# Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree

Deukyeon Hwang and Wook-Hee Kim, *UNIST*; Youjip Won, *Hanyang University*; Beomseok Nam, *UNIST*

https://www.usenix.org/conference/fast18/presentation/hwang

This paper is included in the Proceedings of the 16th USENIX Conference on File and Storage Technologies.
February 12–15, 2018 • Oakland, CA, USA
ISBN 978-1-931971-42-3

# Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree

Deukyeon Hwang
*UNIST*

Wook-Hee Kim
*UNIST*

Youjip Won
*Hanyang University*

Beomseok Nam
*UNIST*

## Abstract

With the emergence of byte-addressable persistent memory (PM), a cache line, instead of a page, is expected to be the unit of data transfer between volatile and non-volatile devices, but the failure-atomicity of write operations is guaranteed in the granularity of 8 bytes rather than cache lines. This granularity mismatch problem has generated interest in redesigning block-based data structures such as B+-trees. However, various methods of modifying B+-trees for PM degrade the efficiency of B+-trees, and attempts have been made to use in-memory data structures for PM.

In this study, we develop Failure-Atomic ShifT (FAST) and Failure-Atomic In-place Rebalance (FAIR) algorithms to resolve the granularity mismatch problem. Every 8-byte store instruction used in the FAST and FAIR algorithms transforms a B+-tree into another consistent state or a *transient inconsistent* state that read operations can tolerate. By making read operations tolerate transient inconsistency, we can avoid expensive copy-on-write, logging, and even the necessity of read latches so that read transactions can be non-blocking. Our experimental results show that legacy B+-trees with FAST and FAIR schemes outperform the state-of-the-art persistent indexing structures by a large margin.

## 1 Introduction

Recent advances in byte-addressable persistent memories (PM), such as phase change memory[48], spin-transfer torque MRAM[17], and 3D Xpoint[2] have opened up new opportunities for the applications to warrant durability or persistency without relying on legacy heavy-duty interfaces offered by the filesystem and/or by the block device [21, 24].

In legacy block devices, B+-tree has been one of the most popular data structures. The primary advantage of a B+-tree is its efficient data access performance due to its high degree of node fan-out, a balanced tree height, and dynamic resizing. Besides, the large CPU cache in modern processors[11] enables B+-tree to exhibit a good cache line locality. As a result, B+-tree shows good performance as an in-memory data structure as well. Thus, the cache-conscious variants of B+-tree including CSS-tree [36], CSB-tree [37], and FAST [19] are shown to perform better than legacy binary counterparts such as T-tree [26].

Byte-addressable PM raises new challenges in using B+-tree because legacy B+-tree operations are implemented upon the assumption that block I/O is failure atomic. In modern computer architectures, the unit of atomicity guarantee and the unit of transfer do not coincide. The unit of atomicity in memory operations corresponds to a word, e.g. 64 bits whereas the unit of transfer between the CPU and memory corresponds to a cache line, e.g. 64 Bytes. This granularity mismatch is of no concern in current memory since it is volatile. When the memory is non-volatile, unexpected system failures may cause the result of incomplete cache flush operations to be externally visible after the system recovers. To make the granularity mismatch problem even worse, modern processors often make the most use of memory bandwidth by changing the order of memory operations. Besides, recently proposed memory persistency models such as *epoch persistency* [9] even allow cache lines to be written back out of order. To prevent the reordering of memory write operations and to ensure that the written data are flushed to PM without compromising the consistency of the tree structure, B+-trees for persistent memory use explicit memory fence operations and cache flush operations to control the order of memory writes [5, 42].

The recently proposed B+-tree variants such as NV-tree [49], FP-tree [34], and wB+-tree [5] have pointed out and addressed two problems of byte-addressable persistent B+-trees. The first problem is that a large number of fencing and cache flush operations are needed to maintain the sorted order of keys. And, the other problem is that the logging demanded by tree rebalancing operations is expensive in the PM environment.

To resolve the first problem, previous studies have proposed a way to update tree nodes in an *append-only* manner and introduced additional metadata to the B+-tree structures. With these augmentations, the size of updated memory region in a B+-tree node remains minimal. However, the additional metadata for indirect access to the keys, and the unordered entries affect the cache locality and increase the number of accessed cache lines, which can degrade search performance.

To resolve the second problem of persistent B+-trees - logging demanded by tree rebalancing operations, NVTree [49] and FP-tree [34] employed selective persistence that keeps leaf nodes in the PM but internal nodes in volatile DRAM. Although the selective persistence makes logging unnecessary, it requires the reconstruction of tree structures on system failures and makes the instant recovery impossible.

In this study, we revisit B+-trees and propose a novel approach to tolerating *transient inconsistency*, i.e., partially updated inconsistent tree status. This approach guarantees the failure atomicity of B+-tree operations without significant fencing or cache flush overhead. If transactions are made to tolerate the transient inconsistency, we do not need to maintain a consistent backup copy and expensive logging can be avoided. The key contributions of this work are as follows.

- We develop *Failure Atomic Shift (FAST)* and *Failure-Atomic In-place Rebalance (FAIR)* algorithms that transform a B+-tree through *endurable transient inconsistent states*. The endurable transient inconsistent state is the state in which read transactions can detect incomplete previous transactions and ignore inconsistency without hurting the correctness of query results. Given that all pointers in B+-tree nodes are unique, the existence of duplicate pointers enables the system to identify the state of the transaction at the time of crash and to recover the B+-tree to the consistent state without logging.
- We make read transactions non-blocking. If every store instruction transforms a B+-tree index to another state that guarantees correct search results, read transactions do not have to wait until concurrent write transactions finish changing the B+-tree nodes.

In the sense that the proposed scheme warrants consistency via properly ordering the individual operations and the inconsistency is handled by read operations, they share much of the same idea as the soft-update technique [12, 31] and *NoFS* [7].

The rest of the paper is organized as follows: In Section 2, we present the challenges in designing a B+-tree index for PM. In Section 3 and Section 4, we propose the FAST and FAIR algorithms. In Section 5, we discuss how to enable non-blocking read transactions. Section 6 evaluates the performance of the proposed persistent B+-tree and Section 7 discusses other research efforts. In Section 8 we conclude this paper.

## 2 B+-tree for Persistent Memory

### 2.1 Challenge: clflush and mfence

A popular solution to guarantee transaction consistency and data integrity is the copy-on-write technique, which updates the block in an out-of-place manner. The copy-on-write update for block-based data structures can be overly expensive because it duplicates entire blocks including the unmodified portion of the block.

The main challenges in employing the in-place update scheme in B+-trees is that we store multiple key-pointer entries in one node and that the entries must be stored in a sorted order. Inserting a new key-pointer entry in the middle of an array will shift on average half the entries. In the recent literature [42, 5, 49], this shift operation has been pointed out as the main reason why B+-trees call many cache line flush and memory fence instructions. If we do not guard the memory write operations with memory fence operations, the memory write operations can be reordered in modern processors. And, if we do not flush the cache lines properly, B+-tree nodes can be updated partially in PM because some cache lines will stay in CPU caches. To avoid such a large number of cache line flushes and memory fence operations, append-only update strategy can be employed [49, 34, 5]. However, the append-only update strategy improves the write performance at the cost of a higher read overhead because it requires additional metadata or all unsorted keys in a tree node to be read [6].

### 2.2 Reordering Memory Accesses

The reordering of memory write operations helps better utilize the memory bandwidth [38]. In the last few decades, the performance of modern processors has improved at a much faster rate than that of memory [10]. In order to resolve the performance gap between CPU speed and memory access time, memory is divided into multiple cache banks so that the cache lines in the banks can be accessed in parallel. As a result, memory operations can be executed out of order.

To design a failure-atomic data structure for PM, we need to consider both volatile memory order and persist order. Let us examine volatile memory order first. Memory reordering behaviors vary across architectures and memory ordering models. Some architectures such as ARM allow store instructions to be reordered with each other[8], while other architectures such as x86 prevent stores-after-stores from being reordered [40, 18], that is, *total store ordering* is guaranteed. However, most architectures arbitrarily reorder stores-after-loads, loads-after-stores, and loads-after-loads unless dependencies exist in them. The Alpha is known to be the only processor that reorders dependent loads [30]. Given that the Alpha processor has deprecated since 2003, we consider that all modern processors do not reorder dependent loads.

