Isle-Tree: A B+-Tree with Intra-Cache Line Sorted Leaves for Non-volatile Memory

Chundong Wang
SIST, ShanghaiTech University, China
cd_wang@outlook.com

Sudipta Chattopadhyay
Singapore University of Technology and Design, Singapore
sudipta_chattopadhyay@sutd.edu.sg

***Abstract*—Byte-addressable non-volatile memory (NVM) is to reshape computer systems. Researchers have proposed crash-consistent in-NVM B+-trees with unsorted or sorted nodes to store key-value (KV) pairs. However, they still yield suboptimal performance: inserting a KV pair into a sorted node shifts numerous KV pairs that may cause multiple cache lines to be flushed, while to search a KV pair in an unsorted node is inefficient. In this paper, we propose Isle-Tree. Each cache line of Isle-Tree's leaf node is sorted while the node is unsorted. For most insertions/deletions, Isle-Tree flushes only one cache line of KV pairs. For searches, sorted cache lines help Isle-Tree avoid unnecessary comparisons. Experiments show that Isle-Tree yields high performance for all insertions, deletions and searches.**

***Index Terms*—B+-tree, Non-volatile Memory, Key-value Store**

I. Introduction

Emerging byte-addressable non-volatile memories (NVM), like spin-transfer torque RAM (STT-RAM), 3D XPoint, and phase change memory (PCM), are poised to radically change the conventional hierarchy of volatile memory and persistent storage for computer systems. NVM embraces both DRAM's byte-addressability and disk's persistency. It can be placed on the memory bus for CPU to directly load and store data.

Several B+-trees have been developed with NVM to build key-value (KV) stores [1–8]. All of them attempt to guarantee the crash consistency of KV pairs without incurring much performance overhead. A system may stop functioning at any time due to the occurrence of a crash, e.g., power failure. Given a B+-tree, say, inserting a KV pair when the system crashes, it shall provide a way to properly recover the tree to a consistent state, in which all valid KV pairs are reconcilable and error-free. As CPUs directly operate with NVM, architectural challenges emerges to affect the crash consistency. One of them is that CPU or memory controller may perform multiple writes from CPU cache to memory in an order deviating from the programmed one [9, 10]. Assume that we allocate a B+-tree node and record its pointer. A reversed order of these two writes upon a crash would result in a dangling pointer.

Using cache line flush and memory fence retains a writing order by forcefully flushing data to NVM with explicit memory barriers, but with non-trivial performance overhead [1, 3, 7, 9, 10]. Worse, NVM generally has slower write speed than DRAM [11–13]. In-NVM B+-tree variants thus have sought different tactics to reduce the use of cache line flush and memory fence. On one hand, researchers proposed unsorted KV pairs in a leaf node to avoid orderly shifting and flushing KV pairs for insertion and deletion. Examples include NV-Tree [3] and FPTree [5]. On the other hand, store dependencies in shifting successive KV pairs of a sorted node impose a natural writing order, and only dirty cache lines caused by shifting need to be shifted. FAST-FAIR [7], and Circ-Tree [8] align with such an approach of sorted leaf nodes.

Though, both approaches have deficiencies. Take a common insertion that inserts a KV pair into a non-full leaf node for illustration. For an unsorted node, the new KV pair can be put in any vacancy of the node. For a sorted node, existing KV pairs are shifted to vacate the proper position for the new one. Multiple cache lines are successively modified and flushed for crash consistency. Concretely, the approach of sorted leaf nodes yields inferior write (insertion/deletion) performance. To search a KV pair, the approach of unsorted nodes, however, is not efficient due to irregular distribution of unsorted keys.

For a crash-consistent in-NVM B+-tree, how to make the best of a cache line, which is the exchange unit between CPU and NVM, is critical. Firstly, no matter if a node is sorted or unsorted, an insertion (resp. deletion) surely causes at least one cache line, i.e., the one containing the newly-inserted (resp. deleted) KV pair, to be flushed for crash consistency and persistency. Secondly, to look up a KV pair with linear search, which is more efficient than binary search over sorted KV pairs due to fewer cache misses in large caches [7, 8], traverses cache line by cache line across a node.

Accordingly, we develop Isle-Tree with novel **i**ntra-cache line **s**orted **le**af nodes. Its main ideas are as follows.

- Isle-Tree maintains a leaf node with multiple cache lines of KV pairs in a contiguous, cache line-aligned NVM space. Each cache line of a leaf node is sorted while the entire node is unsorted (semi-sorted).
- Isle-Tree efficiently handles insertions and deletions with its novel leaf nodes. Take a common insertion for illustration. Isle-Tree inserts a new KV pair into a cache line with vacancies by orderly shifting existing KV pairs inside the cache line. As a result, only this cache line of KV pairs is modified and flushed to NVM.
- Isle-Tree uses a linear search with hops to find a key in its semi-sorted leaf node. It scans from the first key. Given KV pairs sorted in a cache line, once Isle-Tree encounters a

C. Wang's work is supported by the ShanghaiTech Startup Funding.

key that is greater than the target key, it immediately jumps to the next cache line, i.e., *hops*. Hops avoid unnecessary comparisons and improve the utilization of CPU cache.

We have extensively evaluated Isle-Tree. Experiments show that, to insert 10 million KV pairs, Isle-tree spends 55.0%, 124.3% and 49.1% less time than FAST-FAIR, wB+-tree and Circ-tree, respectively, while it is faster by 37.0% and 18.9% than NV-tree and FPtree to search 10 million keys, respectively. An end-to-end test by running YCSB [14] with KV stores built on these B+-trees shows that Isle-Tree provides from 22.4% to 36.2% shorter insertion latency.

