Circ-Tree: A B+-Tree Variant with Circular Design for Persistent Memory

Chundong Wang, Gunavaran Brihadiswarn, Xingbin Jiang, and Sudipta Chattopadhyay

Abstract—Several B+-tree variants have been developed to exploit the byte-addressable non-volatile memory (NVM). We attentively investigate the properties of B+-tree and find that, a conventional B+-tree node is a linear structure in which key-value (KV) pairs are maintained from the zero offset of a node. These KV pairs are shifted in a unidirectional fashion for insertions and deletions. Inserting and deleting one KV pair may inflict a large amount of write amplifications due to shifting existing KV pairs. This badly impairs the performance of in-NVM B+-tree. In this paper, we propose a novel circular design for B+-tree. With regard to NVM's byte-addressability, our Circ-Tree embraces tree nodes in a circular structure without a fixed base address, and bidirectionally shifts KV pairs for insertions and deletions to minimize write amplifications. We have implemented a prototype for Circ-Tree and conducted extensive experiments. Experimental results show that Circ-Tree significantly outperforms two state-of-the-art in-NVM B+-tree variants, i.e., NV-tree and FAST+FAIR, by up to 1.6× and 8.6×, respectively, in terms of write performance. The end-to-end comparison by running YCSB to KV stores built on NV-tree, FAST+FAIR, and Circ-Tree reveals that Circ-Tree yields up to 29.3% and 47.4% higher write performance, respectively, than NV-tree and FAST+FAIR.

Index Terms—Persistent Memory, B+-tree, Non-volatile Memory, Crash Consistency

1 Introduction

The next-generation non-volatile memory (NVM) has DRAM-like byte-addressability and disk-like durability. Computer architects have proposed to place NVM on the memory bus alongside DRAM to build *persistent memory* [1], [2], [3], [4]. However, the write latency of NVM technologies is generally longer than that of DRAM [5], [6], [7], [8]. Moreover, to guarantee crash consistency of data maintained in NVM demands the execution of cache line flush and memory fence, as the CPU or the memory controller may alter the writing order of multiple writes from CPU cache to memory [8], [9], [10], [11], [12], [13]. Despite being effective in preserving a desired writing order, the usage of cache line flushes and memory fences incurs further performance overhead for in-NVM data structures.

B+-tree, as one of the critical building blocks for computer systems, has been tailored to persistently store and manage key-value (KV) pairs in NVM [8], [9], [14], [15], [16], [17]. These in-NVM B+-tree variants attempt to guarantee crash consistency with minimal performance overheads through either managing unsorted KV pairs of a node in an append-only fashion [9], [15], [16], or leveraging architectural supports to orderly shift sorted KV pairs with the reduction of cache line flushes [8]. Despite employing different techniques, all these works inherit an important property from standard B+-tree: their tree nodes are organized in a *linear* structure that starts at the zero offset and spans over a contiguous space. In particular, for a B+-tree with sorted KV pairs, inserting and deleting a KV pair always shift keys in a *unidirectional* fashion, i.e., to the right for insertion and to the left for deletion, given keys in ascending order. Assuming that the KV pair under insertion is with the smallest key of a node, all existing KV pairs in the node have to be shifted. Consequently, a large amount of memory writes with cache line flushes and memory fences are to take place.

B+-tree has been used since the era of hard disk. The linear tree node is perfectly favored by a hard disk which rotates a magnetic head to sequentially write and read data. However, the byte-addressability entitles NVM the very access flexibility on manipulating data structures stored in it. To this end, we propose a novel *circular* tree node and design a **Circ-Tree** for NVM based on our proposed node structure. The main ideas of Circ-Tree are summarized as follows.

- Circ-Tree organizes sorted KV pairs in a circular node structure that has no fixed base address. Upon an insertion or deletion, Circ-Tree shifts existing KV pairs of a node in a *bidirectional* manner.
- Circ-Tree decides whether to shift to the left or to the right for an insertion or deletion by considering which direction would generate fewer shifts of existing KV pairs. This is to reduce memory writes to NVM via cache line flushes and memory fences.

With the novel circular design, Circ-Tree reduces write amplifications caused by shifting KV pairs while retaining good read performance with respect to sorted KV pairs. We have designed and implemented a prototype of Circ-Tree and performed extensive experiments. Evaluation results show that the write performance of Circ-Tree can be up to

- *C. Wang is with the School of Information Science and Technology, ShanghaiTech University, China. This work was partly done when he worked in Singapore University of Technology and Design, Singapore. E-mail: cd_wang@outlook.com.*
- *X. Jiang, and S. Chattopadhyay are with Singapore Universtiy of Technology and Design, Singapore. E-mail: xingbin_jiang@sutd.edu.sg, and sudipta_chattopadhyay@sutd.edu.sg.*
- *G. Brihadiswarn is with University of Moratuwa, Sri Lanka. This work was done when he worked as an intern in Singapore University of Technology and Design, Singapore. Email: gunavaran.15@cse.mrt.ac.lk.*

Manuscript received MM DD, 2020; revised MM DD, 2020.

1.6× and 8.6× that of NV-tree and FAST+FAIR, respectively.

We have also built KV store systems with three B+-tree variants (i.e., Circ-Tree, NV-Tree, and FAST+FAIR) to observe their end-to-end performances. With YCSB [18], Circ-tree yields 29.3% and 47.4% higher write performance than NV-tree and FAST+FAIR, respectively, with all of them directly committing large values into NVM.

The remainder of this paper is organized as follows. In Section 2, we brief the background of NVM and state-of-the-art B+-tree variants for NVM. We show a motivating example in Section 3. We detail the design of Circ-Tree in Section 4 and Section 5. We present evaluation results in Section 6 and conclude this paper in Section 7.

2 Background and Related Works

NVM technologies, such as spin-transfer torque RAM (STT-RAM), phase change memory (PCM), and 3D XPoint, have become prominent with DRAM-like byte-addressability and disk-like durability. Compared to DRAM, NVM generally has asymmetrical write/read speeds, especially with longer write latencies [5], [7], [8], [9], [19], [20].

One way to incorporate NVM in a computer system is to build persistent memory by putting NVM alongside DRAM on the memory bus. Though, new challenges emerge when a data structure is ported from hard disk to persistent memory that is directly operated by a CPU. First, modern CPUs mainly support an atomic write of 8B [4], [8], [21], [22], [23]. In addition, the exchange unit between CPU cache and memory is a cache line that typically has a size of 64B or 128B. Secondly, for multiple store operations from CPU cache to memory, the CPU or the memory controller may perform them in an order differently from the programmed order. Such reordered writes are detrimental to the crash consistency of in-NVM data structures. For example, the allocation of a new object must be completed before the pointer to the object is recorded. If the writing order is reversed but a crash occurs, the pointer might turn to be dangling. Using cache line flushes and memory fences, e.g., `clflush` and `mfence` in the x86 architecture, is an effective method to retain a desired writing order. Cache line flush explicitly flushes a cache line to memory. Memory fence makes a barrier to regulate that memory operations after the barrier cannot proceed unless ones before the barrier complete. However, the execution of cache line flushes and memory fences incurs performance overheads [8], [9], [10], [11], [12]. Recently, Intel has introduced new instructions, i.e., `clflushopt` and `clwb`, to replace `clflush` with reduced performance overheads.

Computer scientists have proposed a number of artifacts for system- and application-level softwares to utilize persistent memory on the memory bus [2], [3], [4], [6], [7], [12], [20], [22], [24], [25], [26], [27], [28], [29], [30], [31], [32]. In particular, several in-NVM B+-tree variants have been developed [8], [9], [14], [15], [16], [17], [33]. CDDS-Tree keeps all nodes sorted in NVM [14]. It calls cache line flushes and memory fences while shifting every KV pairs to orderly write modified data back to NVM. Yang et al. [9] found that, for CDDS-Tree, the cost of using `clflush` and `mfence` to sort its leaf nodes might take up to 90% of overall consistency cost. Accordingly they studied the idea of unsorted tree nodes [33] and proposed NV-tree. NV-tree only enforces crash consistency to leaf nodes and manages them in an unsorted fashion. It appends a KV pair to a leaf node with respective labels for insertion, update and deletion. However, for every write or read request, NV-tree has to scan all unsorted KV pairs in a leaf node to determine the existence of the key under insertion/deletion/search. Oukid et al. [16] looked into this issue and proposed FPTree. A leaf node of FPTree includes a hash value for stored keys, which can be first checked in order to accelerate searching over unsorted KV pairs.

