WALSH: Write-Aggregating Log-Structured Hashing for Hybrid Memory

YUBO LIU, Sun Yat-Sen University, Guangzhou, China
YONGFENG WANG, Sun Yat-Sen University, Guangzhou, China
ZHIGUANG CHEN, Sun Yat-Sen University, Guangzhou, China
YUTONG LU, Sun Yat-Sen University, Guangzhou, China
MING ZHAO, School of Computing and Information Sciences, Arizona State University, Tempe, United States

Persistent memory (PM) brings important opportunities for improving data storage including the widely used hash tables. However, PM is not friendly to small writes, which causes existing PM hashes to suffer from high hardware write amplification. Hybrid memory offers the performance and concurrency of DRAM and the durability and capacity of PM, but existing hybrid memory hashes cannot deliver high performance, low DRAM footprint, and fast recovery at the same time. This paper proposes WALSH, a flat hash with novel log-structured separate chaining designs to optimize the performance while ensuring low DRAM footprint and fast recovery. To address the overhead of hash resizing and garbage collection (GC), WALSH further proposes partial resizing/GC mechanisms and a 4-phase protocol for concurrent hash operations. As a result, WALSH is the first flat index for hybrid memory with embedded write aggregation ability. A comprehensive evaluation shows that WALSH substantially outperforms state-of-the-art hybrid memory hashes; e.g., its insert throughput is up to 2.4X that of related works while saving more than 87% of DRAM. WALSH also provides efficient recovery; e.g., it can recover a dataset with 1 billion objects in just a few seconds.

CCS Concepts: • **Information systems** → **Data structures**; • **Hardware** → **Memory and dense storage;**

Additional Key Words and Phrases: Persistent memory, hybrid memory, hash, key-value store, write amplification

ACM Reference Format:
Yubo Liu, Yongfeng Wang, Zhiguang Chen, Yutong Lu, and Ming Zhao. 2025. WALSH: Write-Aggregating Log-Structured Hashing for Hybrid Memory. *ACM Trans. Storage* 21, 2, Article 13 (February 2025), 26 pages.
https://doi.org/10.1145/3715010

This work was partially funded by National Key R&D Program of China 2023YFB4502704, National Science Foundation awards 2311026, 2126291, and 1955593, National Science Foundation of China 62272499 and 62332021, Guangdong Province Special Support Program for Cultivating High-Level Talents 2021TQ06X160, Pazhou Lab PZL2023KF0001.
Authors' Contact Information: Yubo Liu, Sun Yat-Sen University, Guangzhou, Guangdong, China; e-mail: yubo.liu@nscc-gz.cn; Yongfeng Wang, Sun Yat-Sen University, Guangzhou, Guangdong, China; e-mail: yongfeng.wang@nscc-gz.cn; Zhiguang Chen (Corresponding author), Sun Yat-Sen University, Guangzhou, Guangdong, China; e-mail: zhiguang.chen@nscc-gz.cn; Yutong Lu, Sun Yat-Sen University, Guangzhou, Guangdong, China; e-mail: yutong.lu@nscc-gz.cn; Ming Zhao, School of Computing and Information Sciences, Arizona State University, Tempe, AZ, USA; email: mingzhao@asu.edu.

ACM 1553-3077/2025/02-ART13
https://doi.org/10.1145/3715010

1 Introduction

As a novel storage device, **persistent memory (PM)** brings important opportunities for improving data storage with its in-memory persistence, large capacity, and support for direct CPU access. However, there remains a non-negligible gap in performance between PM and DRAM (e.g., latency and concurrency). More importantly, the mismatch between application write size and PM hardware write unit will significantly affect the efficiency of PM access. Taking 3DXPoint [24] as an example, its hardware access unit is a XPLine, with a size of 256B [50, 52]. This can lead to severe hardware write amplification for PM systems if not addressed adequately.

Hash table is a common index widely used in storage systems. An important advantage of PM hashes compared to DRAM hashes is that they support fast data persistence and recovery [9, 16, 30, 38, 40, 46, 57, 58]. However, existing PM hashes disperse objects across the storage space and are usually used to store small objects (e.g., 8B pointers), which makes them inevitably trigger a lot of small random writes on PM. According to our quantitative analysis, intensive small random writes on PM lead to extremely high hardware write amplification ratio—the ratio of hardware write size to software write size, up to 32X in existing PM hashes. The hardware characteristics of PM severely limit the performance and scalability of PM hashes, especially on write operations.

To break through the hardware limitations of PM, combining the advantages of DRAM and PM as a hybrid memory is an effective approach [35, 37, 49], and this also applies to hash designs. An ideal hybrid memory hash needs to achieve three important goals. **Goal 1:** high performance — leveraging DRAM to boost the performance of PM hashes; **Goal 2:** low DRAM footprint — saving the precious DRAM space as much as possible for applications; **Goal 3:** efficient recovery — maintaining the strong recoverability offered by PM hashes. None of the existing works can meet these three goals simultaneously, as the inherent tradeoff is very challenging to optimize.

Existing write-optimized flat hashes on hybrid memory, such as Viper [4], FlatStore [8, 47], and Halo [20], use the ***non-self-contained*** method to aggregate writes, which means that objects are stored outside the index (values are pointers to objects), and write operations are aggregated through external mechanisms. They maintain a complete flat hash in DRAM to index the PM locations of all objects, and thus avoid the need to update the hash in PM and its associated performance issues. However, the need of a complete hash index in DRAM leads to high DRAM footprint and slow recovery. Other hybrid memory hashes resort to tree structures, instead of a flat structure, for write optimization, such as ChameleonDB [55] and Plush [45]. They aggregate small PM writes by replacing all tables in the LSM-Tree [41] with fixed-size hash tables. However, they are not flat indexes (unable to provide constant time complexity on average) and inherit the shortcomings of LSM trees (e.g., slow search and expensive table compaction).

Motivation. To address the shortcomings of existing hybrid memory flat hashes in high DRAM footprint and low recovery efficiency, it is key to make write aggregation ***self-contained*** in the flat hash, which means that objects are stored directly in the index, and the write aggregation mechanism is designed in conjunction with the hash structure. However, this is difficult to achieve for existing hybrid memory hashes which use open addressing, because the base object locations are directly decided by the hash function. Compared with open addressing hash, we found that separate chain hash brings a good motivation to design a hybrid memory hash with self-contained write aggregation, because the object locations can be dynamically allocated in separate chain hash.

However, designing an efficient write-aggregating hash based on separate chaining is non-trivial. It faces three challenges: **Challenge 1:** How to aggregate write operations of any type (insert, update, and delete) while ensuring efficient recovery? **Challenge 2:** How to optimize the high latency caused by full table resizing in traditional separate chaining hash? **Challenge 3:** How to ensure high concurrency?

This paper proposes WALSH, the first flat hash for hybrid memory with self-contained write aggregation ability. WALSH proposes three techniques to address the challenges mentioned above:

(1) Log-structured separate chaining. It appends all write operations (insert/update/delete) to the heads of chained lists, aggregates them in small DRAM logs, and then flushes the full logs to PM. In this way, WALSH allows all hash writes to be aggregated without maintaining a complete hash in DRAM, thus significantly optimizing write performance and reducing DRAM footprint and recovery overhead, compared to existing hybrid memory hashes.

(2) Partial resizing/GC (garbage collection). To avoid full table resizing/GC (brought by log-structured writes), we propose the log area-based structure, which allows resizing and GC are executed at the small granularity (log area). At the same time, it allows the rehashed objects to be loaded to DRAM in batch and rehashed in DRAM, which avoids small PM accesses during resizing/GC.

(3) 4-phase concurrency protocol. The key concurrency challenge with WALSH is to avoid resizing/GC blocking front-end operations for long periods of time. We divide resizing/GC into four phases using different concurrency strategies, and the expensive rehashing phase can be concurrent with front-end read and write operations.

Our comprehensive evaluation shows that WALSH significantly exceeds the existing write-optimized hybrid memory hashes in write performance while delivering a comparable search performance and drastically saving the DRAM footprint and recovery time. For example, WALSH's insert throughput is up to 2.4X that of Viper while it saves about 87% of DRAM space. In addition, WALSH can recover a dataset of 1 billion objects in just a few seconds.

2 Background and Related Work

2.1 PM Hashes

PM hashes can be used as the storage engine of databases, data management systems, and the like. PM hash operations can be divided into two types: write (insert, update, and delete) and read (search). All operations in PM hashes are atomic. For write operations, they must be correctly recovered or discarded after a system crash. In strict semantics, insert operation will only succeed if the object already exists, so it is necessary to perform an object existence check before performing the insert. Some PM hashes treat inserts and updates as having the same semantics to eliminate the existence check before inserting. Although this can improve insert performance, it makes it impossible to atomically insert a non-existent object. For read operation, it must ensure that the latest version of the target object is available.

PM-only hashes are well studied in previous works, e.g., Level hash [58] employs a level structure to reduce the rehashing overhead; CLevel hash [9, 10] optimizes the rehash concurrency of Level hash; CCEH [40] and PCLHT [30] use cacheline-aware designs to improve the efficiency of PM access; DASH [38] improves the performance and load factor of extendible hash and linear hash [34] by using a bucket load balance mechanism; SEPH [46] optimizes the resizing by spliting the table into multiple segments. These studies optimize hashes for PM from different aspects, but the hardware write amplification of PM still limits the performance of existing PM hashes.