The rest of this paper is organized as follows. In Section 2, we show the background of NVM. In Section 3, we brief state-of-the-art in-NVM B+-trees. In Section 4, we describe the design of Isle-Tree. We present evaluation results of Isle-tree in Section 6 and conclude the paper in Section 7.

II. Background

Computer systems are to benefit from the use of byte-addressable NVM. Compared to DRAM, NVM generally has longer write latency but comparable read latency. Crash-consistent in-memory data management systems, especially in-NVM KV stores have been built with NVM that is placed on the memory bus for CPU to directly load and store data [1–8].

It is non-trivial to develop in-NVM data management systems as architectural challenges emerge for data's crash consistency. One is the size of atomic write, i.e., all-or-nothing written. Conventional disk drives atomically write a sector of 512B. Modern 64-bit CPUs generally support an 8B atomic memory write. Systems developed for NVM must take into account such atomic writes to proceed a write transaction.

Another challenge is the reordered writes between CPU and memory [9, 15]. When writing multiple cache lines to memory, CPU or memory controller may schedule a different writing order instead of the programmed one, which is adverse to crash consistency. Assume we make a new KV pair pointing to an actual value. The value must be filled before feeding its pointer to form the KV pair. If these two writes are reversed while a crash happens, the KV pair refers to an untrustworthy value. One way to retain a writing order is using cache line flush and memory fence. Memory fence (e.g., `mfence` in x86) regulates that memory accesses after a memory fence cannot proceed until ones before it complete. Cache line flush instruction (e.g., `clflush` in x86) yet explicitly flushes a cache line to memory. A combination of cache line flush and memory fence hence attains a desired writing order. Nonetheless, the cost of using them is high. Optimized cache line flush (e.g., `clflushopt` or `clwb` in x86) has been provided with reduced but still non-trivial overhead. Added to this, regarding the longer write latency of NVM, reducing memory writes to NVM surely brings in substantial performance gains.

III. State of the art and Motivation

A. Design and Deficiency of In-NVM B+-trees

A B+-tree has multi-level internal nodes (INs) for indexing and one level of leaf nodes (LNs) to record KV pairs. Among state-of-the-art in-NVM B+-trees, the pioneering CDDS-Tree [1] conforms to the convention of sorted INs and LNs. To insert and delete a KV pair, CDDS-Tree shifts KV pairs to keep keys sorted, with every shifted KV pair flushed.

Yang et al. [3] quantified the high cost of flushing every shifted KV pair in a sorted LN for insertion/deletion, and designed NV-Tree with unsorted LNs. For a common insertion, NV-Tree appends and flushes the new KV pair to the tail of an LN. More, NV-Tree maintains all INs in a contiguous DRAM space without consistency. To split any full IN, NV-Tree 1) asks for a larger contiguous space, and 2) freezes and leverages LNs to rebuild new non-full INs. Incoming insertions and deletions are stalled until such a long rebuilding completes.

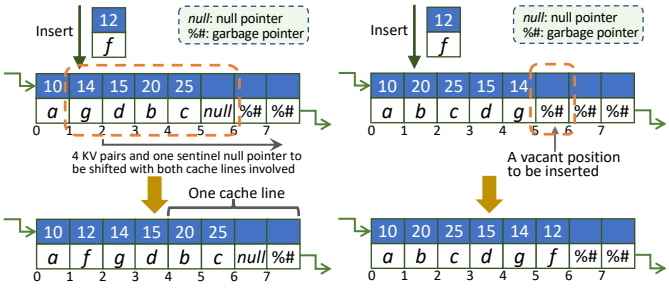
Searching a key in an unsorted node is inefficient. Chen and Jin [4] proposed wB+-tree that uses a slot array, in which the first element records the position of the smallest key, and so forth. wB+-tree proceeds a search by referencing the slot array and fetching corresponding keys for comparison; thus, CPU does two-level indexing. Worse, the slot array must be updated after every insertion/deletion and wB+-tree flushes it frequently for crash consistency. Later, Oukid et al. [5] designed FPTree that employs a fingerprint (1B hash value) for an unsorted key. Searching a key converts to a primary search of fingerprints. However, using fingerprints is ineffectual in practice. Every insertion/search/deletion demands a hash calculation that inflicts extra computation time. The collision of 1B hash values is inevitable and requires a double check over keys, thereby incurring cache misses and comparisons. Moreover, although fingerprints can be recalculated after a crash, FPTree flushes them to NVM for crash consistency.

Later Hwang et al. [7] revealed that store dependencies exist in successively shifting KV pairs to retain a sorted node, i.e., $\{\text{KV}_i \rightarrow \text{KV}_{i+1}, \text{KV}_{i-1} \rightarrow \text{KV}_i, ...\}$, imposing a natural writing order. They designed FAST-FAIR that only flushes dirty cache lines since the order inside a cache line is retained by store dependencies. For an insertion/deletion, FAST-FAIR flushes less frequently than CDDS-Tree, but still flushes multiple cache lines generally.

Recently, Wang et al. [8] discovered that shifting KV pairs in a sorted B+-tree node is unidirectional, e.g., to the right for insertion and left for deletion. They proposed Circ-Tree with a circular LN in which KV pairs are bidirectionally shifted. For an insertion or deletion, Circ-tree shifts KV pairs to the direction that involves fewer KV pairs. Compared to FAST-FAIR, Circ-tree significantly reduces the number of shifted KV pairs and dirty cache lines flushed to NVM.

