Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory *

Pengfei Zuo, Yu Hua, Jie Wu
Huazhong University of Science and Technology

1 Motivation and Challenge

Non-volatile memory (NVM) as persistent memory is expected to substitute or complement DRAM in memory hierarchy, due to strengths of non-volatility, high density, and near-zero standby power. However, NVM typically suffers from the limited endurance and low write performance, compared with DRAM. With the significant changes of memory architectures and characteristics, traditional indexing techniques (fundamental building blocks for modern systems) originally designed for DRAM become inefficient in persistent memory, due to the requirement of data consistency and new NVM device properties.

A large amount of existing work has improved tree-based index structures for efficiently adapting to persistent memory, such as NV-Tree [6], FP-Tree [5], and FAST&FAIR [3]. Tree-based index structures are typically with the lookup time complexity of average O(log(N)) where N is the size of data structures. Unlike tree-based index structures, hashing-based index structures are flat data structures, which are able to achieve constant lookup time complexity, i.e., O(1). Due to providing fast lookup responses, hashing index structures are widely used in main memory systems. For example, they are fundamental components in main memory databases, and used to index in-memory key-value stores, e.g., Redis and Memcached. However, when hashing index structures are maintained in persistent memory, multiple non-trivial challenges exist which are rarely touched by existing work.

Challenge 1: *High Overhead for Consistency Guarantee.* Data structures in persistent memory should avoid any data inconsistency when system failures occur. However, the new architecture that NVM is directly accessed through the memory bus causes high overhead to guarantee consistency. Because we have to employ cache line flush and memory fence instructions, as well as expensive logging or copy-on-write (CoW) mechanisms.

Challenge 2: *Performance Degradation for Reducing Writes*. Memory writes in NVM consume the limited endurance and cause higher latency and energy than reads. Moreover, more writes in persistent memory also cause more cache line flushes and memory fences as well as possible logging or CoW operations. Hence, write reduction matters in NVM. Common hashing schemes such as chained hashing, hopscotch hashing and cuckoo hashing usually cause many extra memory writes for dealing with hash collisions. The write-friendly hashing schemes, such as PFHT [1] and path hashing [7], are proposed to reduce NVM writes in hashing index structures but at the cost of decreasing access performance.

Challenge 3: *Cost Inefficiency for Resizing Hash Table.* With the increase of the load factor of a hash table, the number of hash collisions increases, resulting in the decrease of access performance as well as insertion failure. Resizing is essential for a hash table to increase the size when its load factor reaches a threshold or an insertion failure occurs. However, resizing is an expensive operation due to incurring a large number of insertion operations and high latency, resulting in many NVM writes with cache line flushes and memory fences.

2 Our Approach

To address these challenges, this paper proposes *level hashing*, a write-optimized and high-performance hashing index scheme with low-overhead consistency guarantee and cost-efficient resizing for persistent memory. Specifically, level hashing provides *a sharing-based two-level hash table structure*, i.e., level hash table, to achieve high performance as well as high load factor, and rarely incurs extra writes (addressing ***Challenge 2***). Level hashing leverages *a cost-efficient in-place resizing scheme* to significantly improve the resizing performance (addressing ***Challenge 3***). Moreover, level hashing presents *(opportunistic) log-free schemes* to guarantee data consistency with low overhead (addressing ***Challenge 1***).

• ***Write-optimized Hash Table Structure.*** A level hash table is a new open-addressing structure with the strengths of memory-efficient, write-optimized, and high performance. As shown in Figure 1, the level hash table contains two-level buckets and each bucket has multiple slots. Only the buckets in the top level are addressable by hash functions. The bottom level is not addressable and used to provide standby positions for the top level to store conflicting key-value items. Each bottom-level bucket is shared by two top-level buckets. If a hash collision occurs in a top-level bucket and all positions in the bucket are occupied, the conflicting key-value item can be stored in its corresponding standby bucket in the bottom level. Moreover, we enable each key to have two hash locations via using two different hash functions. A new key-value item is inserted into the less-loaded bucket between the two hash locations. To enable

*Original Paper: Pengfei Zuo, Yu Hua, Jie Wu, "Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory", in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018, pages: 461 – 476. https://www.usenix.org/system/files/osdi18-zuo.pdf

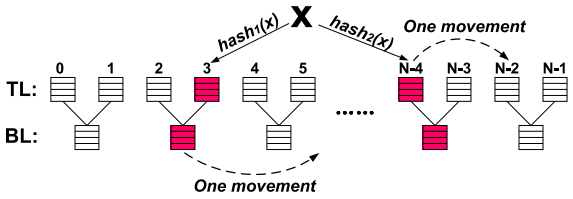

Figure 1: The hash table structure with 4 slots per bucket. *("TL": the top level, "BL": the bottom level.)*

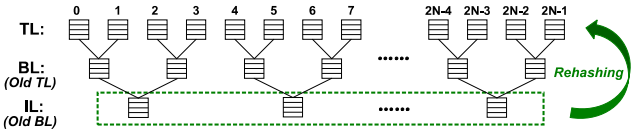

Figure 2: The in-place resizing scheme. *("IL": the interim level.)*

key-value items to be evenly distributed among buckets, if both buckets are full during inserting an item in each level, level hashing allows the movement of at most one item for each insertion.