We compare the total PM write size from a test program and the corresponding write size in the raw PM device (obtained using the `ipmctl` tool). Figure 1 shows that the hardware write amplification reaches more than 2.1 times on small random writes (< 256B), and it is much better in the cases of large writes and sequential writes. For sequential writes, they can be aggregated in the DIMM controller, but this is not feasible for small random writes. High write amplification will reduce the PM efficiency, which also affects the performance of PM hashes.

In existing PM hashes, objects are randomly distributed to the storage space, and hash tables are usually used to store small objects (e.g., 8B pointers), which means most of the hash write

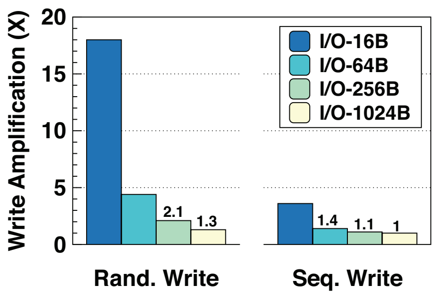


Fig. 1. Hardware write amplification on PM.

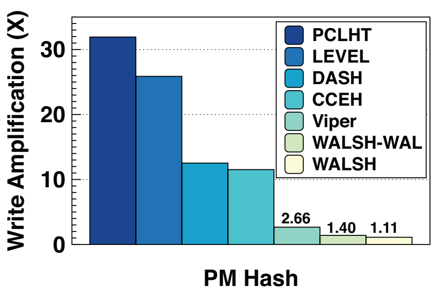


Fig. 2. Hardware write amplification of hashes.

operations are small (< 256B) and random. In addition, a PM hash also needs additional PM writes to maintain structure (e.g., resizing). Therefore, PM hashes can cause serious write amplification. We evaluate the write amplification of state-of-the-art PM flat hashes with a 40-thread insert workload. Figure 2 shows that their write amplification is over 11X (WALSH is only less than 1.4X).

In particular, we believe the write amplification problem exists with current and future PM systems (e.g., CXL-based SSD [25]). First, performing write operations in fixed-size units inside the hardware is a typical design of storage media and interconnect technologies [29]. Second, the smallest unit of the `store` instruction is a cacheline (64B), so write requests smaller than the cacheline size will also cause write amplification. At the same time, the impact of write amplification will be more significant on "slow" PMs (e.g., SSD-based PM) that may appear in the future.

2.2 Hybrid Memory Hashes

Using DRAM to mitigate the hardware shortcomings of PM is an intuitive idea, which leads to the design of hybrid memory hashes. However, existing solutions cannot meet the three important goals introduced in Section 1 at the same time. They can be classified into the following two categories:

(1) Flat Hashes. Viper [4], Halo [20], Bullet [21], and FlatStore [8] (and its optimized version Pacman) [47] maintain a complete flat hash in DRAM to index the objects in PM, and use mechanisms decoupled from the hash indexes to optimize writes. Viper and FlatStore keep a CCEH in DRAM as a object location index and perform writes sequentially to reduce write amplification (nonetheless, Figure 2 shows that Viper still causes more than 2.6X of hardware write amplification); Halo aggregates writes in per-core DRAM buffers and flushes them to PM in batches; Bullet appends writes to PM logs and commits them to the PM hash table in background.

These solutions need to maintain a complete index in DRAM, which takes up a lot of DRAM space. They also suffer from slow recovery if no additional expensive mechanisms are used (e.g., checkpointing), as the whole index needs to be rebuilt in DRAM after crash. Furthermore, the write optimization mechanisms of Halo and Bullet have obvious limitations. For Halo, it uses asynchronous semantics (i.e., return does not mean operation is complete), which greatly limits its usage scenarios, and it cannot aggregate update operations. For Bullet, the background log committing is slower than foreground writes, resulting in the PM logs being often full and blocking write-intensive workloads. In summary, existing flat hashes on hybrid memory cannot make good trade-offs between performance, DRAM footprint, and recovery efficiency.

(2) Tree Hashes. Write aggregation is widely used in tree indexes (e.g., LSM-Trees [1, 2, 7, 27, 41, 53], B^{ε}-Tree [3, 43], STB^{ε}-Tree [12]), and some hybrid memory hashes borrow this idea. ChameleonDB [55] and Plush [45] replace all tables in LSM-Tree [41] with fixed-size hash tables to aggregate small writes. However, they cannot provide constant time complexity on average, i.e., their performance degrades significantly as the depth of the tree increases (e.g., the performance of Plush reduces by about 30% when tree depth grows from 1 to 2). Although ChameleonDB mitigates the negative impact of the LSM structure by caching the objects in all upper levels in a DRAM flat hash, it brings a large DRAM footprint. BOA [11] is also a hash table based on LSM for external memory, but provides no practical implementation and is not specifically designed for PM.

In addition to leveraging hybrid memory to optimize writes, there are some hybrid memory hashes designed for other aspects. TIPS [28] and Pronto [39] proposed frameworks to simply and efficiently convert DRAM index to PM based on hybrid memory. HMEH [56] is an extendable hash that keeps the directories in DRAM but object buckets in PM to accelerate directory operations. HiKV [51] stores objects in a PM hash table and indexes them by a B-tree in DRAM to support multiple types of queries. These hybrid memory hashes/frameworks do not specifically optimize writes for the characteristics of PM, so their write performance is inferior to write-optimized hybrid memory hashes.

3 WALSH Design

WALSH adopts the flat structure instead of trees to provide constant time complexity on average (performance is almost independent of dataset size). However, there is a dilemma between flat hash and write optimization: hash algorithm is characterized by scattering writes across the entire storage space, and this contradicts the need of aggregation/serialization which requires writes to be append-only. Existing flat hashes on hybrid memory all use open addressing which directly determines the base locations of objects by the hash algorithm and makes it difficult to aggregate writes. Hence, they need to keep a complete flat hash in DRAM, resulting in large DRAM footprint and high recovery overhead.

Our solution is a novel log-structured separate chaining hash (detailed in Section 3.1). With separate chaining, an object is hashed to the directory by its key and each directory points to a chained list of objects. WALSH keeps the directories in DRAM and always inserts new objects to the heads of lists, where they are sequentially stored (appended) in small DRAM logs and flushed to PM in batches. Furthermore, WALSH is log structured, where all types of writes including inserts, updates, and deletes are treated as new objects and aggregated in DRAM. This design allows WALSH to keep most of the hash table on PM while using a very small amount of DRAM for aggregating and optimizing writes.

However, we need to address the overhead of hash resizing and GC brought by separate chaining and log-structured hash. WALSH provides partial resizing/GC (Section 3.2) to reduce the granularity of these operations and their corresponding overhead. Then, it proposes 4-phase concurrency control (Section 3.3) to minimize the locking required by resizing/GC and allow normal reads/writes to be handled concurrently.

3.1 Log-Structured Separate Chaining

3.1.1 Two-Level Directory. WALSH stores the directories in DRAM because they tend to be accessed frequently and are relatively small in size. WALSH proposes a two-level directory design to save its DRAM usage (explained later.) As Figure 3 shows, the first level is the *directory block index*, and the second level has multiple fixed-size *directory blocks*. Each entry in the directory block index is the 8B address of a directory block, and each directory block contains multiple directories (32 by default). Each directory consists of two fields, the address pointing to the head of the chained list (8B) and the bloom filter (8B). The chained list is stored in an object log which is explained later. The bloom filter is stored in DRAM and used to speed up negative search. When a new object is appended to the chained list, WALSH uses the first 16 bits of its hash value as the seeds (8 bits/seed) to update the bloom filter in the directory. The directory block index and

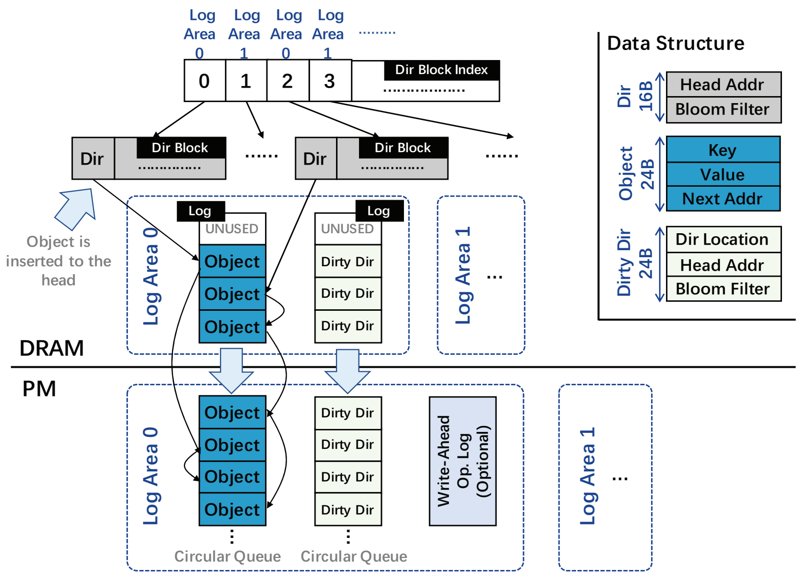


Fig. 3. Overall structure of WALSH. All write operations (insert/update/delete) are treated as new objects and are stored and chained in the object logs. All directory updates are stored in the directory logs. The logs are managed by log areas for concurrency control. The newest object and directory logs from each log area are kept in DRAM for aggregating writes and later flushed to PM when they become full or exceed the flush time interval.