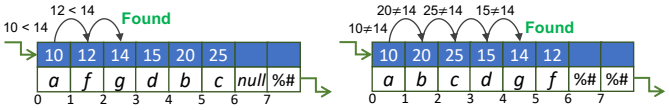
B. Motivational Study

With unsorted or sorted LNs, state-of-the-art in-NVM B+-tree variants have respective designs and deficiencies. Generally, unsorted LN provides incompetent search performance even with accelerating tactics. Despite providing good search performance with sorted KV pairs, FAST-FAIR and Circ-Tree need to shift KV pairs for insertion and deletion, which is likely to cause multiple cache lines to be modified and flushed. In the worst case, FAST-FAIR flushes all cache lines of an LN



(a) Insertion with sorted LN (b) Insertion with unsorted LN

Fig. 1: A Common Insertion with Sorted and Unsorted LNs



(a) Search with sorted LN (b) Search with unsorted LN

Fig. 2: A Search with Sorted and Unsorted LNs

while Circ-Tree flushes half. Frequent cache line flushes lead to low cache efficiency and in turn inferior performance.

Figure 1 and Figure 2 show how an insertion and a search are handled with sorted and unsorted nodes, respectively. We assume four KV pairs take up one cache line. On inserting a new KV pair $\langle 12, f \rangle$ into the sorted one shown in Figure 1a, four existing KV pairs have to be shifted to maintain keys in ascending order with two cache lines successively flushed. Regarding the unsorted node shown in Figure 1b, the new KV pair can be inserted to any vacancy with one cache line flushed. Thus, for common insertions, the performance of unsorted nodes is higher than that of sorted nodes in which multiple cache lines may be flushed due to shifting sorted KV pairs.

Linear search, when running on CPUs with large caches, is faster than binary search with a sorted node as the latter has more cache misses [7, 8]. In Figure 2a and Figure 2b, we search the key 14 with linear search over sorted and unsorted nodes. The ascending order of keys helps CPU train its branch prediction. In Figure 2b, the target key may stay anywhere in an unsorted node. It is unavoidable to compare many keys before finding the target one. For instance, four comparisons in the first cache line cannot be skipped.

No matter if a node is sorted or unsorted, at least one cache line that contains the key to be inserted/deleted must be flushed. Meanwhile, sorted KV pairs facilitate searching. We find that, the cache line, as the exchange unit between CPU and NVM, is a critical factor. CPU loads KV pairs in the granularity of cache line and writes them back with a *cache line* flush. To keep one cache line sorted by shifting few KV pairs is not costly and only flushes that cache line to NVM. Meanwhile, searching a target key in a sorted cache line can avoid unnecessary comparisons once a greater key is encountered. Concretely, we design a B+-tree with **i**ntra-cache line **s**orted **le**af nodes (Isle-Tree). In a nutshell, an LN of Isle-Tree is semi-sorted as each cache line of it is sorted.

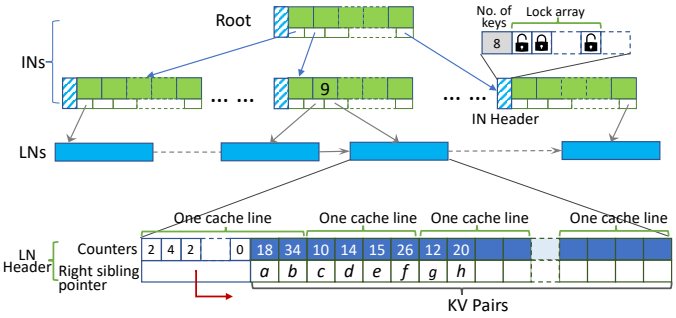


Fig. 3: An Illustration of Isle-Tree and Its Leaf Node

IV. Design of Isle-Tree

A. Overview

Figure 3 captures an in-NVM Isle-Tree with intra-cache line sorted LNs and sorted INs as well as one example LN.

LN: An LN of Isle-Tree occupies a contiguous NVM space that is cache line-aligned and partitioned into multiple cache lines. An LN has a header at its first cache line and an array of KV pairs that form the LN's main body. KV pairs are sorted in each cache line while an LN is unsorted (semi-sorted).

KV Pairs: A value is a pointer (8B memory address) pointing to an actual value. As to the key, we use an 8B integer as a key for illustration. Like previous works [5, 7], we assume that there are no duplicate keys. As a KV pair contains 16B, a typical cache line in 64B holds four KV pairs.

LN Header: As shown by Figure 3, in an header, there are

1) an array of counters, each of which has 1B, to record the number of valid KV pairs in each cache line of the LN,
2) a sibling pointer that connects to the right sibling LN so as to make an LN linked list.

Isle-Tree requires that an LN header at most takes up one cache line. The 8B sibling pointer points to right sibling LN. 1B (0–255) is sufficient for each counter. Both a sibling pointer and a counter can be atomically changed through 8B atomic write. Isle-Tree only includes the array of counters and sibling pointer in the header and enforces crash consistency to them because they are essential in crash recovery (cf. Section IV-F).

Operations: Isle-Tree aims to service insertions, searches, and deletions with high cache efficiency and in turn high performance. In brief, on inserting a KV pair to a non-full LN, Isle-Tree finds a cache line with a vacant position by referencing the array of counters and shifts KV pairs in that cache line to retain the ascending order. By doing so, Isle-Tree involves and flushes at most one cache line of KV pairs.

Inserting a KV pair to a full LN triggers a split. It is easy for Isle-Tree to separate greater half KV pairs from small ones in sorted cache lines and proceed the split. To delete a KV pair, Isle-Tree writes off it by shifting KV pairs inside the cache line where the KV pair is found. To look up or update a KV pair, Isle-Tree applies a linear search with *hops* to avoid unnecessary comparisons in each sorted cache line.

IN: INs of Isle-Tree are standard B+-tree nodes with sorted keys and values that are pointers pointing to children INs or LNs. As shown in Figure 3, the number of values in an IN

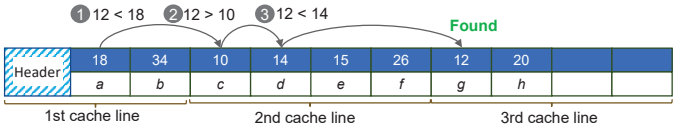


Fig. 4: An Illustration of Isle Searching Key 12

is one greater than the number of keys. An IN of Isle-Tree also has a header that includes metadata like the number of valid keys and an array of locks. Each lock is used to control multi-threading access to a corresponding child IN or LN (cf. Section IV-E). INs can be maintained in NVM [7] or DRAM as they can be reconstructed from LNs [3, 5].

