FAST and FAIR B+-Tree for Byte-Addressable Persistent Memory

Wook-Hee Kim†, Deukyeon Hwang†, Jonghyeon Yoo§, Youjip Won‡, Beomseok Nam§
UNIST (Ulsan National Institute of Science and Technology)†
Hanyang University‡, *Sungkyunkwan University*§

Abstract

This extended abstract presents our FAST and FAIR B+-tree that redesigns insertion, deletion, rebalancing, and search algorithms such that tree structures can be modified in a failure-atomic fashion via a series of store and clflush instructions. We also present the performance of legacy binary T-tree that we modified for byte-addressable persistent memory. Our experimental results show the performance of T-tree is comparable to other state-of-the-art persistent indexes although its implementation is much simpler.

1 Introduction

In the past decades, byte-addressability and persistency have never been considered together. With the emerging byte-addressable persistent memory, various studies have been recently made to leverage the byte-addressability of block-based data structures while guaranteeing the failure-atomicity. One of the key challenges in persistent data structures is that the dirty cache lines in CPU caches can be flushed to persistent memory at any arbitrary time and the ordering of store instructions is not guaranteed. Flushing partially updated dirty cachelines and reordering memory writes can make data structures inconsistent and such transient inconsistency becomes persistent if a system crashes.

To resolve this problem, various works, including NV-tree [6], wB+-tree [1], failure-atomic slotted paging [2], and FP-tree [5], proposed to use *append-only update* strategies so that the consistent part of data structures remain unmodified. However, the append-only update strategy fails to fully leverage the byte-addressability of persistent memory, i.e., the append-only update does not preserve the order of keys. To the best of our knowledge, none of the previous work has studied failure-atomic in-place sorting algorithms except clfB-tree [4]. To in-place update sorted keys, we can make the tree node size as small as a single cacheline, but it increases the tree height and degrades the search performance. Alternatively, we can employ copy-on-write or logging as was done for legacy block device storage, but it increases the number of calls to expensive cacheline flush instructions.

Another challenge in byte-addressable persistent data structures arises when we need failure-atomic updates to multiple cachelines. For example, if a B+-tree node overflows or underflows, we need to split or merge multiple tree nodes to rebalance the tree height. Since it is impossible to atomically update multiple cachelines in modern hardware designs, previous works such as NV-tree and FP-tree proposed *selective persistence* strategy where we store internal B+-tree nodes in volatile DRAM instead of persistent memory, such that we do not need failure-atomic rebalancing operations. That is, if a system crashes, internal tree nodes in volatile memory will be lost. Although the reconstruction of internal tree nodes is possible, as [6] and [5] claim, reconstruction of the entire tree structure is expensive and it will prevent the instant recovery of systems.

2 FAST and FAIR B+-Tree

Instead of the append-only updates and selective persistence, we propose the *failure-atomic shift (FAST)* and *failure-atomic in-place rebalance (FAIR)* algorithms for B+-tree. The key idea behind these two algorithms is that we can make read transactions be aware of what changes write transactions are making to a B+-tree.

FAST: In a tree structure, child pointers always have unique memory addresses. Because of this property, read transactions can detect and ignore a transient inconsistent state partially updated by a write transaction. For example, when we shift pointers in an array of key-pointer pairs, shifting will duplicate pointers in the array. Since such duplicate pointers are not normal, read transactions can ignore the key in between duplicate pointers. I.e., we can shift keys and pointers in a failure-atomic fashion without using logging.

We note that the performance of FAST algorithm is sensitive to the tree node size as it performs more shift operations as the node size increases. In our testbed, we find 512 Bytes and 1 KBytes show the fastest performance. When the node size is set to 4 KB, the insertion time increases by a factor of 3 compared to when the node size is 1 KB.

FAIR: Although updating multiple tree nodes can be failure-atomic if we perform copy-on-write or logging, FAIR algorithm performs in-place updates using a sequence of store instructions so that it can reduce the number of dirty cacheline flushes. To ensure the correctness and invariants of the index, we logically combine two sibling nodes to be updated using a sibling pointer so that they can be treated as a single node. Although tree rebalancing operations require multiple tree node updates, FAIR algorithm controls the order of store instructions and makes read transactions be aware of rebalancing-in-transit operations.