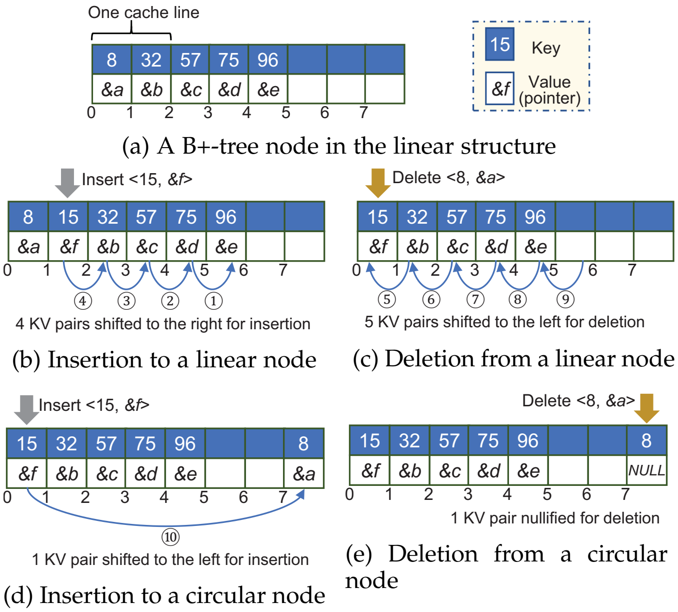
Hwang et al. [8] reconsidered sorted B+-tree nodes and proposed FAST+FAIR that exploits the store dependencies in shifting a sequence of KV pairs in a node on insertion/deletion (e.g., $KV_i \xrightarrow{\text{store}} KV_{i+1}, KV_{i-1} \xrightarrow{\text{store}} KV_i$, etc.). Store dependencies impose a natural writing order among KV pairs and result in reducing the cost of using cache line flushes and memory fences to forcefully write back modified KV pairs to NVM [34], [35], [36]. Nevertheless, FAST+FAIR still needs to orderly flush dirty cache lines holding shifted KV pairs to prevent them from being written to NVM in an altered order.

3 A Motivational Example

State-of-the-art B+-tree variants developed for NVM inherit an important property from the standard B+tree: a B+-tree node is a linear structure that starts at a fixed base address, i.e., zero offset, and spans over a contiguous space; as a result, insertions and deletions onto a B+-tree node always shift KV pairs in a unidirectional fashion. However, in this paper, we show that the linear structure of B+-tree can be logically viewed in a circular fashion with novel operational strategies. We will first introduce a simple example to illustrate the key motivation behind such a design.

Figure 1a shows a linear B+-tree node that keeps keys in ascending order. Without loss of generality, we assume that one CPU cache line holds two KV pairs. In Figure 1b, we insert a KV pair $\langle 15, \&f \rangle$ into the node. Since its key would be the second smallest of the node, four KV pairs must be shifted to the right in order to keep the node sorted. Considering store dependencies in shifting KV pairs, we need to call three cache line flushes and memory fences. This is because three cache lines from the zero offset are modified due to shifting KV pairs and inserting the new KV pair. In other words, inserting $\langle 15, \&f \rangle$ causes writing other four KV pairs with flushing three cache lines. When we delete the smallest key of the node as shown in Figure 1c, all greater KV pairs must be shifted to the left. Again, three dirty cache lines are involved in the deletion and they must be orderly flushed.

The linear structure is well fit for hard disk with a magnetic head rotating to write/read data. The byte-addressable NVM, as directly operated by the CPU, provides an opportunity to revolutionize the linear node structure for B+-tree. Let us assume that in Figure 1b we do not shift four greater KV pairs to the right. Instead we shift the smallest $\langle 8, \&a \rangle$ to the rightmost end of the node and insert $\langle 15, \&f \rangle$ in the leftmost slot (cf. Figure 1d). By shifting one KV pair, just the leftmost and the rightmost cache lines are modified and flushed. As to the deletion in Figure 1c, now that $\langle 8, \&a \rangle$


One cache line
15 Key
&f Value (pointer)
8 32 57 75 96
&a &b &c &d &e
0 1 2 3 4 5 6 7
Insert <15, &f>
4 KV pairs shifted to the right for insertion
Delete <8, &a>
5 KV pairs shifted to the left for deletion
1 KV pair shifted to the left for insertion
1 KV pair nullified for deletion
NULL

(a) A B+-tree node in the linear structure

(b) Insertion to a linear node

(c) Deletion from a linear node

(d) Insertion to a circular node

(e) Deletion from a circular node

Fig. 1: An Example to Compare Classic Linear B+-tree Node to Circ-Tree's Circular Node Design

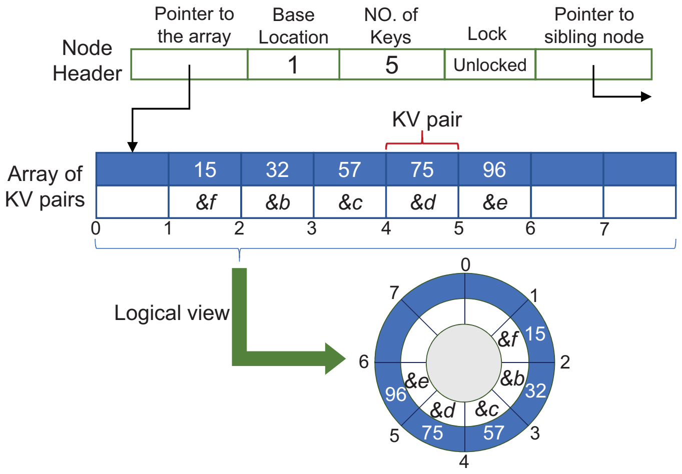


Fig. 2: The Circular Structure of Circ-Tree's Leaf Node

has been shifted to the rightmost cache line of the node, we nullify its value to be NULL without shifting any KV pair (cf. Figure 1e) and hence only one cache line needs to be flushed.

As observed from the preceding example, we transform the linear node of B+-tree into a *circular* structure that supports *bidirectional* shifting. Such a circular design significantly reduces write amplifications caused by shifting KV pairs while retaining sorted B+-tree nodes to support fast lookup. This is the main motivation driving the design of our Circ-Tree. We note that Circ-Tree complements other designs. For example, it can used as one or multiple levels of log-structured merge (LSM) tree developed for NVM [7], [37]. On the other hand, researchers have considered how to make a B+-tree adapt to an ever-changing insertion distribution by maintaining a load factor (a ratio of unused vacancies) for load rebalancing among B+-tree nodes since the era of hard disks [38]. Circ-Tree can be augmented with such an idea as the performance of it may suffer from a fluctuating workload from time to time. However, without loss of generality, we focus on workloads with a stable distribution of insertions in this paper.

4 Tree Node with Circular Deign

4.1 Circular B+-tree Node

Circ-Tree views a linear node as a circular buffer [39] that no longer has a fixed base address at the zero offset. Figure 2 instantiates one leaf node (LN) of Circ-Tree. An LN is composed of two parts, i.e., an array of KV pairs and a node header. An actual value is stored and organized in NVM by calling functions provided by persistent memory libraries, such as Intel Persistent Memory Development Kit (PMDK) [40]. We use the pointer (8B memory address) to the actual value to form each KV pair. As to the key, we use an 8B integer for illustration in this paper. A key can also be in a variable size, e.g., a string with up to 1KB length [41]. Regarding such a key, we jointly store the key and its corresponding value as an entity into NVM and get the address (pointer) to them. Then we calculate a hash value (8B or 16B) for the key. Instead of keeping an actual key in a KV pair, Circ-Tree uses the hash value and the pointer to stored key-value to form a new KV pair with a fixed size and inserts it into an appropriate leaf node.

The reason why Circ-Tree separates its tree node into a header and an array of KV pairs is explained in Section 4.2. As to the node header, it contains the following items:

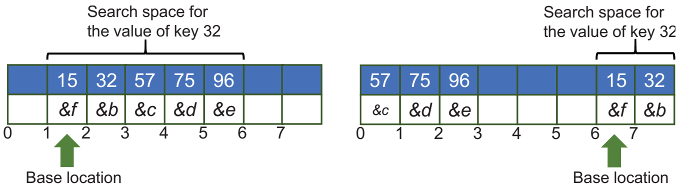
1) a pointer (8B) pointing to the array of KV pairs,
2) the current base location (4B),
3) the number of valid keys in the LN (4B),
4) a lock (8B) for concurrent access, and
5) a pointer (8B) pointing to the right sibling LN.

The base location indicates where the smallest key is stored. The number of keys maintains the boundary of circular space that is being filled with valid KV pairs. We can use an 8B atomic write to atomically modify these two items together as well as either pointer in the node header. Figure 2 also illustrates a logically circular view of the example LN.

An internal node (IN) of Circ-Tree has the same structure as its LN, except that each IN has an unused key so that the number of values is always one more than the number of keys. This is in line with the definition of B+-tree. In the LN, the value of a KV pair is a pointer to a record where the actual value is stored. In the IN, the value is a pointer to the lower-level IN or LN. Circ-Tree enforces crash consistency to all INs and LNs and it incorporates strict modification orders in insertion and deletion for crash recoverablity.

4.2 Optimization for Circular Design

The circular node structure enables Circ-Tree to shift KV pairs bidirectionally, which can be implemented using the modulo operation (% is the modulo operator in C/C++/Python/Java). Let the maximum number of values that can be stored in a node be N. Assuming that a new KV pair should be inserted at the i-th offset from the base



(a) Searching in the contiguous space (Case 1)

(b) Searching in the disjointed space (Case 2)

Fig. 3: An Example of Searching for a Key by Circ-Tree

location (b), the real position p in the array of KV pairs would be

$$p = (b + i)\%\mathsf{N}.$$

The left and right moves by one for the base location would be $(b-1)\%\mathsf{N}$ and $(b+1)\%\mathsf{N}$, respectively.