B. Search and Update

A search with a key starts from the root of Isle-Tree until one LN is reached. Isle-Tree applies a linear search with *hops* onto such a semi-sorted LN. Figure 4 exemplifies how Isle-Tree scans the LN from Figure 3 upon searching a key 12. It initiates the scan from the first KV pair. Given KV pairs sorted in each cache line, after one comparison to a greater key (❶), Isle-Tree deems that key 12 cannot be in the current cache line. So Isle-Tree jumps overs the remaining KV pair. It resumes the scan at the beginning of the second cache line. After two more comparisons (❷ and ❸), it jumps again to the third cache line. Next it finds the key 12 and obtains the corresponding value f. To sum up, by leveraging the intra-cache line sorted KV pairs, Isle-Tree avoids unnecessary comparisons with hops, thereby improving the utilization of CPU cache lines.

An update operation is a search operation plus a substitute of value pointer. After storing the updated actual value into NVM, Isle-Tree uses the new pointer to replace the original one through an 8B atomic write followed by cache line flush and memory fence. Both search and update do not change the metadata in an LN header.

C. Insertion

Common Insertions: Algorithm 1 illustrates the insertion procedure of Isle-Tree. Before inserting, Isle-Tree calls its linear search with the newly-arrived KV pair $\langle k, v\rangle$ until it reaches the target LN with a header h. Isle-Tree then checks if the LN is full (Line 2). If so, it splits the LN (Line 3). Otherwise, Isle-Tree proceeds the insertion with this LN.

By referring to the array of counters, Isle-Tree gets a non-full cache line and the leftmost vacant position in the cache line (ξ and p at Line 7, and p is a local index inside cache line ξ, i.e., $0 \leq p < \mathsf{CL\_Limit}^{(\xi)}$). Then, Isle-Tree starts shifting KV pairs that are with greater keys than k inside the cache line (Lines 8 to 14). Isle-Tree leverages a property of B+-tree in shifting KV pairs, i.e., a normal B+-tree node has no duplicate pointers [7, 8]. In particular, to shift one KV pair, Isle-Tree first shifts the key to the right and then the value. By doing so, duplicate pointers correspond to the same key, thereby ruling out ambiguity in case of a crash.

The new k may be the new greatest key (for loop not executed), or the cache line is vacant (p is 0 and i is -1). In this sense, Isle-Tree adjusts the position where $\langle k, v\rangle$ should

Algorithm 1 Insertion of Isle-Tree (`Insert`$(h, \langle k, v\rangle)$)

Input: A KV pair $\langle k, v\rangle$ and an LN's header h //Isle-Tree first searches out a target LN that is with a header h and an array of KV pairs KV
Output: A completion of inserting $\langle k, v\rangle$

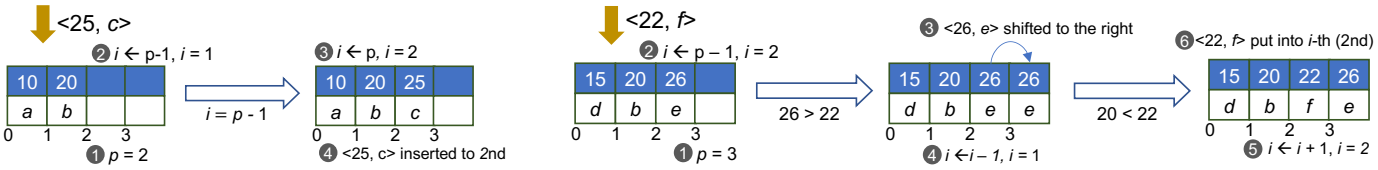
```
1:  //LN_Limit: the max number of KV pairs held by an LN.
2:  if (sum (h.counters)≥ LN_Limit) then
3:      return Split(h, ⟨k, v⟩);
4:  else
5:      //ξ is a sub-array of KV and fitted in a non-full cache line.
6:      //p is the leftmost vacancy of ξ. 0 ≤ p < CL_Limit^(ξ)
          (CL_Limit^(ξ): the max number of KV pairs held by ξ).
7:      (ξ, p) := get_cacheline_vacancy(h.counters, KV);
8:      for (i := p − 1; i ≥ 0; i := i − 1) do
9:          if (ξ[i].key > k)) then
10:             ξ[i + 1].key = ξ[i].key; ξ[i + 1].val = ξ[i].val;
11:         else
12:             break;
13:         end if
14:     end for
15:     i := (i == (p − 1)) ? p : (i + 1);
16:     ξ[i].key = k; ξ[i].val = v;
17:     clflush_with_mfence(&(ξ[i]));
18:     atomic_increase(h.counters, ξ);
19:     clflush_with_mfence(h.counters);
20:     return the completion of inserting ⟨k, v⟩;
21: end if
```

be placed (Line 15). Next, Isle-Tree stores and flushes $\langle k, v\rangle$ to NVM (Lines 16 to 17). Before returning the completion of inserting $\langle k, v\rangle$, Isle-Tree atomically increases and flushes the counter corresponding to cache line ξ (Lines 18 to 20).

Figure 5 shows two insertion examples, of which each cache line is still assumed to hold four KV pairs. In Figure 5a, after scanning, the left most vacant position, i.e., p, is 2 (❶). No KV pair is shifted (❷) as the key to be inserted is the new greatest of the cache line. Isle-Tree just appends the new KV pair into the leftmost vacancy (❸ and ❹). In another example shown by Figure 5b, initially p is 3 (❶) and the for loop shifts one KV pair (❷ and ❸). After Isle-Tree encounters the first smaller key, i.e., 20 (❹), it moves i back (❺) and inserts and flushes the new KV pair there (❻).