Table 1. Symbol Description

Symbol	*Dir_ID*	*Total_Dir_Num*	*Dir_Block_ID*	*Log_Area_ID*	*Area_Num*
Description	Unique ID of directory. Each directory points to an object chained list.	Total number of directories.	Unique ID of directory block. Each directory block contains multiple directories.	Unique ID of log area. Each log area contains multiple directory blocks.	Total number of log areas.

the directory block are initialized to a small size (262,144 directories by default) and dynamically expand as the dataset grows.

Table 1 shows the description of the symbols used in the directory lookup process. To find the directory of a given key in the two-level directory structure, WALSH first calculates its *Dir_ID* using Equation (1), where *Total_Dir_Num* is the total number of directories. Then it calculates the position of the directory block (*Dir_Block_ID*) in the directory block index using Equation (2), where *Dir_Block_Size* is the number of directories in the directory block. Finally, from the directory block, WALSH finds the target directory by the last *N* bits of *Dir_ID* (*N* depends on the number of directories in the directory block).

$$Dir\_ID = Hash(Key)\,\%\,Total\_Dir\_Num \tag{1}$$

$$Dir\_Block\_ID = Dir\_ID\ \%\ Dir\_Block\_Size \tag{2}$$

$$Log\_Area\_ID = Dir\_Block\_ID\ \%\ Area\_Num \tag{3}$$

Compared to single-level directory, the two-level structure allows directories to be allocated at a really fine granularity, in the unit of directory blocks. In order to avoid full table rehashing, WALSH proposes a partial resizing design (detailed in Section 3.2) to allow the table to be partially extended. The conventional single-level structure requires the directories to be allocated contiguously, which is prone to low DRAM utilization in partial resizing because it needs to double the total number of directories. In the worst case, using a single-level directory will increase directory memory usage by *Area_Num* times compared to a two-level directory when resizing. In contrast, with the proposed two-level structure, only a small number of directories need to be allocated during the partial resizing, thereby saving the DRAM usage. In addition, although the directory block index needs to be allocated in advance, its DRAM space overhead is relatively low (a 64MB directory block index can support more than one billion objects).

3.1.2 Logs and Write Aggregation. As shown in Figure 3, there are two types of logs in WALSH: *object logs* are used to store the objects and *directory logs* are used to record the dirty directories. The newest logs are kept in DRAM for aggregating writes while all the rest are stored in two circular queues in PM (one for object logs and the other for directory logs). Each object/directory log contains a certain number of entries (256 by default).

WALSH does not modify objects in place, but instead considers every write operation (insert/update/delete) as a new object and appends it to the head of the chained list that it is hashed to. Multiple chained lists can be aggregated using the same object log and each entry in the object log is an object (both its key and value) of a chained list. For a chained list, objects are chained by `<object log ID, offset in the log>` (this address does not change regardless whether the log is in DRAM or PM). When a new object arrives, it is aggregated using the newest object log, and the list head in the target directory points to the address of the new object. At the same time, the corresponding directory update is also aggregated using the newest directory log. These two logs will be flushed to PM together by two large PM writes when they are full, and then new logs will be created in DRAM to serve subsequent writes. WALSH also flushes logs that are not full but reside in DRAM for more than a preset time (5s by default, which follows the default setting of dirty flushing period of POSIX file systems) to PM in background.

To support the partial resizing/GC and concurrency optimization (detailed in Sections 3.2 and 3.3), WALSH divides the table into multiple log areas (512 by default). Each log area includes multiple object/directory logs, and provides a lock for controlling concurrent access to the log area. The newest object and directory logs from each log area are kept in DRAM for aggregating writes while all the rest are in PM. Figure 3 shows an example of using two log areas. Each directory is assigned to a unique log area using Equation (3), where *Dir_Block_ID* represents the location of the directory block in the directory block index and *Area_Num* is the total number of log areas. Objects are stored in the log areas where their directories are assigned to. Note that *Log_Area_ID* of an object does not change when the hash resizes as the total number of directories in a log area is always doubled.

Increasing the number of log areas can improve concurrency, but this will increase the number of logs in DRAM. Fortunately, the memory cost of this part is small and deterministic. For example, considering the default configuration of 512 log areas, the DRAM logs only occupy 6MB of space. In fact, increasing the number of log areas can help reduce DRAM usage in some cases. Because partial resizing is done at the log area granularity (detailed in Section 3.2), the number of directory blocks that need to be extended for rehashing decreases as the number of log areas increases.

3.1.3 PM Management. WALSH creates a PM pool for each log area and maintains two circular queues for storing object logs and directory logs, respectively (each element in the queues is a log). The `object log ID` in the object address (`<object log ID, offset in the log>`) indicates the position of the object log in the circular queue of the log area.

WALSH prevents PM leak by its inherent PM space management and crash recovery mechanism, although many exiting works (e.g., [30, 40, 58]) do not consider this problem [17, 28]. WALSH uses PMDK [23] to allocate the log pools on PM when it is initialized (which will be recreated if the system crashes during initialization). At runtime, PM space is allocated at the granularity of object/directory logs, and WALSH guarantees that the allocation of log space and making it indexable are atomic by using PMDK transaction to update the allocation information of the PM log pool.

3.1.4 Hash Operations. WALSH provides all the common hash storage functions, including insert, update, search, and delete. In particular, WALSH performs the key existence check. This enhances the usability because it supports atomic insertion of non-existent objects.

Insert. *Step 1* is to find the directory. WALSH calculates the directory ID by hashing the key (Equation (1)) and finds the target directory block (Equation (2)). It also finds the corresponding log area according to the directory ID (Equation (3)). The directory contains the starting address of the chained list and the bloom filter. *Step 2* is to detect whether the key exists by checking the bloom filter and traversing the chained list if the bloom filter returns true. The insert returns failed if the key already exists. *Step 3* is to append the new object to the target log area's DRAM object log and point the list head in the directory to it. It also updates the bloom filter in the directory and appends the dirty directory to the DRAM directory log. *Step 4* is to check whether the object and directory logs need to be flushed to PM and whether resizing and GC are required, and if so, start them.

Update and Delete. In our log-structured hash design, update and delete are similar to insert. They do not modify the old object but instead insert a new object with the same key into WALSH. The difference between them and the insert operation is that they do not need to perform the key existence check. The expired objects will be reclaimed by GC.

Search. *Step 1* is to find the directory and *Step 2* is to check whether the key exists, both of which are the same as the first two steps of handling the Insert operation. If the key exists, *Step 3* is to fetch the object from the log that it is stored in, and return it. Although there may be multiple versions of the object, the search correctness can be guaranteed because the newest version of the object will always be at the forefront of the chained list and returned by the search process.

3.1.5 Support for Strong Durability. By default, WALSH provides the durability which guarantees that only a small amount of data (up to 3MB by default) may be lost in a short time (up to 5s by default). These settings can be flexibly tuned by users or administrators based on their needs, just like the settings for flushing dirty data buffered in DRAM when using a file system [36, 37]. We study their impact to data durability and write performance in Section 4.9.

WALSH also provides a **Write-Ahead operation Log** mode (**WAL** mode) for the scenarios that require immediate persistence (same durability as existing PM hashes). In other words, every successful write operation can be recovered in WAL mode. In the WAL mode, each log area maintains a WAL on PM (Figure 3) and each write will be persistently appended to the WAL with its operation type before it returns. For a log area, every time when the DRAM log is successfully flushed to PM and the log area is not performing rehashing, the WAL will be cleared. An alternative strong durability solution is to persist every write to the PM object log immediately without aggregating, but this solution requires additional random small PM writes and transactions to maintain WALSH's structure (e.g., logging dirty directories); in comparison, WAL appends involve only sequential PM writes. Although the WAL mode of WALSH will sacrifice some write performance compared to its default mode, it is still much better than existing PM hashes with the same level of durability, and saves a lot of DRAM compared to existing hybrid memory hashes.

3.1.6 Support for Variable-Length Objects. WALSH supports keys and values of variable lengths. They are managed in almost the same way as in the fixed-size case. The main difference is that each variable-length object contains a head (24B), which includes the lengths of key and value (8B each). The case of variable-length objects in WALSH shares the same aggregation design, consistency guarantee, and concurrency control as the fixed-size object case. Many existing hashes (e.g., [30, 38, 40, 56–58]) do not support variable-length objects and assume they are stored outside the hash, i.e., the hash stores only their metadata (e.g., pointers). That is also a good way to use WALSH and take advantage of its performance optimizations for the small writes of the pointers. Still, WALSH's support for variable-length objects is a useful feature that allows it to efficiently store and access variable-length objects directly in the hash.

3.1.7 Write Ordering. The order of writes on the PM affects whether WALSH can be consistently restored. All PM writes in WALSH are log-structured, which follows the principle of persisting the log first and then atomically making the log valid to ensure the write order. The persistence granularity is log. WALSH will atomically modify the log tail pointer (by 64B atomic write) after object log and directory log are successfully persisted (by `clflush` and `mfence`) to ensure the write order. For the WAL mode, in addition to the persistence of object and directory logs, writes are also persisted to the operation log immediately, following the persistence order of the log structure. We discuss consistency recovery in Section 3.4.3.