Memory persistency [35] is a framework that provides an interface for enforcing persist ordering constraints on PM writes. Persist order in the *strict persistency* model matches the memory order specified in the memory consistency model. However, the memory persistency model may allow the persist order to deviate from the volatile order under the *relaxed persistency* model [9, 35]. To simplify our discussion, we first present algorithms assuming the strict persistency model. Later, in Section 7, we will discuss the relaxed persistency model.

## 3 Failure-Atomic ShifT (FAST)

### 3.1 Shift and Memory Ordering

In most architectures, the order of memory access operations is not changed arbitrarily if stores and loads have dependencies. Based on this observation, we propose the *Failure-Atomic ShifT* (FAST) scheme for B+-tree nodes. FAST frees the shift operation from explicitly calling a memory fence and a cache line flush instruction without compromising the ordering requirement. The idea of FAST is simple and straightforward. Let us first examine the case where total store ordering (TSO) is guaranteed.

The process of shifting array elements is a sequence of load and store instructions that are all dependent and must be called in a cascading manner. To insert an element in a sorted array, we visit the array elements in reverse order, that is, from the rightmost element to the left ones. Until we find an element $e[i]$ that is smaller than the element we insert, we shift the elements to the right: $e[j+1] \leftarrow e[j], j = N, \ldots, i+1$. The dependency between the consecutive move operations, $e[i+1] \leftarrow e[i]$ and $e[i] \leftarrow e[i-1]$, prohibits the CPU from performing Out-Of-Order writes[38] and guarantees that the records are moved while satisfying the *prefix constraint* [45], i.e., if $e[i] \leftarrow e[i-1]$ is successful, all preceding moves, $e[j] \leftarrow e[j-1], j = i+1, \ldots, N$ have been successful.

If the array size spans across multiple cache lines, it is uncertain which cache line will be flushed first if we do not explicitly call `clflush`. Therefore, our proposed FAST algorithm calls a cache line flush instruction when we shift an array element from one cache line to its adjacent cache line. Since FAST calls cache line flushes only when it crosses the boundary of cache lines, it calls cache line flush instructions only as many times as the number of dirty cache lines in a B+-tree node.

Even if we do not call a cache line flush instruction, a dirty cache line can be evicted from CPU caches and be flushed to PM. In FAST, such a premature cache line flush does not affect the consistency of a sorted array. The only condition that FAST requires is that the dirty cache lines must be flushed in order.

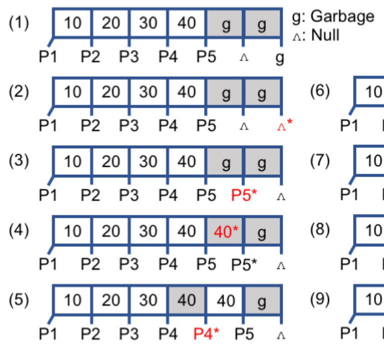


Figure 1: *FAST Inserting (25, Ptr) into B-tree node*

Suppose that an array element is shifted from one cache line to another. Since there exists a dependency between the load and the store instructions, they will not be reordered and we do not need to explicitly call a memory fence instruction. However, since cache line flush instructions can be reordered with store instructions, we call a memory fence instruction per cache line flush.

### 3.2 Endurable Inconsistency during Shift

The shift operation is not failure-atomic by itself because shifting an element can make it appear twice in a row. In case of a system failure, such *transient* duplicate elements can persist as shown in Figure 1, which seems at first glance unacceptably inconsistent. To resolve this issue, we use one property of B+-tree nodes; *B+-tree nodes do not allow duplicate pointers*. Therefore, a key in between duplicate pointers found during tree traversals can be ignored by transactions and considered as a tolerable *transient inconsistent* state. After finding a key of interest in a sorted array, we check if its left and right child pointers have the same addresses. If so, transactions ignore the key and continue to read the next key. This modification in traversal algorithm requires only one more compare instruction per each tree node visit, which incurs negligible overhead to transactions.

### 3.3 Insertion with FAST for TSO

In B+-tree nodes, the same number of keys and pointers (or keys and values) need to be shifted in tandem. We store keys and pointers as an array of structure in B+-tree nodes so that the corresponding keys and pointers are always flushed together.

The insertion algorithm is shown in Algorithm 1. Consider the example shown in Figure 1. We insert a key-value pair $(25, Ptr)$ into the B+-tree node shown in (1) of Figure 1. To make a space for 25, the two rightmost

**Algorithm 1**

*FAST_insert(node, key, ptr)*

```
$node.lock.acquire()$
if $(sibling \leftarrow node.sibling\_ptr) \neq NULL$ then
    if $sibling.records[0].key < key$ then
        – previous write thread has split this node
        $node.lock.release()$
        $FAST\_insert(sibling, key, ptr)$;
        return
    end if
end if
if $node.cnt < node\_capacity$ then
    if node.search_dir_flag is odd then
        – if this node was updated by a delete thread, we
        – increase this flag to make it even so that
        – lock-free search scans from left to right
        node.search_dir_flag++;
    end if
    for $i \leftarrow node.cnt - 1; i \geq 0; i--$ do
        if $node.records[i].key > key$ then
            $node.records[i+1].ptr \leftarrow node.records[i].ptr$;
            $mfence\_IF\_NOT\_TSO()$;
            $node.records[i+1].key \leftarrow node.records[i].key$;
            $mfence\_IF\_NOT\_TSO()$;
            if $\&(node.records[i+1])$ is at cacheline boundary
            then
                $clflush\_with\_mfence(\&node.records[i+1])$;
            end if
        else
            $node.records[i+1].ptr \leftarrow node.records[i].ptr$;
            $mfence\_IF\_NOT\_TSO()$;
            $node.records[i+1].key \leftarrow key$;
            $mfence\_IF\_NOT\_TSO()$;
            $node.records[i+1].ptr \leftarrow ptr$;
            $clflush\_with\_mfence(\&node.records[i+1])$;
        end if
    end for
    $node.lock.release()$
else
    $node.lock.release()$
    $FAIRsplit(node, key, ptr)$;
end if
```

keys, 30 and 40, their pointers $P4$ and $P5$, and the sentinel pointer $Null$ must be shifted to the right.

First, we shift the sentinel pointer $Null$ to the right as shown in (2). Next, we shift the right child pointer $P5$ of the last key 40 to the right, and then we shift key 40 to the right, as shown in (3) and (4). In step (3) and (4), we have a garbage key and a duplicate key 40 as the last key respectively. Such inconsistency can be tolerated by making other transactions *ignore the key between the same pointers* ($P5$ and $P5*$ in the example).

In step (5), we shift $P4$ by overwriting $P5$. This atomic write operation invalidates the old key 40 ($[P4, 40, P4*]$) and validates the shifted key 40 ($[P4*, 40, P5*]$). Even if a system crashes at this point, the key 40 between the redundant pointers ($[P4, 40, P4*]$) will be ignored. Next, the key 30 can be shifted to the right by overwriting the key 40 in step (6). Next, in step (7), we shift $P3$ to the right to invalidate the old key 30 ($[P3, 30, P3*]$) and make space for the key 25, that we insert. In step (8), we store 25 in the third slot. However, the key 25 is not valid because it is in between the same pointers ($[P3, 25, P3*]$). Finally, in step (9), we overwrite $P3*$ with the new pointer $Ptr$, which will validate the new key 25. In FAST, writing a new pointer behaves as a commit mark of an insertion.

Unlike pointers, the size of keys can vary. If a key size is greater than 8 bytes, it cannot be written atomically. However, our proposed FAST insertion makes changes to a key only if it is located between the same pointers. Therefore, even if a key size is larger than 8 bytes and even if it cannot be updated atomically, read operations ignore such partially written keys and guarantee correct search results.

In this sense, every single 8-byte write operation in the FAST insertion algorithm is failure-atomic and crash consistent because partially written tree nodes are always usable. Hence, even if a system crashes during the update, FAST guarantees recoverability as long as we flush dirty cache lines in an order.