Split: Inserting a KV pair to a full LN triggers a split. Figure 6 illustrates how Isle-Trees does so. In short, it asks for two zero-initialized LNs and orderly copies two halves of KV pairs into them, respectively. By atomically changing sibling pointers, the two LNs substitute the orignal full LN. Note that with intra-sorted cache lines, Isle-Tree efficiently separates KV pairs by scanning from the tail of each cache line to find the greater half. Moreover, Isle-Tree flushes both LNs in a batch way, i.e., flushing contiguous cache lines only with beginning and ending memory fences instead of flushing them one by one [1, 3]. Isle-Tree updates the parental IN with the split key (the last copied key of the greater half) and two new LNs. In the end, it inserts the newly-arrived KV pair into either LN, regarding if the new key is greater than the split key.

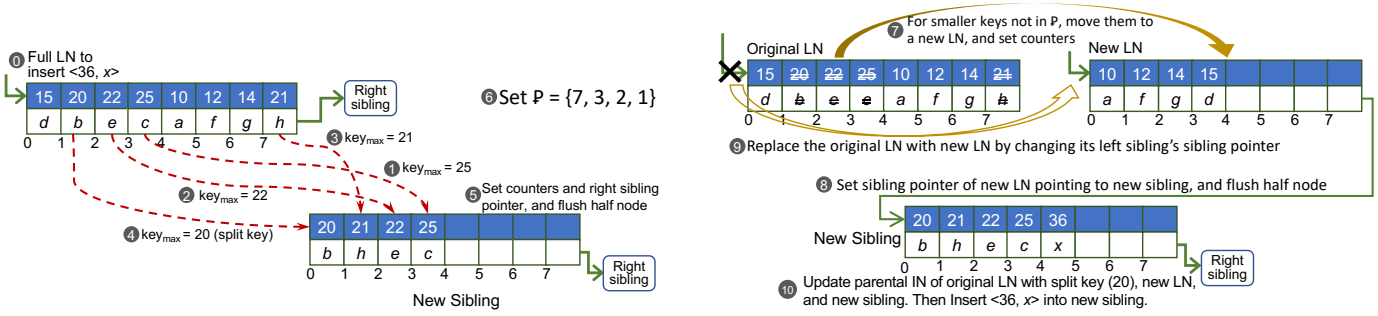
Note that Isle-Tree uses null (zero) pointers as boundaries to identify valid KV pairs. Assume that no null pointer exists as a boundary in a cache line of LN, and a crash happens after



(a) Insertion with new greatest key

(b) Insertion with shifting KV pairs

Fig. 5: An Illustration of Isle-Tree Handling Common Insertion in One Cache Line of an LN



(a) Split (Part 1)

(b) Split (Part 2)

Fig. 6: An Illustration of Isle-Tree Handling Insertion with Split

inserting a new KV pair but before the atomic increase of the counter. In recovery, it is impossible to determine if the counter was increased or not, thereby leaving an unreliable cache line. With a null pointer at each cache line, the recoverability is surely guaranteed (cf. Section IV-F). In a split, Isle-Tree may copy and remove KV pairs from all cache lines. If Isle-Tree insists on keeping the original LN, it must place null pointers in all cache lines and flush them one by one. Such consistency cost is substantial. Comparatively, the way Isle-Tree asks for a zeroed LN and moves remaining KV pairs with a batch flush causes much less performance overhead, and the atomic write to change the sibling pointer rules out inconsistency. Isle-Tree can rely on NVM libraries [16] to provide functions similar to classic `calloc` to allocate zeroed NVM memory space or maintain its own pool of zeroed LNs.

D. Deletion

Due to the space limitation, we brief major steps of Isle-Tree's deletion. If the KV pair to be deleted is with the greatest key of a cache line, Isle-Tree clears the value to be null as a new boundary. Otherwise, Isle-Tree shifts KV pairs to write it off. Isle-Tree starts shifting the target KV pair's immediate right KV pair in the order of value and then key, and so forth, until the KV pair with the greatest key is shifted. Then Isle-Tree zeros the value at the original position of the greatest key to set a new boundary. After flushing the cache line, Isle-Tree atomically decreases and flushes the corresponding counter.

LNs with more than half KV pairs deleted become underutilized. To promote space utilization, Isle-Tree merges underutilized sibling LNs with the same parental IN. To merge LNs, say, A and B, Isle-Tree allocates a zero-initialized new LN, and orderly moves KV pairs from A and B into the new LN. It then sets counters and makes the new LN's sibling pointer link to B's right sibling. Isle-Tree flushes the new LN in a batch fashion, and atomically replaces the sibling pointer of A's left sibling. By doing, A and B are detached and recycled while the new LN substitutes them. The parental IN is updated as two KV pairs for A and B are reduced to one, i.e., the new LN. Isle-Tree does not move KV pairs from A to B or B to A because each move is an actual insertion that requires cache line flushes and memory fences. Flushing a new LN in a batch takes less time.