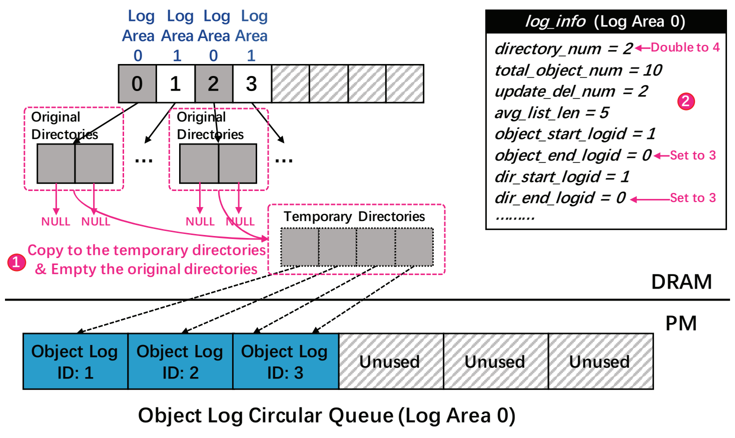
3.2 Partial Resizing and Garbage Collection

Separate chaining and log-structured bring the overhead of resizing and GC. The naive solution is to perform them at the granularity of the full table, which may result in high latency. To address this challenge, WALSH designs a partial resizing/GC mechanism to perform these expensive operations on a small part (by default 1/512) of the table: at the granularity of individual log areas (instead of the full table). (We quantitatively analyze the advantage of partial resizing/GC vs. full-table resizing/GC in Section 4.6.) When a write detects that the state of its log area satisfies the conditions, it will trigger resizing and/or GC in foreground, which will be executed by one write thread with fine-grained concurrency control (using a 4-phase protocol detailed in Section 3.3).

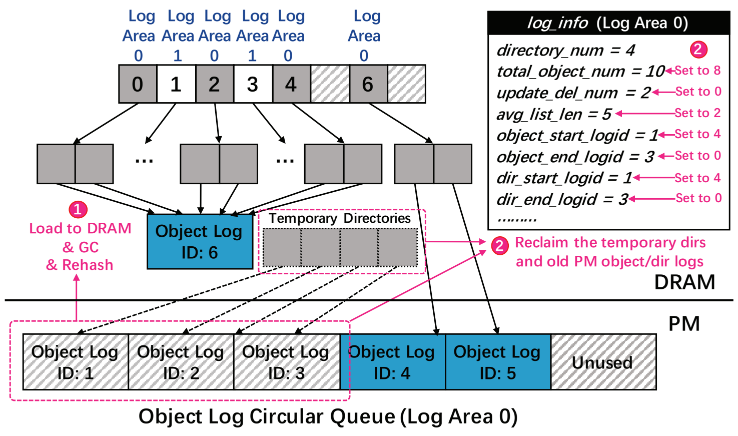
To support resizing and GC, WALSH maintains a *log_info* structure (shown in Figure 4) for each log area, which includes the total number of objects (*total_object_num*), the total number of updates/deletes (*update_del_num*), the average chained list length (*avg_list_len*), the average number of updates/deletes in the chained lists (*avg_updatedel_num*), and the range of valid object/directory logs in the PM circular queue (*object/dir_start_logid*, *object/dir_end_ logid*). WALSH maintains a copy of *log_info* in DRAM to accelerate its access.

3.2.1 Conditions and Processing Steps. For each log area, resizing and GC may be triggered when the average chained list length of the log area exceeds the threshold (four objects by default). The short chained list ensures the efficiency of search, but it increases the frequency of resizing/GC. Thanks to the partial resizing/GC design, their overhead is significantly reduced, allowing WALSH to use short lists while ensuring sufficiently high performance (analyzed in Section 3.5). If the proportion of update/delete operations in the log area does not exceed the threshold (1/16 by default), this means that a large number of objects in the log area cannot be reclaimed, so WALSH performs both resizing and GC at the same time. If the proportion of update/delete operations exceeds the threshold, WALSH performs only GC. Resizing is not needed in this case because there are many expired objects in the chained lists and they can be reclaimed.

Figure 4 shows an example of performing resizing/GC in a log area: **Step 1** is to extend the number of directories of the log area. WALSH first flushes the logs in DRAM to PM to ensure crash consistency and then executes the following sub steps. **Step 1.1** is to create a temporary copy of the existing directories. Searches that occur simultaneously with the resizing and GC need to search both the original and temporary directories. Then, the original directories are reset, i.e., the chained list head and bloom filter are set to `NULL`. **Step 1.2** is to double the *directory_num* and persistently set the *object/dir_end_logid* to the current object/directory log ID. The



(a) Step 1: Extending Directories


Log Area 0
Log Area 1
Log Area 0
Log Area 1
Log Area 0
Log Area 0
0
1
2
3
4
6
1 Load to DRAM & GC & Rehash
Object Log ID: 6
Temporary Directories
log_info (Log Area 0)
directory_num = 4
total_object_num = 10 ← Set to 8
update_del_num = 2 ← Set to 0
avg_list_len = 5 ← Set to 2
object_start_logid = 1 ← Set to 4
object_end_logid = 3 ← Set to 0
dir_start_logid = 1 ← Set to 4
dir_end_logid = 3 ← Set to 0
2 Reclaim the temporary dirs and old PM object/dir logs
DRAM
PM
Object Log ID: 1
Object Log ID: 2
Object Log ID: 3
Object Log ID: 4
Object Log ID: 5
Unused
Object Log Circular Queue (Log Area 0)

(b) Step 2: Rehashing and Collecting Logs

Fig. 4. Process of resizing and GC. They are triggered at the granularity of the log area when the average chained list length of the log area exceeds the threshold. The original directories are copied for concurrent searches. The old object logs are loaded and rehashed in DRAM to avoid small random accesses in PM during resizing and GC.

object/dir_end_logid is used to roll back unfinished resizing/GC after a crash. The directory extension only increases the number of directories in *log_info* to allow future operations to be performed in a larger directory space. New directories will be dynamically allocated by subsequent writes. Note that **Step 1.2** is not needed if resizing is not needed and only GC is performed as discussed above.

Step 2 is to rehash the valid objects and reclaim the expired logs. It includes two sub steps. **Step 2.1** is to load the object logs in the range of *object_start_logid* and *object_end_logid* to DRAM in batches and rehashes them. The rehashing process traverses all chained lists in the log area. For each chained list, it checks and discards the expired objects, and rehashes only the newest versions. At the same time, if the rehashing process finds that the object to be rehashed has a new version (e.g., the object is updated/deleted during rehashing), it will skip this reinsert as the data is out-of-date. This traversal process does not require PM access because all target objects have been loaded

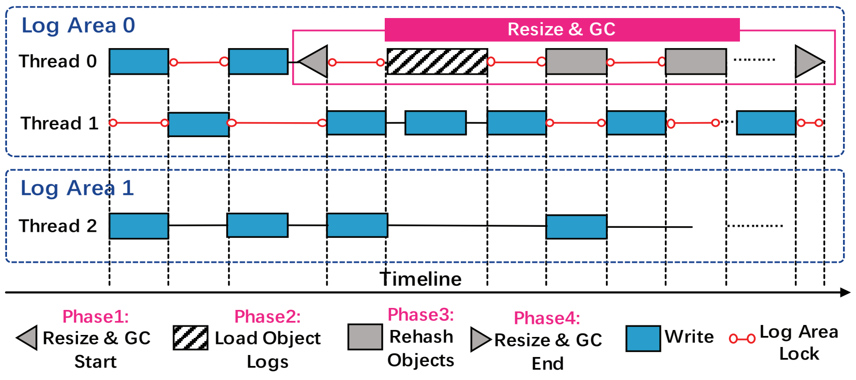


Fig. 5. 4-phase concurrency control. Requests between different log areas can be executed in parallel. Requests in the same log area are subject to the 4-phase concurrency mechanism, which allows rehashing to be concurrent with front-end write operations.

into DRAM in batches. **Step 2.2** is to update the *log_info* and reclaim the old object/directory logs on PM by a PMDK transaction, and finally reclaim the temporary directories in DRAM.

3.2.2 Compacting Directory Logs. As mentioned above, WALSH reclaims the expired object/directory logs in batches during partial resizing/GC. However, since each write in WALSH needs to update the directory and log it to the directory log, it is possible to have expired directories in valid directory logs. WALSH reclaims the expired directories and compacts directory logs to improve the PM space utilization. This directory collection process copies the directories in DRAM to PM (organized as PM directory logs), and then reclaims the old PM directory logs. It can be performed when the system is idle or WALSH is shut down.

3.3 4-Phase Concurrency Control

Although our log-structured separate chaining design is friendly to write aggregation, the need of hash resizing and GC would reduce concurrency if these operations require complete locks and block normal hash accesses. To address this challenge, WALSH splits resizing and GC into four phases, and makes them subject to fine-grained locks.

3.3.1 Concurrent Reads and Writes. Before introducing the concurrency control of resizing and GC, we explain normal read and write concurrency. Concurrent writes in the same log area are subject to the log area lock (such as the Thread 0 and Thread 1 in Figure 5), while multiple writes in different log areas are lock-free (such as the Thread 1 and Thread 2 in Figure 5).

Search is lock-free in the normal case. Similar to some hashes [38], WALSH resorts to an optimistic method to ensure that the newest result is returned from search. Because write is always appended to the head of the chained list, the search process can verify whether the result is the newest by comparing the head of the chained list in the target directory before and after traversing the list. If they are not equal, it means that there are new writes appended to the chained list during the search and the result may be out of date; then the search process will re-traverse the chained list. In the rare cases where a search has failed multiple times (up to 64 times) by concurrent writes, WALSH provides the search the log area lock to avoid starvation.