## 3.4 FAST for Non-TSO Architectures

ARM processors can reorder store instructions if they do not have a dependency. I.e., if a store instruction updates a key and another store instruction updates a pointer, they can be reordered.

First, consider the case where keys are no larger than 8 bytes. Then, keys and pointers can be shifted independently because no key or pointer in either array can be in a partially updated status. Suppose that the $i$th and $i+1$th keys are the same and the $j$th and $j+1$th pointers are the same ($i \neq j$). We can simply ignore one of the redundant elements in both arrays, easily sort out which pointer is for which key's child node, and reconstruct the correct logical view of the B+-tree node.

Second, consider the case where the keys are larger than 8 bytes. One easy option is to call a memory fence instruction for each shift operation. Although this option calls a large number of memory fence instructions, it calls cache line flush instructions only as many times as the number of dirty cache lines as in TSO architectures. Although memory fence overhead is not negligible, it is much smaller than cache line flush overhead. Alternatively, we can make B+-tree store the large keys in a separate memory heap and store pointers to the keys in B+-tree nodes. This option allows us to avoid a large number of memory fence instructions. However, through experiments, we found the indirect access and the poor cache

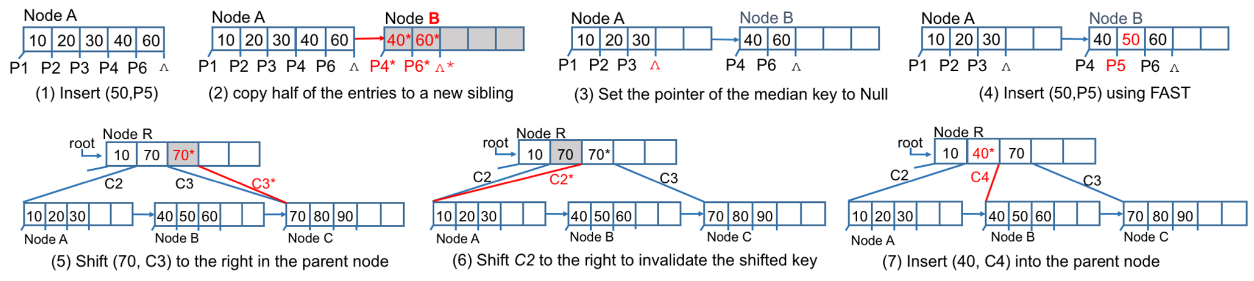

Node A
10 20 30 40 60
P1 P2 P3 P4 P6 ^
(1) Insert (50,P5)
Node A
Node B
40* 60*
P4* P6* ^*
(2) copy half of the entries to a new sibling
(3) Set the pointer of the median key to Null
50
P5
(4) Insert (50,P5) using FAST
root
Node R
10 70 70*
C2 C3 C3*
Node A Node B Node C
(5) Shift (70, C3) to the right in the parent node
C2*
(6) Shift C2 to the right to invalidate the shifted key
10 40* 70
C4
(7) Insert (40, C4) into the parent node

Figure 2: *FAIR Node Split*

locality significantly degrade the search performance.

## 3.5 Deletion with FAST

Deletion can be performed in a similar manner but in reverse order of insertion. Consider the deletion of $(25, Ptr)$ from the tree node illustrated in (9) of Figure 1. First, we overwrite $Ptr$ with $P3$ to invalidate the key 25. Note that this atomic write operation behaves as the commit mark of the deletion. If a system crashes and a subsequent query finds the B+-tree node as shown in (8), the key 25 ($[P3, 25, P3*]$) will be considered as an invalid key because it is in between duplicate pointers.

In the subsequent steps shown in (7)∼(3), the pointer array will have adjacent duplicate pointers that invalidate one of the keys while we shift them to the left. Finally, in step (2), we shift the sentinel pointer *Null* to the left, which completes the deletion process. Similar to the insertion, we flush dirty cache lines only when we are about to make the next cache line dirty.

## 4 Failure-Atomic In-place Rebalancing

## 4.1 FAIR: Node Split

In both legacy disk-based B-trees and recently proposed B+-tree variants for PM [49, 34, 5], logging or journaling has been used when splitting or merging multiple tree nodes. If a tree node splits, (1) we create a sibling node, (2) move half of the entries from the overfull node to the new sibling node, and (3) insert a pointer to the new sibling node and its smallest key to the parent node. Since these three steps must be performed atomically, logging can be used. However, logging duplicates dirty pages. It not only increases the write traffic but also blocks the concurrent access to tree nodes.

We avoid the expensive logging by leveraging the FAST in-place update scheme and the *Failure-Atomic In-place Rebalance* (FAIR) node split algorithm described in Algorithm 2. In our B+-tree design, we store a sibling pointer not just for leaf nodes but also for internal nodes as in the B-link tree [25].

Figure 2 illustrates each step of the FAIR node split algorithm with an example. We examine every individual store operations in splitting a tree node and carefully lay

**Algorithm 2**
*FAIR_split(node, key, ptr)*

```
1: $node.lock.acquire()$
2: if $(sibling \leftarrow node.sibling\_ptr) \neq NULL$ then
3:     if $sibling.records[0].key < key$ then
4:         $node.lock.release()$
5:         $FAST\_insert(sibling, key, ptr)$;
6:         return
7:     end if
8: end if
9: $sibling \leftarrow nv\_malloc(sizeof(node))$;
10: $median \leftarrow node\_capacity/2$
11: for $i \leftarrow median; i < node\_capacity; i++$ do
12:     $FAST\_insert\_without\_lock(sibling,$
        $node.records[i].key, node.records[i].ptr)$;
13: end for
14: $sibling.sibling\_ptr \leftarrow node.sibling\_ptr$;
15: $clflush\_with\_mfence(sibling)$;
16: $node.sibling\_ptr \leftarrow sibling$;
17: $clflush\_with\_mfence(\&node.sibling\_ptr)$;
18: $node.records[median].ptr \leftarrow NULL$;
19: $clflush\_with\_mfence(\&node.records[median])$;
20: – split is done. now insert (key,ptr)
21: if $key < node.records[median].key$ then
22:     $FAST\_insert\_without\_lock(node, key, ptr)$
23: else
24:     $FAST\_insert\_without\_lock(sibling, key, ptr)$
25: end if
26: $node.lock.release()$
27: – update the parent node by traversing from the root.
28: $FAST\_internal\_node\_insert(root, node.level+$
    $1, sibling.records[0].key, node.sibling\_ptr)$;
```

them out so that we do not have to maintain an additional copy of data for crash recovery and consistency is not compromised in the event of system failure.

First, suppose there is only one node in the tree as shown in (1). If we insert a new key 50, the leaf node $A$ will split. In (2), we create a sibling node $B$ using a PM heap manager [33], copy half of the entries, call cache line flushes for the new node, and make the sibling pointer of node $A$ point to the sibling node $B$. The sibling pointer must be written before we delete the migrated entries in the overfull node *A*. It looks as if the consistency of the tree nodes is violated because we have duplicate entries in nodes *A* and *B*; however, the right sibling node *B* will not be used until we delete duplicate entries from *A* because the smallest key in the right sibling node (40) is smaller than the largest key in the overfull node (60).

Suppose a system crashes in state (2). If a query that searches for key 35 is submitted, it will access node *A* following the search path from the root node because the sibling node *B* has not been added to the parent node yet. Since 35 is smaller than 40 (the one in node *A*), the search will stop without accessing node *B*. If the search key is 65, the query will follow the sibling pointer, access the sibling node *B*, and compare to see if the first key of the sibling node *B* is greater than the largest key in the node *A*. If it is, the query will keep checking the next key until it finds a matching key or a larger key than the search key. However, in this example, the search will stop after it finds that the first key 40 in the sibling node *B* is smaller than the largest key 60 in the node *A*. The basic assumption that makes this redundancy and inconsistency tolerable is that the right sibling nodes always have larger keys than the left sibling nodes. Therefore, the keys and pointers in the new right sibling node ($[P4*, 40*, P6*, 60*, Null]$) will not be accessed until we delete them in the overfull node *A*. We delete migrated keys and pointers from the overfull node by atomically setting the pointer of the median to NULL. This will change the tree status to (3). If we search the key 40 after deleting the migrated entries, the query will find that the largest key in node *A* is smaller than the search key, and follow the sibling pointer to find the key 40 in node *B*.

