저작자표시-비영리-변경금지 2.0 대한민국

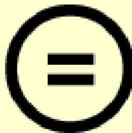
이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

[UCI]I804:31001-200000223568

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

Master's Thesis

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

Deukyeon Hwang

Department of Computer Science and Engineering

Graduate School of UNIST

2019

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

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

Deukyeon Hwang

Department of Computer Science and Engineering

Graduate School of UNIST

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

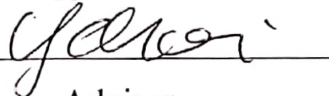
Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree

A thesis/dissertation
submitted to the Graduate School of UNIST
in partial fulfillment of the
requirements for the degree of
Master of Science

Deukyeon Hwang

05/31/2019 of submission

Approved by



Advisor
Young-ri Choi

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

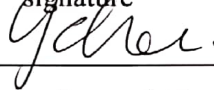
Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree

Deukyeon Hwang

This certifies that the thesis/dissertation of Deukyeon Hwang is approved.

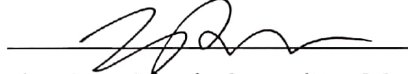
05/31/2019

signature



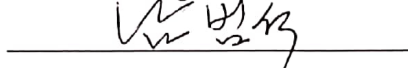
Advisor: Young-ri Choi

signature



Myeongjae Jeon: Thesis Committee Member #1

signature



Beomseok Nam: Thesis Committee Member #2

three signatures total in case of masters

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

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.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Contents

I Introduction 1

II B+-tree for Persistent Memory 3

2.1 Challenge: clflush and mfence 3

2.2 Reordering Memory Accesses 3

III Failure-Atomic B+-tree Algorithms 5

3.1 Failure-Atomic ShifT (FAST) 5

3.2 Failure-Atomic In-place Rebalance 6

IV Lock-Free Search 7

4.1 Lock-Free Search Consistency Model 8

4.2 Lazy Recovery for Lock-Free Search 9

V Experiments 10

5.1 Experimental Environment 10

5.2 Linear Search vs. Binary Search 11

5.3 Range Query 12

5.4 PM Latency Effect 12

5.5 Performance on Non-TSO 14

5.6 TPC-C Benchmark 14

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

5.7 Concurrency and Recoverability 15

VI Related Work 17

VII Conclusion 19

References 20

Acknowledgements 25

Appendices 26

I Pseudo Codes of Algorithms 27

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

List of Figures

1 *FAST Inserting (25, Ptr) into B-tree node* 5

2 *FAIR Node Split* 6

3 *Linear vs. Binary Search* 11

4 *Range Query Speed-Up from using SkipList with Varying Selection Ratio (AVG. of 5 Runs)* 12

5 *Performance Comparison of Single-Threaded Index (AVG. of 5 Runs)* 13

6 *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%]* 15

7 *Performance with Varying Number of Threads (AVG. of 5 Runs)* 15

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

I Introduction

In recent years, byte-addressable persistent memories (PM) have been introduced. For instance, there are phase change memory [1], spin-transfer torque MRAM [2], and 3D Xpoint [3]. Optane DC Persistent Memory had been developed by Intel and evaluated by many researchers [4, 5]. They 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 the block devices [6, 7].

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 [8] 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 [9], CSB-tree [10], and FAST [11] are shown to perform better than legacy binary counterparts such as T-tree [12].

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* [13] 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 [14, 15].

The recently proposed B+-tree variants such as NV-tree [16], FP-tree [17], and wB+-tree [14] 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

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

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 [16] and FP-tree [17] 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 [18, 19] and *NoFS* [20].

The rest of the paper is organized as follows: In Section II, we present the challenges in designing a B+-tree index for PM. In Section III, we propose the FAST and FAIR algorithms. In Section IV, we discuss how to enable non-blocking read transactions. Section V evaluates the performance of the proposed persistent B+-tree and Section VI discusses other research efforts. In Section VII we conclude this paper.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

II 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 [14–16], 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 [14, 16, 17]. 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 [21].

2.2 Reordering Memory Accesses

The reordering of memory write operations helps better utilize the memory bandwidth [22]. In the last few decades, the performance of modern processors has improved at a much faster rate than that of memory [23]. 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 [24], while other architectures such as x86 prevent stores-after-stores from being reordered [25, 26], that is, *total store ordering* (TSO) 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 [27]. Given that the Alpha processor has deprecated since 2003, we consider that all modern processors do not reorder dependent loads.