3.3.2 Concurrent Read/Write and Resizing/GC. To maximize the concurrency between resizing/GC operations and normal reads/writes, WALSH divides the process of resizing and GC into four phases (Figure 5). **Phase 1: Resize & GC Start.** The main job in this phase is to extend the directories, and it is subject to the log area lock called by the write that triggers the resizing and GC operation. Benefiting from the dynamic directory allocation, directory extension is very fast because it only needs to increase the number of directories in the *log_info* structure without immediately allocating the new directories. **Phase 2: Load Object Logs.** It loads all object logs that need to be rehashed and garbage collected from PM to DRAM. This phase is time consuming but it does not need to apply the lock, so it has no obvious impact on the concurrency. **Phase 3: Rehash Objects.** This is a time consuming phase. However, WALSH regards it as ordinary inserts, so each rehashing of an object is only subject to its log area lock, and does not block the entire rehashing process. **Phase 4: Resize & GC End.** This is a lightweight phase, which involves only updating the *log_info* with the log area lock.

During resizing and GC, normal writes can be executed concurrently when log area locks are not held. As the expensive phases (Phases 2 and 3) in resizing/GC do not need locks or only need a single log area lock for a period of time, writes can still be largely concurrent during resizing/GC. Search is still lock-free, but it needs to traverse the chained list in both of the original and temporary directories (detailed in Section 3.2).

3.4 Recovery and Consistency

3.4.1 Normal Recovery. In the normal case, WALSH flushes all logs and directories to PM when it is closed. The directories are stored in PM as directory blocks, which are the same as their DRAM layout. There are two steps to recover WALSH from a normal shutdown. **Step 1** is to load the directory blocks to DRAM and rebuild the directory block index. **Step 2** is to initialize the DRAM log and *log_info* for each log area. Because normal recovery is lightweight, it can perform well in single thread.

3.4.2 Concurrent Crash Recovery. Because the log areas are independent of each other, crash recovery can be executed concurrently across log areas. The recovery of each log area mainly includes two steps:

Step 1 is to recover the *log_info*, including the current log ID and some description information. We use *object/dir_start_logid* and *object/dir_current_logid* in the *log_info* to denote the used range of the PM object/directory log circular queue of the log area. The *object/dir_start_logid* is updated after each resizing/GC operation and *object/dir_current_logid* is updated each time a new log is created. The *object/dir_end_logid* is used to detect whether system crashed during resizing and GC; it is atomically set to the tail of the PM log circular queue before resizing/GC and atomically reset to 0 after resizing/GC is completed. If the *object/dir_end_logid* is not 0 after the crash, it means that the crash occurred in the middle of a resizing/GC operation, and the logs created during resizing/GC will be discarded by rolling back the *object/dir_current_logid* to *object/dir_end_logid*. If the *object/dir_end_logid* is 0, the *object/dir_ current_logid* does not need to be modified.

Step 2 is to rebuild the directories and the directory block index of the log area. The recovery process first loads the directory logs in the range of *dir_start_logid* and *dir_current_logid* to DRAM in batches. Then it traverses all directories in the directory logs and rebuilds the directories by using their last versions. Directory blocks are dynamically allocated and recorded in the directory block index during the directory rebuild. In the WAL mode, it also has to replay the operations in the write-ahead operation log.

3.4.3 Consistency. WALSH can be consistently recovered from system crashes. **Scenario 1:** Crash occurred during write. The log flushing is a transaction because the tail of the PM log circular queue will only be updated atomically after the log is persisted. Therefore, WALSH only needs to discard the logs that have not been flushed to PM. **Scenario 2:** Crash occurred during a resizing/GC operation. These processes are executed transactionally (detailed in Section 3.2). The recovery process can roll back WALSH to the state that it was in before the resizing/GC operation. **Scenario 3:** Crash occurred during initialization/recovery. The initialization and recovery processes are executed transactionally by atomically updating the flags, so WALSH only needs to re-perform them after a crash.

3.5 Tradeoffs Analysis

Our log-structured separate chaining design is friendly to optimizing write performance, DRAM footprint, and recovery overhead, but it also requires WALSH to make good tradeoffs in some other important aspects:

Hash resizing and GC are the overhead of our log-structured separate chaining design. WALSH mitigates this overhead by reducing the granularity of these operations with partial resizing/GC and improving their concurrency with fine-grained locks and 4-Phase concurrency control. The evaluation shows that these optimizations boost the average throughput by 34.5X compared to the full-table resizing/GC solution. Although WALSH still cannot outperform some dynamic hashes (that do not require rehashing) in terms of extremely high percentile latency, it has lower latency for 99.9% of writes than the existing PM/hybrid memory hashes.

Search performance can be affected by separate chaining which WALSH utilizes to aggregate writes. WALSH addresses this problem by dynamically keeping the chained lists at a short length (four objects per list by default), which, however, increases the frequency of resizing and GC. Thanks to the partial resizing/GC design and fine-grained concurrency control, WALSH can strike a good balance between improving read performance and reducing resizing/GC overhead. In addition, the directories are stored in DRAM, which also accelerates the search process. Overall, WALSH is able to achieve its goal of substantially improving write performance without sacrificing much of its read performance.

Cacheline vs. XPLine Accesses. Unlike some related works [30, 38, 40], WALSH is not optimized for cacheline accesses, but it is optimized for XPLine accesses. Benefiting from its inherent write aggregating ability, WALSH substantially reduces XPLine accesses (about 68% to 90% from our evaluation) compared to existing PM hashes for write operations, while providing a comparable search performance.

3.6 Discussion

This subsection discusses some independent aspects.

Durability. WALSH provides both batch and immediate persistence. Although many PM hashes are designed with immediate persistence, we believe that batch persistence is also widely useful for three reasons: (1) Many traditional scenarios do not require that write operations in their object storage engines be persisted immediately. For example, cloud cache can restore data by reloading; big data processing can handle system failures by recalculating. (2) Traditional KV databases (e.g., LevelDB, RockDB, Redis) also use batch persistence as the default configuration, which is suitable for most real workloads. To this end, we believe that sacrificing less durability for a significant performance improvement is reasonable for a hash index. (3) Only a small amount of data (up to 3MB by default) may be lost in a short time (up to 5s by default). Also, the amount of data loss can be controlled by changing the DRAM log size and the maximum interval between PM flushes.

Read-Write Asymmetry. The read and write performance of current PM hardware is asymmetric, and this asymmetry changes according to different access loads. Considering that the data is not in any cache, the write will be faster than the read because it can be cached by the CPU cacheline and PM internal buffer. When the write is persisted (by `clflush`), it will be slower than the read because the data needs to be flushed to the persistent storage unit. WALSH focuses on optimizing write operations for two reasons: (1) Due to the guarantee of write order (a large number

Table 2. Details of Related Works

	CCEH [40]	**LEVEL** [58]	**DASH** [38]	**PCLHT** [30]	**TBB** [22]	**CLHT** [14]
Type	Flat Hash	Flat Hash	Flat Hash	Flat Hash	Flat Hash	Flat Hash
Scheme	PM	PM	PM	PM	DRAM	DRAM
Write Opt.	N/A	N/A	N/A	N/A	N/A	N/A
Durability	Immediately	Immediately	Immediately	Immediately	Volatile	Volatile

	CUCKOO [42]	**DCCEH** [40]	**Plush** [45]	**Viper** [4]	**FlatStore-H** [8, 47]	**Halo** [20]
Type	Flat Hash	Flat Hash	LSM Tree Hash	Flat Hash	Flat Hash	Flat Hash
Scheme	DRAM	DRAM	DRAM-PM	DRAM-PM	DRAM-PM	DRAM-PM
Write Opt.	N/A	N/A	Aggregate	Serialize	Serialize	Aggregate
Durability	Volatile	Volatile	Immediately	Immediately	Imm./Batched	★

	ChameleonDB [55]	**RocksDB** [18]	**MatrixKV** [53]	**NoveLSM** [27]	**WALSH(-WAL)**
Type	LSM Tree Hash	LSM Tree	LSM Tree	LSM Tree	Flat Hash
Scheme	DRAM-PM	DRAM-PM	DRAM-PM	DRAM-PM	DRAM-PM
Write Opt.	Aggregate	Aggregate	Aggregate	Aggregate	Aggregate
Durability	Batched♣	Batched	Batched	Batched	Imm. (Batched)

(♣: No Precise Statement in Its Paper; ★: Asynchronous Semantics).

of `clfushes` are required), write operations in existing PM hashes often cannot efficiently utilize the write cache on the CPU and PM, resulting in significantly worse performance than read performance. (2) Read performance can be easily optimized through external caching mechanisms, which is complementary to the write optimization of WALSH.

Skewed Workload. WALSH can inherently tolerate skewed workloads (a subset of the objects are much more frequently accessed than the others) well. On one hand, WALSH will detect the skewed chained lists and extend the table to disperse hot objects. When operations are concentrated in a specific key range, WALSH needs to frequently expand the directories to ensure that the length of the chained list does not exceed the threshold. Thanks to the partial resizing mechanism, the expansion only needs to act on the corresponding log area. This fine-grained expansion allows WALSH to efficiently cope with any type of skew workload. On the other hand, because update operations are always appended to the heads of chained lists, the newest versions of the hot objects tend to be closer to the heads, which makes searching hot objects efficient.