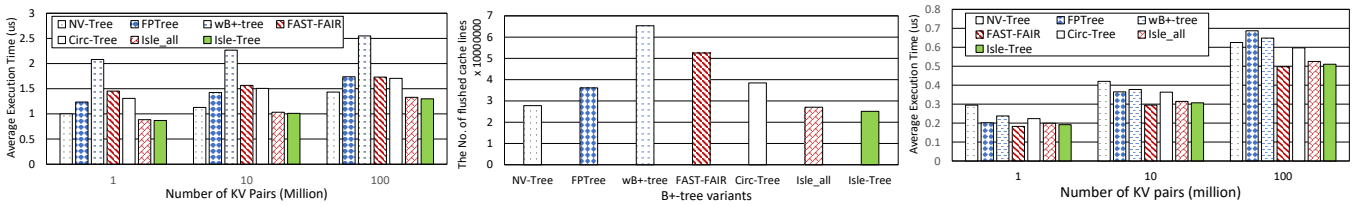
E. Multi-threading Insertion/Deletion/Search

Isle-Tree enables multi-threading with a lock array incorporated in an IN's header (cf. Figure 3). A lock held by a thread makes other threads wait until the lock is released. In addition, a thread may trigger a split (resp. merge) that adds (resp. removes) a KV pair with the current-level IN. The next thread that seizes a released lock must check if the lower-level IN or LN it has intended to access is still the proper one. The thread compares the key it takes to immediate left and right keys in the current-level IN. For example, if a split has occurred, this thread's key may be greater than the immediate right key (newly-added split key), so the thread moves by one to the right lower-level IN or LN to proceed its operation.

F. Recovery

Isle-Tree focuses on recovering LNs because INs can be rebuilt from LNs [3, 5]. By default, Isle-Tree places INs in NVM with crash consistency. It can maintain a flag to mark a normal shutdown [5, 8]. Such a flag is set at startup and reset at normal exit. A crash prevents the flag from being reset. When Isle-tree enters recovery, it first traverses the LN linked list. In an LN, a cache line is consistent if its counter and the number of non-null values are equal, and it has no duplicate value. Scenarios for an inconsistent cache line are as follows.

- The counter is one less than the number of non-null values and there is no duplicate value. So a crash happened after



(a) Average insertion time (b) `clflushopt` executed (10 million) (c) Average search time

Fig. 7: A Comparison among Seven B+-tree Variants for Insertion and Search

Isle-Tree had shifted KV pairs and inserted a new one, but before the atomic increase of the counter. Isle-Tree just increases the counter for the cache line.

- The counter is one less than the number of non-null values and there is a duplicate value. A crash happened when Isle-Tree was shifting KV pairs to the right for an insertion. Isle-Tree shifts KV pairs to the left from the rightmost non-null KV pair until the one with the greatest key. It then zeros the rightmost non-null value for a boundary.
- The counter is one greater than the number of non-null values and there is no duplicate value. This means a crash happened after Isle-Tree had shifted KV pairs and set null pointer to delete a KV pair, but before the atomic decrease of the counter. Isle-Tree just decreases the counter.
- The counter is the same as the number of non-null values but there is a duplicate value. A crash occurred when Isle-Tree was shifting KV pairs to delete a KV pair. Isle-Tree shifts KV pairs to the left from the right KV pair with the duplicate value until all non-null KV pairs are shifted. It then zeros the rightmost non-null value and decreases the counter to finish the deletion.

Split and merge have no impact to the consistency of LNs. Both operations are a replacement by atomically changing some left sibling LN's sibling pointer. Before the replacement, Isle-Tree is with original consistent LN(s). After the replacement, Isle-Tree is with newly-constructed consistent LN(s). After a split (resp. merge), a KV pair should be inserted (resp. deleted) with the parental IN. Given INs enforced consistency in NVM, a crash may make the parental IN inconsistent with one more or less KV pairs, or with duplicate pointers. Such inconsistencies can be fixed by checking between each parental IN and its children LNs (or INs).

V. Evaluation

A. Evaluation Setup

Platforms: We used a machine with an Intel Core i7-7700 CPU (256KB/1MB/8MB L1/L2/L3 cache and 64B cache line) and 64GB DRAM for evaluation. Ubuntu 18.04.1 and GCC/G++ 7.4.0 were installed. The instructions for cache line flush and memory fence are `clflushopt` and `sfence`. We used a part of DRAM space and set the write latency of emulated NVM to be 300ns regarding asymmetrical write/read of NVM.
Implementations: We have implemented two variants of Isle-Tree: Isle_all maintains INs and LNs in NVM with consistency while Isle-Tree only keeps LNs in NVM. We downloaded the source code of FAST-FAIR and implemented NV-Tree, FPTree, wB+-tree (with both slot array and bitmap [4]) and Circ-Tree. Overall we have seven B+-tree variants.
Testing Methodology: We first tested standalone B+-tree variants with 8B/8B KV pairs. Then we built prototyping KV store systems with them and ran YCSB [14] that inserts, searches, and updates an actual value for each key with a configurable number of clients. We set the default node size to be 512B. The main performance metric is the average execution time.

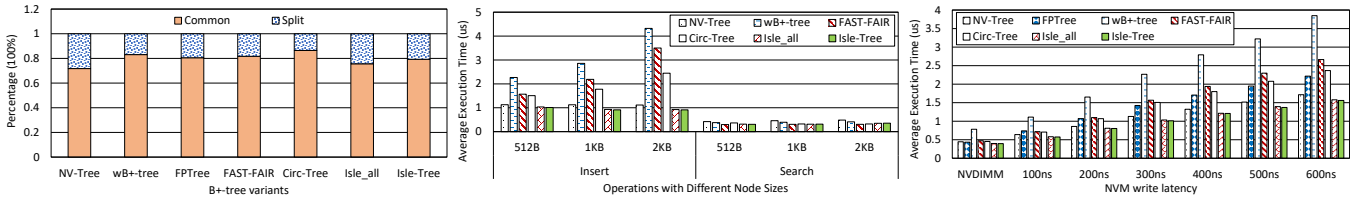
B. Evaluation of Standalone B+-Trees

Insertion Performance: We have inserted 1/10/100 million KV pairs. The keys follow a uniform distribution. As shown in Figure 7a, Isle_all and Isle-Tree consistently outperform state-of-the-art B+-tree variants with varying KV pairs inserted. For example, on inserting 10 million KV pairs, the average execution time of NV-Tree, FPTree, wB+-tree, FAST-FAIR, and Circ-Tree is 11.6%, 41.0%, 124.3%, 55.0%, and 49.1% more than that of Isle-Tree, respectively.

