Buffered Hash Table: Leveraging DRAM to Enhance Hash Indexes in the Persistent Memory

Chen Zhong, Prajwal Challa, Xingsheng Zhao, Song Jiang
University of Texas at Arlington
Arlington, TX
{chen.zhong, vxc5208, xingsheng.zhao}@mavs.uta.edu, song.jiang@uta.edu

***Abstract*—As a high-speed byte-addressable storage media similar to DRAM, Intel Optane DC Persistent Memory (PMem) has drawn the interest from the research community for its high throughput and low latency. These properties propel the migration of in-DRAM data structures, such as hash tables, to the PMem. However, existing PMem hash table designs do not recognize that the PMem is also a block device with an access unit of 256 bytes. Consequently, they carry out writes in sizes that are an order of magnitude smaller than the PMem access unit, leading to high write amplification. To improve their performance, we propose Buffered Hash Table (BHT) design. BHT batches multiple writes into in-DRAM buffers and then merges them into hash table buckets in the PMem, reducing the number of small writes. BHT also employs a PMem-based write-ahead log to prevent data loss. Our experiments show that BHT provides up to 2.3X and 2.8X higher write throughput, assuming the DRAM space is sufficiently available, compared to the state-of-the-art hash indexes, namely CCEH and Dash, respectively.**

***Index Terms*—Persistent Memory, Hash Table, DRAM Buffer**

I. Introduction

Intel Optane DC persistent memory (PMem), the first commercially available high-speed persistent byte-addressable device, has changed the paradigm on the distinct characteristic features of memory and storage devices. Packing both persistence and DRAM-like speeds, the PMem offers features that make it a viable media for storing a program's data structures to support its execution. It is appealing to migrate commonly used index structures, such as hash tables and B+ trees, that consume substantial portion of the memory space to the PMem. The PMem provides a cost-effective platform which is often of much larger capacity and less expensive compared to DRAM, while simultaneously allowing these index structures to survive a system crash. However, performance of the indexes is also critical. It has been reported that index operations may account for 14–94% of query execution time in today's in-memory databases [1] which makes performance of indexes in the PMem a crucial aspect to focus on.

As we know, traditional in-DRAM index designs cannot be used as-is in the PMem, as their underlying dynamic data structures do not guarantee crash consistency, which could leave them in an inconsistent state that cannot be recovered after a crash. Consequently, much effort has been directed towards designing crash-consistent indexes, such as CCEH [2], Level Hashing [3], wB+ tree [4], and FAST & FAIR [5] etc. Many of them were proposed before the release of the Optane PMem and the assumption in their designs is that persistent memory is a "slower but persistent DRAM" [6].

It has been revealed that the Optane DC persistent memory accesses data from/to its 3D-Xpoint media at a 256-byte granularity (XPLine) in recent performance characterization studies [6]–[8]. Any write smaller than 256 bytes becomes a read-modify-write operation and results in a 256-byte write to the physical media, causing potentially large write amplification. It has also been observed that unless there are sequential and large writes, the PMem's write performance can be much worse than its reported peak performance. This PMem write performance issue is specially pronounced for writes to a hash table, where writes are inherently small and are designed to be hashed uniformly to the its buckets.

To tackle the write amplification issue in existing PMem hash index structures, we propose BHT, a DRAM-PMem hybrid hash table design by incorporating in-DRAM write buffers to improve performance of the in-PMem index structures. Instead of redesigning a new hash table, BHT enhances an existing in-PMem hash table design with auxiliary in-DRAM write buffers to reduce write amplification by coalescing writes in the buffers. Each buffer is responsible for buffering key-value (KV) items into a bucket or a segment of buckets. BHT also maintains a PMem-based write-ahead log for recovering lost KV items in the buffers in case a system crash happens.

