WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems

Se Kwon Lee*, K. Hyun Lim†, Hyunsub Song* and Beomseok Nam*, Sam H. Noh*
*Ulsan National Institute of Science and Technology, Korea
†Hongik University, Korea

I. Introduction

Emerging persistent memory technologies such as phase-change memory, spin-transfer torque MRAM, and 3D Xpoint are expected to radically change the landscape of various memory and storage systems [1]. In the traditional block-based storage device, the failure atomicity unit, which is the update unit where consistent state is guaranteed upon any system failure, has been the disk block size. However, as byte-addressable persistent memory will be accessible through the memory bus rather than via a PCI interface, the failure atomicity unit for persistent memory is generally expected to be 8 bytes or no larger than a cache line [1].

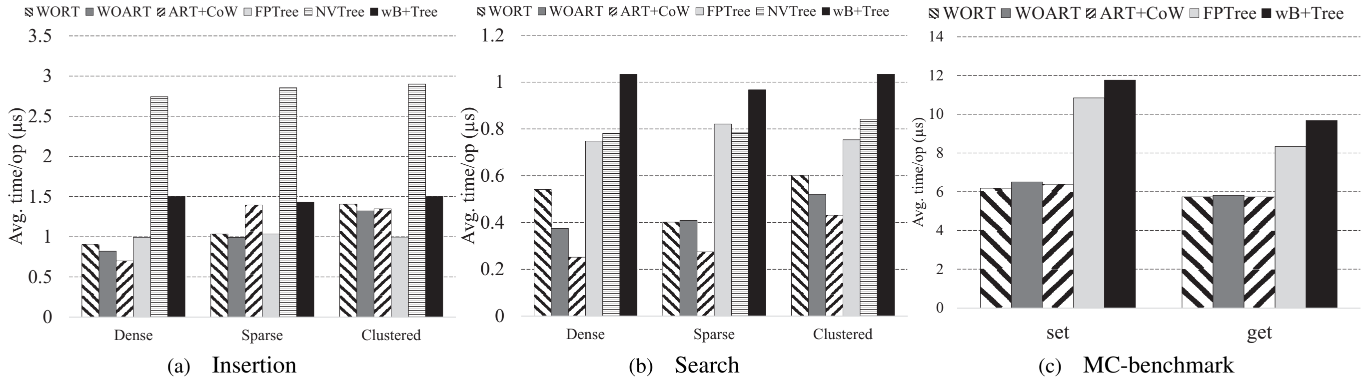
The smaller failure atomicity unit, however, appears to be a double-edged sword in the sense that though this allows for reduction of data written to persistent store as only dirty data need to be written, it can lead to high overhead to enforce consistency. This is because in modern processors, memory write operations are often arbitrarily reordered in cache line granularity and to enforce the ordering of memory write operations, we need to employ memory fence and cache line flush instructions [2]. These instructions have been pointed out as a major cause of performance degradation [3], [4]. Furthermore, if data to be written is larger than the failure atomicity unit, then expensive mechanisms such as logging or copy-on-write (CoW) must be employed to maintain consistency.

Recently, several persistent B-tree based indexing structures such as NVTree [3], wB+Tree [4], and FPTree [5] have been proposed. These structures focus on reducing the number of calls to the expensive memory fence and cache line flush instructions by employing an append-only update strategy. Such a strategy has been shown to significantly reduce duplicate copies needed for schemes such as logging resulting in improved performance. However, this strategy does not allow these structures to retain one of the key features of B-trees, that is, having the keys sorted. Moreover, this strategy is insufficient in handling node overflows as node splits involve multiple node changes, making logging necessary.

While B-tree based structures have been popular in-memory index structures, there is another such structure, namely, the radix tree, that has been less so. The first contribution of this paper is showing the appropriateness and the limitation of the radix tree for PM storage. That is, since the radix tree structure is determined by the prefix of the inserted keys, the radix tree does not require key comparisons. Furthermore, tree rebalancing operations and updates in node granularity units are also not necessary. Instead, insertion or deletion of a key results in a single 8-byte update operation, which is perfect for PM. However, the traditional radix tree is known to poorly utilize memory and cache space. In order to overcome this limitation, the radix tree employs a path compression optimization, which combines multiple tree nodes that form a unique search path into a single node. Although path compression significantly improves the performance of the radix tree, it involves node split and merge operations, which is detrimental for PM.

The limitation of the radix tree leads us to the second contribution of this paper. That is, we present three radix tree variants for PM. For the first of these structures, which we refer to as Write Optimal Radix Tree for PM (WORTPM, or simply WORT), we develop a failure atomic path compression scheme for the radix tree. For the node split and merge operations in WORT, we carefully add memory barriers and persist operations, minimizing the number of writes, memory fence, and cache line flush instructions in enforcing failure atomicity. WORT is optimal for PM, as is the second variant that we propose, in the sense that they require only one 8-byte failure-atomic write per update to guarantee the consistency of the structure.