Isle-Tree attempts to reduce memory writes to NVM by flushing less data. We have recorded the overall numbers of `clflushopt` executed at runtime for seven B+-tree variants when each of them was inserting 10 million KV pairs. Isle-Tree mostly calls `clflushopt` twice, i.e., one for the cache line of KV pairs and the other one for the LN header. For FAST-FAIR and Circ-Tree, they shift KV pairs to keep LNs sorted, which may involve multiple cache lines to be modified and flushed. As shown in Figure 7b, they executed 109.2% and 53.2% more `clflushopt` than Isle-Tree.

By appending a newly-arrived KV pair to the tail of an LN, NV-Tree only flushes the involved tail cache line as well as the LN metadata. As illustrated in Figure 7b, NV-Tree flushes almost the same cache lines as Isle-Tree. Though, NV-Tree still yields lower performance than Isle-Tree. The reason is, when any IN becomes full, NV-Tree stalls all incoming requests until it finishes rebuilding INs from scratch [3]. In addition, for every common insertion, FPTree executes one extra `clflushopt` to flush the fingerprint of inserted key [5]. Thus, FPTree yields worse performance than Isle-Tree due to calling 44.0% more cache line flushes.

The performance gap between Isle-Tree and wB+-tree is due to their respective designs. As mentioned in Section III-A, wB+-tree uses a slot array to maintain ascending order of keys in a node. It also has a bitmap to indicate the availability of positions in each node. wB+-tree updates and flushes both structures for every insertion to track unsorted KV pairs. For an insertion without split, wB+-tree calls at least



(a) Breakdown of common insertions (*common*) and insertions with split (*split*)

(b) Three node sizes

(c) NVM latencies

Fig. 8: The Breakdown of Insertions and the Impact of Node Size and NVM Latencies

four `clflushopt`s [4, 7]. Comparatively, Isle-Tree calls two `clflushopt`s for a common insertion. For splits, wB+-tree employs and flushes redo log for slot array and bitmap besides copying KV pairs. Figure 7b confirms that the number of `clflushopt`s used by wB+-tree is 2.6× that of Isle-Tree.

Search Performance: Figure 7c shows the average search time of seven B+-trees. All of them employ the linear search. Nonetheless, the search performance of NV-Tree, FPTree, wB+-tree and Circ-Tree is inferior. They spent 37.0%, 18.9%, 23.0% and 18.3% more time, respectively, than Isle-Tree in searching 10 million keys. NV-Tree has to scan unsorted KV pairs which disfavors CPU's branch prediction. FPTree, despite leveraging fingerprints to probe where the key should be, calculates the fingerprint for every search and incur additional time cost. Worse, the 1B fingerprint is prone to collisions which entail cache misses for loading and comparing actual keys to double check. For wB+-tree, the slot array of a node presents the ascending order of keys. However, the slot array is an indirect index and keys are still scattered across cache lines. Following the slot array to orderly check keys exhibits irregular accesses of cache lines, which are not favored by CPU's prefetcher and branch predictor. Comparatively, Isle-Tree sequentially accesses cache lines of a node with avoidance of unnecessary comparisons. As a result, Isle-Tree outperforms wB+-tree with higher search performance.

As to Circ-Tree, its circular LN usually makes sorted LNs discontinuous in two segments. Every search with Circ-Tree starts with deciding which segment to be scanned, and CPU needs to do a branch prediction. Comparatively, FAST-FAIR and Isle-Tree scan from the first KV pair and facilitate CPU to prefetch successive cache lines. The linear search with hops of Isle-Tree helps it skip greater KV pairs and leave a cache line as early as possible, thereby improving the utilization of cache lines. Isle-Tree hence yields high search performance.

Breakdown of Insertions: We have further analyzed the insertion performance by separately recording the average execution time for common insertions and insertions with split. A 512B LN can accommodate at most 32 KV pairs ($\frac{512}{8+8}$) regardless of LN metadata, so approximately every 32 common insertions trigger an insertion with split. Hence, the execution time of common insertions theoretically dominates the overall execution time. That is the reason why Isle-Tree tries to complete every common insertion with one cache line of KV pairs to be flushed. Figure 8a gives the percentage breakdown of two types of insertions in the overall execution time for each B+-tree variant. In Figure 8a, the percentage of insertions with split is around 20% of overall execution time except for NV-Tree due to its stalling incoming insertions to rebuild INs. These results quantitatively justify Isle-Tree's strategy in reducing memory writes for common insertions.

C. The Impact of Node Size and NVM Latency

Node Sizes: We have configured two greater node sizes (1KB and 2KB) for B+-tree variants excluding FPTree because a greater node's metadata cannot be fitted in one cache line for FPTree. Figure 8b shows the performance of six B+-trees inserting and searching 10 million KV pairs. The insertion performance of NV-Tree, Isle_all, and Isle-Tree does not fluctuate with varying node sizes. Isle-Tree inserts KV pairs at the cache line level. It is unaffected by the change of node size. Comparatively, the insertion performance of wB+-tree, FAST-FAIR and Circ-Tree badly degrades with greater nodes. For them, shifting KV pairs in a greater node is likely to cause more cache lines of the node to be modified and flushed. For wB+-tree, a greater node contains a longer slot array, which demands more `clflushopt`s for updating with regard to the insertion algorithm of wB+-tree [4]. On the other hand, when searching KV pairs in greater nodes, Isle-Tree still outperforms NV-Tree, being comparable to FAST-FAIR and Circ-Tree.