In practice, the modulo operation is expensive for a CPU because it is a form of integer division. As Circ-Tree frequently and bidirectionally shifts KV pairs for insertions and deletions, we need an efficient calculation to substitute modulo operations. We find that, when N is a power of two, i.e., $\mathsf{N} = 2^m$ ($m > 0$), the modulo operation can be reformed as

$$p = (b + i)\%\mathsf{N} = (b + i)\&(\mathsf{N} - 1),$$

in which $\&$ is the bitwise AND operator. For example, given $\mathsf{N} = 256$, $(133 + 165)\%256 = (133 + 165)\&255 = 42$. The bitwise AND operation is much more CPU-friendly than the modulo operation. As a result, we organize an array that has a size in a power of two, e.g., 2^6, for KV pairs and is competent for using bitwise AND operation for modulo calculations. As a CPU cache line usually has a size of 64B or 128B, we separate the array of KV pairs from the node header to make the former cache line-aligned and-friendly. Meanwhile, the node header has an overall size of 32B and a multiple of it can be fitted in one cache line. This further improves the CPU cache efficiency of Circ-Tree.

5 Search, Insertion, & Deletion

5.1 Search

Search is critical for B+-tree because it is essential part of insertion and deletion. Since Circ-Tree employs a logically circular node structure, valid KV pairs of an LN can occupy either one contiguous space or two segments. Figure 3 exemplifies both possible cases. To avoid cache misses due to traversing disjointed cache lines as shown in Figure 3b, Circ-Tree only searches in a contiguous space. First, whether the distance between locations of the smallest and greatest keys is positive or negative indicates which case in Figure 3 a node corresponds to. Secondly, with two disjointed segments, given a key to be searched, e.g., 32 in Figure 3b, Circ-Tree decides which segment to be searched by comparing the key under search to the key at the zero offset. Therefore, as shown in Figure 3a and Figure 3b, Circ-Tree manages to search only one continuous space in both cases without jumping between discontinuous cache lines. Circ-Tree can use either binary or linear search. Because linear search is more CPU cache-friendly than binary search especially with short arrays [8], Circ-Tree employs the former in its implementation.

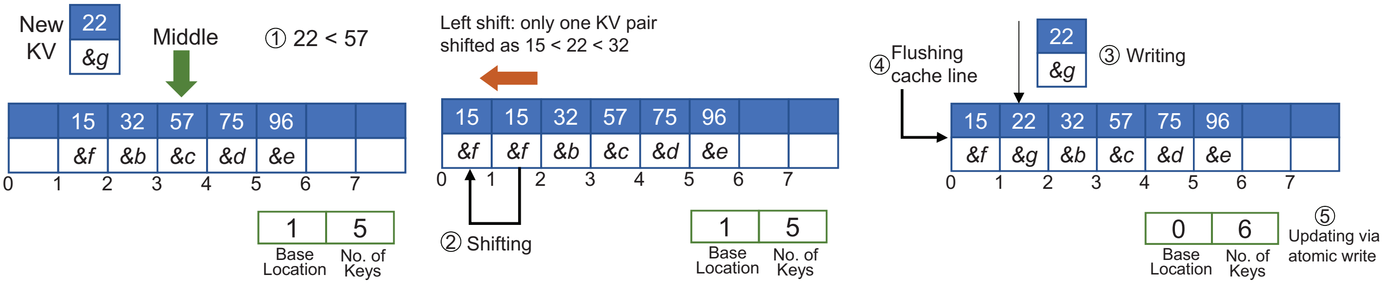
Algorithm 1 Insertion of Circ-Tree (`Insert`($< k, v >$))

```
Input: A KV pair $< k, v >$ to be inserted
1: Search from the root until the target node header $nh$ and KV array $A//nh$ has the base location $b$ and the number of KV pairs $n$
2: if ($nh.n \geq \mathsf{N}$) then
3:     split($nh$, $< k, v >$); //To split the node with $< k, v >$
4: else
5:     if ($k < A[(nh.b + \frac{nh.n}{2})\&(\mathsf{N}-1)].key$) then
6:         for ($i := 0; i < \frac{nh.n}{2}; i := i + 1$) do
7:             $index := (nh.b + i)\ \&\ (\mathsf{N} - 1)$;
8:             if ($k > A[index].key$) then
9:                 $A[(index - 1)\ \&\ (\mathsf{N} - 1)].val := A[index].val$;
10:                $A[(index - 1)\ \&\ (\mathsf{N} - 1)].key := A[index].key$;
11:                if ($A[index]$ is at the start of a cache line) then
12:                    Flush_cacheline($\&A[index - 1]$);
13:                end if
14:            else
15:                break; //Find the appropriate position
16:            end if
17:        end for
18:        $A[(nh.b + i - 1)\ \&\ (\mathsf{N} - 1)].key := k$;
19:        $A[(nh.b + i - 1)\ \&\ (\mathsf{N} - 1)].val := v$;
20:        Flush_KV($\&A[(nh.b + i - 1)\&\mathsf{N} - 1]$);
21:        Update_b_n($nh$, $((nh.b - 1)\ \&\ (\mathsf{N} - 1))$, $(nh.n + 1)$);
22:        Flush_b_n($\&nh$);
23:    else
24:        for ($i := nh.n - 1; i >= \frac{nh.n}{2}; i := i - 1$) do
25:            $index := nh.b + i\ \&\ (\mathsf{N} - 1)$;
26:            if ($k < A[index].key$) then
27:                $A[(index + 1)\ \&\ (\mathsf{N} - 1)].val := A[index].val$;
28:                $A[(index + 1)\ \&\ (\mathsf{N} - 1)].key := A[index].key$;
29:                if ($A[(index + 1)\&(\mathsf{N} - 1)]$ is at the start of a cache line) then
30:                    Flush_cacheline($\&A[(index + 1)\&(\mathsf{N} - 1)]$);
31:                end if
32:            else
33:                break; //Find the appropriate position
34:            end if
35:        end for
36:        $A[(nh.b + i + 1)\ \&\ (\mathsf{N} - 1)].key := k$;
37:        $A[(nh.b + i + 1)\ \&\ (\mathsf{N} - 1)].val := v$;
38:        Flush_KV($\&A[(nh.b + i + 1)\&\mathsf{N} - 1]$);
39:        Update_b_n($nh$, NIL, $(nh.n + 1)$);
40:        Flush_b_n($nh$);
41:    end if
42: end if
```

5.2 Insertion

Algorithm 1 captures the single-threading procedure of insertion for Circ-Tree with the x86 architecture. Inserting a new KV pair starts by traversing from the tree root until reaching the target LN (Line 1). Then Circ-Tree checks if the current LN is full (Line 2). A full LN is split with the newly-arrived KV pair (Line 3). Otherwise, Circ-Tree inserts the new KV pair into the LN (Lines 4 to 42).

Next, Circ-Tree needs to decide the direction to shift KV pairs (Lines 5). Circ-Tree compares the new key to the key at the middle position. If the new key is smaller than the middle one, then the new key is inserted into the logically smaller half. Shifting to the left incurs fewer KV pairs to be moved, because shifting to the right surely moves more than



(a) $22 < 57$, $\langle 22, \&g \rangle$ to be inserted to the lower half by shifting to the left.

(b) Shifting KV pairs that have smaller keys than the new key, i.e., only $\langle 15, \&f \rangle$.

(c) Writing $\langle 22, \&g \rangle$ in the proper position and flushing the corresponding cache line before atomically updating the base location and the number of keys.

Fig. 4: An Example of Inserting a New KV Pair into a Circular LN by Circ-Tree

half KV pairs. Circ-Tree starts shifting KV pairs which have smaller keys than the new key (Line 8) to the left (Lines 7 to 13) until it reaches the first key that is not smaller than the new key (Line 8). This is the position where the newly-arrived KV pair is placed (Line 15). When shifting KV pairs to the left, if Circ-Tree crosses a boundary of cache lines and enters the next left cache line, it flushes the modified dirty cache line for consistency and persistency (Lines 11 to 13). After inserting and flushing the new KV pair into the appropriate position (Lines 18 to 20), the base location of the LN is moved to the left by one (i.e., $((nh.b - 1) \,\&\, (\mathsf{N} - 1))$) and the number of KV pairs is increased by one (i.e., $(nh.n + 1)$). As Circ-Tree keeps these two items in an 8B word, it atomically modifies and flushes the 8B to complete the insertion (Lines 21 to 22). In other words, Circ-Tree leverages this 8B atomic write as the end mark of insertion, and it is useful in crash recovery.

If the new key falls into the logically greater half, Circ-Tree shifts KV pairs to the right (Lines 24 to 40). Circ-Tree moves KV pairs that embrace greater keys than the key to be inserted, and flushes dirty cache lines where necessary (Lines 25 to 31). Then it inserts and flushes the newly-arrived KV pair into the proper location (Lines 36 to 38). These steps are similar to steps in a left shifting except that, in the end, only the number of KV pairs needs to be increased by one (Lines 39 to 40) as the base location does not change in a right shifting. The increase of the number of KV pairs is also accomplished with an 8B atomic write.

Figure 4 describes how Circ-Tree deals with an insertion request into the LN shown in Figure 2. To insert the new KV pair $\langle 22, \&g \rangle$, Circ-Tree first decides the shifting direction (① in Figure 4a). As the new key 22 is smaller than the key at the middle position, shifting to the left incurs fewer moves of KV pairs. As shown in Figure 4b, Circ-Tree shifts one KV pair to the left to make room for $\langle 22, \&g \rangle$ (② in Figure 4b). Note that the shift of $\langle 15, \&f \rangle$ is inside one cache line and does not cross a cache line boundary. Next, Circ-Tree puts the new KV pair in the appropriate position, and flushes the modified cache line due to moving $\langle 15, \&f \rangle$ and writing $\langle 22, \&g \rangle$ (③ and ④ in Figure 4c). In the eventual step, the base location and the number of keys is atomically modified and flushed to the NVM (⑤ in Figure 4c).

