Building Scalable Dynamic Hash Tables on Persistent Memory

Baotong Lu[1], Xiangpeng Hao[2], Tianzheng Wang[2], Eric Lo[1]
[1]The Chinese University of Hong Kong, [2]Simon Fraser University
[1]{btlu, ericlo}@cse.cuhk.edu.hk, [2]{xha62, tzwang}@sfu.ca

1 MOTIVATION

Dynamic hash tables are fundamental building blocks of many data-intensive systems such as database systems and key-value stores. Byte-addressable persistent memory (PM) such as Intel Optane DCPMM promises persistence, high capacity, low cost and high performance on the memory bus. These features make it very attractive to build hash tables that directly operate and persist on PM with high performance and instant recovery. Although there have been a new breed of hash tables [3, 6] specifically designed for PM, they were mostly based on DRAM emulation before actual PM was available, and we notice two critical issues when they are deployed on DCPMM: (1) they do not scale well, and (2) desirable properties are often traded off. The main culprit of low scalability is Optane DCPMM's limited bandwidth which is ~3–14× lower than DRAM's [5]. Notably, DCPMM exhibits very limited performance for small, random accesses, which are inherent access pattern for hash tables. Excessive PM accesses could easily saturate the system and prevent the system from scaling. We describe two major sources of excessive PM accesses that were not given enough attention before, followed by a discussion of important but missing functionality in prior work.

Excessive PM Reads. Much prior work focused on reducing writes and cacheline flushes to PM, however, we note that it is also imperative to reduce PM reads; yet many existing solutions reduce PM writes by incurring more PM reads. Different from the device-level behavior (PM reads being faster than writes), *end-to-end* write latency (i.e., the entire data path including CPU caches and write buffers in the memory controller) is often lower than reads [5]. In particular, existence checks in record probing constitute a large proportion of such PM reads: to find out if a key exists, one or multiple buckets (e.g., with linear probing) have to be searched.

Heavyweight Concurrency Control. Most prior work sidestepped the impact of concurrency control. Bucket-level read-write locking has been widely used [3, 6], but it incurs additional PM writes to acquire/release read locks, further pushing bandwidth consumption towards the limit.

Missing Functionality. We observe in prior designs, necessary functionality is often traded off for performance (though scalability is still an issue on real PM). (1) High load factor is important but often sacrificed by organizing buckets using larger segments in exchange for smaller directories (fewer cache misses) [3]. (2) Prior designs lack efficient support for variable-length keys. (3) Instant recovery is a unique, desirable feature on PM, but is often omitted in prior work which requires a linear scan of the metadata whose size scales with data size. (4) Prior approaches also often side-step the PM programming issues (e.g., PM allocation), which impact the proposed solution's scalability and adoption in practice.

* Published in VLDB 2020 [2]: http://www.vldb.org/pvldb/vol13/p1147-lu.pdf.
† Baotong Lu's work was partially performed at Simon Fraser University.

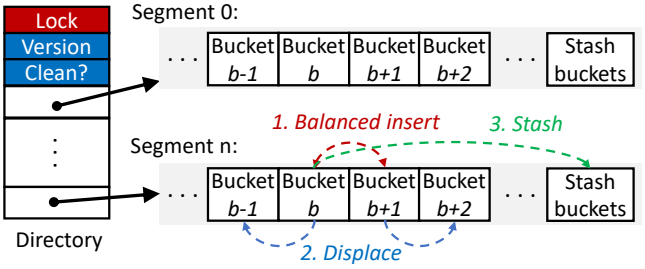

Figure 1: Overall architecture of Dash-EH.

2 DASH DESIGN

We present *Dash*, a holistic approach to dynamic and scalable hashing on real PM without trading off desirable properties. Dash uses a combination of new and existing techniques that are carefully engineered to achieve this goal. Techniques in Dash could be applied to various static and dynamic hash schemes. We focus on dynamic hashing and apply Dash to two classic approaches: extendible hashing and linear hashing, which are widely used in database and storage systems. We give an overview of Dash techniques in the context of extendible hashing (Dash-EH); more details on linear hashing can be found in our VLDB 2020 paper [2]. Our implementation is open-source at: https://github.com/baotonglu/dash.

2.1 Overview

Similar to prior work [3], Dash-EH groups buckets into *segments* to reduce directory size. As shown in Figure 1, each directory entry points to a segment which consists of a fixed number of normal buckets and stash buckets for overflow records from normal buckets which did not have enough space for the inserts.

2.2 Fingerprinting

A bucket in Dash consists of multiple key-value slots (4 cacheline in current implementation) to tolerate hash collisions. Bucket probing (i.e., search in one bucket) is a basic operation needed by all the operations supported in hash tables (e.g., search and insert) to check for key existence. However, linearly scanning the slots could incur lots of cache misses and is the major source of PM reads. We employ fingerprinting that was used in PM trees [4] to reduce PM accesses. Fingerprints are one-byte hashes of keys for predicting whether a key possibly exists; they are stored continuously at the head of each bucket. By first checking whether any fingerprint matches the search key's fingerprint, a probing operation only accesses slots with matching fingerprints, skipping all the other slots. Dash supports variable-length keys by storing pointers to them. Although dereferencing pointers could incur extra overhead, fingerprinting largely alleviates this problem by filtering out most unnecessary key probings.