Memory persistency [28] 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

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

model may allow the persist order to deviate from the volatile order under the *relaxed persistency* model [13, 28]. To simplify our discussion, we first present algorithms assuming the strict persistency model. Later, in Section VI, we will discuss the relaxed persistency model.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

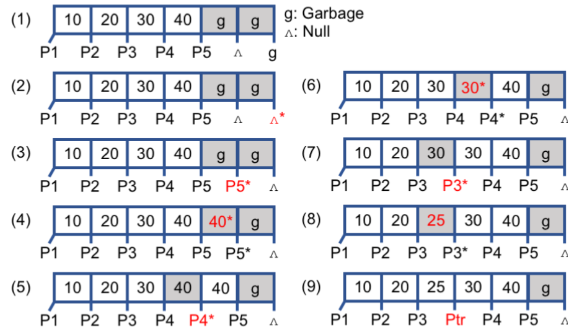


Figure 1: *FAST Inserting (25, Ptr) into B-tree node*

III Failure-Atomic B+-tree Algorithms

We propose *Failure-Atomic ShifT* (FAST) and *Failure-Atomic In-place Rebalance* (FAIR) instead of append-only and selective persistence manners which have been proposed by previous researches. The main idea of FAST and FAIR algorithms is that a read transaction can detect and ignore transient inconsistency by a write transaction. In the following subsections, Section 3.1 and 3.2, we explain how read transactions tolerate inconsistencies by failure.

3.1 Failure-Atomic ShifT (FAST)

FAST allows read transactions to detect and ignore transient inconsistencies by duplicate pointers. FAST uses two facts that A B+-tree node has unique pointers and the size of each pointer is 8-byte which is the data size for CPU to update its value atomically. The condition of a valid key is that its left and right keys are different. When a read transaction searches a key, it compares the left-hand and the right-hand pointers of a key. A read transaction considers the key as the invalid and moves to the next key if the left and right pointers are same. For example, there are two 40s at the step (5) in Figure 1. The left 40 which is colored as grey is invalid because both left and right pointers are P4 while the right 40 is valid due to the different pointers.

Transient inconsistency occurs when keys and pointers in a B+-tree node are shifted in order. Figure 1 shows the process of inserting a pair *(25, Ptr)* into a B+-tree node. To shift keys and pointers from left to right, it starts from shifting the rightmost pointer first and then the rightmost key. The following sequence is also shifting pointer and key in order. During the shift operation of B+-tree, there are two same keys which are a key and its copy. As shown at the step (5) in Figure 1, updating a pointer atomically makes one of them valid and makes the other invalid at the same time. At the last step, updating a new pointer *Ptr* behaves as a commit.

To guarantee the shifting order of keys and pointers in a B+-tree node, we should use barrier instruction such as *mfence* to prevent from reordering store and store instructions. Fortunately, *total store ordering* (TSO) systems allow us to remove memory fence instruction between store

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

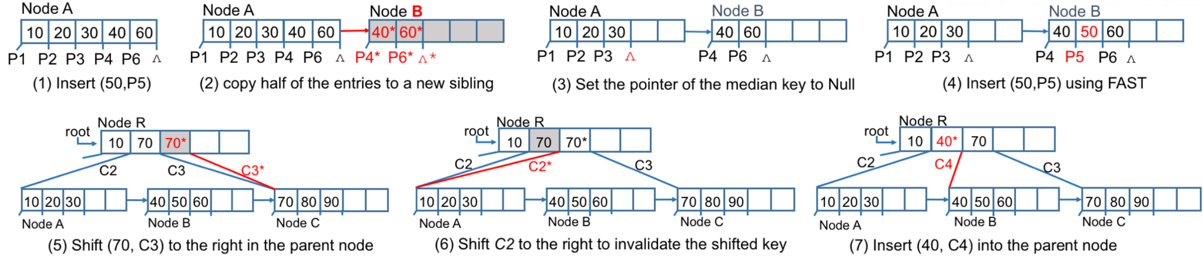


Figure 2: *FAIR Node Split*

instructions. It helps to reduce overheads by calling additional instructions. Non-TSO systems still need to guarantee the order of instructions by calling memory fence instructions explicitly.