If every store instruction in FAST and FAIR algorithms guarantees that no read transaction will ever access inconsistent tree nodes it is guaranteed that read transaction will always return the correct results. This observation implies that FAST and FAIR algorithms enable non-blocking lock-free search. Although FAST and FAIR algorithms are designed for byte-addressable persistent memory, we note that they can be used for volatile DRAM to eliminate the necessity of read lock (or latches) so that read transactions can be non-blocking. The details of FAST and FAIR algorithms and how they enable lock-free search are referred to [3].

3 Failure-Atomic T-Tree for PM

FAST algorithm is not just for B+-tree but it can be used for any data structure that needs sorted records. T-tree is a binary index used by various in-memory database systems. Although T-tree nodes have two child pointers, they can hold multiple keys in each node when we set the tree node size to the cacheline size. Since the keys in a T-tree node need to be sorted, we use the FAST algorithm to shift keys in a failure-atomic manner. Note that T-tree performs AVL-tree-like rotation operations to rebalance the tree height. Since the rotation operations are too complicated to make them failure-atomic, we decided to use explicit logging. However, we note that the logging overhead is not very significant because we do not log the keys but only a few child pointers such that the log size is kept minimal.

4 Experiments

In the experiments shown in Figure 1, we evaluate the insertion throughput of persistent indexing structures when we index 10 million records. We emulate the write latency of persistent memory by injecting `nop` instructions on Intel Xeon E7-4809 v3 processor. Our experimental results show that FAST and FAIR B+-trees outperform other state-of-the-art persistent indexing structures, as was reported in [3].

As for the performance of T-tree, T-tree shows comparable performances to FP-tree and wB+-tree although it is outperformed by FAST and FAIR B+-tree. We observe that T-tree accesses a fewer number of cachelines than FAST and FAIR B+-tree although its LLC miss ratio (38%) is higher than that of FAST and FAIR B+-tree (29%). Besides this, the cachelines that T-tree accesses are not adjacent to each other,

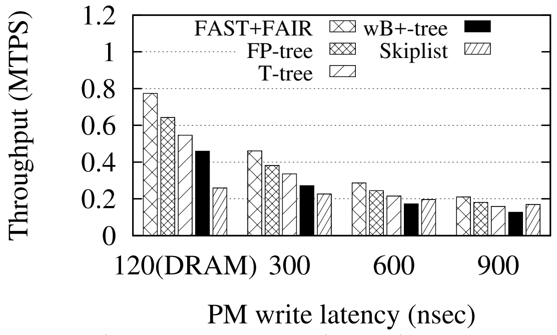

Figure 1: Insert : Varying Write Latency

hence it fails to leverage pipelining and memory level parallelism, which explains why it performs worse than FAST and FAIR B+-tree. Although T-tree performs worse than FAST and FAIR B+-tree, we note that the implementation of T-tree is much simpler than FAST and FAIR B+-tree.

5 Conclusions

In this extended abstract, we present FAST and FAIR B+-tree for byte-addressable persistent memory. FAST and FAIR B+-tree not only shows superior insertion and range query performance, but it also enables lock-free search. We also briefly present the performance of binary T-tree as a novel contribution, which we believe has a great potential to show good performance for the next-generation byte-addressable persistent memory.

6 Acknowledgements

This work was supported by the R&D program of NST (grant No. B551179-12-04-00) and ETRI R&D program (No. 18ZS1220), National Research Foundation of Korea (NRF) (No. NRF-2018R1A2B3006681 and NRF-2016M3C4A7952587), and Institute for Information & Communications Technology Promotion(IITP) (No. 2018-0-00549) funded by Ministry of Science and ICT, Korea.

References

[1] Chen, S., and Jin, Q. Persistent B+-Trees in non-volatile main memory. *PVLDB 8*, 7 (2015), 786–797.

[2] Seo, J., Kim, W. H., Baek, W., Nam, B., and Noh, S. failure-atomic slotted paging for persistent memory. *ACM ASPLOS* (2017).

[3] Hwang, D., Kim, W. H., Won, Y., and Nam, B. Endurable transient inconsistency in byte-addressable persistent memory. *USENIX FAST* (2018).

[4] Kim, W.-H., Seo, J., Kim, J., and Nam, B. clfB-tree: Cacheline Friendly Persistent B-tree for NVRAM. *ACM Transactions on Storage (TOS) 14*, 1 (2018).

[5] 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. *SIGMOD* (2016).

[6] Yang, J., Wei, Q., Chen, C., Wang, C., and Yong, K. L. NV-Tree: reducing consistency const for NVM-based single level systems. *USENIX FAST* (2015).