Split Circ-Tree splits an LN when the LN's space is consumed up upon inserting a newly-arrived KV pair. The main steps of Circ-Tree in splitting are as follows.

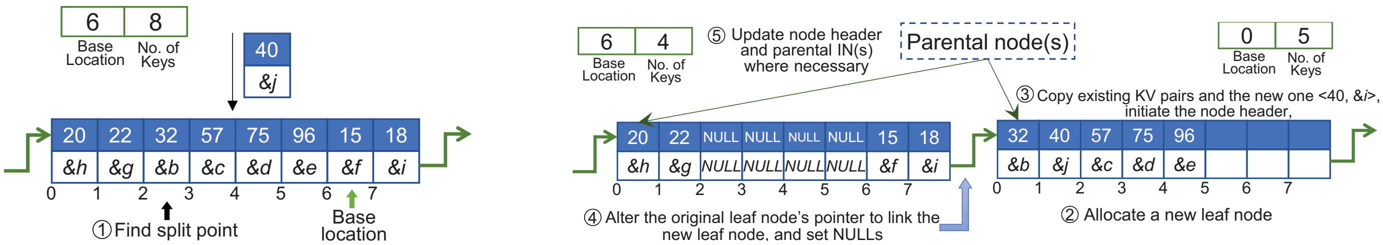
1) Circ-Tree first determines the split point for the LN, i.e., the middle position of the LN.
2) Circ-Tree allocates a new LN. From the split point, the greater half of KV pairs are copied into the newly-allocated LN. If the newly-arrived KV pair falls into the range of greater half, then we copy it alongside without shifting any KV pairs.
3) After copying, Circ-Tree fills the new LN's node header accordingly and sets the sibling pointer pointing to the right sibling of the original LN.
4) Circ-Tree alters sibling pointer pointing to the newly-allocated LN via an atomic write. It then clears copied KV pairs with NULLs (zeros). If the newly-arrived KV pair falls into the range of smaller half, Circ-Tree inserts it into the original LN.
5) Circ-Tree changes the number of keys in the original LN's node header via an atomic write, and starts updating upper-level INs where necessary.

Figure 5 shows an example of splitting an LN by Circ-Tree. Circ-Tree first finds the split point (① in Figure 5a). Then it allocates a new LN (② in Figure 5b). We note that any newly-allocated node for Circ-Tree is shredded with zeros (NULLs) [30]. Then Circ-Tree starts copying half KV pairs as well as the newly-arrived one to the new LN, sets the base location and the number of keys as well as the the sibling pointer pointing to the right sibling of the original LN with two successive atomic writes, respectively (③ in Figure 5b). Next, Circ-Tree alters the original LN's sibling pointer to point to the new LN via an 8B atomic write, and nullifies copied KV pairs in the original LN with NULLs (④ in Figure 5b). In the end, Circ-Tree updates the number of keys in the original LN's node header with an 8B atomic write and starts updating parental IN(s) where necessary (⑤ in Figure 5b). This strict modification order is obeyed to sustain Circ-Tree's crash recoverability.

5.3 Deletion

The deletion with a key needs to locate a target LN as well as the exact position where the KV pair is in the LN. In particular, Circ-Tree considers two cases when deleting a KV pair from an LN.

- If the key to be deleted is the smallest or the greatest key of the LN, its value and key are cleared to be NULL


6
8
Base Location
No. of Keys
40
&j
20 22 32 57 75 96 15 18
&h &g &b &c &d &e &f &i
0 1 2 3 4 5 6 7
①Find split point
Base location
6
4
Base Location
No. of Keys
⑤ Update node header and parental IN(s) where necessary
Parental node(s)
0
5
Base Location
No. of Keys
③ Copy existing KV pairs and the new one <40, &j>, initiate the node header,
20 22 NULL NULL NULL NULL 15 18
&h &g NULL NULL NULL NULL &f &i
0 1 2 3 4 5 6 7
32 40 57 75 96
&b &j &c &d &e
0 1 2 3 4 5 6 7
④ Alter the original leaf node's pointer to link the new leaf node, and set NULLs
② Allocate a new leaf node

(a) Finding the split point of LN and which half the new KV pair will join (①)

(b) Creating new LN (②), copying KV pairs (③), linking the new LN into LN linked list as well as setting NULLs (④), and updating node header and parental INs (⑤)

Fig. 5: An Example of Splitting an LN with a New KV pair by Circ-Tree

with cache line flush and memory fence subsequently executed.

- If the key under deletion falls inside the key range of the LN, Circ-Tree shifts KV pairs to the left or right. The shifting direction still depends on which direction shall incur fewer KV shifts.

In the first case, using NULLs to nullify the KV pair at either end of an LN helps to set boundaries that can be leveraged to identify valid KV pairs in recovery. As to the second case, a removal of an in-between KV pair entails shifting KV pairs. If the KV pair to be deleted has a greater key than the middle KV pair, Circ-Tree shifts greater KV pairs to the left and then sets the KV pair of the greatest key at its original position to be NULLs. Otherwise, Circ-Tree shifts smaller KV pairs to the right and clears the KV pair of the smallest key at its original position using NULLs. Eventually Circ-Tree decreases the number of valid KV pairs by one and/or updates the base location in the node header via an 8B atomic write.

Merge Continuous deletions dwindle the space of nodes. A B+-tree node becomes underutilized when its number of valid KV pairs drops below N/2. Circ-Tree considers merging an underutilized node's KV pairs with its right sibling if both of them are under the same parental IN and the right sibling has sufficient vacant space to accommodate all KV pairs of the underutilized node. The reason why Circ-Tree only merges a node with its right sibling is for multi-threading, which will be explained later. During a merge, KV pairs are first inserted into the right sibling. Circ-Tree then atomically modifies the number of KV pairs and the base location for the right sibling. Then it resets the number of KV pairs in the underutilized node to be zero via an atomic write. Next the node linked list is adjusted by detaching the underutilized node. Circ-Tree then updates upper-level INs where necessary.

In short, the procedures of merge is similar to that of split and both of them must abide by strict modification orders instantiated in Figure 5 to avoid crash inconsistency.

5.4 Multi-threading Access

As shown in Figure 2, each node header includes a lock to support multi-threading accesses. Figure 6a captures two threads aiming to insert KV pairs to the same LN. As Thread-1 has locked the LN and is doing insertion (① in Figure 6a), Thread-2 is being blocked (② in Figure 6a).

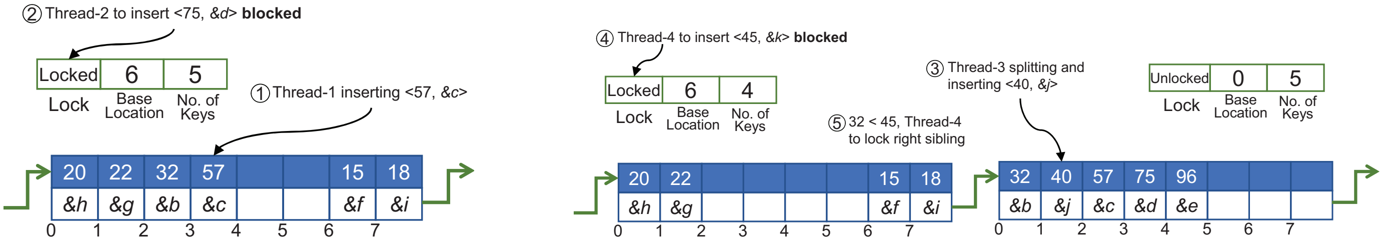
Figure 6b illustrates a more complicated scenario as the current thread is splitting an LN that the other thread has been waiting for (③ and ④ in Figure 6b). To avoid a KV pair to be incorrectly inserted, Circ-Tree checks whether the key under insertion is greater than the smallest one in the LN's right sibling. If so, the current thread releases the lock but locks the right sibling (⑤ in Figure 6b). Otherwise, it proceeds with the insertion.

Parental IN may need to be split with multi-threading access. In this scenario, before releasing the LN's lock, the current thread locks and operates the parental IN. It may further lock upper-level INs up to the root. If an IN is being locked by another thread, the current thread will wait and may need to traverse to the IN's right sibling to proceed. Once the thread finishes an insertion into the highest IN where necessary, it unlocks the IN and goes down to unlock lower-level INs and the LNs. In other words, Circ-Tree recursively locks and unlocks an LN and its parental INs.

Circ-Tree deals with a merge under multiple threads similarly to a split. When a thread is merging KV pairs of an underutilized node to its right sibling, the other insertion/deletion/search thread that waits for the lock of the underutilized node later finds the number of KV pairs changes to be zero. The current thread will also go to the right sibling to continue. In the multi-threading environment, the sibling pointer of an underutilized node that has been merged is not immediately cleared if the lock in the node header is on hold.

5.5 Recovery of Circ-Tree

The recoverability of Circ-Tree is entitled by 1) the 8B atomic update of the base location and the number of valid keys as well as two pointers in the node header, 2) the NULL boundaries and non-duplicate valid values in the array of KV pairs, and 3) strict execution orders of insertion and deletion. The property of non-duplicate values in a B+-tree node has been exploited [8]. However, as Circ-Tree bidirectionally shifts KV pairs, it needs further efforts for crash recovery. To detect the occurrence of crashes, a special flag is installed in the root node of Circ-Tree. It is flagged up when Circ-Tree starts and cleared in case of a normal exit. If a crash happens, Circ-Tree will be conscious of the uncleared flag and initiate a recovery. It traverses all tree nodes in a bottom-up way, i.e., from LNs to INs. Circ-Tree scans successive LNs to discover possible inconsistency scenarios.