If the size of a tree node spans to multiple cache lines, we need to flush the current dirty cache line before modifying the next cache line. Dirty cache lines in the CPU cache are randomly flushed to persistent memory. Unless we call *clflush* explicitly after modifying data at the boundary of a cache line, the tree node would be inconsistent by a system failure. That is, we need to guarantee the order of flushing cache lines while shifting keys and pointers. Algorithm 1(Appendix I) shows the pseudo code of FAST.

3.2 Failure-Atomic In-place Rebalance

In existing B+-tree variants [14, 16, 17], logging or copy-on-write (COW), which has heavy overhead, is a common way to guarantee data consistency when a B+-tree node splits. FAIR algorithm allows us to avoid copy overheads by expensive logging or COW. Like FAST, FAIR algorithm lets read transactions tolerate transient inconsistency. Our B+-tree node has a sibling pointer for leaf nodes as well as internal nodes as in B-link tree [29].

Figure 2 illustrates FAIR node split process step by step with an example. Consistency of B+-tree nodes should be guaranteed at each step. During a node split, read transactions should consider two cases – One is the case of step (2) and the other is that of step (3) or (4) in Figure 2. A sibling node B is created and half of data are copied from node A to B. A sibling node is invisible before connecting two nodes with a sibling pointer. At (2), read transactions look up keys of two nodes through a sibling pointer. If a system failure occurs at (2), read transactions can detect inconsistency because the order of keys is incorrect when reaching at node B. In (3), node A and B looks like a virtual single node which has all keys in two nodes with a sibling pointer. It is not problematic for read transactions to find a key in spite of a system crash. The rest of split process is to update the pointer of the new sibling node to its parent node by using FAST. The pseudo code of FAIR algorithm is shown in Algorithm 2 (Appendix I).

IV 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 [30], 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 [29]. 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 $P3$ 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 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.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

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 (Appendix I).

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 5.2.

4.1 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 [31].

In the database community, various levels of isolation such as *serializable mode*, *non-repeatable read mode*, *read committed mode*, and *read uncommitted mode* [31] 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 [31]. 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 5.7.

UNIST
ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

4.2 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.

V Experiments

We evaluate the performance of FAST and FAIR against the state-of-the-art indexing structures for PM - wB+-tree with *slot+bitmap nodes* [14], FP-tree [17], WORT [32], and Skiplist [33].

wB+-tree [14] 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 [17] 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 [32] 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.

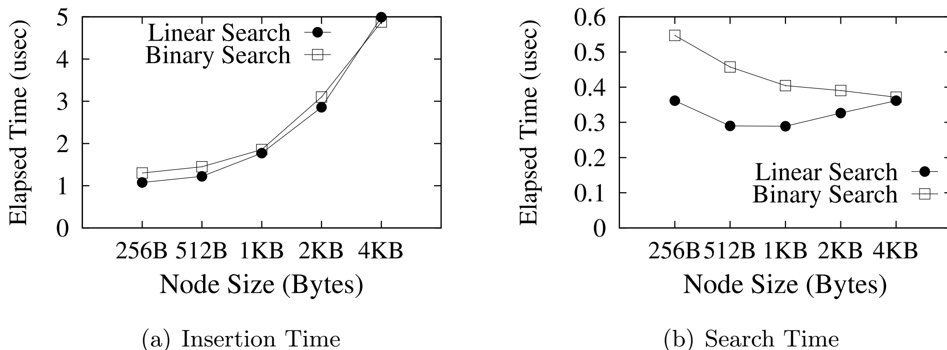
SkipList [33] is a probabilistic indexing structure that avoids expensive rebalancing operations. An update of the skip list needs pointer updates in multi-level linked lists. But only the lowest level-linked list needs to be updated in a failure-atomic way. In a Log-Structured NVMM System [33], SkipList 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.

5.1 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* [34] as was done in [35–37]. Quartz models application-perceived PM latency by inserting stall cycles in each predefined time interval

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY



(a) Insertion Time

(b) Search Time

Figure 3: *Linear vs. Binary Search*

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.