The second and third structures that we propose are both based on Adaptive Radix Tree (ART) that was proposed by Leis et al. [6]. ART resolves the tradeoff between search performance and node utilization by employing an adaptive node type conversion scheme that dynamically changes the size of a tree node based on node utilization. This requires additional metadata and more memory operations than the traditional radix tree, but has still been shown to outperform other cache conscious in-memory indexing structures [6]. However, ART in its present form does not guarantee failure atomicity. For the second radix tree variant, we present Write Optimal Adaptive Radix Tree (WOART), which is a PM extension of ART. WOART redesigns the adaptive node types of ART and carefully supplements memory barriers and persist operations to prevent processors from reordering memory writes and violating failure atomicity. Finally, as the third variant, we present ART+CoW, which is another extension of ART that makes of Copy-on-Write (CoW) to maintain consistency. Unlike B-tree variants where CoW can be expensive, with the radix tree, we show that CoW incurs considerably less overhead.



(a) Insertion (b) Search (c) MC-benchmark

Fig. 1: Performance evaluation

II. Performance Evaluation

To test the effectiveness of WORT and WOART, we implement both radix trees and compare their performance with state-of-the-art PM indexing structures. The experiments are run on a workstation with an Intel Xeon E5- 2620 v3 2.40GHz X 2, 15MB LLC, and 256GB DRAM running the Linux kernel version 4.7.0. We compile all implementations using GCC-4.4.7 with the -O3 option.

For the workloads, we make use of three synthetically generated distributions of 8-byte integers. Unlike B-tree based indexes, the radix tree is sensitive to the key distribution due to its deterministic nature. To see how the indexes react to extreme cases, we consider three distributions. In Dense key distribution, we generate sequential numbers from 1 to 128M, so that all keys share a common prefix. In Sparse key distribution, keys are uniformly distributed, thus they share a common prefix only in the upper level of the tree structure. In Clustered key distribution, we merge Dense and Sparse key distributions to model a more realistic workload. Specifically, we generate 2 million small dense distributions, each consisting of 64 sequential keys. In Clustered key distribution, the middle level nodes share common prefixes.

A. Synthetic Workload

Figure 1a shows the average insertion time for inserting 128 million keys for the three different distributions. We see from the results that in general the radix based trees perform considerably better than the NVTree and wB+Tree. FPTree performs the best among the B-tree variants and, in some cases, better than the radix tree variants. However, this comparison must be made with caution as FPTree assumes that the internal nodes are in DRAM. The range of benefit and the best radix tree variant depends on the workload. Considering only the radix trees for the distributions in Figure 1a, we see that for Clustered distribution, insertion time is roughly 1.5x higher than for the other two distributions. This is due to the higher number of LLC misses incurred as the common prefix of the Clustered distribution is much more fragmented due to the scattered tree nodes than the other two distributions.

Figure 1b shows the average search time for searching 128 million keys for the three different distributions. We see that the radix tree variants always perform better than the B-tree variants. Notice that the depth of the tree is slightly higher for the radix tree variants. However, the number of LLC misses is substantially smaller, which compensates for the higher depth. The reason there is less LLC misses is because the radix tree can traverse the tree structure without performing any key comparisons, which incurs less pollution of the cache. Recall, that in constrast, B-tree variants must compare the keys to traverse the tree.

B. Experiments with Memcached

In order to observe the performance of our proposed index structures for real life workloads, we implement all the tree structures used in the previous experiments within Memcached [7]. We run mc-benchmark, which performs a series of insert queries (SET) followed by a series of search queries (GET) [8]. The key distribution is uniform, which randomly chooses a key from a set of string keys, whose size is 20 bytes. The left part of Figure 1c shows the results for SET operations. We observe that the radix tree variants perform considerable better than the B-tree variants by roughly 50%. The right part of Figure 1c shows the results for GET queries. Similarly to the SET query, the radix tree variants perform better than the B-tree variants.

References

[1] S. R. Dulloor et al. ”System Software for Persistent Memory.” in Proc. ACM EuroSys, 2014.
[2] Intel Corporation Intel® 64 and IA-32 Architectures Software Developer’s Manual. *http://www.intel.com/content/www/us /en/processors/architectures-software-developer-manuals.html*.
[3] J. Yang et al. ”NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems.” in Proc. USENIX FAST, 2015.
[4] S. Chen and Q. Jin. ”Persistent B+-Trees in Non-Volatile Main Memory.” in PVLDB, 2015.
[5] I. Oukid et al. ”FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory.” in Proc. ACM SIGMOD, 2016.
[6] V. Leis et al. ”The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases.” in Proc. IEEE ICDE, 2013.
[7] MEMCACHED What is Memcached? *https://memcached.org*.
[8] GitHub Memcache port of Redis benchmark *https://github.com/ antirez/mc-benchmark*.