Generalizability. Although this work uses 3DXPoint for prototyping since it is the only commercially available product, we can simply modify the configuration of WALSH (e.g., write buffer size) to make it suitable for the future PMs with different write units.

NUMA. Similar to most related hash studies [4, 28, 30, 38, 40, 58], WALSH is not specifically optimized for NUMA. In fact, there are some techniques [5, 48] that can transparently and efficiently run a non-NUMA-aware index on NUMA architecture, which are also useful for WALSH.

4 Evaluation

The evaluation covers a wide range of scenarios, including *write*- and *read*-intensive, *uniform* and *skew*, *micro benchmarks,* and *real-word workloads.* We compare WALSH to a wide set of related works (Table 2), including state-of-the-art hybrid memory hashes (our focus) as well as PM and DRAM hashes (as baselines).

Flat Hashes on Hybrid Memory. We evaluated Viper [4], FlatStore-Hash [8, 47], and Halo [20]. For conciseness, we mainly discuss the comparison with Viper here. Both Viper and FlatStore maintain a CCEH in DRAM as a object location index and serializes small PM writes outside the hash. Our experiments show that performance, DRAM footprint, and recovery time of Viper and FlatStore are on the same order. Halo uses asynchronous semantics (i.e., the return of the operation does not mean completion), and it also maintains a complete index in DRAM which suffers from the same limitations as Viper.

Tree Hashes on Hybrid Memory. Plush is chosen as a representative hash of this type. The main difference between tree hashes and flat hashes is that tree hashes cannot guarantee constant time complexity on average. Our experiments show that the performance of Plush drops by about 30% when its tree depth grows from 1 to 2. ChameleonDB [55] is omitted because it is similar to Plush, but it ensures stable performance on datasets of different sizes by using a lot of DRAM space as a cache (over twice the DRAM footprint of WALSH).

PM Hashes. Four state-of-the-art PM hashes are used in our evaluation, including Level hash (LEVEL) [58], CCEH [40], DASH (extendible hash version) [38], and PCLHT [30].

DRAM Hashes. Four mainstream DRAM hashes are considered, including TBB [22], CLHT [15], Cuckoo hash (CUCKOO) [19, 33], and DRAM-CCEH (DCCEH) [40].

Write Aggregating Sorted Trees. Three write aggregating trees are also considered: MatrixKV [53] and NoveLSM [27] are based on PM-optimized LSM-Tree; RocksDB [18] is based on LSM-Tree without PM optimization. NVLSM [54] leverages PM to optimize the compaction of LSM tree, and it is omitted in the experiments because its performance is at the same level of RocksDB. We also omit detailed discussion of other write aggregating trees that are not optimized for PM (e.g., PebblesDB [44], TRIAD [1], KVell [31, 32], SILK [2], SplinterDB [12]) or designed for the PM-DISK (no DRAM) storage scheme (e.g., SLM-DB [26]).

Hardware Setup. We run the experiments on a server with an Intel Xeon Golden 6230N CPU (20 physical cores and 27.5MB L3 cache, hyperthreading enabled), 192GB of RAM (32GB×6, DDR4, 2666MHz), and 256GB Intel Optane 3DXPoint DCPMM (128GB×2). The PMs are used in the `app-direct` mode. The server runs Ubuntu 22.04 with Linux 5.15 and PMDK 1.12. PMDK is set to the `fsdax` mode and runs upon EXT4-DAX.

Software Setup. All code is compiled using GCC 11.2 with optimization enabled. The object size is 8B key and 8B value if not otherwise specified. Some tested hashes will crash with a large dataset; we use the dataset with which all other hashes work correctly for each experiment. We evaluate WALSH in both the default mode (denoted as *WALSH*, batch durability, 512 log areas) and write-ahead operation log mode (denoted as *WALSH-WAL*, immediate durability, 32 log areas). The maximum average length of chained lists (threshold for resizing/GC a log area) is set to 4; the number of entries per log is set to 256; the initial number of directories is 262,144 (all tested hashes are initialized from a very small size relative to the test dataset size). For the other hashes and trees, we use the default settings provided by their authors.

4.1 Micro Benchmark

We perform 500 million inserts, updates, deletes, and positive and negative searches to evaluate the performance of WALSH (batch durability), WALSH-WAL (same durability as other hashes), and the related PM/hybrid memory hashes. We use the random function in C++ to generate keys in the dataset used in the experiments while ensuring that they are non-repetitive. All operations in the experiments are also random and non-repetitive. The difference between sequential and random access is not significant in a hash index because objects are evenly distributed over the table. Note that for WALSH and WALSH-WAL, partial resizing/GC take place as described in Section 3.2. Specifically, for WALSH with 512 log areas, resizing+GC is performed 4,608 times during the Insert workload, and GC is performed 512 times during the Update and Delete workload.

Insert (Figure 6(a)). Compared to the PM hashes, WALSH achieves 1.3X to 17.5X speedup and WALSH-WAL achieves up to 9.4X speedup because most PM writes in WALSH/WALSH-WAL are large and sequential, which significantly reduce the hardware write amplification (as Figure 2 shows) and mitigate the impact of poor concurrency of PM. Compared to hybrid memory flat hashes, WALSH's performance is about 1.3X that of Viper in 40 threads, but WALSH-WAL has no advantage. Moreover, WALSH's DRAM footprint is only 13.5% of Viper in this experiment. Plush is

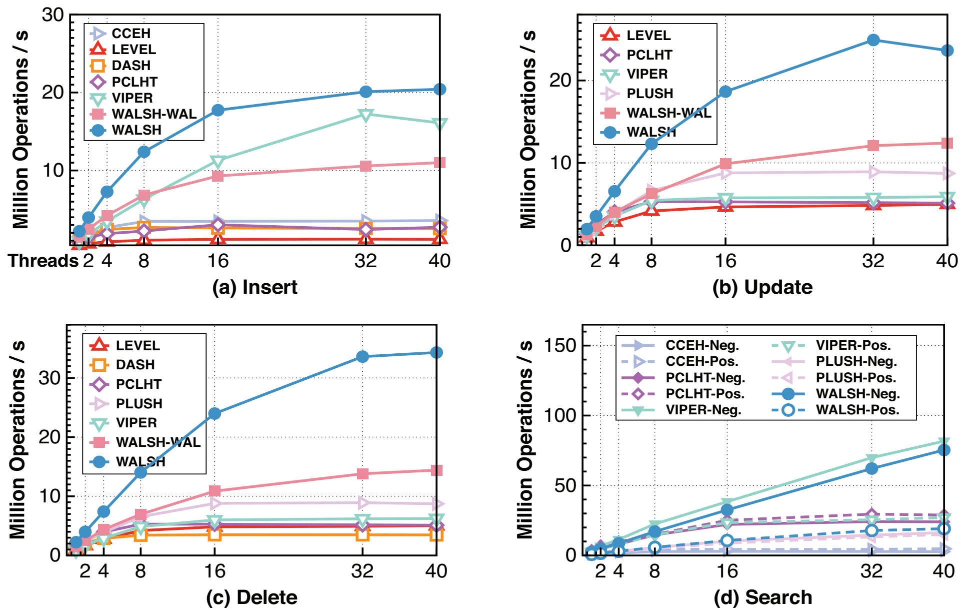


Fig. 6. Micro benchmark. Compared to Viper (same type as WALSH), WALSH reduces the DRAM footprint by **86.5%**.

not included in this experiment as it only provides weaker semantics, i.e., it does not check object existence before insert.

Update and Delete (Figure 6(b) and 6(c), some hashes do not provide update/delete interface natively). The performance of WALSH and WALSH-WAL is up to 9.9X and 2.5X that of the other hashes, respectively. Due to the log-structured design, update and delete are executed similarly to insert in WALSH (both modes), so they can also benefit from aggregating and serializing PM writes in WALSH. Among the other hybrid memory hashes, Viper cannot optimize small writes in update operations because updates will be performed in-place; Plush is designed based on LSM-tree, which brings non-constant time complexity and expensive table compaction.

Search (Figure 6(d)). WALSH provides a similar performance in both modes. For clarity, only the best and worst PM hashes are shown. Since WALSH keeps the chained lists short through resizing and GC, its search performance can reach 70% to 1.7X of existing PM hashes in positive search, which is reasonable since most of the hash is stored in PM. However, WALSH achieves up to 27.4X speedup for negative search: benefiting from the bloom filter in the directory, WALSH has a high probability of not needing to traverse the chained list in PM, which allows a lot of negative searches to be performed only in DRAM. Viper performs better than WALSH in positive search (by 30%) and negative search (by 7%), but at the cost of much higher DRAM space usage (7.4X); and WALSH performs much better than Viper (5.5X) in write performance. The performance of WALSH is up to 1.3X and 4.6X that of Plush in positive and negative search, respectively, because Plush inherits the disadvantage of LSM tree in search. These results show that WALSH makes a good tradeoff between write and search performance and DRAM usage.

At the same time, we evaluate the performance of raw PM at 16B random write (each operation is accompanied by `clflush` and `mfence`) and random read as the hardware baseline, which is the same I/O load as the micro benchmark. Table 3 shows the results. For write operations, there is a large gap between the performance of PM hashes and the PM hardware limit, while hybrid

Table 3. Raw PM Performance

Threads	**1**	**2**	**4**	**8**	**16**	**32**	**40**
16B Write	3.5	6.3	9.6	12.0	12.2	12.5	12.1
16B Read	3.3	7.0	10.6	19.8	34.7	41.1	56.4

Random read/write, in millions of operations per second.

