key, 8B value) entries 70% full, then perform 100K random back-to-back insertions in a single thread. w: wb-tree, f: fp-tree, l: lb-tree.)

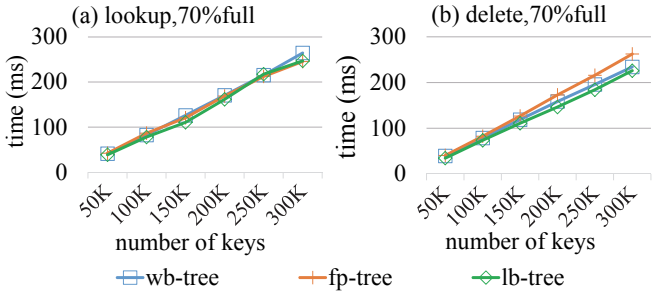


Figure 14: Search and deletion performance varying the number of operations. (We bulkload a tree with 2 billion (8B key, 8B value) entries 70%, then perform 50K–300K random search or deletion operations in a single thread.)

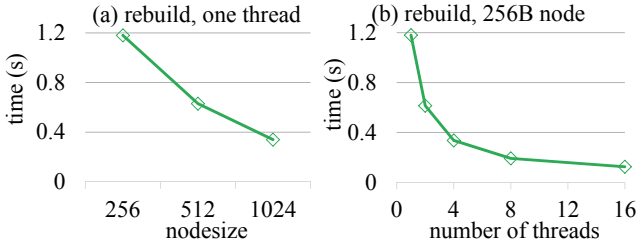


Figure 15: Non-leaf rebuild time during recovery. (During recovery, leaf nodes contain 2 billion (8B key, 8B value) entries 70% full. We vary the node size and the number of threads for rebuilding the non-leaf nodes.)

WB$^+$-Tree takes 27ns to search the leaf node. FP-Tree and LB$^+$-Tree reduce this time to 21ns. However, the tree is 10 levels high. The root-to-leaf search time is on average 823ns, which overshadows the improvement in the search time inside the leaf node.

A deletion first performs a search to locate the target entry in a leaf node. It deletes the entry by using a single NAW to clear the corresponding bit in the leaf bitmap. Since they perform this same procedure, the three trees have similar deletion performance.

Non-leaf Rebuild Time During Recovery. Figure 15 shows the time to rebuild the non-leaf nodes of a tree containing 2 billion entries. As shown in Figure 15(a), the rebuild time decreases as the leaf node size increases. This is expected because the non-leaf rebuild cost is proportional to the number of leaf nodes. As the node size increases, the total number of leaf nodes for storing 2 billion entries decreases, leading to shorter rebuild time.

As shown in Figure 15(b), we can use multiple threads to build the non-leaf nodes in parallel. During normal execution, we record the NVM locations of a few tens of leaf nodes that are roughly equally spaced across the leaf level. (These locations are checked and updated in the rare event that a leaf node becomes completely empty and gets deleted.) They divide the leaf nodes into multiple disjoint segments. During recovery, multiple threads each build a subtree on a leaf segment, then the main thread puts the subtrees into a single tree by merging the subtree roots. From the figure, we see that as the number of threads increases from 1 to 16, the rebuild time for 256B nodes reduces from 1.2s to 0.1s.