(a) Multi-threading concurrent access to an LN

(b) Multi-threading concurrent access to an LN being split

Fig. 6: An Example of Handling Multi-threading Accesses by Circ-Tree

1) The number of keys is one smaller than the non-NULL values in the array of KV pairs. This is caused by an incomplete insertion.
   a) If there are duplicate non-NULL values, which means the crash occurred during shifting KV pairs, Circ-Tree will use the base location to decide how to undo the insertion. In brief, if the value at the base location's logical left is non-NULL, which means shifting to the left was being performed prior to the crash, Circ-Tree shifts to the right from the logical leftmost KV pair until the duplicate value is shifted back; otherwise, Circ-Tree shifts KV pairs to the left until the duplicate value is shifted back.
   b) If no duplicate non-NULL values exist, that means shifting KV pairs had been completed but the crash happened before atomically updating the base location and the number of KV pairs. Circ-Tree atomically modifies them to bring the LN to a consistent state.
2) The number of keys is one greater than the non-NULL values in the array of KV pairs. This is caused by an incomplete deletion. Similarly to aforementioned cases with an incomplete insertion, Circ-Tree uses the number of non-NULL values, the existence of duplicate values and the base location to fix the inconsistency.
3) Two sibling LNs contain duplicate valid values. That means a crash has happened before the completion of a split or merge procedure.
   a) If both LNs are consistent concerning their respective arrays and node headers, the crash must have happened 1) before the two LNs' parental IN was updated in split, since updating the LN's node header and parental IN is the last step in split (cf. ⑤ in Figure 5 for split), and 2) before the number of KV pairs was reset to be zero in merge. By analyzing these two LNs and their parental IN, Circ-Tree can proceed and complete the split or merge.
   b) If the number of non-NULL KV pairs in one LN is not the same as the number of keys in this LN's node header, that means either setting NULLs has not been done in splitting this LN (cf. ④ in Figure 5), or resetting the number of keys as zero in the LN's node header during merge has been done. Circ-Tree fixes such inconsistencies by removing duplicate KV pairs from the LN's array and updating its node header. It might update the LN's parental IN if the other LN is not recorded in the parental IN during split.

After scanning LNs, Circ-Tree scans and fixes inconsistent INs with similar strategies in addition to a double check of each IN's children nodes. Because a crash may occur at the last step of split (resp. merge), i.e., just before ⑤ in Figure 5b, the number of pointers stored in an IN can be one less (resp. more) than the actual children nodes although no duplicate pointers exist in the IN. Given such an IN, Circ-Tree traverses its corresponding children in the next-level node linked list and compares their addresses to the pointers recorded in the IN. If one child is found to be missing (resp. present) in the IN, Circ-Tree inserts (resp. deletes) it within the IN. Circ-Tree does so up to the root in a bottom-up fashion. In recovery, Circ-Tree also reinitializes all locks in node headers to be unlocked. In addition, when using Circ-Tree to build a KV store system, we demand that, to insert a KV pair with concrete value, the value itself must be persisted into NVM before inserting the key and the pointer to the committed value into Circ-Tree. Modifying a value is conducted in the copy-on-write way before replacing the corresponding pointer in the Circ-Tree.

6 Evaluation

In this section, we first evaluate Circ-Tree as a standalone B+-tree variant by comparing to other volatile or in-NVM B+-tree variants under single-threading and multi-threading workloads. Then we test the efficacy of Circ-Tree when it is used as the indexing structure in a KV store system. To do so, we employ the prevalent YCSB [18] for benchmarking.

6.1 Evaluation Setup

Platform We have used a Dell OptiPlex 7050 in which there is an Intel Core^{TM} i7-7700 CPU with 256KB/1MB/8MB for L1/L2/L3 caches, respectively. The CPU provides `clflushopt` for cache line flush. The instruction used for memory fence is `sfence`. The operating system is Ubuntu 18.04.2 and the compiler version is GCC/G++ 7.3.1.

The machine has 64GB DRAM and we use a part of it to emulate the NVM space. NVM technologies generally have asymmetrical write/read latencies. We keep the read latency of NVM the same as that of DRAM, and emulate the write latency of NVM by adding an extra delay after each `clflushopt` instruction [20], [42]. Following previous works [7], [8], we set the default write latency of NVM as 300ns. We note that adding a delay aims to reflect the write latency of typical NVM technologies but does not prevent multiple cache lines from being flushed in parallel as supported by using clflushopt. We note that, in spite of the hardware limit by using DRAM for emulation, the idea of circular node structure is widely applicable to byte-addressable NVM technologies, including both Intel 3D XPoint and ones that are being under development, such as STT-MRAM and ReRAM. Shifting KV pairs bidirectionally makes the best of such NVM technologies and secures the efficacy of Circ-Tree.

Competitors We have used C++ to implement two versions of Circ-Tree. One is a Circ-Tree with circular node structure and linear search starting from the base location of a node to the last valid KV pair (Circ-Tree_ls). The other Circ-Tree performs linear search only in a contiguous space of a circular node. We followed the libpmem2[1] of Intel PMDK to allocate and deallocate NVM space. The reason is that, we must maintain crash consistency of Circ-tree by ourselves, and the memory addresses provided by such library facilitate cache line flushes.

We employ other five B+-tree variants to compare against Circ-Tree. We implemented a standard volatile B+-tree (Volatile_std) without forcefully flushing any data to NVM (without clflushopt or sfence but retaining the write latency). We also implemented another volatile B+-tree (Volatile_circ), the node of which is in the circular fashion. NV-tree, FPTree, and FAST+FAIR three two state-of-the-art in-NVM B+-tree variants. The former two are with unsorted leaf nodes while the latter one employs sorted nodes. FAST+FAIR has open-source code[2] while we implemented NV-tree and FPTree[3] strictly aligning with their respective literatures [9], [16]. All implementations have been compiled with -O3 option.

We used 512B, 1KB, 2KB, and 4KB for the IN and LN sizes. We note that the node size refers to the size of the array of KV pairs and a larger node holds more KV pairs. For a fair comparison, in any node of each tree, we made the array of KV pairs cache line-aligned and also separated the node header into an individual cache line. In addition, the original design of FPtree requires the header of an LN, including 1B fingerprints for all KV pairs, a pointer to a sibling LN, and the LN's bitmap, to be fitted in one cache line. Regarding a KV pair of 16B (8B key and 8B pointer) and a cache line of 64B, only the node size of 512B ($\frac{512\text{B}}{16\text{B}} = 32 \leq$ 64B) satisfies such a requirement. As a result, FPTree would not be involved in comparisons with node sizes greater than 512B.

For end-to-end comparison, we built four KV store prototypes with Circ-Tree, Circ-Tree_ls, NV-tree, and FAST+FAIR, respectively. These KV store systems include interfaces to receive and handle access requests issued by YCSB. The key from a YCSB workload is a string with a prefix ('*user*') and a number. We removed the prefix and used an unsigned 8B integer to treat the number as a key. The default value of YCSB has ten fields with 100B per field, so we made a two-dimensional (2D) array for each value. The first dimension holds pointers to ten fields. To add a new key-value, the actual value must be committed and flushed to NVM before inserting its KV pair in an indexing tree. To update one field of a value, we used copy-on-write to first write the field elsewhere and change the field's pointer in the value's 2D array.

Workloads To evaluate standalone B+-tree variants, we followed the widely-used uniform distribution [6], [8], [9], [16] to generate one million, ten million, and 100 million unsigned 64-bit keys that do not exhibit access skewness. Each key was inserted with an 8B pointer as a value into a tree. We used the SessionStore (numbered as 'workloada') workload with YCSB. It first inserts a predefined number of KV pairs and then follows a search/update ratio of 50%/50% over keys selected in accordance with a Zipfian distribution. The number of inserted KV pairs in our experiment is 1 million per client thread while the total number of search and update requests is also 1 million (0.5/0.5 million) per client thread. In addition, deletion is just a reverse operation of insertion for B+-tree by shifting KV pairs in opposite directions, so the difference among deletion performances of trees is similar to that of insertion performances. Due to the space limitation, we focus on showing the performances of insertion and search.

Metrics We use the average latency per insertion/search as the main metric to measure performance. A shorter average latency means a higher performance. However, using arithmetic mean to calculate average latency is inaccurate due to its biases to very short or very long latencies [43]. For a standard B+-tree, inserting a new greatest key into a node takes significantly shorter time than inserting a key that incurs splits up to the root. As a result, we have chosen the geometric mean to calculate the average latency [43], [44]. For end-to-end comparison, YCSB reports a series of latencies from which we choose the 99th percentile latency to rule out the impact of very short or very long operations. It means that 99% of overall write/read requests can be completed below such a latency. In addition, each number of the results presented in following subsections is the average value in geometric mean by running the respective experiment for five times.

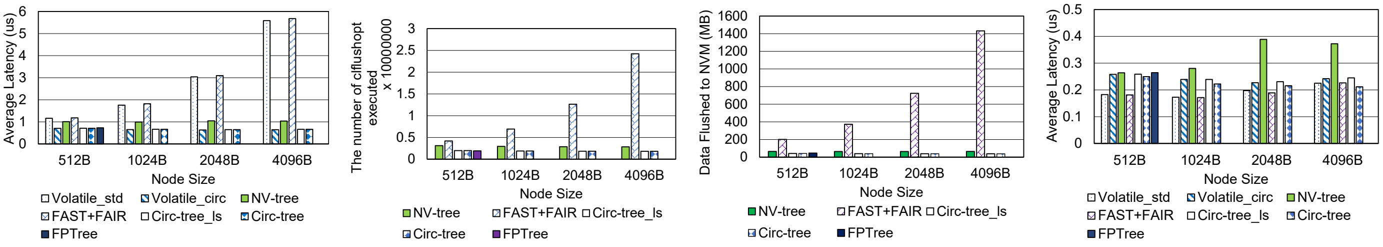
6.2 Performance Comparisons on Insertion and Search

Figure 7a captures the average latencies of seven trees on inserting one million keys with four node sizes. From Figure 7a, we first observe that Circ-Tree substantially outperforms NV-tree, FPTree, and FAST+FAIR with much shorter latencies. For example, with 4KB node, the average latencies of NV-tree and FAST+FAIR are 1.6× and 8.6× that of Circ-Tree. We have recorded the number of clflushopt executed by NV-tree, FPTree, FAST+FAIR, Circ-Tree_ls, and CIRC-Tree to flush data to NVM and the overall amount of flushed data. These two numbers help to explain the reason for Circ-Tree to yield much higher write performance. As indicated by Figure 7b and Figure 7c, Circ-Tree incurred the fewest clflushopt and flushed the least amount of data. With 4KB node size, the numbers of executed clflushopt of NV-tree and FAST+FAIR are 1.6× and 13.3× that of Circ-Tree, respectively, while the amount of data flushed by NV-tree and FAST+FAIR are 1.7× and 37.8× that of Circ-Tree, respectively. The substantial reductions of executed clflushopt and flushed data in turn justify the reduction of write amplifications achieved by Circ-Tree.

[1] https://pmem.io/pmdk/libpmem2/

[2] Available at https://github.com/DICL/FAST_FAIR.

[3] The hash function used for calculating 1B fingerprints for FPTree is from https://stackoverflow.com/questions/7666509/hash-function-for-string, as instructed by the authors of FPTree paper.



(a) Average insert latency (μs) (b) Number of `clflushopt` (c) The amount of flushed data (d) Average latency (search, μs)

Fig. 7: A Comparison of Seven Trees on Inserting and Searching 1 Million Keys

Let us first make a comparative analysis between Circ-Tree and FAST+FAIR. FAST+FAIR keeps keys sorted in a linear node structure and shifts KV pairs in a unidirectional way. Circ-Tree employs a circular node and bidirectionally shifts KV pairs. In practice, the impact of linear structure and unidirectional shifting is cascading and profound. Let us consider a special case in which we continue to insert a new smallest key to a 4KB linear node until the node is full. A 4KB node can hold at most 256 KV pairs. From inserting the second KV pair, all existing KV pairs must be shifted. Therefore, in all, $(0 + 1 + 2 + \cdots + 255) = 32,640$ KV pairs have to be shifted. Nonetheless, for a circular node, no shift is required for any insertion. Concretely, Circ-Tree surely conducts much fewer memory writes to NVM than FAST+FAIR and in turn significantly reduces write amplifications.

The reason why Circ-Tree outperforms NV-tree is multi-fold. First, although NV-tree uses the append-only fashion to avoid shifting KV pairs, the way it splits an LN is time-consuming. This is because it must first scan all unsorted KV pairs to separate smaller half from greater half, and then copy and flush them into two newly-allocated LNs. Secondly, NV-tree needs a flag for each KV pair to label the validity of the KV pair, which decreases space utilization of an LN and holds fewer KV pairs than FAST+FAIR and Circ-Tree. Hence, given the same insertion workload, more splits are expected for NV-tree. These explain why NV-tree flushed more data than FAST+FAIR and Circ-Tree. Thirdly, the design of NV-tree demands that it has to stall and rebuild INs even though only one IN becomes full. This is because NV-tree organizes all INs in a contiguous memory space which cannot be adjusted except a rebuilding [9]. Finally, every insertion for NV-tree has to traverse all KV pairs stored in an unsorted node to find out whether a previous valid version exists.

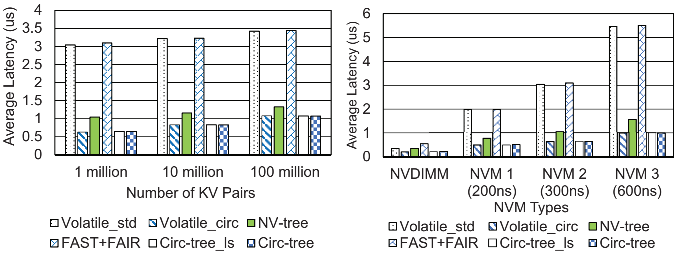
A comparison between FPTree and Circ-Tree with 512B node size shows that Circ-Tree achieves comparable average insertion and a bit higher (5.6%) search performances than FPTree. Nevertheless, Circ-Tree yields 28.9% higher performance for insertions with splits. An insertion with split costs much longer time due to separating and copying KV pairs. In other words, the worst-case response time (WCRT) of Circ-Tree is shorter than that of FPTree. On an insertion without split, FPTree calculates and compares fingerprints; then it calls cache line flushes for three times to successively persist the newly-inserted KV pair, new fingerprint and updated bitmap [16]. On an insertion with split, FPTree copies and flushes *all* KV pairs to a newly-allocated LN. Then it clears half bits in the bitmaps of original and newly-allocated LNs respectively for lower half and greater half keys. Comparatively, Circ-Tree bidirectionally shifts KV pairs on an insertion without split and copies only half KV pairs on an insertion with split. Therefore, Circ-Tree is faster than FPTree considering insertions with split while on average the two achieve comparable performance over all insertions. As to search, FPTree's fingerprint calculation takes time, and the collisions of 1B fingerprints may lead to multiple comparisons and cache misses. Thus, it eventually yields similar performance to Circ-Tree with linear search.

From Figure 7a, we also observe that the performance gap between FAST+FAIR and Circ-Tree becomes wider with a larger node size. From 512B to 4KB, the average latency of FAST+FAIR is 1.7×, 2.7×, 4.8×, and 8.6× that of Circ-Tree, respectively. A smaller node holds fewer KV pairs. Hence, the performance gain brought by bidirectional shifting decreases. However, given the same amount of data stored, a smaller node size entails much more INs for indexing. For instance, assume that there are $2^{20} \approx 1$ million keys that are already ordered, so they can be densely stored in B+-tree nodes. Given 512B node that holds at most 32 KV pairs, the height of a B+-tree is four with one, 32, and 1024 INs at three upper levels, respectively. These INs take about $(1 + 32 + 1024) \times 512 \approx 529$KB NVM space. As to the 4KB node that holds a maximum of 256 KV pairs, the height of B+-tree is three with one and 16 INs at two upper levels, respectively. These INs cost $(1 + 17) \times 4096 =$68KB NVM space. Therefore, 512B node demands $\frac{529}{68} \approx 7.8\times$ NVM space for INs compared to 4KB node. Besides, more INs demand extensive memory allocations and links of nodes into a tree. We note that INs are only for indexing and can be always reconstructed from LNs [9]. As a result, Circ-Tree prefers a larger node size.

Another observation with Figure 7a is that the average latency for insertion of Circ-Tree_ls is similar to that of Circ-Tree. The reason is that the two only differ in searching but shifting and writing KV pairs to NVM is the dominant factor affecting the insertion performance.

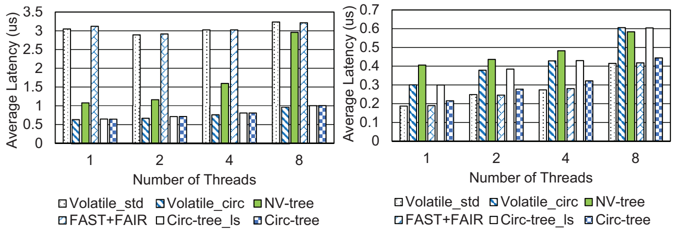
Figure 7d captures the average latency of seven trees in searching for one million inserted KV pairs with four node sizes. Owing to unsorted KV pairs in LNs, NV-tree yields the worst search performance. The average latency of Circ-Tree is shorter than Volatile_circ and Circ-Tree_ls because the former only searches in a contiguous memory space and avoids jumping between disjointed cache lines.

Moreover, with smaller node sizes, Circ-Tree is slower than Volatile_std and FAST+FAIR. A smaller node size helps



(a) Average latency (insert, μs) (b) Average latency (insert, μs)

Fig. 8: A Comparison of Six Trees on (a) Inserting 1/10/100 Million KV Pairs, and (b) Three NVM Types



(a) Average latency (insert, μs) (b) Average latency (search, μs)

Fig. 9: A Comparison of Six Trees with Multi-threading

them to maintain cache efficiency by prefetching. However, searching a smaller node or a part of it by Circ-Tree does not differ much. This explains with 512B node, even the average latency of NV-tree is close to that of Circ-Tree. Circ-Tree calls conditional branches to speculate about whether KV pairs are contiguous and which segment it should search. The cost of such branches offsets the gain introduced by searching a contiguous part of a node. However, with a larger node size, e.g., 4KB, the average latency for search of Circ-Tree is 7.1% shorter than that of Volatile_std and FAST+FAIR.