Table 4. Workload Details of YCSB

Workload	**Read/Update/Insert**	**Types**	**Distribution**
Workload A	50% / 50% / 0%	Balanced	Zipfian (skewness of 0.99)
Workload B	95% / 5% / 0%	Read mostly	Zipfian (skewness of 0.99)
Workload C	100% / 0% / 0%	Read only	Zipfian (skewness of 0.99)
Workload D	95% / 0% / 5%	Read latest	Zipfian (skewness of 0.99)
Workload UH	20% / 80% / 0%	Update heavy	Uniform

hashes can exceed the PM hardware limit. This is because the hybrid hashes can utilize DRAM to optimize the write behavior on the PM, that is, turning small random writes into large sequential writes. For read operations, both PM and hybrid hashes cannot exceed the PM hardware limit in the case of positive search, because the search process is often accompanied by multiple PM accesses. However, hybrid hashes (WALSH and Viper) can approach/exceed the PM hardware limits in the case of negative search because they keep the entire index in DRAM (Viper) or use bloom filters (WALSH) to avoid/reduce the number of PM accesses for search.

4.2 Macro Benchmark

We evaluate WALSH/-WAL on two representative macro benchmarks to show the performance in real-world workloads. YCSB characterizes the real workloads on Yahoo! Cloud [13], while dbbench characterizes the real workloads on Facebook [6].

4.2.1 YCSB. Table 4 describes the workloads. Workload A to D are the *standard workloads* in YCSB. Because the standard workloads are all skewed, we additionally generate Workload UH for testing the uniform write-intensive scenario. For each workload, we load a dataset containing 300 million objects and run 300 million operations with 40 threads. The workloads cover various types of I/O patterns, including zipfian (Workload A to D), latest (Workload D), and uniform (Workload UH). It is worth noting that in a hash index, the difference between sequential and random patterns is not significant, because objects are evenly distributed across the table through the hash algorithm (i.e., it is difficult for us to give an access sequence that makes it a sequential access on the hash table). Figure 7 shows the results.

Write-Intensive Scenarios (Workload A and UA). For these workloads, the performance of WALSH and WALSH-WAL is 1.2X to 11.5X of the other tested hashes (Plush hangs on Workload A). The performance advantage mainly comes from the fact that WALSH reduces a lot of small random writes on PM and improves the write concurrency. In addition, similar to the micro benchmark (Section 4.1), WALSH and WALSH-WAL have much lower DRAM footprint than Viper, another hybrid memory flat hash in this experiment.

Read-Intensive Scenarios (Workload B, C, and D). For these workloads, WALSH and WALSH-WAL perform better (up to 3.5X) than all the other hashes except for PCLHT. As discussed in Section 3.5, the cacheline-aware design of PCLHT brings high cache locality in search, and allows it to outperform WALSH/WALSH-WAL by 30% for the read-only workload C. But our strength in write performance manifests even in the read-mostly Workload B and D; with only 5%

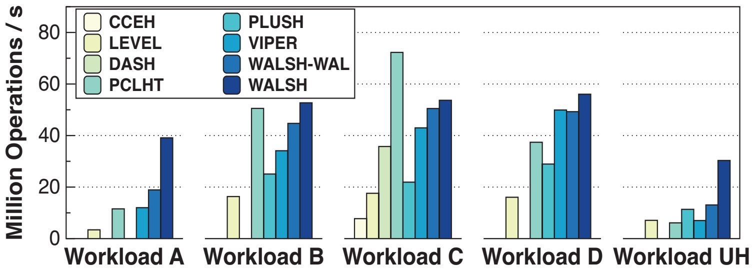


Fig. 7. YCSB performance. Workload A to D are YCSB's standard workloads, reflecting the real I/O behavior in cloud scenarios. Workload UH is our own customized load.

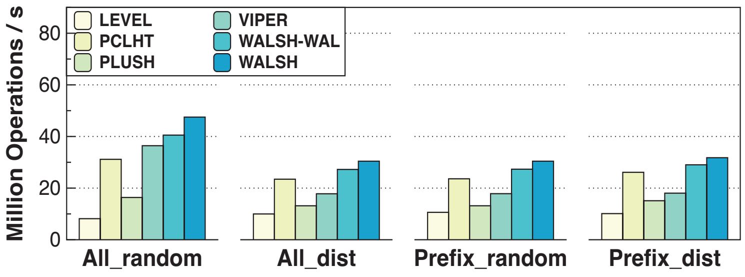


Fig. 8. dbbench performance. Workloads are characterized from the real-world traces in Facebook.

Table 5. Workload Details of dbbench

Workload	**I/O Pattern**
All_random	Objects are randomly distributed across the whole key space
All_dist	Hot objects are placed together in the whole key space
Prefix_random	Hot objects are randomly distributed in each key range
Prefix_dist	Hot objects are distributed in several key range

writes in the workloads, WALSH/WALSH-WAL can deliver a performance comparable to PCLHT, despite its disadvantage in search. For the write-intensive workloads (A and UH), WALSH/WALSH-WAL substantially outperform PCLHT (by more than 1.6X).

WALSH/WALSH-WAL also outperform Viper for all the read-intensive YCSB workloads, although they are worse for the Search micro benchmark. This is because the YCSB workloads are skewed (in comparison, the Search micro benchmark is uniform), and WALSH /WALSH-WAL are friendly to skewed workloads (the log-structured design allows frequently updated objects to be close to the heads of the lists).

4.2.2 dbbench. Table 5 shows the details of the workloads. We use four workloads provided by Cao, et al. [6] to conduct the experiments. The core difference between these workloads and YCSB is that they consider the key-space locality, which is characterized from the real world traces of Facebook. We load 300 million objects into the tested indexes and run the benchmark with 40 threads. All workloads consist of 84% update operations and 16% lookup operations. Because all workloads include update operations, we skip the hashes that do not support it.

Figure 8 shows the results. WALSH/WALSH-WAL outperforms the PM hashes (LEVEL, PCLHT, PLUSH) by 10% to 34% and outperforms the other hybrid hash (Viper) by 10% to 43%. These workloads are read intensive, which limits the performance advantage of WALSH. At the same time,

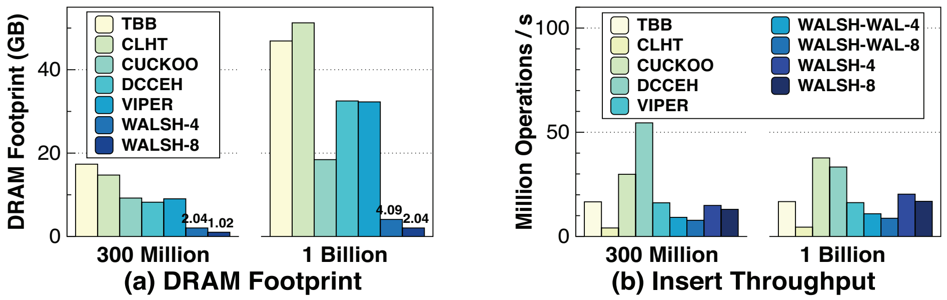


Fig. 9. DRAM footprint and insert throughput of DRAM/Hybrid memory hashes. For WALSH, all DRAM usage is calculated, including directories, logs, etc.

the All_dist and Prefix_dist cases show that WALSH can handle workloads with strong key-range locality well. This is because the partial resizing mechanism can ensure that the chained lists in the hot key-range are of the short lengths with low overhead. Furthermore, since the new version of the object generated by the update will be at the front of the chained list, this helps to further optimize the search efficiency under strong locality workload.

4.3 DRAM Footprint

We study the tradeoff between the write performance and DRAM footprint of WALSH and the other flat hashes on DRAM/hybrid memory. We evaluate WALSH with two resizing/log collection threshold values: the maximum average chained list length of 4 (*WALSH-4*, *WALSH-WAL-4*) and 8 (*WALSH-8*, *WALSH-WAL-8*). For each hash, we use 40 threads to perform 300 million and 1 billion inserts and measure the DRAM usage after all insertions are completed.

Figure 9 shows the results. Compared to hybrid memory hashes, WALSH/WALSH-WAL exceed or approach Viper in throughput (up to 1.3X) but saves more than 87.3% of DRAM space. Compared with DRAM hashes, WALSH/WALSH-WAL outperform or are close to TBB and CLHT (up to 4.5X), and can achieve about 53.8% to 60.1% of CUCKOO and DCCEH in terms of throughput on the 1 billion dataset; but WALSH/WALSH-WAL can save about 77.8% to 96% of DRAM space. These experiments show that WALSH has a significantly smaller DRAM footprint than the existing flat hashes on DRAM and hybrid memory, but it is highly competitive in write performance.

4.4 Dataset Size Sensitivity

For write operations, the results of Figure 9(b) show that the insert performance is almost the same as the dataset size increases from 300 million to 1 billion. For read operations, since the chained list length will be kept below 4 through resizing, the search performance is not sensitive to the dataset size. In principle, WALSH is a flat hash index that guarantees a constant computational complexity on average.