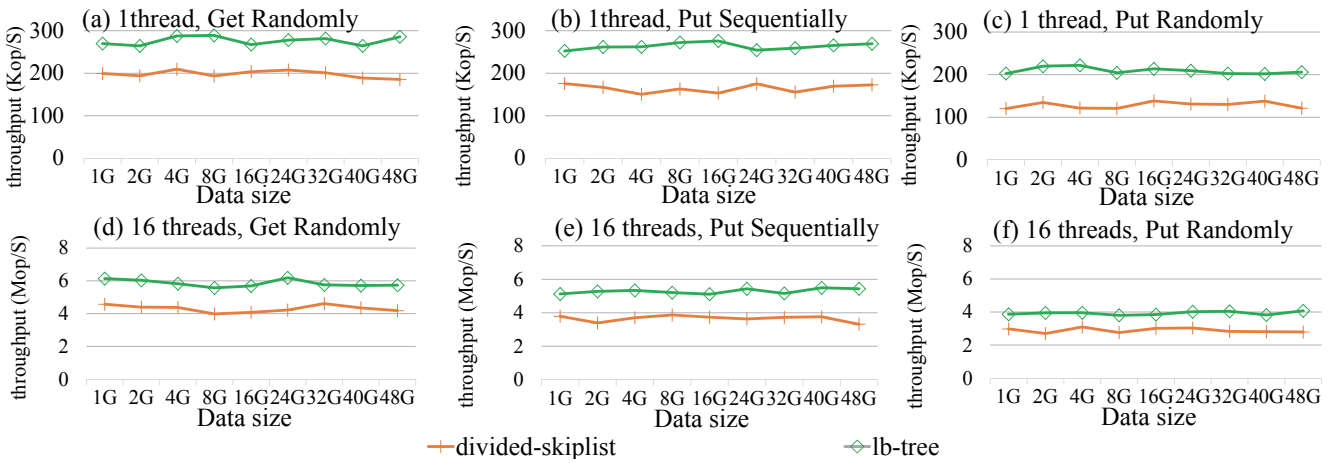


Figure 16: X-Engine performance with LB$^+$-Tree vs. skip list. (Before the experiments, we bulkload an index to have 1GB, 2GB, ..., 48GB data. Then we perform 100K Get operations randomly, 100K Put operations sequentially, or 100K Put operations randomly in every thread. We use *db_bench* to produce (8B key, 20B value) entries. The number of threads is either 1 or 16.)

From the figure, we conclude that the non-leaf rebuild time can be kept reasonably low (e.g., with 16 threads) during recovery.

4.3 X-Engine Experiments

X-Engine is a storage engine for supporting OLTP transactions on Alibaba's backbone e-commerce platform [12]. It implements an LSM-tree design with a tiered storage. We run X-Engine on the test machine with DRAM and 3DXPoint memory. Our experiments focus on the in-memory part of X-Engine without performing any storage I/O operations. The general goal is to explore the potential design point where the 3DXPoint memory capacity is sufficiently large to fit the entire database into NVM main memory.

As described in Section 4.1, the main index structure in X-Engine is the skip list. We replace it with either our LB$^+$-Tree or divided-skiplist, a persistent skip list design for 3DXPoint memory. We use X-Engine's test tool, *db_bench*, to evaluate the engine throughput through its key-value interfaces.

Before the experiments, we bulkload an index to have 1GB–48GB data. We vary the index data size to cover a wide range of application scenarios. Then we perform 100K Get operations randomly, 100K Put operations sequentially, or 100K Put operations randomly in every thread. We use *db_bench* to produce (8B key, 20B value) entries. The number of threads is either 1 or 16.

As shown in Figure 16, in the single thread experiments, compared to divided-skiplist, LB$^+$-Tree achieves 1.31–1.54x speedups for Get operations, 1.44–1.80x speedups for sequential Put operations, and 1.46–1.83x speedups for random Put operations. In 16-thread experiments, LB$^+$-Tree achieves 1.25–1.40x speedups for Get operations, 1.35–1.64x speedups for sequential Put operations, and 1.28–1.46x speedups for random Put operations.

We conclude that LB$^+$-Tree has better performance than skip list on 3DXPoint memory.

4.4 Memcached Experiments

Memcached is a popular open-source in-memory key-value store. The key data structure of Memcached is a hash index in memory, which serves the key-value requests, such as Get and Set.

In this set of experiments, we replace the hash index in Memcached 1.4.17 to use the three B$^+$-Trees, i.e. WB$^+$-Tree, FP-Tree, and LB$^+$-Tree. In this way, with proper communication interface (not yet implemented), the modified Memcached would be capable of serving range scan queries efficiently, and protect the cached key-value entries against power failures.

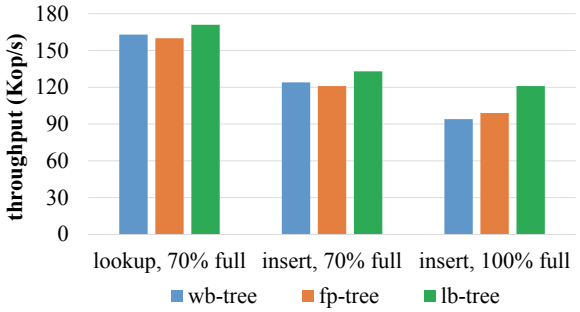


Figure 17: Memcached performance with three B$^+$-Tree structures as its core index. (We bulkload a tree with 50 million entries 70% full or 100% full, and use mc-benchmark to get (lookup) or set (insert) 500K random keys. The keys are 20-byte random strings.)

We bulkload a tree with 50M entries 70% full or 100% full, and use `mc-benchmark` to get or set 500K random keys. The keys are 20-byte random strings. Figure 17 reports the throughput of the three operations for the trees. We see that LB$^+$-Tree achieves the best performance in all cases. LB$^+$-Tree achieves 1.05-1.29x higher throughput than WB$^+$-Tree, and 1.06–1.22x higher throughput than FP-Tree. The memcached experiments confirm our findings in the micro-benchmark experiments.