5.2 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 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.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

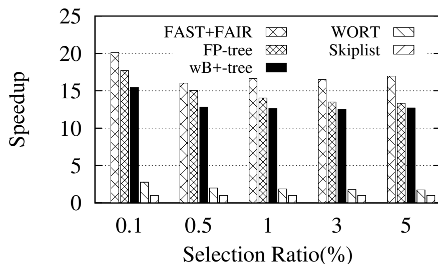


Figure 4: *Range Query Speed-Up from using SkipList with Varying Selection Ratio (AVG. of 5 Runs)*

5.3 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.

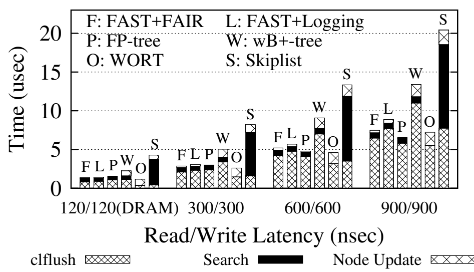
5.4 PM Latency Effect

In the experiment shown in Figure 5(a) and 5, 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 [17]. 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 [17].

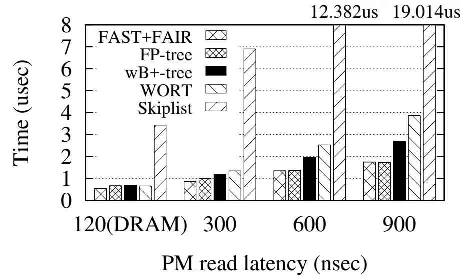
Figure 5(a) shows that FAST+FAIR, FP-tree, and WORT show comparable insertion performances and they outperform wB+-tree and SkipList by a large margin. In detail, the insertion time is composed of three parts, *Cache line flush*, *Search*, and *Node Update* times. wB+-tree calls a 1.7 times larger number 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

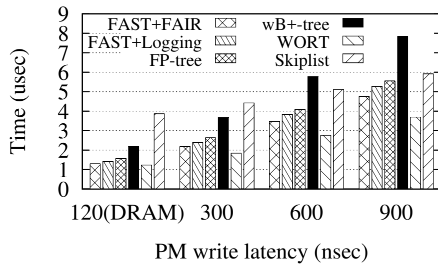
ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY



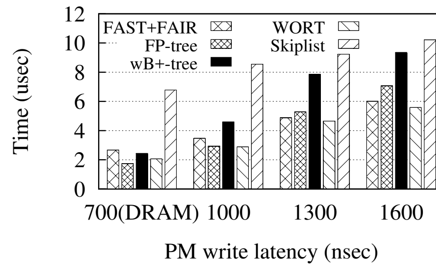
(a) Breakdown of Time Spent for B-tree Insertion



(b) Search : Varying Read Latency



(c) Insert : Varying Write Latency on TSO Architecture



(d) Insert : Varying Write Latency on Non-TSO Architecture

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

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 leverage 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 5(b) 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 [34]. 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 5(c), 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.

5.5 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 5(d) 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.

5.6 TPC-C Benchmark

In real-world applications, workloads are often mixed with writes and reads. In the experiments shown in Figure 6, 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 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.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

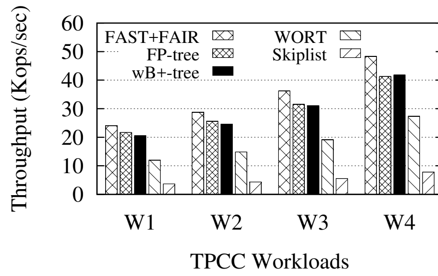
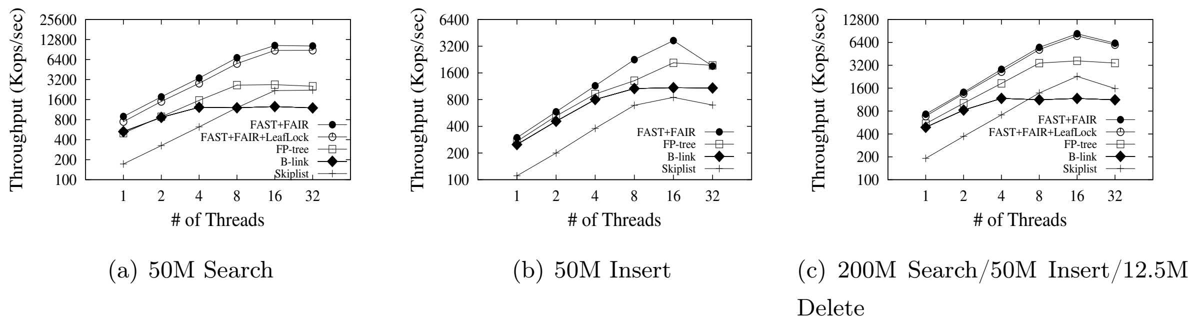