In this paper we implement BHT on a state-of-the-art PMem hash table design, named Cacheline-Conscious Extendible Hashing (CCEH) [2]. CCEH has a carefully designed crash-consistency scheme for efficiency and minimizes disruptive performance impact of hash table rehashing by employing local hash segment resizing. However, without considering Optane PMem's block-like performance behavior, CCEH's writes suffer from a very large write amplification. Note that as reads are assumed to take place randomly across the hash table buckets, BHT is not designed for improving read performance. A separate in-DRAM read cache may be deployed for improving read performance if there exists a relatively small set of frequently accessed KV items. However, such a read cache is not in the scope of this study. The write buffers may be dynamically applied only during write-intensive execution period. We have implemented BHT on Intel Optane DC PMem and extensively compare its performance with state-of-the-art indexes, namely CCEH and Dash [9]. Our evaluation results show that BHT's throughput performance outperforms

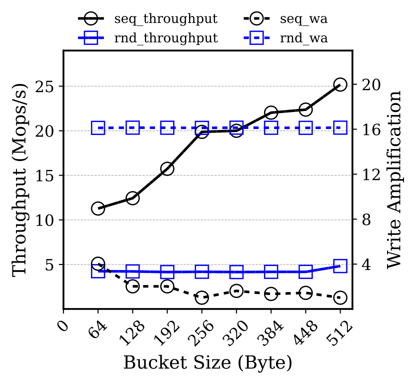


Fig. 1: Throughput and WA with different bucket sizes for sequential and random insertions in the hash table.

both CCEH and Dash by 2.3X and 2.8X, respectively, when sufficient DRAM space is provided as write buffers.

This paper is organized as follows. Section II introduces the motivation for integrating DRAM buffers into a in-PMem hash index structure and the background of PMem hash indexes. Section III presents the BHT design. Section IV describes our evaluation setup and results in comparison to state-of-the-art indexes. Section V discusses related work. And Section VI concludes the paper.

II. Motivation and Background

This study is motivated by observations of large performance variations with difference access patterns on the PMem.

A. *Performance Impact of Access Patterns*

To observe performance behaviors of a hash table in the PMem, we design an experiment in which a bucket-based hash table in the PMem is populated using two different methods. In the first method, a sequence of keys are hashed into different buckets until all buckets are full. That is, the buckets are written in a random order. In the second method, the sequence of keys are carefully arranged so that the buckets are filled sequentially one at a time. That is, keys are written into each bucket continuously and sequentially. In the hash table, there are three million buckets, each as a pre-allocated fixed-size memory space. Each write includes a 8-byte key and a 8-byte value. We use xxHash [10] as the hash function. Figure 1 shows throughput of these two population methods (either random or sequential writes) with different bucket sizes. In the experiment we also measure amount of data written to the PMem's media using the `ipmctl` tool. We define write amplification (WA) as the ratio of amount of data written to the media and amount of program data written to the hash table.

As shown in the figure, the throughput of sequential writes is about 2.2X-5.1X as high as that of random writes. Note that, as a hash function always attempts to scatter keys evenly across different buckets, the random-write throughput represents the normal performance one would expect on an in-PMem hash table. Correspondingly, the random writes have a consistent 16X WA (256B/16B). As the data set is much larger than PMem's 16KB internal write-back buffer, each random and small write has to be individually written back to the PMem's media in a block. In contrast, sequential writes have much smaller WAs (from 1X to 5X). When the bucket size is a multiple of blocks (e.g., 256 bytes or 512 bytes), the WA is as low as 1. The WA becomes larger when the bucket size is more distant from its next multiple of 256 bytes. The sequential-access throughput increases with the increase of bucket size. For sequential accesses, PMem's internal prefetching mechanism is activated to accelerate the read-modify-write operations. Therefore, it is critical to use sequential and large writes for PMem's high performance. The potential of performance improvement by organizing small writes into larger ones is very large.

B. *In-PMem Extendible Hash Table*

As hash tables naturally receive random writes into its buckets and its in-PMem variants have a large performance improvement potential, we focus on addressing the issue of small writes in hash tables in this work. In particular, we develop our design on a highly optimized hash table – Extendible Hashing [11]. Unlike traditional hashing, extendible hashing avoids a full table rehashing. Instead, when a collision occurs at a filled bucket, it carries out rehashing locally. Extendible hash table is divided into two layers. The first layer is a bucket address table named the directory. Entries of the directory are indexed by either MSBs (Most Significant Bits) or LSBs (Least Significant Bits) of keys and store the pointers to the buckets. The Global Depth, G associated with each directory entry, represents the number of bits that are used to categorize the hash buckets, allowing for a maximum of 2^G buckets. The Local Depth is used to decide the operation (rehashing) to be performed in case an overflow occurs. Extendible hashing avoids full table rehashing by only updating the pointers of directory entries and rehashing individual buckets.

C. *Cacheline-Conscious Extendible Hashing*

Cacheline-Conscious Extendible Hashing (*CCEH*), is a state-of-the-art extendible hash table design for persistent memory. It inherits the design of extendible hashing. Knowing that each memory access is due to a miss at the CPU cache and therefore is at the unit of cacheline (64B), it sets up each bucket's capacity at 64 bytes. In the extendible hashing, as number of buckets increases, corresponding increase of directory size is memory-consuming. CCEH proposes an intermediate layer between directory layer and buckets. This new layer uses a collection of buckets, referred as *segment*, to manage data, aiming to reduce the number of directory entries.

CCEH uses a three-level structure design where each entry of the directory stores a pointer to a segment that consists of a fixed number of buckets. Each bucket's size is aligned with cacheline size of 64B where up to 4 key-value items can be stored for high memory-level parallelism. Segments are indexed by G MSB bits of a hash key. Even with the presence of a three-level structure, this indexing approach allows CCEH to use L LSBs as a bucket index to locate the bucket in a segment directly.

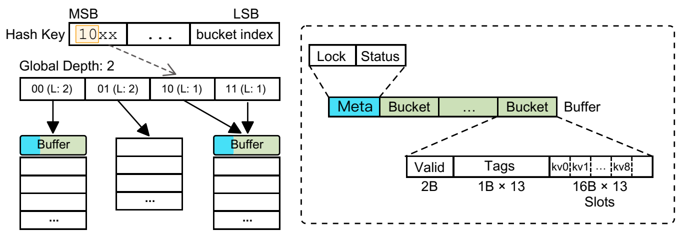


Fig. 2: Illustration of structure of Buffered Hash Table

When a hash collision in a bucket occurs, CCEH allocates a new segment and then copies data from overflowed bucket's segment to the new segment based on the MSB. To avoid unnecessary copy-on-writes and split overhead, CCEH adopts *lazy deletion* upon segment split where migrated data is not immediately deleted after segment split in the old segment. This data instead will be labeled as invalid and is overwritten in subsequent insert transactions to reduce number of writes. Since segment is a collection of multiple extendible hashing buckets, it is very likely that the segment upon a single bucket overflow will be split even though other buckets in the same segment are not yet full. To improve space utilization, CCEH uses linear probing. To guarantee index structure's crash consistency, CCEH uses 8-byte write atomicity and flush/fence operations so that index's integrity will not be compromised by a system crash.

While CCEH adopts a Cacheline-Conscious structure for cache efficiency, carefully arranges write sequence for crash consistency, and judiciously applies locks for high concurrency, it misses a significant performance optimization opportunity – high efficiency with the PMem's block write. This is an issue shared by other existing index structures. Specifically, small writes, each of a 16-byte KV item, are uniformly distributed into a large number of segments in CCEH. Keys are further uniformly hashed to buckets in the segment. These are all random small writes, representing Achilles' heel of the PMem's write performance.

III. The Design

In this section, we present BHT (Buffered Hash Table), an optimized CCEH hash table design that leverages DRAM buffers to avoid small random writes into CCEH's segments for high write efficiency. While this idea seems to resemble use of write-back buffer in an OS for improving disk's write performance, the buffering service for optimizing PMem's write performance cannot be provided in an OS. Different from the disk, whose access has to be via OS's I/O layer, the PMem is a byte-addressable device and is designed to be accessed directly by user programs. It becomes prohibitive for OS to be involved in the management of the write buffers.

Therefore, the design of the buffering component is integrated into the hash-table structure. In the meantime, BHT represents an example design for adding the buffer-based optimization on an existing hash table. The design is intended to lead to a general approach that can be similarly and conveniently applied to other data structures. Therefore, a major BHT 's design principle is that addition of the buffering function is non-disruptive to the existing CCEH data structure design.

A. *Integrating Buffers to CCEH*

To follow the principle of minimizing modification of existing data structures, the design identifies contiguous memory space where (time-wise) non-continuous and/or (space-wise) non-contiguous accesses frequently occur, and assigns a write buffer to turn them into one or multiple sequential block writes. It does not attempt to change memory layout of the existing in-PMem data structure for this purpose, as it could demand substantial modifications of the data structure, which could be in conflict with its other design objectives.

For the CCEH hash table, each segment consisting of multiple buckets represents a contiguous memory chunk receiving random accesses. Accordingly, BHT associates a buffer to one segment. It does not choose to attach buffers to individual buckets because a bucket can be too small (64 bytes by default) to effectively reduce write amplification.

By (logically) attaching a write buffer to a segment in CCEH, BHT requires a few straightforward changes to the CCEH's operations. Any writes to a segment will be first received into the segment's buffer, instead of directly into the in-PMem segment. For a delete or update request, if the corresponding key is not currently in the buffer, a new record is inserted into the buffer. In particular, for delete the record is a special tombstone indicating the key has been deleted. A read will also first check into the buffer, and then into the segment if it's a miss. When the buffer is filled, all the KV items in the buffer are written to the segment in a batch, an operation named *flip*. In particular, during a flip all KV items in the segment and those in the buffer are read and merged to form a segment in the DRAM (and occasionally two segments if collisions cannot be resolved within one segment, as CCEH does for splitting a segment). A delete or update is materialized at this moment. The segment is then written to the PMem as a new segment, rather than overwriting existing one. This batched out-of-place write enables not only a highly-efficient sequential write, but also concurrent reads during a flip operation. BHT sets up a lock for each segment to prevent writes into a segment and its buffer when they are involved in a flip. However, reads to the segment and the buffer are allowed during the flip because both the buffer and the segment are not being modified.

Note that not every segment is necessarily equipped with a buffer as we assume a limited DRAM budget for this purpose. Any accesses to non-buffered segments are carried out in the same way as CCEH does. As the main purpose of the buffer is to transform random writes into batched sequential writes, it has side effects on read requests. On one hand, a read might find its requested data in the buffer holding recently written KV items without accessing the slower PMem. On the other hand, a read miss carries performance penalty, especially when access locality is weak and the workload shifts to a read-intensive phase. To address the issue, BHT organizes KV items in the buffer into a hash table for quick search. Each buffer is associated with a 1-byte counter tracking number of read misses since the last flip of the buffer. This counter is reset to zero after a flip. It is incremented by one with each read miss. When the counter overflows, the buffer is flipped (even if it is not yet filled). In this way, when the buffer does not help improve the performance by having enough read hits, it will be forced to become empty. Each buffer is associated with a 1-bit flag indicating if it is empty. A read operation into a segment will first check its corresponding bit and access its buffer only when it is necessary. That is, if a workload consists mostly of reads, BHT does not degrade its performance.

Figure 2 illustrates the structure of a buffered CCEH hash table. In the table, two segments are selected to have write buffers. Similar to the original CCEH, a key is hashed and its MSB is used to identify a directory slot, where pointers to the segment and the buffer are stored. A buffer contains some metadata about the buffer, including a *lock* bit for access concurrency control and a *status* bit indicating if the buffer is empty. As each buffer is organized as a hash table, KV items are placed into these buckets. In the example shown in the figure, each bucket is 256 bytes long and can hold up to 13 KV slots, each for a 16-byte KV item (a 8-byte key and a 8-byte value, as assumed in the CCEH design [2]). Each bucket also has a valid bit map, where each bit indicates if its corresponding slot contains valid KV item. Furthermore, to speed up searching keys in a bucket, we generate a 1-byte-tag array by hashing each key in the bucket into a tag. BHT then uses an SIMD instruction ("*_mm_cmpeq_epi8_mask*"), which is widely available in today's processors, for a quick search of the tags in parallel to screen out non-matching keys.

Admittedly, BHT requires additional DRAM space to receive its performance improvement. Considering the fact that real-world systems with the PMem are most likely installed with a certain amount of DRAM [12], BHT makes it possible to leverage a portion of the DRAM to close the performance gap between DRAM and PMem and to make PMem more amenable as an alternative of DRAM. However, the amount of DRAM space available for this purpose can be limited. To understand the impact of DRAM usage, assume a segment size of N bytes, a buffer size of n bytes, and (small) m-byte KV items. Each buffer flip reduces write amplification (WA) from $256/m$ to N/n for n KV items. As an example, the segment size is 16KB, buffer size is 4KB, and 16B KV items, a flip reduces WA from 16 to 4 for 256 KV items.

B. Avoiding Loss of Data Buffers

To prevent KV items in the DRAM buffers from getting lost after a system crash, BHT adopts the WAL (write-ahead log) approach that is widely used in the design of KV stores, such as LevelDB [13], [14] and RocksDB [15], [16]. In this approach, any new inserts (including updates) that are sent to the buffers are also appended at the tail of the in-PMem log. For a delete request, a special tombstone record is appended to the log. KV items are appended to the log once they are written to DRAM. In traditional log-based storage systems, small writes are first buffered in the DRAM until the resultant data size matches the underlying storage system's access granularity and is then written in an append-only format. Doing so converts multiple random writes to a sequential write so as to extract performance from the storage medium, while avoiding high write amplification. However, to write small data to a log in the PMem, a DRAM buffer that aligns with the size of PMem access unit is not required. Random writes to a log are transformed to sequential writes automatically by the PMem's internal write-combining buffer which is of 16KB in size. Consequently, once an item is written to the DRAM, the item is immediately written to the log thereby avoiding a possible scenario of data loss due to a crash before data reaches the threshold size to be written to PMem.

To confine the size of the log, KV items in the buffer that have been persisted to the PMem are garbage-collected in the log. As the buffer in BHT fills up, it is flipped with all of its KV items written to the PMem. These KV items are then considered as garbage in the log. To enhance the efficiency of garbage collection, BHT tracks the oldest non-garbage KV item for each buffer. Assuming that there is an insert or delete request sent to the emptied buffer after its flip, the corresponding log record for the new request becomes this buffer's oldest one in the log. To carry out a space reclamation on the log, BHT only needs to track the oldest record among all buffers' oldest ones in the log. Any records before the record can be all safely removed.

IV. Evaluation

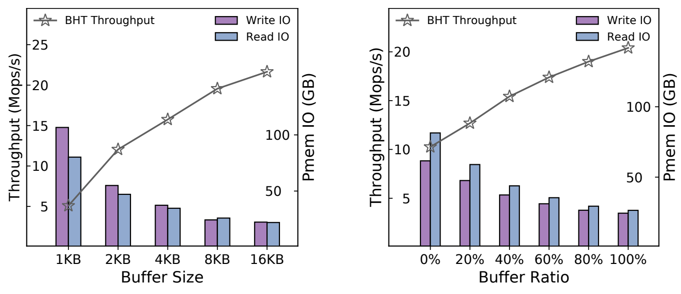
To understand performance of BHT , we implement it on a state-of-the-art PMem hash index, CCEH, and evaluate it on the Intel Optane DC Persistent Memory. We refer to this CCEH implementation simply as BHT from now on. Our implementation is based on a newer CCEH implementation provided by its initial authors [17]. We break down the performance factors and show their effects on our design.

A. Evaluation Setup

All the experiments are run on a server with the 1st generation Optane DCPMM that has dual Intel Xeon Gold 6320 2.1GHz CPUs (20-core) with 1.3MiB L1i/L1d cache, 40MiB L2 cache, and 55MiB L3 cache. It is equipped with 188GB DRAM and 6 × 128GB Optane DCPMMs. All the threads in an experiment are pinned to cores using `numactl`. A uniformly distributed write workload of 120 Million KV items is employed in all of the experiments, where key and value are of 8 bytes each.

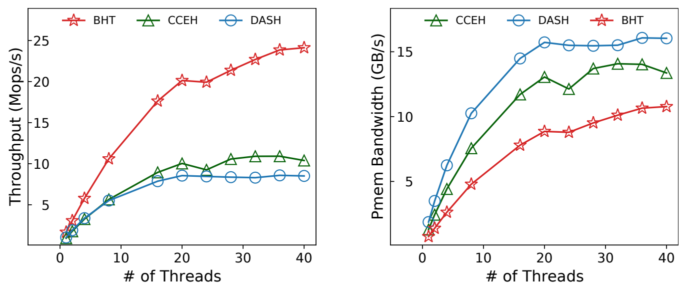
B. Impact of buffer Size

In order to understand the impact of adding in-DRAM buffers to CCEH, we investigate the performance of BHT with varying buffer sizes. Every segment of CCEH in the PMem is equipped with a buffer in the DRAM. A fixed number of threads (20) is used in this experiment. Figure 3a depicts the throughput and the total amount of read and write from the PMem's media with varying buffer sizes. As each



(a) Varying Buffer Size (b) Buffers to Segments Ratio

Fig. 3: BHT's throughput and PMem I/O with 20 threads and varying (a) buffer size (b) ratio of buffer to segment counts



(a) Write throughput (b) I/O Bandwidth

Fig. 4: Throughput and PMem write bandwidth.

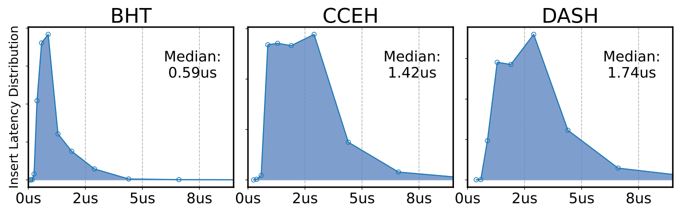


Fig. 5: Insert latency distribution for BHT, CCEH and Dash.

PMem segment's buffer size increases, writes to the segment become more sequential (in larger batches), and throughput of BHT increases. This result is consistent with the observation shown in Figure 1 where larger bucket sizes correspond to higher throughput. It is also noted that even though the workload in the experiment is purely write, we do have both PMem write and read I/O. This is resultant of BHT design of managing PMem data in the read-modify-*batched-out-of-place-write* manner. An increase in buffer size helps transform higher number of random writes into batched out-of-place sequential writes. A reduction in number of in-place read-modify-write operations is translated into lower amount of total PMem read and write to the PMem media. The reduction in the PMem I/O observed when the buffer size is increased from 8KB to 16KB is little compared to amount of additional DRAM space allocated. So as to effectively utilize the DRAM, we fix BHT's buffer size at 8KB in the following experiments.