5. CONCLUSION

In this paper, we study persistent B$^+$-Tree structure for 3DXPoint memory. Based on the observations of 3DXPoint features, we propose LB$^+$-Tree. We introduce three techniques to improve LB$^+$-Tree's insertion performance: (i) Entry moving, (ii) Logless node split, and (iii) Distributed headers. Micro-benchmark results and experiments in two real-world systems (i.e. X-Engine and Memcached) show that LB$^+$-Tree achieves significantly better performance than state-of-the-art NVM optimized B$^+$-Trees for insertions while obtaining similar search and deletion performance.

6. ACKNOWLEDGMENTS

This work is partially supported by National Key R&D Program of China (2018YFB1003303), NSFC (61572468), Alibaba collaboration project (XT622018000648), and K.C.Wong Education Foundation. Shimin Chen is the corresponding author.

7. REFERENCES

[1] Intel optane dc persistent memory architecture overview. https://techfieldday.com/video/intel-optane-dc-persistent-memory-architecture-overview/.

[2] D. Apalkov, A. Khvalkovskiy, S. Watts, V. Nikitin, X. Tang, D. Lottis, K. Moon, X. Luo, E. Chen, A. Ong, A. Driskill-Smith, and M. Krounbi. Spin-transfer torque magnetic random access memory (STT-MRAM). *JETC*, 9(2):13:1–13:35, 2013.

[3] J. Arulraj, J. J. Levandoski, U. F. Minhas, and P. Larson. Bztree: A high-performance latch-free range index for non-volatile memory. *PVLDB*, 11(5):553–565, 2018.

[4] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 235–246, 2001.

[5] S. Chen, P. B. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 21–31, 2011.

[6] S. Chen and Q. Jin. Persistent b+-trees in non-volatile main memory. *PVLDB*, 8(7):786–797, 2015.

[7] S. Cho and H. Lee. Flip-n-write: a simple deterministic technique to improve PRAM write performance, energy and endurance. In *42st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42 2009), December 12-16, 2009, New York, New York, USA*, pages 347–357, 2009.

[8] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. C. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 133–146, 2009.

[9] D. B. Dgien, P. M. Palangappa, N. A. Hunter, J. Li, and K. Mohanram. Compression architecture for bit-write reduction in non-volatile memory technologies. In *IEEE/ACM International Symposium on Nanoscale Architectures, NANOARCH 2014, Paris, France, July 8-10, 2014*, pages 51–56, 2014.

[10] D. H. Graham. Intel optane technology products - what's available and what's coming soon. https://software.intel.com/en-us/articles/3d-xpoint-technology-products.

[11] Y. Guo, Y. Hua, and P. Zuo. DFPC: A dynamic frequent pattern compression scheme in nvm-based main memory. In *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*, pages 1622–1627, 2018.

[12] G. Huang, X. Cheng, J. Wang, Y. Wang, D. He, T. Zhang, F. Li, S. Wang, W. Cao, and Q. Li. X-engine: An optimized storage engine for large-scale e-commerce transaction processing. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.*, pages 651–665, 2019.

[13] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

[14] T. Johnson and D. E. Shasha. Utilization of b-trees with inserts, deletes and modifies. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 29-31, 1989, Philadelphia, Pennsylvania, USA*, pages 235–246, 1989.

[15] J. Liu and S. Chen. Initial experience with 3d xpoint main memory. In *35th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2019, Macao, China, April 8-12, 2019*, pages 300–305, 2019.

[16] Memcached. http://memcached.org/.

[17] C. Mohan, D. Haderle, B. G. Lindsay, H. Pirahesh, and P. M. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[18] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. Fptree: A hybrid SCM-DRAM persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 371–386, 2016.

[19] 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, May 16-18, 2000, Dallas, Texas, USA.*, pages 475–486, 2000.

[20] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y. Chen, R. M. Shelby, M. Salinga, D. Krebs, S. Chen, H. Lung, and C. H. Lam. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4-5):465–480, 2008.

[21] A. F. Webster and S. E. Tavares. On the design of s-boxes. In *Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings*, pages 523–534, 1985.

[22] B. Yang, J. Lee, J. Kim, J. Cho, S. Lee, and B. Yu. A low power phase-change random access memory using a data-comparison write scheme. In *International Symposium on Circuits and Systems (ISCAS 2007), 27-20 May 2007, New Orleans, Louisiana, USA*, pages 3014–3017, 2007.

[23] J. J. Yang and R. S. Williams. Memristive devices in computing system: Promises and challenges. *JETC*, 9(2):11:1–11:20, 2013.

[24] V. Young, P. J. Nair, and M. K. Qureshi. DEUCE: write-efficient encryption for non-volatile memories. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 33–44, 2015.

[25] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, pages 14–23, 2009.