2.3 Bucket Load Balancing

Segmentation reduces directory size (hence cache misses) at the cost of load factor: the entire segment needs to be split if any bucket is full even if other buckets in the segment still have much free space. To improve load factor, Dash uses a combination of techniques for new inserts to balance the load among buckets while limiting PM reads needed. Figure 1 lists three techniques Dash used.

Balanced Insert. To insert a record whose key is hashed into bucket b ($hash(key) = b$), Dash probes both buckets b and $b+1$ and inserts the record into the bucket that is less full (Figure 1 step 1). This improves load factor by amortizing the load of hot buckets while limiting PM accesses (at most two buckets).

Displacement. If both the target bucket b and probing bucket $b+1$ are full, Dash-EH tries to displace (move) an existing record from bucket b or $b+1$ to their neighbor buckets, making room for the new record (Figure 1 step 2). In essence, displacement follows a similar strategy to balanced insert, but is for existing records.

Stashing. If the insert cannot be accommodated with balanced insert displacement, we insert the record to one of the segment's stash buckets. While stashing can be effective in improving load factor, it incurs additional PM reads during probing. To reduce such accesses, we set up record metadata in a normal bucket and only refer actual record accesses to stash buckets. Our evaluation shows that using 2–4 stash buckets per segment can improve load factor to over 90% without imposing significant overhead.

2.4 Optimistic Concurrency

Dash employs optimistic locking, an optimistic flavor of bucket-level locking inspired by optimistic concurrency control. Insert operations will follow traditional bucket-level locking to lock the affected buckets. Search operations are allowed to proceed without holding any locks (thus avoiding writes to PM) but need to verify the read record. For this to work, in Dash the lock consists of (1) a single bit that serves the role of "the lock" and (2) a version number for detecting conflicts. This design saves PM writes during search operations, improving scalability.

2.5 Crash Consistency

Insert, delete and structural modification operations (SMOs) must guarantee consistency on PM for correct recovery. The typical approach is logging, which however is heavyweight. Instead of logging, Dash adds a bitmap in each bucket where each bit indicates the validity of a slot; accessing the bitmap only requires single-word atomic loads and stores. Specifically, the insert operation first write and persist the new record, and then atomically update the corresponding bit in bitmap. A crash happens after or before bitmap persistence indicates whether the insert operation succeeds or not. Delete operation simply reset the corresponding bit in bitmap.

A segment will be split when a thread exhausted all options to insert a new record. SMOs such as segment split must also be made crash consistent on PM while maintaining high performance. To achieve this, each segment has a `state` variable to indicate whether the segment is in an SMO and whether it is the one being split or the new segment. Upon recovery, the SMO process could be redo or undo by detecting the `state` variable and using side link to find the new segment.

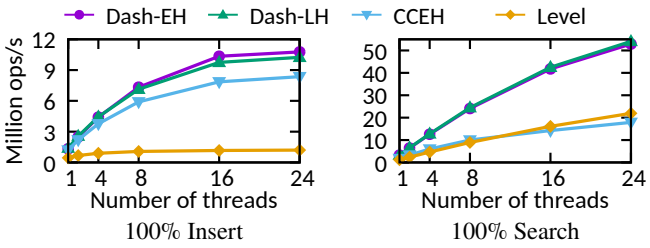

Figure 2: Throughput with a varying number of threads.

2.6 Instant Recovery

Dash provides truly instant recovery by requiring a constant amount of work (reading and possibly writing a one-byte counter), before the system is ready to accept user requests. We add a global version number V and a `clean` marker shown in Figure 1, and a per-segment version number. Upon system restart, if `clean` is not true (i.e., no clean shutdown), we increment V by one and start to handle requests. For both clean shutdown and crash cases, "recovery" only involves reading `clean` and possibly bumping V. The actual recovery work is amortized over segment accesses.

To access a segment, the thread first checks whether the segment version matches V. If not, the thread (1) recovers the segment to a consistent state (e.g., continue the ongoing SMO) before doing its original operation (e.g., insert), and (2) sets the segment's version number to V so that future accesses can skip the recovery pass.

3 EVALUATION AND CONCLUSION

We implemented Dash-EH and Dash-LH (linear hashing) using PMDK [1], which provides primitives for crash-safe PM management and synchronization. We conduct experiments on a server with 24-core, 768GB of Optane DCPMM (6×128GB DIMMs on all six channels) in AppDirect mode. We use uniformly distributed random 8-byte keys in our workloads.

Under single thread, Dash hash tables outperform state-of-the-art by 1.9×–2.4× for search operation as a result of reduced PM accesses. As shown in Figure 2, Dash also achieves near-linear scalability for search operations because of its lightweight concurrency control. For inserts, although neither Dash-EH nor Dash-LH scales linearly as inserts inherently exhibit many random PM writes, Dash is the most scalable solution, being up to 1.3× faster than CCEH [3].

REFERENCES

[1] Intel. 2018. Persistent Memory Development Kit. (2018). http://pmem.io/pmdk/libpmem/.
[2] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable Hashing on Persistent Memory. *Proc. VLDB Endow.* 13, 8 (2020), 1147–1161.
[3] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 31–44.
[4] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD*. 371–386.
[5] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies, FAST 2020*. 169–182.
[6] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 461–476.