Figure 6: *TPC-C benchmark (AVG. of 5 Runs) : W1 [NewOrder 34%, Payment 43%, Status 5%, Delivery 4%, StockLevel 14%], W2 [27%, 43%, 15%, 4%, 11%], W3 [20%, 43%, 25%, 4%, 8%], W4 [13%, 43%, 35%, 4%, 5%]*



(a) 50M Search

(b) 50M Insert

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

Figure 7: *Performance with Varying Number of Threads (AVG. of 5 Runs)*

5.7 Concurrency and Recoverability

In the experiments shown in Figure 7, 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* [31]. 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 [31]. 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 [17]. 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.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

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 7, 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 7(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 7(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 7(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.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

VI Related Work

Persistent hashing scheme: Indexing is one of key parts to gain the performance of accessing data in most applications. As persistent memories are commercializing, many applications including academic projects and huge business products should run on PM systems. Although B+-tree has good performance of both exact match query and range query, hash table is a popular data structure which is used to implement kinds of dictionary, key-value store, database, and so on. Hash table is much faster and more efficient for exact match query than tree-like structures. Because of that, hash tables for PM are also hot topics in the current [38–40], while early persistent index studies had focused on tree such as B+-tree or radix tree [14–17, 32]. We leave comparison of applications using tree structures or hash table on real-life PM systems.

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 [41–44]. Among various lock-free data structures, Braginsky et al.'s lock-free dynamic B+-tree [41] and Levandoski's Bw-tree [45] are the most relevant 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 [13, 28, 46] have lately proposed novel memory persistency models, such as *strict persistency* [28] and *epoch persistency* [13]. Strict persistency does not distinguish memory consistency from memory persistency, but *epoch persistency* [13] 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 trans-

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

actional 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 [47–51]. 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 [49]. 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 [52] and a tree needs rebalancing operations.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

VII 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.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

References

[1] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase change memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2201–2227, 2010.

[2] Y. Huai, "Spin-transfer torque MRAM (STT-MRAM): Challenges and prospects," *AAPPS bulletin*, vol. 18, no. 6, pp. 33–40, 2008.

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

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

[5] G. Gill, R. Dathathri, L. Hoang, R. Peri, and K. Pingali, "Single machine graph analytics on massive datasets using intel optane DC persistent memory," *CoRR*, vol. abs/1904.07162, 2019. [Online]. Available: http://arxiv.org/abs/1904.07162

[6] W.-H. Kim, B. Nam, D. Park, and Y. Won, "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.

[7] W. Lee, K. Lee, H. Son, W.-H. Kim, B. Nam, and Y. Won, "WALDIO: Eliminating the filesystem journaling in resolving the journaling of journal anomaly," in *Proceedings of the 2015 USENIX Anual Technical Conference*, 2015.

[8] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz, "Cpu db: recording microprocessor history," *Communications of the ACM*, vol. 55, no. 4, pp. 55–63, 2012.

[9] J. Rao and K. A. Ross, "Cache conscious indexing for decision-support in main memory," in *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*, 1999.

[10] ——, "Making B+-trees cache conscious in main memory," in *Proceedings of 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2000.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

[11] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, "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.

[12] T. J. Lehman and M. J. Carey, "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.

[13] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. C. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[14] S. Chen and Q. Jin, "Persistent B+-Trees in non-volatile main memory," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 8, no. 7, pp. 786–797, 2015.

[15] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *9th USENIX conference on File and Storage Technologies (FAST)*, 2011.

[16] J. Yang, Q. Wei, C. Chen, C. Wang, and K. L. Yong, "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.

[17] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, "FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory," in *Proceedings of 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2016.

[18] M. Dong and H. Chen, "Soft updates made simple and fast on non-volatile memory," in *2017 USENIX Annual Technical Conference.* Santa Clara, CA: USENIX Association, 2017, pp. 719–731. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/dong

[19] M. K. McKusick, G. R. Ganger *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.

[20] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Consistency without ordering," in *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST)*, 2012.