Figure 2 (5)∼(7) illustrate how we can endure transient inconsistency when we update the parent node. In step (5), node *A* and its sibling node *B* can be considered as a single virtual node. In the parent node *R*, we add a key-pointer pair for node *B*, i.e., $(40, C4)$ in the example. Since the parent node has keys greater than the key 40 that we add, we shift them using FAST as shown in (5) and (6). After we create a space in the parent node by duplicating the pointer *C*2, we finally store $(40, C4)$ as shown in (7).

### 4.2 FAIR: Node Merge

In B+-tree, underutilized nodes are merged with sibling nodes. If a deletion causes a node to be underutilized, we delete the underutilized node from its parent node so that the underutilized node and its left sibling node become virtually a single node. Once we detach the underutilized node from the parent node, we check whether the underutilized node and its left sibling node can be merged. If the left sibling node does not have enough entries, we merge them using FAST. If the left sibling node has enough entries, we shift some entries from the left sibling node to the underutilized node using FAST, and insert the new smallest key of the right sibling node to its parent node. This node merge algorithm is similar to the split algorithm, but it is performed in the reverse order of the split algorithm. Thus, a working example of the node merge algorithm can be illustrated by reversing the order of the steps shown in Figure 2.

## 5 Lock-Free Search

With the growing prevalence of many-core systems, concurrent data structures are becoming increasingly important. In the recently proposed *memory driven computing* environment that consists of a large number of processors accessing a persistent memory pool [1], concurrent accesses to shared data structures including B+-tree indexes must be managed efficiently in a thread-safe way.

As described earlier, the sequences of 8 byte store operations in FAST and FAIR algorithms guarantee that no read thread will ever access inconsistent tree nodes even if a write thread fails while making changes to the B+-tree nodes. In other words, even if a read transaction accesses a B+-tree partially updated by a suspended write thread, it is guaranteed that the read transaction will return the correct results. On the contrary, read transactions can be suspended and a write transaction can make changes to the B+-tree that the read transactions were accessing. When the read transactions wake up, they need to detect and tolerate the updates made by the write transaction. In our implementation, tree structure modifications, such as page splits or merges, can be handled by the concurrency protocol of the B-link tree [25]. However, the B-link tree has to acquire an exclusive lock to update a single tree node atomically. However, leveraging the FAST algorithm, we can design a non-blocking lock-free search algorithm to allow concurrent accesses to the same tree node as described below.

To enable a lock-free search, all queries must access a tree node in the same direction. That is, while a write thread is shifting keys and pointers to the right, read threads must access the node from left to right. If a write thread is deleting an entry using left shifts, read threads must access the node from right to left. Suppose the following example. A query accesses the tree node shown in Figure 1(1) to search for key 22. After the query reads the first two keys - 10 and 20, it is then suspended before accessing the next key 30. While the query thread is suspended, a write thread inserts 25 and shifts keys and pointers to the right. When the suspended query thread wakes up later, the tree node may be different from what it was before. If the tree node is in one of the states (1)∼(6), or (9), the read thread will follow the child pointer *P*3 without a problem. If the tree node is in state (7) or (8), the read thread will find the left and right child pointers are the same. Then, it will ignore the current key

**Algorithm 3**
*LockFreeSearch(node, key)*

```
repeat
    $ret \leftarrow NULL$;
    $prev\_switch \leftarrow node.switch$;
    if $prev\_switch$ is even then
        – we scan this node from left to right
        for $i \leftarrow 0; records[i].ptr \neq NULL; i++$ do
            if $(k \leftarrow records[i].key) = key \&\& records[i].ptr \neq (t \leftarrow records[i+1].ptr)$ then
                if $(k = records[i].key)$ then
                    $ret \leftarrow t$;
                    break;
                end if
            end if
        end for
    else
        – this node was accessed by a delete query
        – we have to scan this node from right to left
        for $i \leftarrow node.cnt - 1; i >= 0; i--$ do
            – omitted due to symmetry and lack of space
        end for
    end if
until $prev\_switch = node.switch$
if $ret = NULL$ && $(t \leftarrow node.sibling\_ptr)$ then
    if $t.records[0].key \leq key$ then
        return $t$;
    end if
end if
return $ret$;
```

and move on to the next key so that it follows the correct child pointer. From this example, we can see that shifting keys and pointers in the same direction does not hurt the correctness of concurrent non-blocking read queries.

If a read thread scans an array from left to right while a write thread is deleting its element by shifting to the left, there is a possibility that the read thread will miss the shifted entries. But if we shift keys and pointers in the same direction, the suspended read thread cannot miss any array element even if it may encounter the same entry multiple times. However, this does not hurt the correctness of the search results.

To guide which direction read threads scan a tree node, we use a counter flag which increases when insertions and deletions take turn. That is, the flag is an even number if a tree node has been updated by insertions and an odd number if the node has been updated by deletions. Search queries determine in which direction it scans the node according to the flag, and it double checks whether the counter flag remains unchanged after the scan. If the flag has changed, the search query must scan the node once again. A pseudo code of this lock-free search algorithm is shown in Algorithm 3.

Because of the restriction on the search direction, our lock-free search algorithm cannot employ a binary search. Although the binary search is known to perform faster than the linear search, its performance can be lower than that of the linear search when the array size is small, as we will show in Section 6.2.

## 5.1 Lock-Free Search Consistency Model

A drawback of a lock-free search algorithm is that a deterministic ordering of transactions cannot be enforced. Suppose a write transaction inserts two keys - 10 and 20 into a tree node and another transaction performs a range query and accesses the tree node concurrently. The range query may find 10 in the tree node but may not find 20 if it has not been stored yet. This problem can occur because the lock-free search does not serialize the two transactions. Although the lock-free search algorithm helps improve the concurrency level, it is vulnerable to the well-known *phantom reads* and *dirty reads* problems [41].

In the database community, various levels of isolation such as *serializable mode*, *non-repeatable read mode*, *read committed mode*, and *read uncommitted mode* [41] have been studied and used. Considering our lock-free search algorithm is vulnerable to dirty reads, it operates in read uncommitted mode. Although read uncommitted mode can be used for certain types of queries in OLAP systems, the scope of its usability is very limited.

To support higher isolation levels, we must resort to other concurrency control methods such as key range locks, snapshot isolation, or multi-version schemes [41]. However, these concurrency control methods may impose a significant overhead. To achieve both a high concurrency level and a high isolation level, we designed an alternative method to compromise. That is, we use read locks only for leaf nodes considering the commit operations only occur in leaf nodes. For internal tree nodes, read transactions do not use locks since they are irrelevant to phantom reads and dirty reads. Since transactions are likely to conflict more in internal tree nodes rather than in leaf nodes, read locks in leaf nodes barely affect the concurrency level as we will show in Section 6.7.

## 5.2 Lazy Recovery for Lock-Free Search

Since the reconstruction of a consistent logical view of a B+-tree is always possible with an inconsistent but correctable B+-tree, we perform a recovery in a lazy manner. We do not let read transactions fix tolerable inconsistency because read transactions are more latency sensitive than write transactions. In our design, instead, we make only write threads fix tolerable inconsistencies. Such a lazy recovery approach is acceptable because FAST allows at most one pair of duplicate pointers in each node. Thus, it does not waste a significant amount of memory space, and its performance impact on search is negligible. Besides, the lazy recovery is necessary for a lock-free search. In lock-free searches, read threads and a write thread can access the same node. If read threads must fix the duplicate entries that a write thread caused, read threads will compete for an exclusive write lock. Otherwise, read threads have to check whether the node is inconsistent due to a crash or due to an inflight insert or delete, which will introduce significant complexities and latency to reads.

To fix inconsistent tree nodes, we delete the garbage key in between duplicate pointers by shifting the array to the left. For a dangling sibling node, we check if the sibling node can be merged with its left node. If not, we insert the pointer to the sibling node into the parent node.

In the FAIR scheme, the role of adding a sibling node to the parent node is not confined to the query that created the sibling node. Even if a process that split a node crashes for some reason, a subsequent process that accesses the sibling node via the sibling pointer triggers a parent node update. If multiple write queries visit a tree node via the sibling pointer, only one of them will succeed in updating the parent node and the rest of the queries will find that the parent has already been updated.

## 6 Experiments

We evaluate the performance of FAST and FAIR[1] against the state-of-the-art indexing structures for PM - wB+-tree with *slot+bitmap nodes* [5], FP-tree [34], WORT [23], and Skiplist [16].

**wB+-tree** [5] is a B+-tree variant that stores all tree nodes in PM. wB+-tree inserts data in an append-only manner. To keep the ordering of appended keys, wB+-tree employs a small metadata, called *slot-array*, which manages the index of unsorted keys in a sorted order. In order to atomically update the slot-array, wB+-tree uses a separate bitmap to mark the validness of array elements and slot-array. Because of these metadata, wB+-tree calls at least four cache line flushes when we insert data into a tree node. Besides this, wB+-tree also requires expensive logging and a large number of cache line flushes when nodes split or merge.

**FP-Tree** [34] is a variant of a B+-tree that stores leaf nodes in PM but internal nodes in DRAM. It proposes to reduce the CPU cache miss ratio via *finger printing* and employs hardware transactional memory to control concurrent access to internal tree nodes in DRAM. FP-Tree claims that internal nodes can be reconstructed without significant overhead. However, the reconstruction of internal nodes is not very different from the reconstruction of the whole index. We believe one of the most important benefits of using PM is the instantaneous recoverability. With FP-Tree, such an instantaneous recovery is impossible. Thus, strictly speaking, FP-Tree is not a persistent index.

**WORT** [23] is an alternative index for PM based upon a radix tree. Unlike B+-tree variants, the radix tree does not require key sorting nor rebalancing tree structures. Since the radix tree structure is deterministic, insertion or deletion of a key requires only a few 8 byte write operations. However, radix trees are sensitive to the key distribution and often suffers from poor cache utilization due to their deterministic tree structures. Also, radix trees are not as versatile as B+-trees as their range query performance is very poor.

**SkipList** [16] is a probabilistic indexing structure that avoids expensive rebalancing operations. An update of the skip list needs pointer updates in multi-level linked lists. But only the lowest level-linked list needs to be updated in a failure-atomic way. In a Log-Structured NVMM System [16], SkipList was used as a volatile address mapping tree, but SkipList shares the same goal with our B+-tree. That is, both indexing structures do not need logging, enable lock-free search, and benefit from a high degree of parallelism.

### 6.1 Experimental Environment

We run experiments on a workstation that has two Intel Xeon Haswell-Ex E7-4809 v3 processors (2.0 GHz, 16 vCPUs with hyper-threading enabled, and 20 MB L3 cache) that guarantee total store ordering and 64 GB of DDR3 DRAM. We compiled all implementations using g++ 4.8.2 with -O3 option.

We use a DRAM-based PM latency emulator - *Quartz*[43] as was done in [44, 3, 20]. Quartz models application-perceived PM latency by inserting stall cycles in each predefined time interval called *epoch*. In our experiments, the minimum and maximum epochs are set to 5 nsec and 10 nsec respectively. We assume that PM bandwidth is the same as that of DRAM since Quartz does not allow us to emulate both latency and bandwidth at the same time.

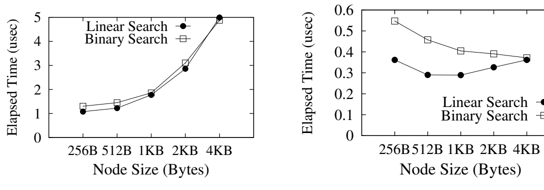
### 6.2 Linear Search vs. Binary Search

In the first set of experiments shown in Figure 3, we index 1 million random key-value pairs of 8 bytes each and evaluate the performance of the linear search and the binary search with varying size of B+-tree nodes. We assume the latency of PM is the same as that of DRAM.

As we increase the size of the tree nodes, the tree height decreases in log scale but the number of data to be shifted by the FAST algorithm in each node linearly increases. As a result, Figure 3(a) shows the insertion performance degrades with larger tree node sizes.

Regarding search performance, Figure 3(b) shows the binary search performance improves as we increase the tree node size because of the smaller tree height and the

[1]Codes are available at https://github.com/DICL/FAST_FAIR



(a) Insertion Time (b) Search Time

Figure 3: *Linear vs. Binary Search*

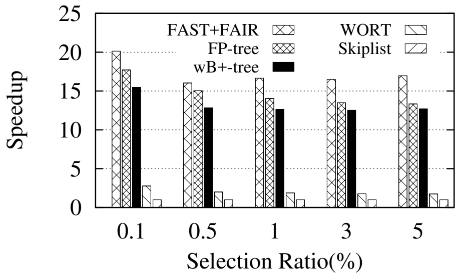


Figure 4: *Range Query Speed-Up from using SkipList with Varying Selection Ratio (AVG. of 5 Runs)*

fewer number of key comparisons. However, the binary search often stalls due to poor cache locality and the failed branch prediction. As a result, it performs slower than the linear search when the tree node size is smaller than 4 KB. Although the linear search accesses more cache lines and incurs more LLC cache misses, memory-level parallelism (MLP) helps read multiple cache lines at the same time. As a result, the number of effective LLC cache misses of the linear search is smaller than that of the binary search.

Overall, our B+-tree implementation shows good insertion and search performance when the tree node size is 512 bytes and 1 KB. Hence, we set the B+-tree node size to 512 bytes for the rest of the experiments unless stated otherwise. Note that the 512-byte tree node size occupies only eight cache lines, thus FAST requires eight cache line flushes in the worst case and four cache line flushes on average. Since the binary search makes lock-free search impossible and the linear search performs faster than binary search when the node size is small, we use the linear search for the rest of the experiments.

## 6.3 Range Query

A major reason to use B+trees instead of hash tables is to allow range queries. Hash indexes are known to perform well for exact match queries, but they cannot be used if a query involves the ordering of data such as *ORDER BY* sorting and *MIN/MAX* aggregation. Thus many commercial database systems use a hash index as a supplementary index of the B+-tree index.

In the experiments shown in Figure 4, we show the relative performance improvement over SkipList for range query performance with various selection ratios. The selection ratio is the proportion of selected data to the number of data in an index. We set the read latency of PM to 300 nsec and inserted 10 million 8-byte random integer keys into 1 KB tree nodes. A B+-tree with FAST and FAIR processes range queries up to 20 times faster than SkipList and consistently outperforms other persistent indexing structures (6∼27% faster than FP-tree and 25∼33% faster than wB+-tree) due to its simple structure and sorted keys.

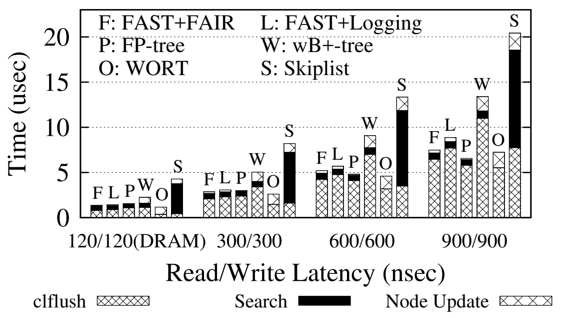


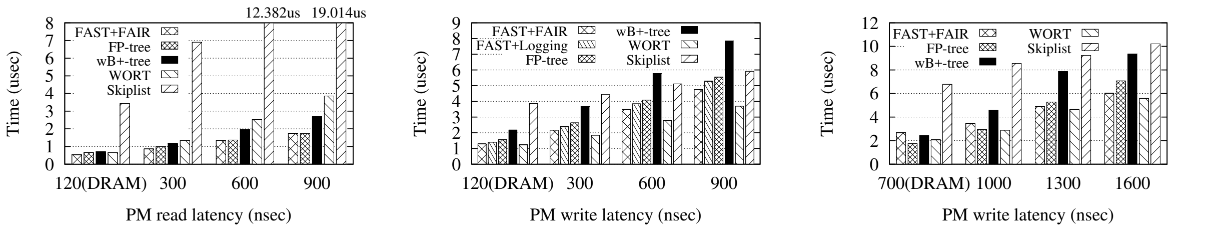
Figure 5: *Breakdown of Time Spent for B-tree Insertion (AVG. of 5 Runs)*

## 6.4 PM Latency Effect

In the experiment shown in Figure 5 and 6, we index 10 million 8 byte key-value pairs in uniform distribution and measure the average time spent per query while increasing the read and write latency of PM from 300 nsec. For FP-tree, we set the size of the leaf nodes and internal nodes to 1 KB and 4 KB respectively as was done in [34]. The node size of wB+-tree is fixed at 1 KB because each node can hold no more than 64 entries. Both configurations are the fastest performance settings for FP-tree and wB+-tree [34].