NVM Latencies: The default write latency of NVM used in evaluation was 300ns. We also configured varying write latencies from 0 (identical to NVDIMM [3]) to 600ns. Figure 8c presents the average execution time to insert 10 million KV pairs with seven NVM latencies. A longer latency degrades performance of all B+-trees; however, the gaps between Isle-Tree and others become wider. For example, from NVDIMM to 600ns, the difference between Isle-Tree and FAST-FAIR rises from 20.0% to 70.8%. This is because Isle-Tree aims to reduce memory writes to NVM by flushing less data. With slower NVM technologies, the effect of reducing memory writes becomes more significant.

D. End-to-End Evaluation with KV Stores

To evaluate the applicability and multi-threading of Isle-Tree in real-world environments, we have built prototyping KV store systems with each B+-tree variant as the index structure and exported interfaces to answer requests issued by YCSB's concurrent clients. Due to the space limitation, we show results of running YCSB's workloada ('SessionStore') with six B+-trees except wB+-tree. With workloada, YCSB first inserted 1 million KV pairs, each of which contains a string key and an

(a) Insertion (b) Search (c) Update

Fig. 9: The 99p Latencies of Insertion, Search and Update with Prototyping KV Stores

actual string value; then YCSB searched and updated stored KV pairs in 1 million requests with a ratio of 50%/50%. We ran YCSB with 1, 2, 4, and 8 clients to impose concurrent write/read to every prototyping KV store system. YCSB reported several latencies as the end-to-end performance. In order to rule out biases of abnormal response time caused by factors like software scheduling and network disturbance, we chose the 99th percentile (99p) latency, which means that 99% insertion/search/update requests can be satisfied within such a latency. The three diagrams in Figure 9 correspond to the 99p latencies for insertion, search, and update, respectively.

From Figure 9a, we can observe that with varying clients, Isle-Tree consistently outperforms other B+-tree variants. Take 4 clients for example. The 99p latency of Isle-Tree is 33.1%, 27.7%, 36.2%, 22.4% shorter than that of NV-Tree, FPTree, FAST-FAIR, and Circ-Tree, respectively. Therefore, storing actual values through Isle-Tree dramatically benefits from Isle-Tree's reducing memory writes to NVM and improving cache efficiency, because an actual value also relies on CPU to persist it into NVM through CPU cache. On the other hand, with more and more clients, all B+-tree variants require longer latencies to process insertion requests due to the contention among multiple threads. In addition, the bars of NV-Tree and FAST-FAIR in Figure 9a differ from those in Figure 7a. The average execution time in Figure 7a contains biases to shorter insertion latencies, while 99p latency can be viewed as the latency ranked as the 99% longest one of all insertions. At that point, NV-Tree and FAST-FAIR must heavily do rebuilding and shift KV pairs across all cache lines of an LN, respectively.

For search and update, each B+-tree variant first locates a KV pair by searching and fetches the actual value or updates a part of it, respectively. Reading the actual value and partially replacing it cost the most time for search and update, respectively, and the search process through an indexing B+-tree does not affect much. That explains why Isle-Tree achieves similar or a bit higher search and update performance compared to other B+-tree variants in Figure 9b and Figure 9c.

VI. Conclusion

In this paper, we propose Isle-Tree that handles most insertions and deletions by flushing only one cache line of KV pairs. The split and merge of Isle-Tree are made in a replacement fashion. Isle-Tree also employs a linear scan with hops to avoid unnecessary comparisons. Extensive experiments with standalone B+-trees and prototyping KV stores confirm that Isle-Tree yields both high write and read performance.

References

[1] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, ser. FAST'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 1–15.

[2] S. Chen, P. B. Gibbons, and S. Nath, "Rethinking database algorithms for phase change memory," in *5th Biennial Conference on Innovative Data Systems Research (CIDR '11)*, January 2011, pp. 1–11.

[3] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "NV-Tree: Reducing consistency cost for NVM-based single level systems," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*. Santa Clara, CA: USENIX, 2015, pp. 167–181.

[4] S. Chen and Q. Jin, "Persistent B+-trees in non-volatile main memory," *Proc. VLDB Endow.*, vol. 8, no. 7, pp. 786–797, Feb. 2015.

[5] 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*, ser. SIGMOD '16. New York, NY, USA: ACM, 2016, pp. 371–386.

[6] P. Chi, W. Lee, and Y. Xie, "Adapting B^+ -tree for emerging nonvolatile memory-based main memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 9, pp. 1461–1474, Sep. 2016.

[7] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in byte-addressable persistent B+-Tree," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*. Oakland, CA: USENIX Association, 2018, pp. 187–200.

[8] C. Wang, G. Brihadiswarn, X. Jiang, and S. Chattopadhyay, "Circ-tree: A B+-Tree variant with circular design for persistent memory," https://arxiv.org/abs/1912.09783, 2019.

[9] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-ordering consistency for persistent memory," in *Computer Design (ICCD), 2014 32nd IEEE International Conference on*, Oct 2014, pp. 216–223.

[10] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "ThyNVM: Enabling software-transparent crash consistency in persistent memory systems," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2015, pp. 672–685.

[11] J. Xu and S. Swanson, "NOVA: A log-structured file system for hybrid volatile/non-volatile main memories," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA: USENIX Association, Feb. 2016, pp. 323–338.

[12] J. Ou, J. Shu, and Y. Lu, "A high performance file system for non-volatile main memory," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. New York, NY, USA: ACM, 2016, pp. 12:1–12:16.

[13] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Redesigning lsms for nonvolatile memory with NoveLSM," in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '18. USA: USENIX Association, 2018, p. 993–1005.

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 143–154.

[15] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung, "iDO: Compiler-directed failure atomicity for nonvolatile memory," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. IEEE Press, 2018, p. 258–270.

[16] Intel, "Persistent memory development kit," http://pmem.io/pmdk/.