[21] P. Chi, W.-C. Lee, and Y. Xie, "Making B+-tree efficient in PCM-based main memory," in *Proceedings of the 2014 international symposium on Low power electronics and design.* ACM, 2014, pp. 69–74.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

[22] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2. ACM, 2000, pp. 128–138.

[23] V. Cuppu, B. Jacob, B. Davis, and T. Mudge, "A performance comparison of contemporary DRAM architectures," in *the 26th International Symposium on Computer Architecture*, 1999.

[24] N. Chong and S. Ishtiaq, "Reasoning about the ARM weakly consistent memory model," in *Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness (MSPC'08)*. ACM, 2008, pp. 16–19.

[25] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "x86-tso: a rigorous and usable programmer's model for x86 multiprocessors," *Communications of the ACM*, vol. 53, no. 7, pp. 89–97, 2010.

[26] INTEL, "Intel® 64 Architecture Memory Ordering White Paper," August 2007, sKU 318147-001.

[27] P. E. McKenney, "Memory ordering in modern microprocessors," *Linux Journal*, vol. 136, no. Aug., 2005.

[28] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 265–276.

[29] P. L. Lehman and S. B. Yao, "Efficient locking for concurrent operations on B-trees," *ACM Transactions on Database Systems*, vol. 6, no. 4, pp. 650–670, 1981.

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

[31] A. Silberschatz, H. Korth, and S. Sudarshan, *Database Systems Concepts.* McGraw-Hill, 2005.

[32] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, "WORT: Write optimal radix tree for persistent memory storage systems," in *Proceedings of the 15th USENIX conference on File and Storage Technologies (FAST)*, 2017.

[33] Q. Hu, J. Ren, A. Badam, and T. Moscibroda, "Log-structured non-volatile main memory," in *Proceedings of the USENIX Annual Technical Conference*, 2017.

[34] H. Volos, G. Magalhaes, L. Cherkasova, and J. Li, "Quartz: A lightweight performance emulator for persistent memory software," in *15th Annual Middleware Conference (Middleware '15)*, 2015.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

[35] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[36] J. Arulraj, A. Pavlo, and S. R. Dulloor, "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*. ACM, 2015, pp. 707–722.

[37] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won, "NVWAL: Exploiting NVRAM in write-ahead logging," in *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[38] X. Zhang, D. Feng, Y. Hua, J. Chen, and M. Fu, "A write-efficient and consistent hashing scheme for non-volatile memory," in *Proceedings of the 47th International Conference on Parallel Processing*. ACM, 2018, p. 87.

[39] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 461–476.

[40] M. Nam, H. Cha, Y.-r. Choi, S. H. Noh, and B. Nam, "Write-optimized dynamic hashing for persistent memory," in *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, 2019, pp. 31–44.

[41] A. Braginsky and E. Petrank, "A lock-free B+tree," in *24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2012.

[42] F. Ellen, P. Fataourou, E. Ruppert, and F. van Breugel, "Non-blocking binary search trees," in *the 29th ACM Symposium on Principles of Distributed Computing (PODC)*, 2010.

[43] M. Fomitchev and E. Ruppert, "Lock-free linked lists and skiplists," in *the 23rd ACM Symposium on Principles of Distributed Computing (PODC)*, 2004.

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

[45] J. J. Levandoski, D. B. Lomet, and S. Sengupta, "The Bw-Tree: a B-tree for new hardware platforms," in *Proceedings of the 29th International Conference on Data Engineering (ICDE)*, 2013.

[46] Y. Lu, J. Shu, and L. Sun, "Blurred persistence in transactional persistent memory," in *Proceedings of the 31st International Conference on Massive Storage Systems and Technology (MSST)*, 2015.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

[47] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the 9th ACM European Conference on Computer Systems (EuroSys)*, 2014, pp. 15:1–15:15.

[48] V. Leis, A. Kemper, and T. Neumann, "Exploiting hardware transactional memory in main-memory databases," in *Proceedings of the 30th International Conference on Data Engineering (ICDE)*, 2014.

[49] J. Seo, W.-H. Kim, W. Baek, B. Nam, and S. H. Noh, "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.

[50] Z. Wang, H. Qian, J. Li, and H. Chen, "Using restricted transactional memory to build a scalable in-memory database," in *ACM SIGOPS/Eurosys European Conference on Computer Systems (EuroSys)*, 2014.