Figure 5 shows that FAST+FAIR, FP-tree, and WORT show comparable insertion performances and they outperform wB+-tree and SkipList by a large margin. In detail, the insertion time is composed of three parts, *Cache line flush*, *Search*, and *Node Update* times. wB+-tree calls a 1.7 times larger number of cache line flushes than FAST+FAIR. We also measured the performance of a FAST-only B+-tree with legacy tree rebalancing operations that have a logging overhead. Because of the logging overhead, FAST+Logging performs 7∼18% slower than FAST+FAIR.

The poor performance of SkipList is because of its poor cache locality. Without clustering similar keys in contiguous memory space and exploiting the cache locality, byte-addressable in-memory data structures such as SkipList, radix trees, and binary trees fail to lever-



(a) Search : Varying Read Latency (b) Insert : Varying Write Latency on TSO Architecture (c) Insert : Varying Write Latency on Non-TSO Architecture

Figure 6: *Performance Comparison of Single-Threaded Index (AVG. of 5 Runs)*

age memory level parallelism. Hence, block-based data structures such as B+-trees that benefit from clustered keys need to be considered not only for block device storage systems but also for byte-addressable PM. FP-tree benefits from faster access to internal nodes than FAST+FAIR and WORT, but it calls a slightly larger number of cache line flushes than FAST+FAIR (4.8 v.s 4.2) because of fingerprints and leaf-level logging.

Figure 6(a) shows how the read latency of PM affects the *exact match query* performance. FP-tree shows a slightly faster search performance than FAST+FAIR when the read latency is higher than 600 nsec because it has faster access to volatile internal nodes. When the read latency is 900 nsec, WORT spends twice as much time as FAST+FIAR to search the index. Interestingly, the average number of LLC cache misses of WORT that we measured with *perf* is 4.9 while that of FAST+FAIR is 8.3 in the experiments. Although B+-tree variants have a larger number of LLC cache misses than WORT, mostly they access adjacent cache lines and benefit from serial memory accesses, i.e., hardware prefetcher and memory level parallelism. To reflect the characteristics of modern hardware prefetcher and memory-level parallelism and to avoid overestimation of the overhead of serial memory accesses, Quartz counts the number of memory stall cycles for each LOAD request and divides it by the memory latency to count the number of serial memory accesses and estimate the appropriate read latency for them [43]. Therefore, the search performances of B+-tree variants are less affected by the increased read latency of PM compared to WORT and SkipList.

In the experiments shown in Figure 6(b), we measure the average insertion time for the same batch insertion workload while increasing only the write latency of PM. As we increase the write latency, WORT, which calls fewer cache line flushes than FAST+FAIR, outperforms all other indexes because the number of cache line flushes becomes a dominant performance factor and the poor cache locality of WORT gives less impact on the performance. FAST+FAIR consistently outperforms FP-tree, wB+-tree, and Skip List as it calls a lower number of `clflush` instructions.

## 6.5 Performance on Non-TSO

Although stores-after-stores are not reordered in X86 architectures, ARM processors do not preserve total store ordering. To evaluate that the performance of FAST on non-TSO architectures, we add `dmb` instruction as a memory barrier to enforce the order of store instructions and measure the insertion performance of FAST and FAIR on a smartphone, *Nexus 5*, which has a 2.26 GHz Snapdragon 800 processor and 2 GB DDR memory. To emulate PM, we assume that a particular address range of DRAM is PM and the read latency of PM is no different from that of DRAM. We emulate the write latency of PM by injecting `nop` instructions. Since the word size of Snapdragon 800 processor is 4 bytes, the granularity of failure-atomic writes is 4 bytes, and the node size of wB+-tree and FP-tree is limited to 256 bytes accordingly. Figure 6(c) shows that, when the PM latency is the same as that of DRAM, our proposed FAST+FAIR shows worse performance than FP-tree although FP-tree calls more cache line flush instructions. This is because FAST+FAIR calls more memory barrier instructions than FP-tree on ARM processors (16.2 vs. 6.6). However, as the write latency of PM increases, FAST+FAIR outperforms other indexes because the relative overhead of `dmb` becomes less significant compared to that of the cache line flushes (`dccmvac`). In our experiments, the performance of FAST+FAIR is up to 1.61 times faster than wB+-tree.

## 6.6 TPC-C Benchmark

In real-world applications, workloads are often mixed with writes and reads. In the experiments shown in Figure 7, we evaluate the performance using TPC-C benchmark. TPC-C benchmark is an OLTP benchmark that consists of 5 different types of queries (New-Order, Payment, Order-Status, Delivery, and Stock-Level). We varied the percentage of these queries to generate four different workloads so that the proportion of search queries increases from W1 to W4. The read and write latency of PM are both set to 300 nsec. FAST+FAIR consistently outperforms other indexes because of its good insertion performance and superior range query performance due

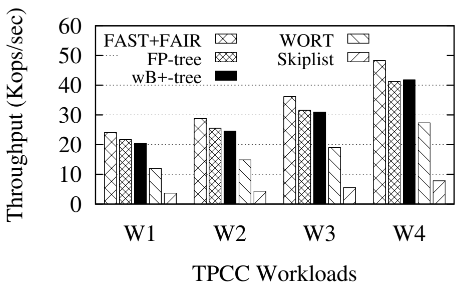


Figure 7: *TPC-C benchmark (AVG. of 5 Runs) : W1 [NewOrder 34%, Payment 43%, Status 5%, Delivery 4%, StockLevel 14%], W2 [27%, 43%, 15%, 4%, 11%], W3 [20%, 43%, 25%, 4%, 8%], W4 [13%, 43%, 35%, 4%, 5%]*

to the sorted data in its leaf nodes. While WORT shows the fastest insertion performance, it suffers from a poor range query performance in this benchmark.

## 6.7 Concurrency and Recoverability

In the experiments shown in Figure 8, we evaluate the performance of multi-threaded versions of FAST+FAIR, FP-tree, SkipList, and B-link tree. Although wB+-tree and WORT are not designed to handle concurrent queries, they can employ well-known concurrency control protocols such as the *crabbing protocol* [41]. However, we do not implement and evaluate multi-threaded versions of wB+-tree and WORT. Instead, we present the performance of *B-link tree* for reference because it is known that *B-link tree* outperforms the crabbing protocol [41]. Note that B-link tree is not designed to provide the failure-atomicity for byte-granularity writes in PM and B-link tree does not allow the lock-free search. We implemented the concurrent version of FP-tree using Intel's Transactional Synchronization Extension (TSX) as was done in [34]. For this experiments, the write latency of PM is set to 300 nsec and the read latency of PM is set to be equal to that of DRAM. FAST+FAIR and SkipList eliminate the necessity of read locks but they require write locks to serialize write operations on tree nodes. Our implementations of FAST+FAIR and B-link tree use `std::mutex` class in C++11 and Skiplist uses a spin lock with gcc built-in CAS function, `__sync_bool_compare_and_swap`. But they can also employ the hardware transactional memory for higher concurrency. We compiled these implementations without the -O3 optimization option because the compiler optimization can reorder instructions and it affects the correctness of lock-free algorithm.

Although these experiments are intended to evaluate the concurrency, they also show the instant recoverability of FAST and FAIR algorithms. In a physical power-off test, we need to generate a large number of partially updated transient inconsistent tree nodes and see whether read threads can tolerate such inconsistent tree nodes. In the lock-free concurrency experiments, a large number of read transactions access various partially updated tree nodes. If the read transactions can ignore such transient inconsistent tree nodes, instant recovery is possible.

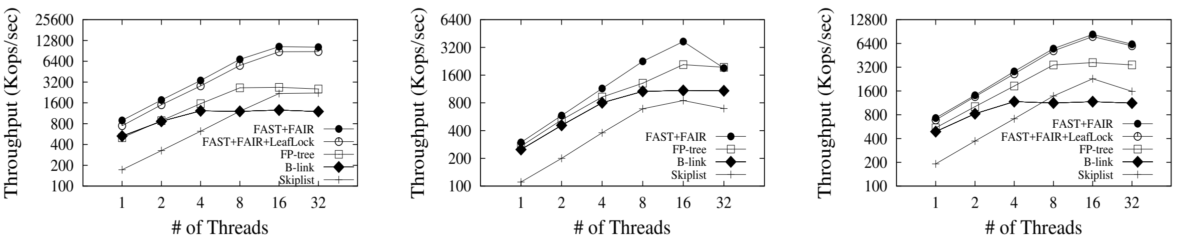
In the experiments shown in Figure 8, we run three workloads - *50M Search*, *50M Insertion*, and *Mixed*. We insert 50 million 8 byte random keys into the index and run each workload: For *50M Insertion* workload, we insert additional 50 million keys into the index. For *50M Search* we search 50 million keys. And for *Mixed* workload, each thread alternates between four insert queries, sixteen search queries, and one delete query. We use *numactl* to bind threads explicitly to a single socket to minimize the socket communication overhead and we distribute the workload across a number of threads.

Figure 8(a) shows that FAST+FAIR gains about a 11.7x faster speedup when the number of threads increases from one to sixteen. However, the speed-up saturates over 16 threads because our testbed machine has 16 vCPUs in a single socket. For FP-tree and B-link tree, the search speed-up becomes saturated when we use 8 and 4 threads respectively. When we run 8 threads, FP-tree takes advantage of the TSX and shows a throughput about 2.2x higher than B-link tree. Since SkipList also benefits from lock-free search, it scales to 16 threads but from a much lower throughput. Although FAST+FAIR+LeafLocks requires read threads to acquire read locks in leaf nodes, FAST+FAIR+LeafLocks shows a comparable concurrency level with FAST+FAIR. Note that the lock-free FAST+FAIR operates at read uncommitted mode while FAST+FAIR+LeafLocks operates at serializable mode.

In terms of write performance, FP-tree does not benefit much from the TSX as it shows a similar performance to B-link tree. It is because FP-tree performs expensive logging when a leaf node splits although it benefits from the faster TSX-enabled lock. In Figure 8(b), FAST+FAIR achieves about 12.5x higher insertion throughput when 16 threads run concurrently. In contrast, FP-tree and B-link tree achieve only a 7.7x and 4.4x higher throughput. Figure 8(c) shows that the scalability of FP-tree and B-link tree is limited because of read locks while FAST+FAIR takes advantage of the lock-free search. For the mixed workload, FAST+FAIR achieves up to a 11.44x higher throughput than a single thread.

## 7 Related Work

**Lock-free index**: In parallel computing community, various non-blocking implementations of popular data structures such as queues, lists, and search trees have been studied [4, 14, 15, 32]. Among various lock-free data structures, Braginsky et al.'s lock-free dynamic B+-tree [4] and Levandoski's Bw-tree [28] are the most rel-



(a) 50M Search (b) 50M Insert (c) 200M Search/50M Insert/12.5M Delete

Figure 8: *Performance with Varying Number of Threads (AVG. of 5 Runs)*

evant to our work. Unlike their lock-free B+-tree implementations, our current design still requires write latches for write threads because persistent B+-trees need to address durability as an additional challenge. We leave lock-free writes for persistent index as a future work.

**Memory persistency**: In order to resolve the ordering issues of memory writes in PM-based systems, numerous works [9, 35, 29] have lately proposed novel memory persistency models, such as *strict persistency* [35] and *epoch persistency* [9]. Strict persistency does not distinguish memory consistency from memory persistency, but *epoch persistency* [9] requires persist barriers so that persist order may deviate from the volatile memory order. These persistency models complement our work. FAST and FAIR guarantee the consistency of B+-tree under strict persistency model. If memory consistency is decoupled from memory persistency and the persist order deviates from the volatile memory order as in *relaxed persistency*, FAST and FAIR must call a persist barrier for every cache line flush instruction because they must enforce the order of cache line flushes to PM. However, within the same cache line, we do not need to call a persist barrier for each shift operation because array elements in the same cache line are guaranteed to be flushed to PM even under the relaxed persistency model. The only condition that FAST and FAIR require is that the dirty cache lines must be flushed in order. Therefore, FAST and FAIR place minimal persistence overhead under both strict and relaxed persistency models. That is, FAST calls persist barriers only as many times as the number of dirty cache lines in a B+-tree node under the relaxed persistency. On the other hand, other persistent indexes such as wB+-tree and FP-tree that employ append-only strategy need to call a persist barrier for each store instruction since their store instructions are not dependent. Due to the unavailability of PM that implements various persistency models, we leave a performance evaluation of our FAST and FAIR schemes under relaxed persistency model to our future work

**Hardware transactional memory**: The advent of commercially available hardware transactional memory such as the Intel's Restricted Transactional Memory (RTM) and Hardware Lock Elision (HLE) can be used to support coarse-grained atomic cache line writes [13, 27, 39, 46, 47]. Hardware transactional memories guarantee a dirty cache line remains in the write combining store buffer so that isolation can be preserved. However, memory persistency models including even strict persistency do not guarantee multiple cache lines will be flushed atomically even with the help of hardware transactional memory [39]. Hence, if a system crashes while flushing multiple cache lines, its consistency can not be guaranteed. Hence, hardware transactional memory in its current form cannot replace our FAST and FAIR algorithms as long as the tree node size is larger than a single cache line [22] and a tree needs rebalancing operations.

## 8 Conclusion

In this work, we have designed, implemented, and evaluated Failure-Atomic ShifT (FAST) and Failure-Atomic In-place Rebalance (FAIR) algorithms for legacy B+-trees to get the most benefit out of byte-addressable persistent memory. FAST and FAIR solves the granularity mismatch problem of PM without using logging and without modifying the data structure of B+-trees.

FAST and FAIR algorithms transform a consistent B+-tree into another consistent state or a *transient inconsistent* state that read operations can endure. By making read operations tolerate transient inconsistency, we can avoid expensive copy-on-write and logging. Besides, we can isolate read transactions, which enables non-blocking lock-free search.

## Acknowledgement

We would like to give our special thanks to our shepherd Dr. Jorge Guerra and the anonymous reviewers. Their comments and suggestions helped us correct and improve this work. This research was supported by Samsung Research Funding Centre of Samsung Electronics under Project Number SRFCSRFC-IT1501-04. The first and the second authors contributed equally and the corresponding author is Beomseok Nam.

## References

[1] HP Enterprise Lab, Memory Driven Computing. *https://www.labs.hpe.com/next-next/mdc*.

[2] Intel and Micron produce breakthrough memory technology. *https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology*.

[3] ARULRAJ, J., PAVLO, A., AND DULLOOR, S. R. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), ACM, pp. 707–722.

[4] BRAGINSKY, A., AND PETRANK, E. A lock-free B+tree. In *24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2012).

[5] CHEN, S., AND JIN, Q. Persistent B+-Trees in non-volatile main memory. *Proceedings of the VLDB Endowment (PVLDB) 8*, 7 (2015), 786–797.

[6] CHI, P., LEE, W.-C., AND XIE, Y. Making B+-tree efficient in PCM-based main memory. In *Proceedings of the 2014 international symposium on Low power electronics and design* (2014), ACM, pp. 69–74.