6.3 Impact of Workload and NVM

We also tested inserting ten million and 100 millions keys in uniform distributions with 2KB node. Figure 8a captures the average latencies for six trees. Circ-Tree still achieved the highest performance with heavier workloads. Nonetheless, all trees have longer average latency when the workload increases. For example, from ten million to 100 million, the average latencies of FAST+FAIR and Circ-Tree increase by 6.5% and 29.5%, respectively. With increasing number of keys stored in a B+-tree variant, inserting new keys to the tree becomes more time-consuming. The reason for such increased overhead is twofold. First, more keys lead to the increase of the height of a B+-tree variant. Thus, the traversal time from the tree root to a target LN turns to be longer, because 1) more cache misses occur due to more nodes to be loaded and searched, and 2) more comparisons are performed to locate the appropriate node at the next levels. Secondly, splits that spread to upper levels may involve more INs.

We used 300ns as the default write latency for NVM in experiments. Researchers have used NVDIMM with the same write speed as DRAM [9], [13], and NVM with 200ns and 600ns write latencies [6], [13], [20]. We also considered these three configurations in enumlating NVM and did experiments of inserting one million keys with 2KB node. Figure 8b shows the average latencies of six trees with four NVM types. With slower NVM, the cost of writing data to NVM is higher, so the write amplifications caused by shifting KV pairs are more significant. This explains why the performance gaps between Circ-Tree and FAST+FAIR are 2.5×, 3.9×, 4.8×, and 5.5× with increasing write latencies.

6.4 Multi-threading Performance

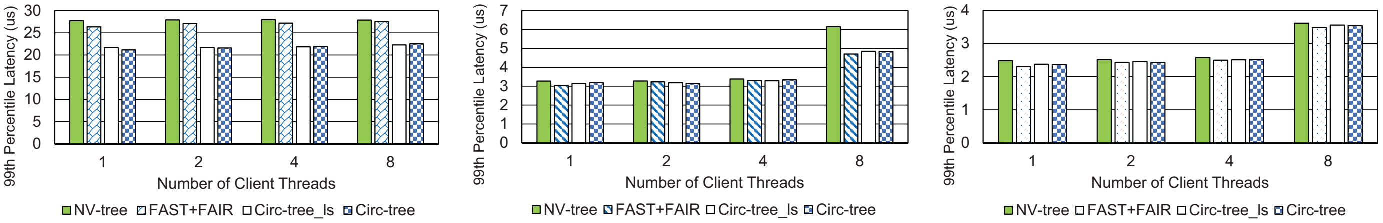
We have performed multi-threading tests to evaluate the performance of Circ-Tree. We varied the number of threads to be 1, 2, 4, and 8. We generated eight sets, each of which has one million keys in the uniform distribution. There are no duplicate keys across these sets. Each thread inserted and searched with a set. We used 2KB as the node size for every tree. Figure 9 shows the average latencies for insertion and search with varying the number of threads for six trees. Circ-Tree and Circ-Tree_ls still significantly outperform NV-tree and FAST+FAIR even in the presence of multiple threads. For example, with eight threads, the average latencies for insertion of NV-tree and FAST+FAIR are 2.9× and 3.2× that of Circ-Tree, respectively.

With more threads, B+-tree variants require more time to handle concurrent insertion requests. One reason is similar to the one mentioned for inserting 1/10/100 million keys, as more threads gradually make more KV pairs inserted. Moreover, when more threads concurrently insert KV pairs, the lock period due to operating with the same nodes becomes longer. Trees that are more efficient in the context of single-threading insertion, suffer more from a longer lock period (i.e. the duration a tree node might be locked for insertion or search). For instance, the average latencies for insertion of FAST+FAIR and Circ-Tree increased by 6.0% and 19.5%, respectively, when the number of operating threads increases from four to eight. Since Circ-Tree leverages bidirectional shifting to reduce write amplifications, the insertion performance of Circ-Tree is impaired more by the longer lock period. For NV-tree, however, its average latency almost doubled when the number of operating threads was increased from four to eight. Specifically, when more threads are inserting KV pairs, INs of NV-tree are more likely to become full. A full IN entails a rebuilding of all INs, which blocks all insertion threads. With more threads inserting data, NV-tree stalls more.

As shown in Figure 9b, more threads also lead to longer average latencies for search as they contend to access the same nodes. Though, both FAST+FAIR and Circ-Tree suffer from the longer lock period with more threads, so the performance gap between them decreases from 11.6% to 5.9% with four and eight threads, respectively. In addition, more threads also confirm the capability of Circ-Tree's search in a contiguous space. A comparison between Circ-Tree_ls and Circ-Tree with 8 threads shows that, the average latency for search of Circ-Tree is 36.1% shorter than that of Circ-Tree_ls.

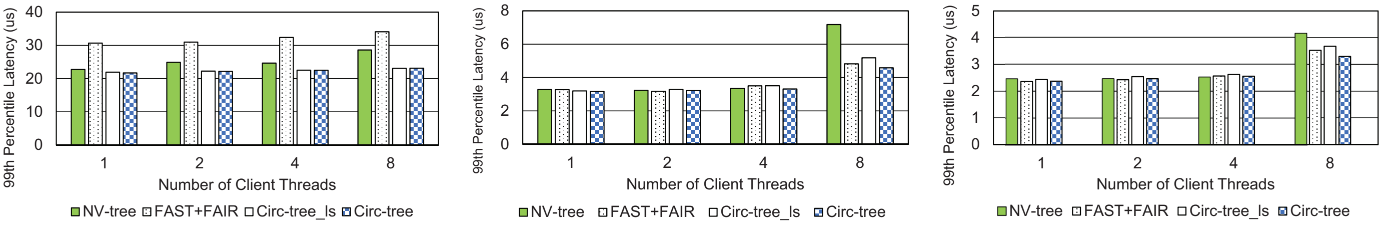
6.5 Recovery Time

We emulated inconsistency scenarios because the NVM used in evaluation was based on volatile DRAM. We first inserted 1/10/100 million keys. Then we orderly saved all



(a) 99th percentile latency (insertion, μs) (b) 99th percentile latency (search, μs) (c) 99th percentile latency (update, μs)

Fig. 10: A Comparison of Four KV Stores (2KB Node) on SessionStore Worload of YCSB



(a) 99th percentile latency (insertion, μs) (b) 99th percentile latency (search, μs) (c) 99th percentile latency (update, μs)

Fig. 11: A Comparison of Four KV Stores (4KB Node) on SessionStore Worload of YCSB

TABLE 1: The Recovery Time of Circ-Tree (unit: ms)

Number of KV Pairs Stored	Node Size			
	512B	1KB	2KB	4KB
1 Million	14.9	8.8	5.1	3.1
10 Million	154.1	91.2	54.0	34.4
100 Million	1,772.9	1,026.8	581.2	352.7

INs and LNs into a text file. When saving nodes into the file, we randomly selected some nodes and either decreased the number of keys by one or shifted KV pairs to generate duplicate non-NULL values. Next we reconstructed the tree by allocating, filling, and linking nodes. In the final step, we called the recovery procedure. For each number of keys (1/10/100 million), we repeated the recovery test for five times, and all inconsistency issues were found and fixed. Table 1 shows the average recovery time calculated in geometric mean. With a smaller node size, the recovery spent more time because much more INs were processed. With the 4KB node, the recovery time of Circ-Tree for 100 million KV pairs is just 352.7ms (0.35 second) with NVM having the same read latency as DRAM. We believe this recovery time is acceptable in practice.

6.6 End-to-End Comparison with YCSB

We built four KV stores using Circ-Tree_ls, Circ-Tree, and FAST+FAIR for indexing. We configured the number of client threads to be 1, 2, 4, and 8, and tested with two node sizes, i.e., 2KB and 4KB, by loading and running YCSB's SessionStore workload. Figure 10 and Figure 11 capture the 99th percentile latency with two node sizes, respectively.

As to the 99th percentile latencies shown in Figure 10a and Figure 11a for inserting with varied client threads, the differences among B+-tree variants are not as significant as those with standalone B+-trees. The reason is that, committing 1000B value to NVM per insertion costs much longer time than inserting a KV pair of 8B/8B into an indexing tree and the former dominates the insertion latency. Though, Circ-Tree still yields higher performance than NV-tree and FAST+FAIR. With 2KB node, Circ-Tree achieved 29.3% and 25.4% shorter latency than NV-tree and FAST+FAIR, respectively, with two threads. With 4KB node, Circ-Tree achieved 23.7% and 47.4% shorter latency than NV-tree and FAST+FAIR, respectively, with eight threads.

A comparison between Figure 10a and Figure 11a also indicates that, with a larger node, 1) the latency of NV-tree decreases, 2) the latency of FAST+FAIR increases, and 3) the latencies of Circ-Tree_ls and Circ-Tree remain consistent. Such an observation matches the observation we obtain with standalone B+-tree variants with larger nodes. In the meantime, the impact of more threads is not as much as what we have seen with Figure 9a, because the long duration of committing 1000B value per insertion overtakes the time incurred by contentions due to multi-threading.

Figure 10b and Figure 10c (resp. Figure 11b and Figure 11c) capture the 99th percentile latencies for searching and updating, respectively, with 2KB node (resp. with 4KB node). Search and update are both searching a B+-tree variant with additional read and write operations with NVM, respectively. We can obtain three observations from these four diagrams. First, in-NVM B+-tree variants yield comparable performances except NV-tree handling search requests issued from eight threads. Secondly, as reading 1000B is less heavyweight than writing 100B to NVM, the latencies in Figure 10b (resp. Figure 11b) are longer than those in Figure 10c (resp. Figure 11c). Thirdly, comparing these diagrams to Figure 9b (i.e., searching standalone B+-tree variants with multi-threading workload), the additional costs of reading and writing data with NVM have minimized the performance differences among trees.

The preceding results with YCSB confirm the practical usability of Circ-Tree in real-world applications.

7 Conclusion

The next-generation byte-addressable NVM has emerged as the persistent memory to revolutionize computer systems. In this paper, we consider a fundamental change in the design of B+-tree, i.e. to logically view its linear node structure in a circular fashion in the context of persistent memory. We have built Circ-Tree with the concept of circular node structure and designed insertion, deletion and search strategies suited for the circular node.

We have evaluated Circ-Tree with extensive experiments. Evaluation results show that Circ-Tree significantly outperforms state-of-the-art NV-tree and FAST+FAIR by up to 1.6× and 8.6×, respectively. An end-to-end comparison with YCSB workload onto KV store systems based on Circ-Tree, NV-tree, and FAST+FAIR shows that the write latency of Circ-Tree is up to 29.3% and 47.4% shorter than that of NV-tree and FAST+FAIR, respectively.

Acknowledgments

This work is partly supported by the Ministry of Education of Singapore under the grant MOE2018-T2-1-098. C. Wang's work is supported by the Startup Funding of ShanghaiTech University, China.

References

[1] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 421–432, New York, NY, USA, 2013. ACM.

[2] Iulian Moraru, David G. Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, TRIOS '13, pages 1:1–1:17, New York, NY, USA, 2013. ACM.

[3] Jian Xu and Steven 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)*, pages 323–338, Santa Clara, CA, February 2016. USENIX Association.

[4] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[5] Jiaxin Ou, Jiwu Shu, and Youyou Lu. A high performance file system for non-volatile main memory. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 12:1–12:16, New York, NY, USA, 2016. ACM.

[6] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. HiKV: A hybrid index key-value store for DRAM-NVM memory systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 349–362, Santa Clara, CA, July 2017. USENIX Association.

[7] S. Kannan et al. Redesigning LSMs for nonvolatile memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 993–1005, Boston, MA, 2018. USENIX Association.

[8] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 187–200, Oakland, CA, 2018. USENIX Association.

[9] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 167–181, Santa Clara, CA, 2015. USENIX Association.

[10] Youyou Lu, Jiwu Shu, Long Sun, and O. Mutlu. Loose-ordering consistency for persistent memory. In *Computer Design (ICCD), 2014 32nd IEEE International Conference on*, pages 216–223, Oct 2014.

[11] Ren-Shuo Liu, De-Yu Shen, Chia-Lin Yang, Shun-Chih Yu, and Cheng-Yuan Michael Wang. NVM Duet: Unified working memory and persistent store architecture. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 455–470, New York, NY, USA, 2014. ACM.

[12] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. Log-structured non-volatile main memory. In *Proceedings of the 2017 USENIX Conference on Annual Technical Conference*, USENIX ATC '17, pages 703–717, Santa Clara, CA, 2017. USENIX Association.

[13] Mingkai Dong and Haibo Chen. Soft updates made simple and fast on non-volatile memory. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 719–731, Santa Clara, CA, 2017. USENIX Association.

[14] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy 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*, FAST'11, pages 1–15, Berkeley, CA, USA, 2011. USENIX Association.

[15] Shimin Chen and Qin Jin. Persistent B+-trees in non-volatile main memory. *Proc. VLDB Endow.*, 8(7):786–797, February 2015.

[16] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A hybrid SCM-DRAM persistent and concurrent B-Tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 371–386, New York, NY, USA, 2016. ACM.

[17] Chundong Wang, Sudipta Chattopadhyay, and Gunavaran Brihadiswarn. Crash recoverable ARMv8-oriented B+-tree for byte-addressable persistent memory. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2019, pages 33–44, New York, NY, USA, 2019. ACM.

[18] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[19] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li. Quartz: A lightweight performance emulator for persistent memory software. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 37–49, New York, NY, USA, 2015. ACM.

[20] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 461–476, Carlsbad, CA, October 2018. USENIX Association.

[21] Intel. Intel 64 and IA-32 architectures software developer manuals. Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C and 3D, December 2016.

[22] Qingsong Wei, Chundong Wang, Cheng Chen, Yechao Yang, Jun Yang, and Mingdi Xue. Transactional nvm cache with high performance and crash consistency. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 56:1–56:12, New York, NY, USA, 2017. ACM.

[23] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency without ordering. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 101–116, Berkeley, CA, USA, 2012. USENIX Association.

[24] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *SOSP '09*, pages 133–146, New York, NY, USA, 2009. ACM.

[25] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 105–118, New York, NY, USA, 2011. ACM.

[26] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM.

[27] Eunji Lee, Hyokyung Bahn, and Sam H. Noh. Unioning of the buffer cache and journaling layers with non-volatile memory. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 73–80, San Jose, CA, 2013. USENIX.

[28] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 3–18, New York, NY, USA, 2015. ACM.

[29] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 672–685, New York, NY, USA, 2015. ACM.

[30] Amro Awad, Pratyusa Manadhata, Stuart Haber, Yan Solihin, and William Horne. Silent shredder: Zero-cost shredding for secure non-volatile main memory controllers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 263–276, New York, NY, USA, 2016. ACM.

[31] Cheng Chen, Jun Yang, Qingsong Wei, Chundong Wang, and Mingdi Xue. Fine-grained metadata journaling on NVM. In *MSST '16*, pages 1–12. IEEE, May 2016.

[32] Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 373–385, Berkeley, CA, USA, 2018. USENIX Association.

[33] Shimin Chen, Phillip B. Gibbons, and Suman Nath. Rethinking database algorithms for phase change memory. In *5th Biennial Conference on Innovative Data Systems Research (CIDR '11)*, January 2011.

[34] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. Efficient persist barriers for multicores. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 660–671, New York, NY, USA, 2015. ACM.

[35] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. Delegated persist ordering. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, pages 58:1–58:13, Piscataway, NJ, USA, 2016. IEEE Press.

[36] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence: A static analysis approach to automatic fence insertion. *ACM Trans. Program. Lang. Syst.*, 39(2):6:1–6:38, May 2017.

[37] Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, and Ke Yi. Tree indexing on solid state drives. *Proc. VLDB Endow.*, 3(1–2):1195–1206, September 2010.

[38] Michael A. Bender and Haodong Hu. An adaptive packed-memory array. *ACM Trans. Database Syst.*, 32(4):26–es, November 2007.

[39] Gilbert Laurenti, Karim Djafarian, and Herve Catan. Circular buffer management, March 2002. US Patent 6,363,470.

[40] Intel. Persistent memory development kit. http://pmem.io/pmdk/.

[41] J. Ahn, C. Seo, R. Mayuram, R. Yaseen, J. Kim, and S. Maeng. ForestDB: A fast key-value storage system for variable-length string keys. *IEEE Transactions on Computers*, 65(3):902–915, 2016.

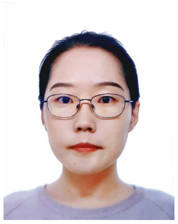
[42] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. NVWAL: Exploiting NVRAM in write-ahead logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 385–398, New York, NY, USA, 2016. ACM.

[43] Philip J. Fleming and John J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221, March 1986.

[44] Gernot Heiser. Systems benchmarking crimes, October 2018. https://www.cse.unsw.edu.au/ gernot/benchmarking-crimes.html.

Chundong Wang received the Bachelor's degree in computer science from Xi'an Jiaotong University in 2008, and the Ph.D. degree in computer science from National University of Singapore in 2013. Currently Chundong works in ShanghaiTech University as a tenure-track assistant professor. Before joining ShanghaiTech, he successively worked in Data Storage Institute, A*STAR, Singapore and Singapore University of Technology. He has published a number of papers in IEEE TC, ACM TOS, DAC, DATE, LCTES, USENIX ATC, USENIX FAST, etc. His research interests include data storage, non-volatile memory and computer architecture.

Gunavaran Brihadiswaran is a final-year undergraduate student at the Department of Computer Science and Engineering, University of Moratuwa, Sri Lanka. He completed a 6-month internship in Singapore University of Technology and Design in 2018. His research interests include bioinformatics and computational biology, parallel computing and machine learning.



Xingbin Jiang received the B.S. degree in Electronic and Information Engineering from University of Science and Technology Beijing, China, in 2012, and the M.S. degree in Materials Engineering from the National Center for Nanoscience and Technology, Chinese Academy of Sciences, in 2015. She is currently working as a Research Assistant at the Information Systems Technology and Design (ISTD) Pillar at Singapore University of Technology and Design (SUTD).

Her current research interests include IoT system security, network security, kernel security, and wireless security.

Sudipta Chattopadhyay received the Ph.D. degree in computer science from the National University of Singapore, Singapore, in 2013. He is an Assistant Professor with the Information Systems Technology and Design Pillar, Singapore University of Technology and Design, Singapore. In his doctoral dissertation, he researched on Execution-Time Predictability, focusing on Multicore Platforms. He seeks to understand the influence of execution platform on critical software properties, such as performance, energy, robustness, and security. His research interests include program analysis, embedded systems, and compilers.

Mr. Chattopadhyay serves in the review board of the IEEE Transactions on Software Engineering.