• ***Cost-efficient In-place Resizing.*** To reduce NVM writes and improve the resizing performance, we propose *a cost-efficient in-place resizing scheme*. The basic idea is to put a new level on top of the old hash table and only rehash the items in the bottom level of the old hash table when expanding a level hash table, as shown in Figure 2. During the resizing, we first allocate the memory space with $2N$ buckets as the new top level and put it on top of the old hash table. The level hash table becomes a three-level structure during the resizing. The third level is called the interim level (IL). The in-place resizing scheme rehashes the items in the IL into the top-two levels. After all items in the IL are rehashed into the top-two levels, the memory space of the IL is reclaimed. After the resizing, the new hash table becomes a two-level structure again. We observe that the new hash table with $3N$ buckets is exactly double size of the old hash table with $1.5N$ buckets, which meets the demand of real-world applications. Moreover, the in-place resizing rehashes only the bottom level of the old hash table instead of the entire table. The bottom level only contains $1/3 (= 0.5N/1.5N)$ of all buckets in the old hash table, thus significantly reducing data movements and NVM writes during the resizing, as well as improving the resizing performance.

• ***Low-overhead Consistency Guarantee.*** To reduce the overhead of guaranteeing consistency, level hashing achieves log-free deletion, insertion, and resizing operations, and opportunistic log-free update operation, by leveraging the tokens to be performed in the atomic-write manner. A token associated with each slot is used to indicate whether the slot is empty in open-addressing hash tables. In a bucket, the header area stores the tokens of all slots and the remaining area stores the slots each with a key-value item. A token is one bit that indicates whether the corresponding slot is empty. Thus modifying the tokens only needs to perform an atomic write. Level hashing guarantees the consistency of write operations by leveraging the atomic write without the need of logging.

• ***Performance Evaluation.*** We have implemented level hashing and evaluated it in both real-world DRAM and simulated NVM platforms. Extensive experimental results show that the level hashing speeds up insertions by $1.4\times-3.0\times$, updates by $1.2\times-2.1\times$, and resizing by over $4.3\times$ while maintaining high search and deletion performance, compared with start-of-the-art hashing schemes including BCH [2], PFHT [1] and path hashing [7]. The concurrent level hashing improves the request throughput by $1.6\times-2.1\times$, compared with the start-of-the-art concurrent hashing scheme, i.e., libcuckoo [4].

3 Potential Impact

This paper will enable a broad impact. First, since the proposed level hashing achieves significant performance improvement over existing hashing schemes in both DRAM and NVM, the level hashing can replace existing hashing index mechanisms as a key building block in existing DRAM-based and future NVM-based main memory databases and key-value stores. Second, this paper gives the insights of several key challenges for hashing indexes in persistent memory and proposes the first persistent hashing index scheme, which will open the door to a new research topic about efficient hashing index structures for persistent memory. Third, relevant academic and industrial researches will benefit from our presented idea and the source code released at GitHub (https://github.com/Pfzuo/Level-Hashing) that has obtained over 50 stars in about three months.

References

[1] Debnath, B., Haghdoost, A., Kadav, A., Khatib, M. G., and Ungureanu, C. Revisiting hash table design for phase change memory. In *Proc. of INFLOW* (2015).

[2] Fan, B., Andersen, D. G., and Kaminsky, M. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proc. of NSDI* (2013).

[3] Hwang, D., Kim, W.-H., Won, Y., and Nam, B. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *Proc. of FAST* (2018).

[4] Li, X., Andersen, D. G., Kaminsky, M., and Freedman, M. J. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proc. of EuroSys* (2014).

[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. In *Proc. of SIGMOD* (2016).

[6] Yang, J., Wei, Q., Chen, C., Wang, C., Yong, K. L., and He, B. NV-Tree: reducing consistency cost for nvm-based single level systems. In *Proc. of FAST* (2015).

[7] Zuo, P., and Hua, Y. A write-friendly hashing scheme for non-volatile memory systems. In *Proc. of MSST* (2017).