[7] CHIDAMBARAM, V., SHARMA, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Consistency without ordering. In *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST)* (2012).

[8] CHONG, N., AND ISHTIAQ, S. Reasoning about the ARM weakly consistent memory model. In *Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness (MSPC'08)* (2008), ACM, pp. 16–19.

[9] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B. C., BURGER, D., AND COETZEE, D. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)* (2009).

[10] CUPPU, V., JACOB, B., DAVIS, B., AND MUDGE, T. A performance comparison of contemporary DRAM architectures. In *the 26th International Symposium on Computer Architecture* (1999).

[11] DANOWITZ, A., KELLEY, K., MAO, J., STEVENSON, J. P., AND HOROWITZ, M. Cpu db: recording microprocessor history. *Communications of the ACM 55*, 4 (2012), 55–63.

[12] DONG, M., AND CHEN, H. Soft updates made simple and fast on non-volatile memory. In *2017 USENIX Annual Technical Conference* (Santa Clara, CA, 2017), USENIX Association, pp. 719–731.

[13] DULLOOR, S. R., KUMAR, S., KESHAVAMURTHY, A., LANTZ, P., REDDY, D., SANKARAN, R., AND JACKSON, J. System software for persistent memory. In *Proceedings of the 9th ACM European Conference on Computer Systems (EuroSys)* (2014), pp. 15:1–15:15.

[14] ELLEN, F., FATAOUROU, P., RUPPERT, E., AND VAN BREUGEL, F. Non-blocking binary search trees. In *the 29th ACM Symposium on Principles of Distributed Computing (PODC)* (2010).

[15] FOMITCHEV, M., AND RUPPERT, E. Lock-free linked lists and skiplists. In *the 23rd ACM Symposium on Principles of Distributed Computing (PODC)* (2004).

[16] HU, Q., REN, J., BADAM, A., AND MOSCIBRODA, T. Log-structured non-volatile main memory. In *Proceedings of the USENIX Annual Technical Conference* (2017).

[17] HUAI, Y. Spin-transfer torque MRAM (STT-MRAM): Challenges and prospects. *AAPPS bulletin 18*, 6 (2008), 33–40.

[18] INTEL. Intel® 64 Architecture Memory Ordering White Paper, August 2007. SKU 318147-001.

[19] KIM, C., CHHUGANI, J., SATISH, N., SEDLAR, E., NGUYEN, A. D., KALDEWEY, T., LEE, V. W., BRANDT, S. A., AND DUBEY, P. FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2010).

[20] KIM, W.-H., KIM, J., BAEK, W., NAM, B., AND WON, Y. NVWAL: Exploiting NVRAM in write-ahead logging. In *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2016).

[21] KIM, W.-H., NAM, B., PARK, D., AND WON, Y. Resolving journling of journal anomaly in Android I/O: Multi-version B-tree with lazy split. In *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST)* (2014).

[22] KIM, W.-H., SEO, J., KIM, J., , AND NAM, B. clfB-tree: Cacheline friendly persistent B-tree for NVRAM. *ACM Transactions on Storage (TOS), Special Issue on NVM and Storage* (2018).

[23] LEE, S. K., LIM, K. H., SONG, H., NAM, B., AND NOH, S. H. WORT: Write optimal radix tree for persistent memory storage systems. In *Proceedings of the 15th USENIX conference on File and Storage Technologies (FAST)* (2017).

[24] LEE, W., LEE, K., SON, H., KIM, W.-H., NAM, B., AND WON, Y. WALDIO: Eliminating the filesystem journaling in resolving the journaling of journal anomaly. In *Proceedings of the 2015 USENIX Anual Technical Conference* (2015).

[25] LEHMAN, P. L., AND YAO, S. B. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems 6*, 4 (1981), 650–670.

[26] LEHMAN, T. J., AND CAREY, M. J. A study of index structures for main memory database management systems. In *Proceedings of the 12th International Conference on Very Large Data Bases (VLDB)* (1986).

[27] LEIS, V., KEMPER, A., AND NEUMANN, T. Exploiting hardware transactional memory in main-memory databases. In *Proceedings of the 30th International Conference on Data Engineering (ICDE)* (2014).

[28] LEVANDOSKI, J. J., LOMET, D. B., AND SENGUPTA, S. The Bw-Tree: a B-tree for new hardware platforms. In *Proceedings of the 29th International Conference on Data Engineering (ICDE)* (2013).

[29] LU, Y., SHU, J., AND SUN, L. Blurred persistence in transactional persistent memory. In *Proceedings of the 31st International Conference on Massive Storage Systems and Technology (MSST)* (2015).

[30] MCKENNEY, P. E. Memory ordering in modern microprocessors. *Linux Journal 136*, Aug. (2005).

[31] MCKUSICK, M. K., GANGER, G. R., ET AL. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *USENIX Annual Technical Conference, FREENIX Track* (1999), pp. 1–17.

[32] MICHAEL, M. M. High performance dynamic lock-free hash tables and list-based sets. In *the 14th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2002).

[33] MORARU, I., ANDERSEN, D. G., KAMINSKY, M., TOLIA, N., BINKERT, N., AND RANGANATHAN, P. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the ACM Conference on Timely Results in Operating Systems (TRIOS)* (2013).

[34] OUKID, I., LASPERAS, J., NICA, A., WILLHALM, T., AND LEHNER, W. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *Proceedings of 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2016).

[35] PELLEY, S., CHEN, P. M., AND WENISCH, T. F. Memory persistency. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)* (2014), pp. 265–276.

[36] RAO, J., AND ROSS, K. A. Cache conscious indexing for decision-support in main memory. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)* (1999).

[37] RAO, J., AND ROSS, K. A. Making B+-trees cache conscious in main memory. In *Proceedings of 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2000).

[38] RIXNER, S., DALLY, W. J., KAPASI, U. J., MATTSON, P., AND OWENS, J. D. Memory access scheduling. In *ACM SIGARCH Computer Architecture News* (2000), vol. 28, ACM, pp. 128–138.

[39] SEO, J., KIM, W.-H., BAEK, W., NAM, B., AND NOH, S. H. Failure-atomic slotted paging for persistent memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2017).

[40] SEWELL, P., SARKAR, S., OWENS, S., NARDELLI, F. Z., AND MYREEN, M. O. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM 53*, 7 (2010), 89–97.

[41] SILBERSCHATZ, A., KORTH, H., AND SUDARSHAN, S. *Database Systems Concepts*. McGraw-Hill, 2005.

[42] VENKATARAMAN, S., TOLIA, N., RANGANATHAN, P., AND CAMPBELL, R. H. Consistent and durable data structures for non-volatile byte-addressable memory. In *9th USENIX conference on File and Storage Technologies (FAST)* (2011).

[43] VOLOS, H., MAGALHAES, G., CHERKASOVA, L., AND LI, J. Quartz: A lightweight performance emulator for persistent memory software. In *15th Annual Middleware Conference (Middleware '15)* (2015).

[44] VOLOS, H., TACK, A. J., AND SWIFT, M. M. Mnemosyne: Lightweight persistent memory. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2011).

[45] WANG, Y., KAPRITSOS, M., REN, Z., MAHAJAN, P., KIRUBANANDAM, J., ALVISI, L., AND DAHLIN, M. Robustness in the salus scalable block store. In *Proc. of USENIX NSDI 2013* (Apr 2013).

[46] WANG, Z., QIAN, H., LI, J., AND CHEN, H. Using restricted transactional memory to build a scalable in-memory database. In *ACM SIGOPS/Eurosys European Conference on Computer Systems (EuroSys)* (2014).

[47] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)* (2015).

[48] WONG, H.-S. P., RAOUX, S., KIM, S., LIANG, J., REIFENBERG, J. P., RAJENDRAN, B., ASHEGHI, M., AND GOODSON, K. E. Phase change memory. *Proceedings of the IEEE 98*, 12 (2010), 2201–2227.

[49] YANG, J., WEI, Q., CHEN, C., WANG, C., AND YONG, K. L. NV-Tree: reducing consistency const for NVM-based single level systems. In *Proceedings of the 13th USENIX conference on File and Storage Technologies (FAST)* (2015).