[51] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, "Fast in-memory transaction processing using RDMA and HTM," in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, 2015.

[52] W.-H. Kim, J. Seo, J. Kim, , and B. Nam, "clfB-tree: Cacheline friendly persistent B-tree for NVRAM," *ACM Transactions on Storage (TOS), Special Issue on NVM and Storage*, 2018.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

Acknowledgements

I would first like to specially thank Associate Professor Beomseok Nam of College of Software at Sungkyunkwan University. We meet at first 9 years ago. He has been the best advisor for me since that time. It was pretty exciting for me to share idea with the professor. He always steered me in the right direction. He was and is still my role-model. I would also like to thank my advisor, Associate Professor Young-ri Choi of Computer Science and Engineering at Ulsan National Institute of Science and Technology (UNIST). Her advice helps for me to complete this thesis and prepare Ph.D. program in Austin, Texas. I sincerely thank Assistant Professor Myeongjae Jeon of Computer Science and Engineering at UNIST. He gave me helpful comments as a committee. Jinwoong Kim and Wook-hee Kim are the best friends who I meet at UNIST. We are great helpers to each other for life as well as research activities. They were not hesitate to give me any kind of advice. Wherever we are, we are going to be the best friends as before. In addition, I would also like to remember all lab-mates of DICL and CISSR. I sincerely was happy to take time together. Finally, I must express my very profound gratitude to my parents and family for providing me with unfailing support and continuous encouragement throughout my years of Master.

This accomplishment would not have been possible if I were alone.
Thank you so much.

Deukyeon Hwang

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Appendices

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

I Pseudo Codes of Algorithms

Algorithm 1

$FAST_insert(node, key, ptr)$

```
1: node.lock.acquire()
2: if (sibling ← node.sibling_ptr) ≠ NULL then
3:     if sibling.records[0].key < key then
4:         – previous write thread has split this node
5:         node.lock.release();
6:         FAST_insert(sibling, key, ptr);
7:         return
8:     end if
9: end if
10: if node.cnt < node_capacity then
11:     if node.search_dir_flag is odd then
12:         – 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
13:         node.search_dir_flag++;
14:     end if
15:     for i ← node.cnt − 1; i ≥ 0; i−− do
16:         if node.records[i].key > key then
17:             node.records[i + 1].ptr ← node.records[i].ptr;
18:             mfence_IF_NOT_TSO();
19:             node.records[i + 1].key ← node.records[i].key;
20:             mfence_IF_NOT_TSO();
21:             if &(node.records[i + 1]) is at cacheline boundary then
22:                 clflush_with_mfence(&node.records[i + 1]);
23:             end if
24:         else
25:             node.records[i + 1].ptr ← node.records[i].ptr;
26:             mfence_IF_NOT_TSO();
27:             node.records[i + 1].key ← key;
28:             mfence_IF_NOT_TSO();
29:             node.records[i + 1].ptr ← ptr;
30:             clflush_with_mfence(&node.records[i + 1]);
31:         end if
32:     end for
33:     node.lock.release()
34: else
35:     node.lock.release()
36:     FAIR_split(node, key, ptr);
37: end if
```

UNIST
ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

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: $FAST_internal_node_insert(root, node.level + 1, sibling.records[0].key, node.sibling_ptr)$;

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Algorithm 3

$LockFreeSearch(node, key)$

```
1: repeat
2:   ret ← NULL;
3:   prev_switch ← node.switch;
4:   if prev_switch is even then
5:     – we scan this node from left to right
6:     for i ← 0; records[i].ptr ≠ NULL; i++ do
7:       if (k ← records[i].key) = key&&records[i].ptr ≠ (t ← records[i+1].ptr) then
8:         if (k = records[i].key) then
9:           ret ← t;
10:          break;
11:        end if
12:      end if
13:    end for
14:  else
15:    – this node was accessed by a delete query
16:    – we have to scan this node from right to left
17:    for i ← node.cnt − 1; i >= 0; i−− do
18:      – omitted due to symmetry and lack of space
19:    end for
20:  end if
21: until prev_switch = node.switch
22: if ret = NULL && (t ← node.sibling_ptr) then
23:   if t.records[0].key ≤ key then
24:     return t;
25:   end if
26: end if
27: return ret;
```

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY