IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received November 30, 2021, accepted December 15, 2021, date of publication December 20, 2021,
date of current version December 28, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3136459

Analysis and Optimization of Persistent Memory Index Structures' Write Amplification

YOUNGJOO WOO[1], TAESOO KIM[2], SUNGIN JUNG[1], AND EUISEONG SEO[3], (Member, IEEE)

[1]Electronics and Telecommunications Research Institute (ETRI), Daejeon 34129, Republic of Korea
[2]Georgia Institute of Technology (Georgia Tech), Atlanta, GA 30332, USA
[3]Department of Computer Science and Engineering, Sungkyunkwan University, Suwon 16419, Republic of Korea

Corresponding author: Euiseong Seo (euiseong@skku.edu)

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) Grant funded by the Korea Government (MSIT) under Grant 2014-3-00035 (Research on a High Performance and Scalable Manycore Operating System) and Grant 2021-0-00773 (Research on Edge-Native Operating Systems for Edge Micro-Data-Centers).

ABSTRACT Using persistent memory (PM) for an in-memory database (IMDB) system significantly extends data processing capacity and reduces service downtime for the recovery process in case of system failure. However, it requires an index structure tailored for the characteristics of PM to recover the consistent state from system failure. The processor flushes data with the cache line size to the memory controller, whereas the PM media writes with the media-write unit size, which is four times larger than the cache line size. This mismatch in write sizes amplifies the quantity of data written by an index structure at the physical media level. Additionally, the PM-aware index structures explicitly and frequently flush cache lines to ensure memory write orders, which worsen the write amplification factor. The write amplification not only degrades the performance but also shortens the lifespan of PM media. This study analyzes the write amplification and performance of various PM-aware index structures at the memory controller and physical media levels. Furthermore, this paper proposes four optimization techniques based on our observation: removal of unintended bloat, cache flush coalescing, early eviction of volatile data, and XPLine-aware unit sizing. Our analysis demonstrated that up to 3.9 times as many writes could occur depending on the choice of the index structure. We also revealed that the proposed optimization techniques could save up to 51% of physical media write operations and improve throughput by up to 63%.

INDEX TERMS Persistent memory, index structure, write amplification, in-memory databases.

I. INTRODUCTION

An in-memory database (IMDB) system can considerably benefit from using persistent memory (PM) as a part of its main memory. When using PM, a single server can serve a more extensive database. Additionally, frequent backup to the secondary storage for failure recovery is unnecessary since data stored in PM immediately becomes persistent. Since the latest state remains in the main memory after a system failure, the service downtime for the recovery from failure can be significantly reduced. For example, the recovery time of a database could be reduced to 35 from 4745 s by supplementing PM to a DRAM-only system [1].

To recover from system failure, the index structure of an IMDB should be able to reconstruct the consistent state from the remaining data in memory at any point. For this, several PM-aware index structures have been proposed [2]–[18]. However, the PM-aware indexes commonly send writes down to the PM using the explicit cache flush and memory fence instructions to ensure write-ordering whenever the structure is modified. Because of this, in comparison to the index structures designed for DRAM, which try to maximize the cache utilization by reducing cache flushes, more memory writes are inevitable when using the PM-aware index structures.

In addition, write amplification can occur inside the PM module because the size of a media access unit is four times larger than that of the unit request sent from the processor. The significant write amplification at the PM media level increases the degree of congestion in the PM controller [19],

The associate editor coordinating the review of this manuscript and approving it for publication was Qichun Zhang.

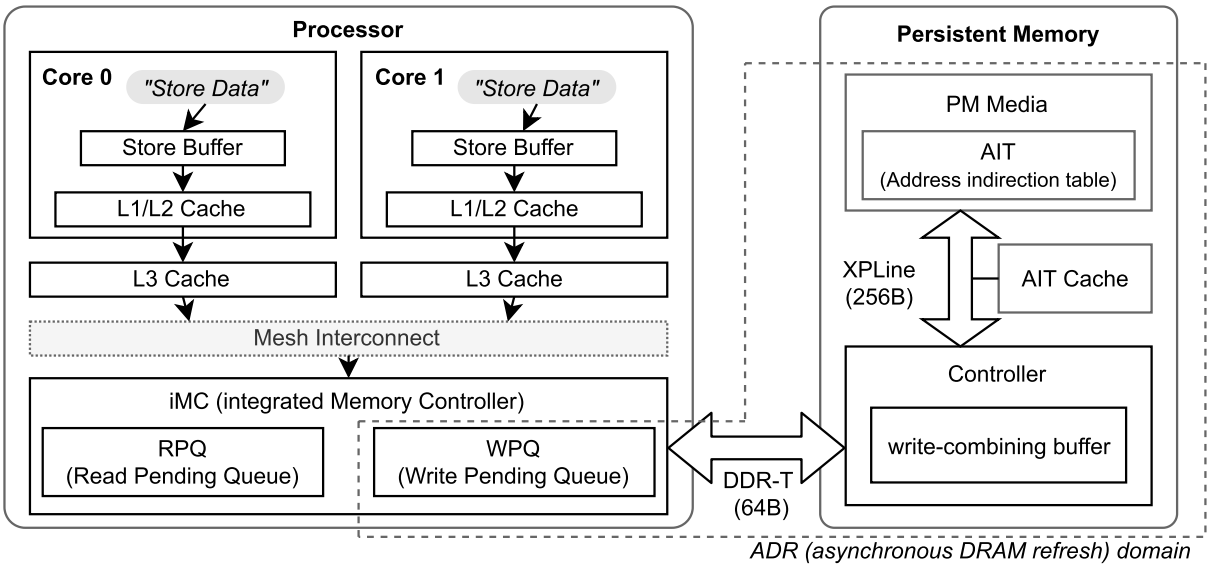


FIGURE 1. Write path to persistent memory [19], [22].

and thus results in considerable performance degradation. In addition, it adversely affects the lifespan of the PM media [5], [20], [21].

The quantity of actual PM media writes, which is the consequence of the write amplification, heavily depends on the design of the index structure. For example, in a workload that reads and writes are combined, the level hashing index produced an average of 5.0 PM media writes per insert operation, whereas P-BwTree performed 19.6 media writes, approximately 3.9 times larger. This paper analyzes the relationship between the index structure design and its write amplification factors at the on-chip cache level and PM media level, respectively, through source code analysis and experiments with benchmark workloads. Furthermore, this paper proposes four optimization techniques that reduce the write amplification produced by a PM-aware index structure. Our findings and optimization techniques can help developers choose or design an appropriate index structure that fits the target workload characteristics.

The contributions of this paper are summarized as follows.

1) This paper reveals that the amount of data written to the PM media can be significantly amplified from the size of the original data written by applications because of the complication in the PM architecture.
2) This paper analyzes the causes of write amplification in relation to the selection of the index structure in the database.
3) This paper points out four write-amplifying design flaws found in the index structures we analyzed and proposes optimization techniques to mitigate them.

The rest of this paper is as follows. Section II introduces the background and motivation of this research. Section III analyzes the write amplification and performance depending on the PM-aware index design and workload characteristics, and Section IV proposes the four optimization techniques. Finally, Section V concludes the research.

II. BACKGROUND AND MOTIVATION

A. PERSISTENT MEMORY HIERARCHY

User-level programs can read from and write to PM using *load* and *store* instructions. The stored data are physically written to the PM media through the data flow path illustrated in Fig. 1. When a core stores data, they are written to the store buffer in the processor core first, and then, it goes down to the on-chip caches. Subsequently, flushed or evicted data from the caches are sent with the 64 bytes cache line granularity to the write pending queue (WPQ) in the integrated memory controller (iMC). The asynchronous DRAM refresh (ADR) guarantees that the data reaching this domain is to be power-failure safe even if it has not yet been physically written to the PM media. Because the ADR includes the WPQ, the order in which data is written after the WPQ does not affect data consistency and durability [22].

The iMC sends read/write requests to the controller inside the PM with the 64-byte unit, the cache line size, through the DDR-T interface. However, the PM controller reads or writes data to or from the media in 256 bytes granularity, referred to as the XPLine size. The PM controller has a write-combining buffer to merge neighboring write operations to a single XPLine [19].

In order to keep the index structure in a consistent state after a power failure, programs should send modifications to ADR in order. For this purpose, cache line flush or non-temporal store (ntstore) is used with *sfence* (store fence). The cache line flush instructions, such as *clflush*, *clflushopt*, and *clwb*, invalidate or write back the specified cache line so that the data goes down to the WPQ immediately. The ntstore instructions such as *movntq* and *movnti* can be used to bypass cache hierarchy and write directly to the WPQ. The *sfence*

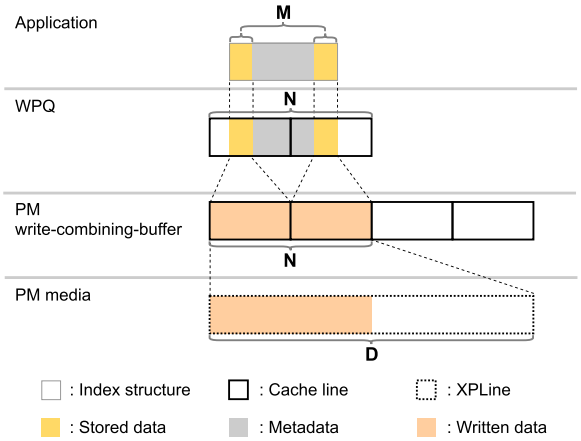


FIGURE 2. Points of write amplification in PM architecture.

instruction following a cache line flush or ntstore prevents the stores after the fence instruction from being reordered or flushed with the stores prior to the fence instruction [22], [23].

In the PM hierarchy, writes are amplified at two points. A write operation smaller than 64 bytes is expanded to the cache line size during the flush or eviction operations. For example, as shown in Fig. 2, when an application stores M bytes of data, which are colored in yellow, the size of the write operation for storing the data and its accompanied metadata is inflated to N bytes in the WPQ. In addition, write amplification happens when a write request smaller than the XPLine size is expanded to a 256-byte XPLine during the media write process. In Fig. 2, the N bytes data in the WPQ is again expanded to D bytes at the PM media level because the unit of media writes is the XPLine size, which is much larger than the cache line size. For these reasons, PM's write amplification is heavily affected by data structures and data access patterns used by the application.

The write amplification of the PM architecture is distinguished from that of the NAND flash SSDs, most of which occur internally due to the garbage collection or wear-leveling operations performed by the flash translation layer (FTL).

B. PM-AWARE INDEX STRUCTURES

Index structures produce memory writes while processing insert, update, and delete operations. The insert or delete operations may require structural modification operations (SMOs), such as merging or splitting nodes, which usually cause massive memory writes. This section analyzes how many memory writes occur while performing inserts on 8 different PM-aware index structures. These 8 indexes consist of three hash-based indexes (P-CLHT (Cache Line Hash Table) [24], level hashing [5] and CCEH (Cacheline-Conscious Extendible Hashing) [4]), three tree-based indexes (FAST&FAIR [3], P-Masstree [25], and P-BwTree [26], [27]), and two trie-based indexes (P-ART (Adaptive Radix Tree) [28] and P-HOT (Height Optimized Trie) [29]). An index structure with a P- prefix in its name is a conversion for PM from its original DRAM-based index structure [2].

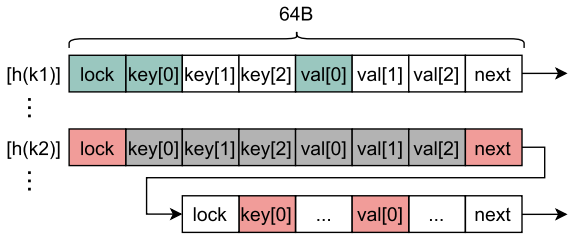


FIGURE 3. Data structure of P-CLHT.

1) HASH-BASED INDEX STRUCTURES

All hash-based indexes covered in this paper use cache line-sized buckets. Because a key-value pair is stored in a bucket, an insertion normally modifies a single cache line. In addition, the hash-based indexes commonly provide consistency during the insert operation by writing the key after the value in order. To keep this constraint, P-CLHT flushes the stores of the value and the key, respectively. This incurs two memory writes per insert operation. Conversely, level hashing and CCEH guarantee the order by adding a fence instruction after the value store. They flush the value and the key together, and thus only one memory write occurs per insertion. Additionally, because all three hash-based indexes use locks for concurrency control, additional memory writes may occur.

The three hash-based indexes have significant differences in how they resolve hash collisions, which occur when inserting a key-value pair into a full bucket, and how they expand the hash table.

P-CLHT resolves collisions through bucket chaining. The second bucket in Fig. 3 depicts the case where P-CLHT tries to add a key, k2, to a bucket full of other keys. P-CLHT adds a new bucket for the key to be added, writes the value and key there, and then flushes the new bucket. After that, the address of the new bucket is written to the *next* field of the existing bucket and flushed. It modifies two cache lines and thus produces two memory writes for adding a new bucket and inserting the key. When the number of chained buckets exceeds the predefined threshold, P-CLHT expands its hash table. When expanding the table, P-CLHT allocates a table four times the size of the existing table and rehashes all keys. Therefore, resizing the hash table causes a considerable quantity of memory writes. Additionally, since the entire table must be locked during table resizing, it limits the scalability and prolongs the tail latency of P-CLHT.

Level hashing is a variant of Bucketized Cuckoo hashing (BCH) [18], [30], [31]. It is designed to reduce write operations in consideration of the PM characteristics. Level hashing has two separate hash tables, each using a different hash function. Therefore, a key is allowed to be inserted into two different top-level buckets, as illustrated in Fig. 4. In addition, the key has four possible positions, including two bottom-level buckets shared by two adjacent top-level buckets. If the inserting key cannot find any empty slot in these four buckets, level hashing selects a victim key from the buckets and move it to another position to obtain an

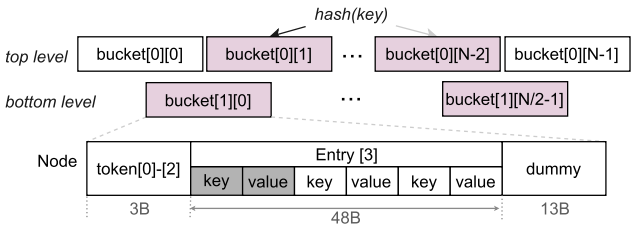


FIGURE 4. Data structure of level hashing.

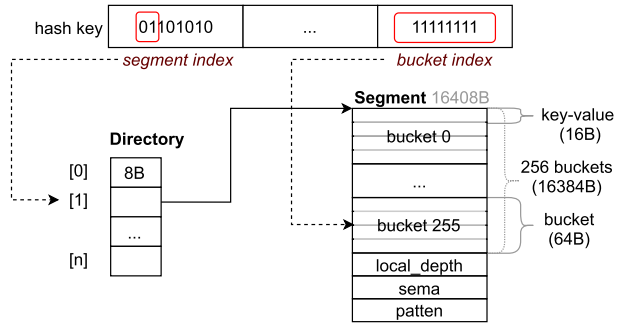


FIGURE 5. Data structure of CCEH [4].

empty slot like BCH. However, unlike BCH, which randomly selects a victim and repeats key movement until an empty slot becomes available, level hashing selects a victim that can free up an empty slot by moving only the victim. Moving the victim key requires two memory writes, and inserting the new key into an empty slot needs an additional memory write. If there are no keys that can be moved, table resizing occurs. Level hashing creates a new top-level table twice the old top-level table and rehashes the old bottom-level to the new top-level to resize the table. Therefore, the cost of table resizing is less than that of P-CLHT, which rehashes the entire table.

Both P-CLHT and level hashing insert keys and resolve collisions with few memory writes. However, large quantities of memory writes inevitably occur for table resizing because they use static hashing. Conversely, extendible hashing, used by CCEH, does not require full-table rehashing. However, extendible hashing requires a directory that maps a hash key to its corresponding bucket. CCEH updates directories only when hash table expansion is required. Using a small bucket size increases the space overhead for the directory. Therefore, CCEH maps hash keys to segments, which are bundles of buckets, to reduce the directory size, as described in Fig. 5. It uses partial bits of the hash key to find the bucket within the segment. A CCEH segment is stored in 257 cache lines, including 256 cache line size buckets and a header. When a bucket collision occurs, a new segment is created through segment split, and the entire new segment is flushed. Therefore, it requires 257 memory writes. For subsequent directory updates, an insignificant number of additional write operations are performed. However, when resizing the directory, additional memory writes may occur.

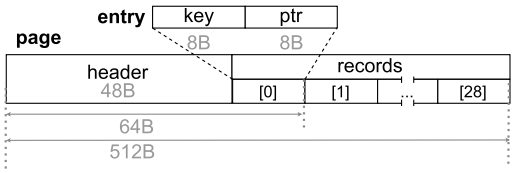


FIGURE 6. Data structure of FAST&FAIR.

2) B+-TREE-BASED INDEX STRUCTURES

FAST&FAIR is a B+-tree customized for PM. P-BwTree is also a variation of the B+-tree that provides non-blocking insert and get operations. P-Masstree is a trie with a fanout of 2^{64}, in which each node is a B+-tree. When using an 8-byte integer key, a P-Masstree consists of a single trie node, which acts as a simple B+-tree. Therefore, in this paper, P-Masstree is classified as a B+-tree-based index structure.

An internal node of B+-tree is composed of a header that stores node information and an array of key and child node pointer pairs. A leaf node has an array of key-value pairs instead of the key-child node pointer array. This key-value array must provide a way to access the keys in sorted order. FAST&FAIR maintains the key-value array in the sorted state by accordingly shifting the existing pairs for every insertion. Conversely, P-Masstree appends the inserted key at the end of the array without sorting the array. In turn, it updates the 8-byte permutation table, which supports ordered access. This difference affects the number of memory writes per insert of these two index structures.

The *page* structure, which is used as a node of FAST&FAIR, has a header and an array of *records* in its 512-byte space, as illustrated in Fig. 6. Depending on where the key is inserted into the record array, 2 to 8 memory writes may occur. Since a B+-tree adds a new node through node split, a node is at least half-filled. Therefore, a FAST&FAIR node has a minimum of 15 keys and can have a maximum of 29 keys. Moreover, the worst-case memory writes average per insert is 6.6, where the keys inserted are always appended to the left-most entry until the node becomes full. Conversely, P-Masstree adds the key-value pair to the array and updates the permutation value, generating only two memory writes, once each.

Node split is performed when a node overflows. For node split, FAST&FAIR prepares an empty node, copies half of the keys in the original node, writes a new key in the new node, and deletes the old keys from the original node. During these steps, 8, 5, 2 to 5, and 5 memory writes are performed, respectively. Conversely, P-Masstree allocates a new node, copies half of the keys, and updates the original node. To do this, it carries out five memory writes for flushing the new node, which is 304 bytes long, and two memory writes for flushing the original node. Because the fanout of P-Masstree is smaller than FAST&FAIR, split occurs more frequently. However, its modification operations generate a fewer number of writes as stated.

P-Bwtree is designed to prevent direct tree structure modification for non-blocking reads and writes when performing

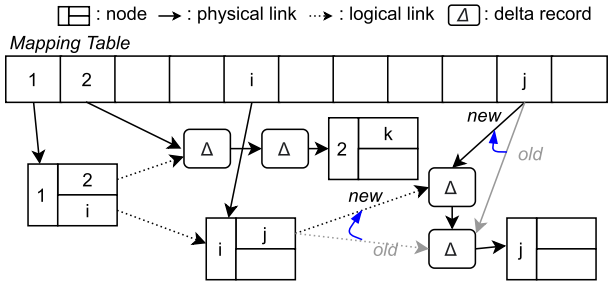


FIGURE 7. Data structure of P-Bwtree [27].

operations such as insert, delete, and update. As shown in Fig. 7, each operation logs the required modifications to a delta record and adds it to the delta chain located in front of the modified node. Thus, the delta records can be recognized prior when searching for a key in the node. Every node has a unique id, and the memory address to the node can be found through the mapping table. When a delta record is added to a node, the starting address of the node is changed. Therefore, the mapping table should be updated to point to the changed starting address for the corresponding node ID. For example, the second entry of the mapping table points to the last added delta record of node 2's delta chain so that the modification history of node 2 can be scanned. When a modification occurs in node *j*, a new delta record is added to the beginning of the delta chain of node *j*, and the pointer to node *j* in the mapping table is changed to the newly added delta record pointer. A parent node points to its child nodes through the mapping table. Therefore, node *i*'s pointer to node *j* is not needed to be updated. As a result, an insert operation produces only two memory writes for logging 56 bytes of the delta record and updating the mapping table.

A node must be consolidated whenever the delta chain length becomes longer than 8, which causes additional writes. The node consolidation allocates the space for the new node and key-value pair arrays. It also updates the mapping table. During this, it causes up to 34 memory writes depending on the number of child nodes. The split operation of P-Bwtree requires 38 memory writes for adding a delta record and creating two nodes with 64 keys, respectively.

3) TRIE-BASED INDEX STRUCTURES

Trie is a tree structure in which all descendants of nodes share a prefix key and indexes child nodes using a part of the remaining bits. Tries can reduce height by increasing span and fanout to improve traverse performance. However, as a trade-off, the reduced height leads to increased memory space consumption for each node. To mitigate this overhead, P-ART uses 8-bit span that allows nodes to have up to 256 children while reducing the memory usage using the node representation that consumes less memory when the node has a small number of children. P-HOT sets the maximum fanout of a node and adjusts the span of the node to reduce the height. In general, P-HOT has fewer nodes and less memory consumption than P-ART [29].

TABLE 1. System configurations used for experiments.

		Specification
Processor	Model	Intel Xeon Platinum 8280M × 2
	Number of Cores	28 (2 SMTs per core)
	Clock Frequency	2.7 GHz
	L1I & L1D Cache	32 KiB & 32 KiB per core
	L2 Cache	1 MB per core
	L3 Cache	1.375 MB per core
Memory		DDR4 2666 MHz 32 GB × 12
Persistent Memory [32]	Model	Optane Persistent Memory 128GB × 12
	User Capacity	126.4 GiB
	Limited Warranty	5 years
	Endurance	259 PBW (100 % Writes 256B)
	Write Bandwidth	1.85 GB/s (100 % Writes 256B)
Software	OS & kernel	Ubuntu 18.04.5 with Linux 5.4.12

When inserting a key-value pair, P-ART converts the integer key into a leaf node and adds the address of the leaf node to an empty slot of the corresponding node. This requires only 2–3 memory writes. However, when the node has no empty slots, P-ART allocates a new node for node growth. When the size of the grown node is 16, 48, and 256 entries, the growth operation causes 4, 12, and 34 additional memory writes, respectively.

The maximum fanout of P-HOT is 32, and the node layout and size vary according to the node span and the number of children. Because P-HOT replaces an existing node with a new node to insert a new key, it needs at least 2–9 memory writes depending on the size of the node. If a node overflows due to the insertion, additional memory writes may occur due to SMOs to resolve it.

III. ANALYSIS OF INDEX STRUCTURES' IMPACT

A. ANALYSIS ENVIRONMENT

The configuration of the server used for the analysis is presented in Table 1. We set the PM module to the AppDirect mode, in which applications are aware that there are two types of memory in the system and enabled interleaving. The OS recognizes six PM modules attached to one socket as a single storage device in this setting. Although the system of Table 1 has two processors, to prevent performance distortion caused by remote NUMA domain accesses, only the processor and memory included in a single NUMA domain were used for our analysis.

The endurance cycle of the PM on Table 1 is from its specification. To understand its expected life span under real-world conditions, we measured the actual wear-out rate while conducting our analysis by monitoring the device health status and the total media write counter using the `ipmctl` tool. A SMART (Self-Monitoring, Analysis, and Reporting Technology) warning is raised when the remaining spare capacity reported in the health status falls below the predefined threshold [33]. When this value reaches zero, it is considered the end of its life span. We found out that one percent of the remaining spare capacity is decreased for every 49 TB of media writes on average. Accordingly, the actual endurance of the PM was 313.4 PBW (petabytes written) based on this observation.

We used the RECIPE benchmark [2] for our analysis. It runs the load phase and then the run phase. The load phase inserts 64 million key-value pairs into an empty index while the run phase performs 64 million operations on the index built in the load phase. The workload traces for load and run phases were obtained from the `A`, `B`,and `C` workloads of Yahoo! Cloud Serving Benchmark (YCSB) [34]. Workload Load consists of only insert operations. Workload A consists of 50% of gets and 50% of inserts, and workload B consists of 95% of gets and 5% of inserts. Workload C performs only get operations. We used uniformly distributed 8-byte random numbers as the keys to be inserted.

Among the three cache line flush instructions, both *clflush* and *clflushopt* invalidate the cache line owning the operand at every level. However, *clwb* may leave the operand cache line in a cache level after writing it back to the lower layers. In addition, both *clwb* and *clflushopt* allow reordering of write operations, and thus can achieve higher throughput. In our evaluation, we used clwb to maximize the performance. However, the choice of the flush operation does not affect the amount of both WPQ insert and media write operations.

We used two values, the number of WPQ-inserts per insert operation and the number of media writes per insert operation, to measure the degree of write amplification.

The WPQ-insert counter is one of the uncore performance monitoring counters that increments by one for every cache line write. It increases not only by explicit memory write operations, such as cache line flush and ntstore, but also by implicit memory writes, including cache eviction. The WPQ-insert counter counts DRAM writes and PM writes separately, and we used only the PM WPQ-insert count for our analysis.

We used the media write counter, which is one of the performance monitoring counters the PM controller provides. The PM controller performance counters increase by one for every 64-byte operation, which is the unit size that iMC sends a request. Therefore, a single XPLine write increases the media write counter by four.

B. HASH-BASED INDEXES

Fig. 8 shows the throughput, the number of WPQ-inserts, and the number of media writes of each of the hash-based indexes during the execution of workload Load, A, B, and C.

P-CLHT performed table resizing 8 to 9 times while executing workload Load. Conversely, it did not resize its hash table when running workload A, B, and C. Therefore, the memory writes measured while running workload A and B were from the inserts without SMOs. Fig. 8(e) and Fig. 8(h) show that P-CLHT produced approximately three WPQ-inserts for key, value, and lock variable updates, per insert as expected in Section II-B1.

P-CLHT uses per bucket lock variable for the insert operation. As shown in Fig. 8(d), the throughput of P-CLHT scaled well through such fine-grained locking for write-intensive workloads. Additionally, the get operation requires only one cache line read to locate a key in most cases where a chained bucket is not added. This enabled fast read performance, as shown in Fig. 8(g) and Fig. 8(j). However, Fig. 8(b) shows that P-CLHT caused the largest number of memory writes among hash-based indexes because it performed the whole rehashing for table resizing. Moreover, when table resizing was performed, the scalability significantly degraded because the entire table must be locked. Therefore, to maximize the advantages that the write amplification factor of P-CLHT is small and performance is fast, it is necessary to initialize the hash table large enough to eliminate the resize cost.

Fig.8(j) showed that the read performance dropped beyond 32 threads where the number of threads per physical core becomes two. The more threads a single core runs, the relative L1- and L2-cache sizes will become smaller. In fact, the total amount of L2 cache accesses was 20.9% larger when with 32 threads in comparison to that of the 28 threads case. The L3 cache accesses were also increased by 20.7%. These result in the amplification of media write requests and media writes, which increased by 12.7% and 11.0%, respectively. Consequently, the IPC (instruction-per-clock) decreased by 9.0%.

Like P-CLHT, level hashing resized the table only during workload Load. Since there was no table resizing, workload A demonstrated the lowest WPQ-insert amplification. However, as illustrated in Fig. 8(e), level hashing running workload A produced an average of 1.5 WPQ-inserts per insert, which was more than expected. This was because one node of the node array used as the level hashing table was allocated with a size larger than 64 bytes, unlike its design. The analysis and solution of this phenomenon are explained in Section IV-A.

Level hashing acquires a lock for a get operation, resulting in additional memory writes. Fig. 8(k) verifies this point. This could be verified with Fig. 8(k) where the workload did not perform any writes. The WPQ-insert-per-get showed 0.15 at 8 threads in the experiment. The WPQ-insert-per-get stays below 1.0 because Level hashing uses a global lock so that a significant portion of writes to the lock were absorbed by the on-chip caches. For the lock accesses, Level hashing demonstrated WPQ-insert amplification over 2.0 for the read-dominant workload B, and media writes were further amplified. Fig. 8(h) and Fig. 8(i) show the numbers of memory and media writes due to 64 million total operations divided by 3.2 million, which is the number of insert operations. In other words, it includes writes not only caused by inserts but also by gets.

Level hashing demonstrated low get throughput because it uses a shared global lock as stated and requires checking up to four cache lines to locate the given key. However, level hashing adopts an optimized hash table resizing technique that maintains the existing top-level and rehashes only the bottom level with a new table. As a result, it generated fewer memory writes for resizing than P-CLHT. Fig. 8(b) shows that the number of WPQ-inserts in level hashing with a single thread was 45.3% smaller than in P-CLHT.

Unlike the two hash-based indexes explained above, CCEH expands the table through segment splits rather than full

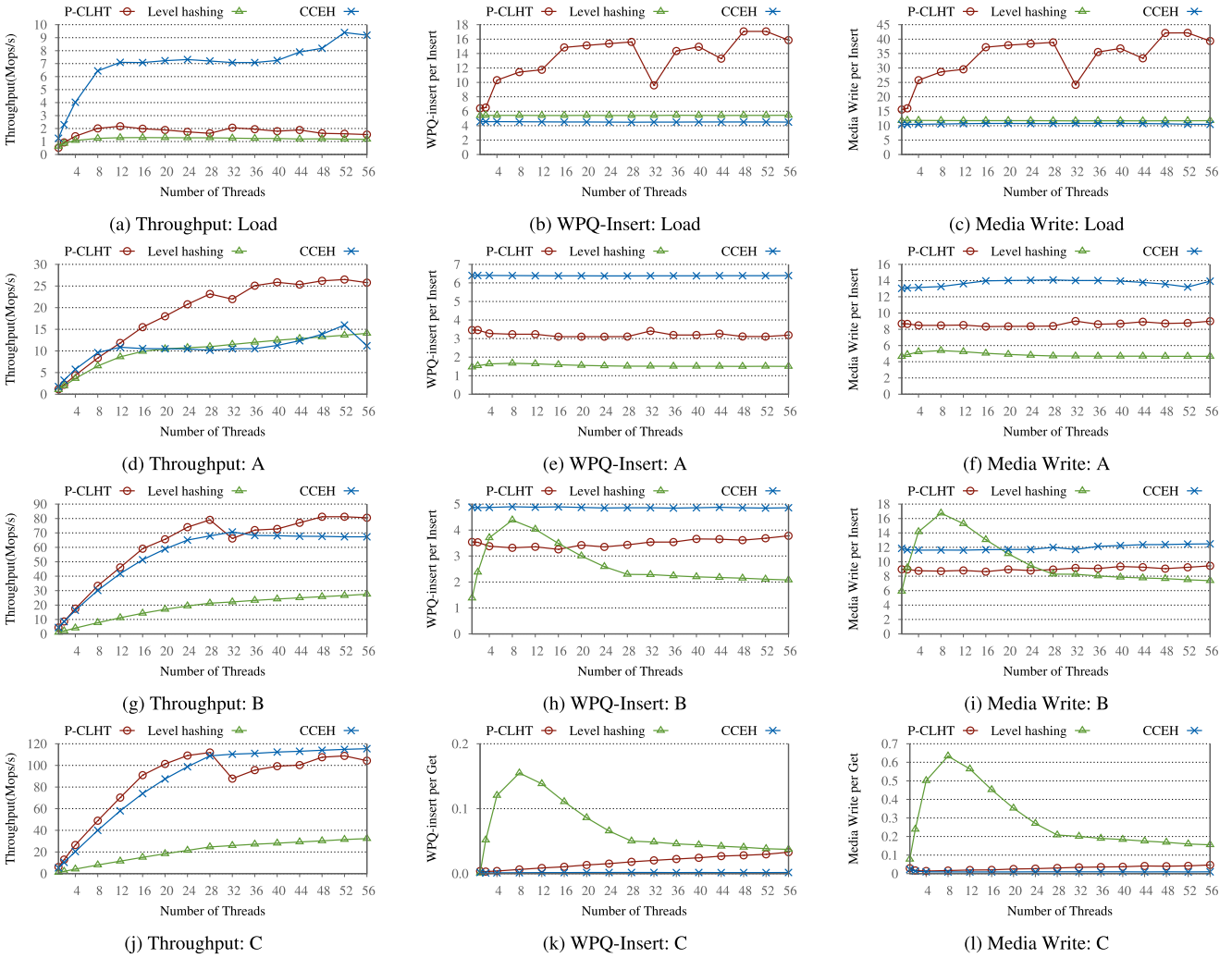


FIGURE 8. Throughput, WPQ-inserts, and media writes of hash-based indexes.

rehashing, resulting in low write amplification for workload Load. However, because segment splitting also occurred during workload A and B, its write amplification was higher than the others for workload A and B. An insert operation creates a small random write pattern. However, segment splitting produces a sequential write. As a result, the media-write-to-WPQ-insert ratio of CCEH was relatively small.

CCEH uses a per segment lock for the segment modification, such as insert or split. This coarse-grained locking, compared to the per bucket lock of P-CLHT, degrade its write scalability. However, CCEH can locate a key by reading only two cache lines, one for the hash directory and the other for the key-value bucket. Additionally, the hash directory is highly likely to be in the processor cache. Consequently, CCEH achieved high throughput for workload B and C.

Table 2 presents how many media writes occurred per WPQ-insert depending on the index structure and the workload. Insert operations on hash-based indexes typically translate into small random writes, resulting in high media write amplification. However, in P-CLHT, since two consecutive flushes for writing key-value pairs are in the same cache line and thus merged, the media-write-to-WPQ-insert ratio in both workload A and B was reduced. In addition, the SMOs, which show sequential write patterns, performed during the workload execution further reduced the media-write-to-WPQ-insert ratio.

TABLE 2. Media writes per WPQ-insert.

Index		LOAD	A	B
Hash-based	P-CLHT	2.49	2.71	2.60
	Level hashing	2.16	3.12	3.60
	CCEH	2.39	2.21	2.46
Tree-based	FAST&FAIR	1.83	1.89	1.87
	P-Masstree	2.97	3.01	3.06
	P-BwTree	2.21	2.15	1.85
Trie-based	P-ART	2.25	2.54	2.52
	P-HOT	1.87	1.97	2.17

C. TREE-BASED AND TRIE-BASED INDEXES

Fig. 9 shows the WPQ-inserts per insert operation, media writes per insert operation, and throughput of the tree-based indexes and trie-based indexes. Both tree- and trie-based

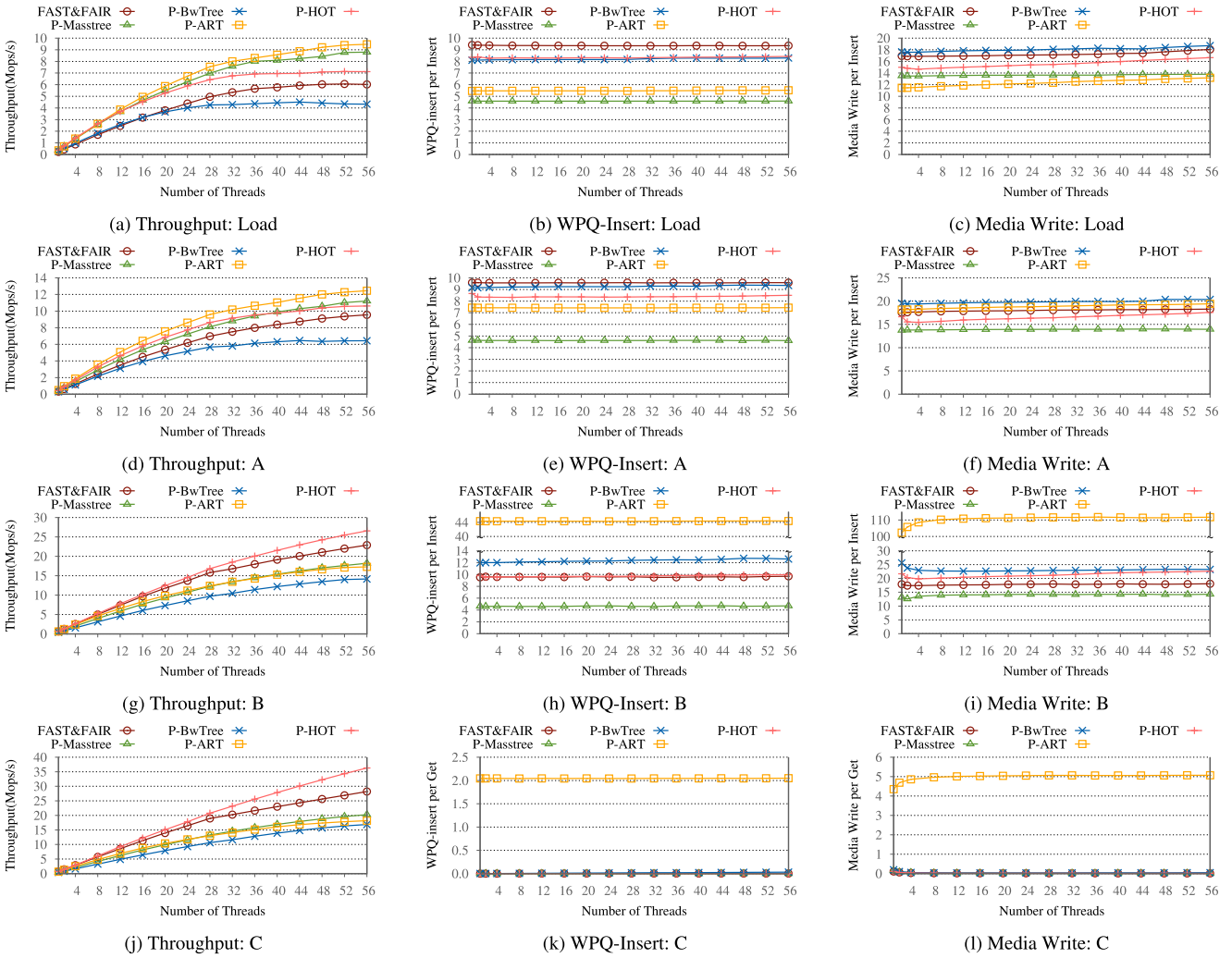


FIGURE 9. Throughput, WPQ-inserts, and media writes of tree- and trie-based indexes.

indexes are compared together because they have a lot in common, such as extending the structure through node splitting and supporting range lookup operations.

FAST&FAIR showed the second-highest performance for workload B and C, while it demonstrated the second-slowest performance in workload Load and A. This is because FAST&FAIR shifts the existing keys to maintain the aligned state while inserting the key into the node. For this reason, FAST&FAIR demonstrated the most WPQ-insert per insert among tree- and trie-based indexes. However, since the keys in the node are kept in an ordered state, get and scan are simplified. Thus, it provided high performance for these operations. Since the shift operations are performed through sequential writes, its media-write-to-WPQ-insert ratio was the lowest among all, as shown in Table 2.

As opposed to FAST&FAIR, P-Masstree was fast on workload Load and workload A but slow on workload B and C. Because of the use of the permutation table for insertion, P-Masstree's WPQ-insert per insert was the smallest among tree- and trie-based indexes, and thus its insert performance was fast. However, because the permutation table complicates the comparison operation, its get performance was low.

The permutation table was also the cause of the sizeable media-write-to-WPQ-insert ratio of P-Masstree. While performing an insert operation, appending a key-value to the array and updating a table modify two different cache lines. They may not be adjacent and result in two different XPLine writes. Consequently, Fig. 9(c) shows that P-Masstree caused more media writes than P-ART. While, P-Masstree produced a smaller number of WPQ-insert than P-ART, as shown in Fig. 9(b).

P-BwTree was the slowest among tree-based indexes, and the number of media writes caused by an insert operation was the largest, with an average of 18.0. The delta node writes and the mapping table updates resulted in random write patterns, and the node consolidation caused a large quantity of memory writes. P-Bwtree was designed in this way for supporting non-blocking synchronization. However, the PM layer has to pay the cost of endurance and performance degradation for this approach. Conversely, other indexes using

(a) Level hashing (b) FAST&FAIR (c) P-CLHT

FIGURE 10. Impact of optimizations when running workload A with 28 threads (RB: removal of bloat, FC: cache flush coalescing, VE: volatile-data early eviction, and XS: XPline-aware unit sizing).

blocking synchronization provided better performance and scalability with fewer memory writes.

P-ART induced relatively fewer WPQ-insert-per-insert of 5.5 when executing workload Load. However, the memory writes by P-ART tended to increase as the proportion of get operations increased. It was 7.4 for workload A and 44.4 for workload B. P-ART uses Read-Optimized Write EXclusion (ROWEX) as its synchronization mechanism [35]. ROWEX causes memory writes even when performing get operations to manage the version information. However, it does not block the reader, and memory writes for ROWEX are not explicitly flushed. Therefore, the read performance of P-ART was on par with P-Masstree, which produced significantly fewer memory writes as shown in Fig. 9(k).

In the workload Load, since P-HOT performs the trie modification for each insert through the node replacement, its WPQ-inserts were relatively large. The large number of WPQ-inserts in P-HOT caused many write streams from multiple threads to be multiplexed in the write buffer as the number of threads increases. This lowered the sequentiality of the writes to the media. As a result, the chances that the write-combining buffer merges the buffered write operations became low, which led to the increased media write amplification [19], as shown in Fig. 9(c) and Fig. 9(f).

When using 8-byte random integer keys of uniform distribution, HOT is known to have a relatively larger depth of leaves and thus to have lower performance than ART. However, since P-HOT on PM does not require memory write when performing get operations, it caused much fewer WPQ-inserts and media writes during workload B, resulting in higher performance than P-ART.

IV. PM-AWARE INDEX OPTIMIZATION

During our analysis, we derived four performance optimization techniques for PM-aware indexes. During the analysis, we discovered a few common design flaws that caused write amplification in the index structures. This section explains the observed defects and proposes four techniques for mitigating the problems. These approaches are independent of each other and can be applied orthogonally.

A. REMOVAL OF UNINTENDED BLOAT

A node of an index structure is designed to be aligned with the cache line size to reduce the number of cache flushes. However, the compiler inserts padding so that the start address of each field inside the node is a multiple of 8. Therefore, the actual node size may become larger than the cache line, contrary to developers' intention. This issue was found in the design of FAST&FAIR and level hashing.

As illustrated in Fig. 6, a 512-byte page of FAST&FAIR has a 48-byte header followed by a record array with 28 entries. A one-byte variable and a two-byte variable are placed at the end of the header, followed by the dummy field inserted for the cache line alignment. In fact, as illustrated in Fig. 11, the two variables take 8 bytes by padding five bytes after, and the dummy also takes 8 bytes. Therefore, the actual size of the header becomes 56 bytes, not 48 bytes. Because of this, a key and its value in a record can be placed on different cache lines for record 0, 8, 16, and 24. This results in additional read and write operations. Additionally, the number of key-value entries that one page can have is also reduced to 28. When the dummy field is removed, a page occupies 48 bytes as intended.

The node data structure of level hashing is designed to fit in a cache line size, as illustrated in Fig. 4. However, as shown in Fig. 12, the token takes 8 bytes, and the dummy field occupies 16 bytes. Consequently, the node size becomes 72 bytes, not 64 bytes. Therefore, two cache flush instructions are required to flush a single bucket. Because the level hashing table is implemented as an array of nodes, buckets are sequentially allocated in the PM. As shown in Fig. 12, from the second bucket, a bucket is stored from the middle of the cache line because of the 8-byte shift. Consequently, the token and key-value entries will be stored in different cache lines. This problem can be resolved by resizing the dummy or moving the token array after the entry array to remove the padding.

Fig. 10(a) and Fig. 10(b) show the effect of the removal of the unintended bloat. WPQ-inserts decreased by 31.3% in level hashing and by 27.4% in FAST&FAIR. The media write reduction was 11.2% in level hashing and 2.0% in FAST&FAIR. The reduced cache flush resulted in reduced core stall cycles and improved throughput. The throughput was improved by 6.9% in level hashing and 27.5% in FAST&FAIR. The reason that the performance improvement of FAST&FAIR was large is that the tree height and split decreased as the cardinality of the node increased.

B. CACHE FLUSH COALESCING

For an insert operation to guarantee data consistency in a hash-based index, the value must be stored in the PM

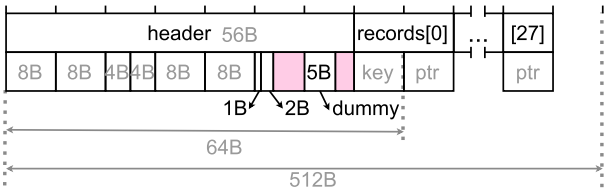


FIGURE 11. Page of FAST&FAIR allocation layout.

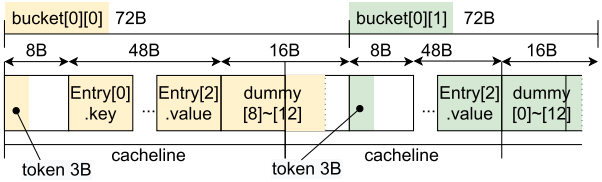


FIGURE 12. Buckets of level hashing allocation layout.

before the key becomes persistent. Hash-based indexes use explicit cache flush after each write to ensure this write-ordering. However, if it is certain that the key and value are on the same cache line, and writing the key and the value is performed consecutively, it is sufficient to execute the cache flush instruction only once after the key write. Among the index structures we investigated, P-CLHT meets these conditions.

Without the cache flush instruction between the value store and the key store, the two store instructions can be reordered by the compiler or the processor. Indeed, reordering of these store commands should still not be allowed. If reordering of them is allowed, the consistency may be broken because the updated key can be observed before its value is updated by other cores. Fortunately, the x86_64 architecture does not allow reordering between local stores. If an architecture allows store reordering, we can place a fence instruction between these two store instructions. However, to prevent reordering by the compiler, we have to annotate a reordering barrier in the source code.

P-CLHT is implemented to flush the value and then to flush the key immediately afterward as shown in Fig. 13. The first flush can be safely removed. When the cache flush coalescing is applied, the memory writes generated by an insert operation of P-CLHT are reduced from three to two. Fig. 10(a) shows that the number of WPQ-inserts was reduced by 30.8%. As shown in Fig. 10(c), the throughput increased by 31.8%. Because cache flushes repeatedly performed on the same cache line are merged in the write-combining buffer, there was no significant difference in the number of media writes.

C. EARLY EVICTION OF VOLATILE-DATA

Writes of volatile-data do not require an explicit cache flush. However, if one cache line has both volatile-data and persistent data, in some cases, explicitly flushing volatile-data can reduce media writes and improve throughput by merging writes in the PM controller.

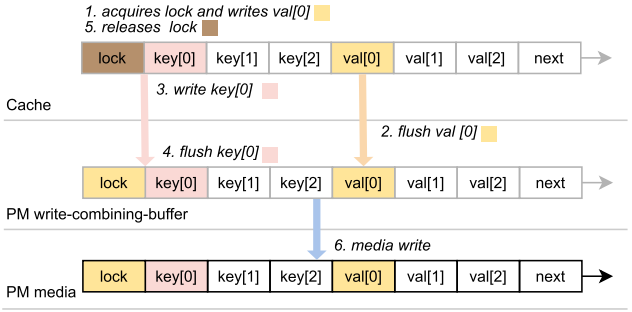


FIGURE 13. Insert operation procedure of P-CLHT.

This approach is applicable to P-CLHT. P-CLHT has a lock variable for each bucket as mentioned in Section II-B1. Writing a Key-value pair must ensure that data is safely written to the PM media at the moment the modifying operation completes. However, the lock variable is a volatile object, which does not need to be persistent at the media-level. The state of the lock variable has no effects when determining the validity of the key-value during the recovery after a crash. For this, when releasing a lock, the lock variable modification is not flushed.

As shown in Fig. 13, the flushed modifications for acquiring the lock and writing a value and a key are combined into one media write, but the modification from releasing the lock, which has not been flushed, is still in the cache after the insert operation is completed. Because the accesses to a hash-based index are difficult to have temporal locality, the modifications stored in a cache are likely to be evicted to the PM media by the following operations. However, if this lock release were flushed immediately, it would be merged in the write-combining-buffer in the same way that the key write and value write are combined.

Fig. 10(c) shows that this optimization reduced media writes by 41.6% compared to when the cache flush reduction was applied. Although the number of WPQ-inserts increased by 5.7%, throughput was improved by 23.7% due to the significant reduction in media writes. This optimization technique improved the performance as the number of threads increased, resulting in a performance improvement of 40.2% and media write reduction of 42.1% when running with 56 threads.

D. XPLine-AWARE UNIT SIZING

Designing data structures with consideration of the XPLine size can reduce the number of media reads and writes.

FAST&FAIR's page structure spans two XPLines. Since a node of B+-tree is created through a split, it is at least half full. Therefore, the shift operations to process an insert always cause 8 media writes. Changing the page size to 256 bytes can reduce the number of media writes for an insert from 8 to 4. Additionally, as the fanout of the tree becomes smaller, the number of cache flushes for the shift operations also decreases. Conversely, the smaller fanout increases the tree height and thus traversal time. Moreover, it also causes split

operations to occur more frequently. Fig. 10(b) shows that XPLine-aware page sizing reduced WPQ-inserts by 10.5% and media writes by 7.6% compared to the case with the bloat removal. However, the throughput was decreased by 7.1% due to the increased tree traversal time.

We also increased the bucket size of P-CLHT from 64 bytes to 256 bytes. This can reduce media writes from 8 to 4 when writes to consecutive buckets are executed. This optimization reduced the WPQ-inserts and media writes by 10.4% and 15.1%, respectively, from the combination of flush coalescing and volatile-data eviction. However, the throughput was barely improved because the other optimization techniques already lowered the degree of contention at the media level and controller level.

V. CONCLUSION

The PM can remarkably increase the capacity of an IMDB system with its low price and decent performance. Additionally, due to its non-volatility characteristics, recovery can be performed with the remaining data in the PM in case of a system failure, and service downtime can be minimized through fast recovery. For this purpose, several PM-aware index structures that can maintain consistency even in system failure have been proposed.

This paper analyzed the quantity of memory writes, and media writes, and throughput during benchmark execution using 8 different index structures fabricated for the PM. Our analysis revealed the various performance and write amplification characteristics of the PM-aware index structures. Additionally, we proposed four optimization techniques based on the analysis. Representatively, P-CLHT achieved a 49.5% reduction in total media writes and a 35.7% improvement in throughput by applying the proposed optimization technique.

REFERENCES

[1] *Intel Optane DC Persistent Memory: Benefit From Greater Capacity, Affordability and Persistence*. Santa Clara, CA, USA: Intel Corporation, Mar. 2019.

[2] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram, "Recipe: Converting concurrent DRAM indexes to persistent-memory indexes," in *Proc. 27th ACM Symp. Oper. Syst. Princ. (SOSP)*, Oct. 2019, pp. 462–477.

[3] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in byte-addressable persistent B+-tree," in *Proc. 16th USENIX Conf. File Storage Technol. (FAST)*, Feb. 2018, pp. 187–200.

[4] M. Nam, H. Cha, Y.-R. Choi, S. H. Noh, and B. Nam, "Write-optimized dynamic hashing for persistent memory," in *Proc. 17th USENIX Conf. File Storage Technol. (FAST)*, Feb. 2019, pp. 31–44.

[5] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *Proc. 13th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, Oct. 2018, pp. 461–476.

[6] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, "WORT: Write optimal radix tree for persistent memory storage systems," in *Proc. 15th USENIX Conf. File Storage Technol. (FAST)*, 2017, pp. 257–270.

[7] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, "FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, Jun. 2016, pp. 371–386.

[8] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *Proc. 9th USENIX Conf. File Stroage Technol. (FAST)*, 2011, p. 5.

[9] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "NV-tree: Reducing consistency cost for NVM-based single level systems," in *Proc. 13th USENIX Conf. File Storage Technol. (FAST)*, 2015, pp. 167–181.

[10] S. Chen and Q. Jin, "Persistent B$^+$-trees in non-volatile main memory," *Proc. VLDB Endowment*, vol. 8, no. 7, pp. 786–797, Feb. 2015.

[11] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson, "Bztree: A high-performance latch-free range index for non-volatile memory," *Proc. VLDB Endowment*, vol. 11, no. 5, pp. 553–565, Jan. 2018.

[12] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Redesigning LSMs for nonvolatile memory with NoveLSM," in *Proc. USENIX Annu. Technical Conf. (USENIX ATC)*, 2018, pp. 993–1005.

[13] J. Liu, S. Chen, and L. Wang, "LB+Trees: Optimizing persistent index performance on 3DXPoint memory," *Proc. VLDB Endowment*, vol. 13, no. 7, pp. 1078–1090, Mar. 2020.

[14] B. Lu, X. Hao, T. Wang, and E. Lo, "Dash: Scalable hashing on persistent memory," *Proc. VLDB Endowment*, vol. 13, no. 8, pp. 1147–1161, Apr. 2020.

[15] S. Ma, K. Chen, S. Chen, M. Liu, J. Zhu, H. Kang, and Y. Wu, "ROART: Range-query optimized persistent ART," in *Proc. 19th USENIX Conf. File Storage Technol. (FAST)*, Feb. 2021, pp. 1–16.

[16] F. Nawab, J. Izraelevitz, T. Kelly, C. B. Morrey, D. R. Chakrabarti, and M. L. Scott, "Dalí: A periodically persistent hash map," in *Proc. 31st Int. Conf. Distrib. Comput. (DISC)*, Oct. 2017, pp. 37:1–37:16.

[17] L. Wang, Z. Zhang, Z. Zhang, and B. He, "PA-tree: Polled-mode asynchronous B+ tree for NVMe," in *Proc. 36th IEEE Int. Conf. Data Eng. (ICDE)*, Apr. 2020, pp. 553–564.

[18] B. Debnath, A. Haghdoost, A. Kadav, M. G. Khatib, and C. Ungureanu, "Revisiting hash table design for phase change memory," in *Proc. 3rd Workshop Interact. NVM/FLASH With Oper. Syst. Workloads (INFLOW)*, 2015, pp. 1–9.

[19] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *Proc. 18th USENIX Conf. File Storage Technol. (FAST)*, Feb. 2020, pp. 169–182.

[20] V. Gogte, W. Wang, S. Diestelhorst, A. Kolli, P. M. Chen, S. Narayanasamy, and F. T. Wenisch, "Software wear management for persistent memories," in *Proc. 17th USENIX Conf. File Storage Technol.*, Feb. 2019, pp. 45–63.

[21] S. Cho and H. Lee, "Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2009, pp. 347–357.

[22] S. Scargall, *Programming Persistent Memory—A Comprehensive Guide for Developers*, 1st ed. New York, NY, USA: Apress, 2020.

[23] *Instruction Set Reference, A-Z*, Intel 64 IA-32 Archit. Softw. Developer Manuals, Intel Corp., Santa Clara, CA, USA, Nov. 2020, vol. 2.

[24] T. David, R. Guerraoui, and V. Trigonakis, "Asynchronized concurrency: The secret to scaling concurrent search data structures," *SIGARCH Comput. Archit. News*, vol. 43, no. 1, pp. 631–644, Mar. 2015.

[25] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *Proc. 7th ACM Eur. Conf. Comput. Syst. (EuroSys)*, 2012, pp. 183–196.

[26] J. J. Levandoski, D. B. Lomet, and S. Sengupta, "The Bw-tree: A B-tree for new hardware platforms," in *Proc. IEEE 29th Int. Conf. Data Eng. (ICDE)*, Apr. 2013, pp. 302–313.

[27] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, and D. G. Andersen, "Building a bw-tree takes more than just buzz words," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, May 2018, pp. 473–488.

[28] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: ARTful indexing for main-memory databases," in *Proc. IEEE 29th Int. Conf. Data Eng. (ICDE)*, Apr. 2013, pp. 38–49.

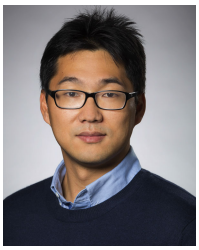
[29] R. Binna, E. Zangerle, M. Pichl, G. Specht, and V. Leis, "HOT: A height optimized trie index for main-memory database systems," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, May 2018, pp. 521–534.

[30] A. D. Breslow, D. P. Zhang, J. L. Greathouse, N. Jayasena, and D. M. Tullsen, "Horton tables: Fast hash tables for in-memory data-intensive computing," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2016, pp. 281–294.

[31] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing," in *Proc. 10th USENIX Conf. Netw. Syst. Design Implement. (NSDI)*, 2013, pp. 371–384.

[32] Intel Corporation. *The Challenge of Keeping up With Data*. Accessed: Jun. 2021. [Online]. Available: https://www.intel.co.kr/content/www/kr/ko/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html
[33] Intel Corporation. (2020). *Intel Optane DC Persistent Memory Quick Start Guide Revision 1.1*. [Online]. Available: https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/data-center-persistent-mem/Intel-Optane-DC-Persistent-Memory-Quick-Start-Guide.pdf
[34] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput. (SoCC)*, 2010, pp. 143–154.
[35] V. Leis, F. Scheibner, A. Kemper, and T. Neumann, "The ART of practical synchronization," in *Proc. 12th Int. Workshop Data Manage. New Hardw. (DaMoN)*, 2016, pp. 1–8.

YOUNGJOO WOO received the B.S. degree in electronic engineering from Inha University, South Korea, in 2009, the M.S. degree in computer science from the Ulsan National Institute of Science and Technology (UNIST), in 2012, and the Ph.D. degree from Sungkyunkwan University, in 2019. She is currently a Postdoctoral Researcher with the Electronics and Telecommunications Research Institute (ETRI). Her research interests include operating systems, many-core systems, energy-efficient computing, and cloud computing.



TAESOO KIM received the B.S. degree from KAIST, in 2009, and the S.M. and Ph.D. degrees from MIT, in 2011 and 2014, respectively. He is an Associate Professor with the School of Cybersecurity and Privacy and the School of Computer Science, Georgia Institute of Technology (Georgia Tech). He also serves as the Director for the Georgia Tech Systems Software and Security Center (GTS3). He is interested in building a system that has underline principles for why it should be secure. Those principles include the design of a system, analysis of its implementation, and clear separation of trusted components. He was a recipient of various awards, including NSF CAREER, in 2018, Internet Defense Prize, in 2015, and several best paper awards, including USENIX Security'18 and EuroSys'17.

SUNGIN JUNG received the B.E. and M.E. degrees in computer engineering from Pusan National University, in 1987 and 1989, respectively, and the Ph.D. degree in computer engineering from Chungnam National University, Daejeon, South Korea, in 2006. Since 1990, he has 28 years of experience in the area of the operating systems at the Electronics and Telecommunications Research Institute (ETRI), and was a UNIX kernel developer during the 1990s. He had participated in developing the UNIX kernel for SMP, ccNUMA, and MPP systems in cooperation with Novell and SCO. In the early 2000s, he began the Linux Kernel Project for Carrier Grade Linux (CGL) and DataCenter Linux (DCL) workgroups of Linux Foundation. Since 2014, he has been the Director of the Basic Research Center for manycore OS research. In addition, he is serving in OSS activities, such as OSS policy and international events. His research interests include operating systems kernel, cloud computing, intermittent computing, and OSS.

EUISEONG SEO (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science from KAIST, in 2000, 2002, and 2007, respectively. He is currently a Professor with the Department of Computer Science and Engineering, Sungkyunkwan University, Republic of Korea. Before joining Sungkyunkwan University, in 2012, he had been an Assistant Professor at the Ulsan National Institute of Science and Technology (UNIST), Republic of Korea, from 2009 to 2012, and a Research Associate at The Pennsylvania State University, from 2007 to 2009. His research interests include systems software, embedded systems, and cloud computing.