C. *Impact of Number of Buffers*

It is enticing to consider allocating a buffer to every segment in the PMem for better throughput and overall lower I/O traffic. However, DRAM is a heavily contested resource by applications and OS as well. In production systems it may not be feasible to allocate large amounts of DRAM to the PMem. To understand BHT's performance in a DRAM-constrained system we evaluate the performance of BHT under varying amount of DRAM. Figure 3b shows the performance of BHT for different number of buffers (as a proportion of segment count with the hash table at the time when all of the 120 million of KV items have been inserted.). If each segment is allocated with a buffer (the 100% ratio), the highest throughput is achieved. However, it can be seen that even with only 40% of all segments are allocated with buffers, BHT achieves around 1.6X higher throughput than an index without any buffers (equivalent to the original CCEH). When we compare the amount of write I/O to the PMem media at the 0% buffer ratio and that at the 100% ratio, the WA is reduced by 2.1X.

D. *Comparison with State-of-the-arts*

To investigate performance advantage of BHT, we compare it with two state-of-the-art PMem indexes (CCEH [2] and Dash [9]). Dash is a dynamic PMem hash table design that incorporates techniques like fingerprinting to achieve high performance. For accurate comparison we use same configurations for BHT and CCEH and we use default configurations of Dash. BHT is configured to allocate a buffer for each segment. Figure 4a shows their throughput with different number of threads. It can be observed write throughput of BHT outperforms CCEH and Dash by a factor of 2.3X and 2.8X at 40 threads, respectively. BHT's throughput is higher than other two indexes even at low thread counts. This is a consequence of CCEH and Dash's design to receive updates onto PMem in an in-place small-write manner, leading to large number of random writes. BHT on the other hand uses buffers to batch writes to the PMem avoiding high write-amplification and writes to the PMem in an out-of-place manner so as to extract the PMem's peak sequential write rate. Increase in random writes at higher thread count has a more pronounced effect on performance loss for CCEH and Dash.

To reveal insights behind the performance gaps, we measure the bandwidth, defined as amount of I/O on the PMem media per second, in the service of write requests on the hash indexes shown in Figure 4a and show it in Figure 4b. The ratio between throughput for an index at a certain number of threads and its corresponding I/O bandwidth indicates the I/O amplification. As BHT has consistently much higher throughput, it has much lower I/O bandwidth. Each small write in CCEH or Dash triggers a read-modify-update of XPLine in the PMem internally, incurring very high consumption of I/O bandwidth. BHT batches writes which are aligned with the PMem access unit to have minimal number of PMem writes for small data, avoiding expensive bandwidth consumption by the PMem.

Figure 5 shows the write latency of the indexes at a fixed thread count of 20 threads. By using buffers to first receive KV items and keeping PMem's bandwidth low, most of BHT's write latency is distributed at lower values compared to that of CCEH and Dash.

V. Related Works

As in-memory indexing techniques are not originally designed to provide crash consistency, new index structures have been developed for non-volatile memory while keeping failure-atomicity in mind. Unfortunately, none of them, in particular hash table indexes, has considered the block-access property in Intel Optane DC PMem and exploited the performance improvement opportunity.

Path Hashing [18] is a hashing scheme designed for PMem that uses a logical inverted binary tree structure for organizing and servicing data. A double hashing approach is used for better space utilization. Path Hashing resolves collisions by rehashing its original table to a new table that is twice the size. Level hashing [3] is an advancement of path hashing where two hash tables are managed, each at a different level having independent hash functions. Dash [9] is a dynamic PMem hash table design that incorporates techniques like fingerprinting and optimistic locking for higher performance. Data is managed in a fashion similar to cuckoo hashing where two adjacent buckets are probed. Hash collisions in segments are managed by placing data in a separate stash buckets. Once stash buckets overflow a rehash is triggered. In these index designs, a PMem is treated simply as a byte-addressable device. And small writes are allowed to directly go to it for immediate service without any efforts on improving their spatial locality. This critical issue is addressed in BHT.

VI. Conclusion

In this paper, we propose a hash index design, BHT (Buffered Hash Table), that integrates DRAM to an existing hash table structure to tackle the issue of the hash index's intrinsic small random writes on Intel Optane DC persistent memory, a device internally with a 256B access unit. Using DRAM as a staging area to coalesce writes and later merging them helps convert small random writes to a large sequential write to the PMem. For high data durability even though writes are placed in the DRAM, BHT employs a write-ahead log to ensure the index structure can survive a system crash. We implement our design into a state-of-the-art index structure, CCEH, to demonstrate its performance advantage. In the meantime, this design approach can be applied similarly to other PMem hash indexes and likely other index structures such as B+ tree and skip lists. Evaluations show that BHT outperforms state-of-the-art indexes, including CCEH and Dash, by a factor of up to 2.3X and 2.8X, respectively.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work is supported by the National Science Foundation under grant CCF-1815303.

References

[1] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers: Accelerating index traversals for in-memory databases," ser. MICRO-46. New York, NY, USA: Association for Computing Machinery, 2013, p. 468–479. [Online]. Available: https://doi.org/10.1145/2540708.2540748

[2] M. Nam, H. Cha, Y. ri 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)*. Boston, MA: USENIX Association, Feb. 2019, pp. 31–44. [Online]. Available: https://www.usenix.org/conference/fast19/presentation/nam

[3] 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.

[4] S. Chen and Q. Jin, "Persistent b+-trees in non-volatile main memory," *Proc. VLDB Endow.*, vol. 8, no. 7, pp. 786–797, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol8/p786-chen.pdf

[5] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in Byte-Addressable persistent B+-Tree," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*. Oakland, CA: USENIX Association, Feb. 2018, pp. 187–200. [Online]. Available: https://www.usenix.org/conference/fast18/presentation/hwang

[6] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 169–182. [Online]. Available: https://www.usenix.org/conference/fast20/presentation/yang

[7] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao, "Characterizing and modeling non-volatile memory systems," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 496–508.

[8] A. Kalia, D. Andersen, and M. Kaminsky, "Challenges and solutions for fast remote persistent memory access," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 105–119. [Online]. Available: https://doi.org/10.1145/3419111.3421294

[9] B. Lu, X. Hao, T. Wang, and E. Lo, "Dash: Scalable hashing on persistent memory," *Proc. VLDB Endow.*, vol. 13, no. 8, pp. 1147–1161, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p1147-lu.pdf

[10] "xxhash - extremely fast non-cryptographic hash algorithm," https://cyan4973.github.io/xxHash/, (Accessed on 02/26/2022).

[11] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible hashing—a fast access method for dynamic files," *ACM Trans. Database Syst.*, vol. 4, no. 3, p. 315–344, sep 1979. [Online]. Available: https://doi.org/10.1145/320083.320092

[12] H. T. Kassa, J. Akers, M. Ghosh, Z. Cao, V. Gogte, and R. Dreslinski, "Improving performance of flash based Key-Value stores using storage class memory as a volatile memory extension," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 821–837. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/kassa

[13] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996. [Online]. Available: https://doi.org/10.1007/s002360050048

[14] S. Ghemawat and J. Dean, "Leveldb," 2011. [Online]. Available: https://github.com/google/leveldb

[15] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum, "Optimizing space amplification in rocksdb," in *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, 2017. [Online]. Available: http://cidrdb.org/cidr2017/papers/p82-dong-cidr17.pdf

[16] F. R. Team, "A persistent key-value store for fast storage environments," 2021. [Online]. Available: http://rocksdb.org

[17] M. Nam and H. Cha, "CCEH," https://github.com/DICL/CCEH, 2020.

[18] P. Zuo and Y. Hua, "A write-friendly and cache-optimized hashing scheme for non-volatile memory systems," *IEEE Trans. Parallel Distributed Syst.*, vol. 29, no. 5, pp. 985–998, 2018. [Online]. Available: https://doi.org/10.1109/TPDS.2017.2782251