4.5 Tail Latency

We evaluate tail latency of flat hashes by inserting 500 million objects with 40 threads and calculating the cumulative distribution of the latency. Figure 10 shows that the insertion latency of 99.9% writes in WALSH (both modes) is lower (<80%) than the other flat hashes on PM and hybrid memory, while it is similar to or even better than DRAM hashes. There are two main reasons for this result. First, the write aggregation design of WALSH amortizes the high latency of PM. Second, the good concurrency of WALSH reduces the latency caused by blocking between threads. But WALSH has no advantage in extremely high percentile latency (99.999th percentile) compared

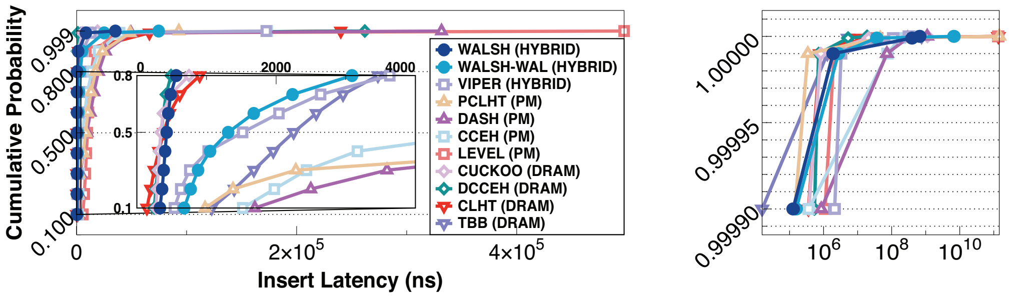


Fig. 10. Insertion latency (P0 to P100).

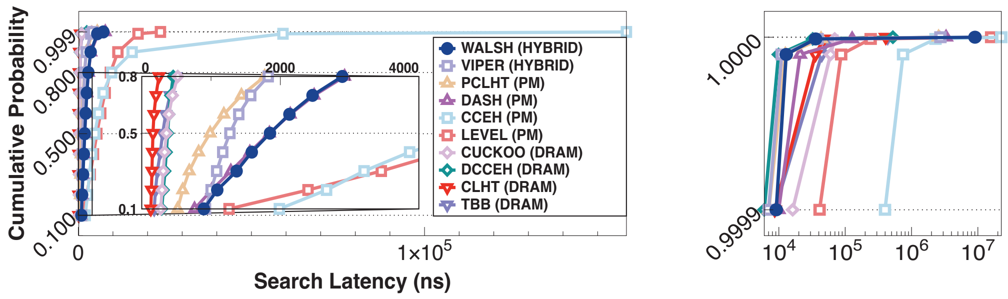


Fig. 11. Search latency (P0 to P100).

to other dynamic hashes (no need for rehashing in resizing), but it still outperforms other static hashes (need rehashing in resizing).

Figure 11 shows the tail latency of search. DRAM hashes are undoubtedly much better than PM and hybrid hashes in terms of search latency (used to show the upper bound). Consider the case of P99.999, WALSH's latency is 85%, 98%, and 38% lower than Level hash, CCEH, and DASH, respectively. These PM hashes require multiple PM accesses during a search, while WALSH limits the length of the chained list through the efficient partial resizing mechanism, thereby limiting the number of PM accesses during the search process. In P99.999 case, WALSH's latency is 3% and 14% higher than PCLHT and Viper, respectively. However, they make huge trade-offs in other important aspects: PCLHT sacrifices a lot of write performance to ensure cacheline locality for search, e.g., its insert throughput is 75% less than WALSH-WAL with the same durabilty guarantees; Viper uses a lot of DRAM space, which is about 5X more than WALSH/WALSH-WAL. This indicates that WALSH/WALSH-WAL makes a better tradeoff between the performance and other important aspects.

4.6 Design Analysis

WALSH includes three contributions: (1) log-structured separate chaining, (2) partial resizing/GC, and (3) 4-Phase concurrency protocol. The first contribution is the basic structure of WALSH, which is used as the baseline. We then test the improvement after adding the other two contributions. WALSH addresses the challenges on hash resizing and GC brought by log-structured separate chaining with partial resizing/GC mechanism and 4-Phase concurrency control. We evaluate their contributions to insert and update performance. The *baseline* in Figure 12 is WALSH with full table resizing/GC and complete lock for the entire resizing/GC operation. Although the overhead of resizing and GC cannot be ignored (compared to the ideal case without

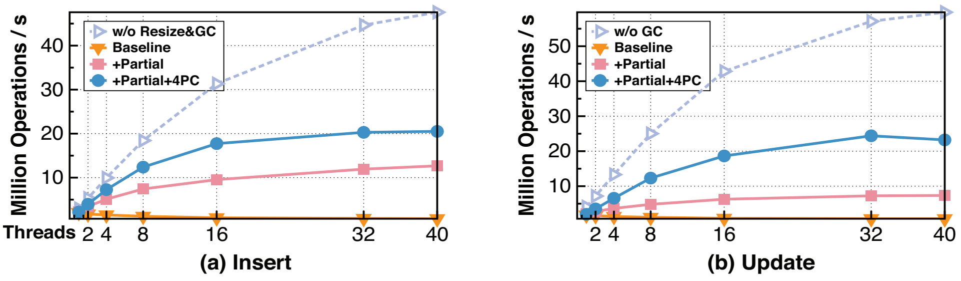


Fig. 12. Design analysis. The "*baseline*" is WALSH with full table resizing/GC and complete lock for the entire resizing/GC operation. The "*+Partial*" is WALSH with only the partial resizing optimization. The "*+Partial+4PC*" is WALSH with all optimizations. The "*w/o Resize&GC*" is the ideal case.

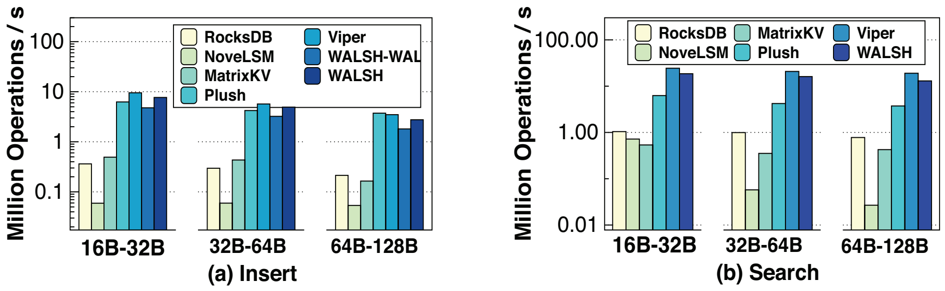


Fig. 13. Performance of variable-length objects.

resizing/GC), our optimizations significantly reduce their impact. Figure 12 shows that adding the partial resizing/GC mechanism (*+Partial*) can improve the performance by up to 19.2X and 10.5X for insert and update respectively, and adding 4-Phase concurrency control (*+Partial+4PC*) can further improve by 1.9X and 3.4X compared to enabling only partial resizing/GC.

4.7 Variable-Length Objects

We compare WALSH with Viper, Plush, and some typical write aggregating trees in the case of variable-length objects. Only WALSH, Viper, and Plush natively support variable-length objects among the tested PM/hybrid hashes; the block device layer of NoveLSM and MatrixKV is run on PM. RocksDB is run on EXT4 without DAX as it is not PM-aware. In terms of durability, WALSH-WAL, Viper, and Plush provide immediate durability while WALSH and the other trees provide batch durability. In addition, all LSM-tree-based indexes (RocksDB, NoveLSM, MatrixKV, and Plush) do not perform existence checks when inserting objects.

We perform 50 million inserts and positive searches for evaluation. The object size ranges from 16B/32B to 64B/128B (key size/value size). WALSH is mainly designed for small objects; for large objects (e.g., >256B), they benefit little from write aggregation. Figure 13 shows the results. The comparison between WALSH and the other PM/hybrid memory hashes is similar to the previous results from 8B-object workloads. For the LSM-tree-based indexes, Plush has a write performance similar to WALSH and Viper because the dataset is smaller compared to its initial size (its DRAM footprint is static, but much higher than WALSH in this experiment), and its read performance is worse than these flat hashes (<=86%); other LSM-Trees perform an order of magnitude slower than WALSH (with similar DRAM footprint), since they are not flat unsorted indexes, but they do provide richer features (e.g., range query) that hashes typically do not support.

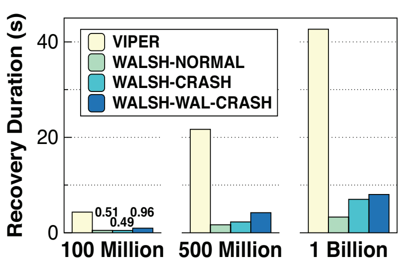


Fig. 14. Recovery duration.

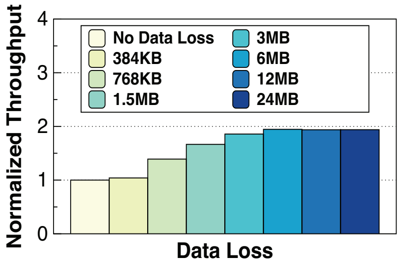


Fig. 15. Data durability and performance tradeoff.

4.8 Recovery

We compare the recovery overhead of WALSH to another flat hash on hybrid memory, Viper. WALSH/WALSH-WAL are evaluated in the normal shutdown case (close function is called) and the crash case (close function is not called) under different sizes of dataset (100 million to 1 billion objects). We use single thread for normal recovery and 8 threads for crash recovery (multithreading is not needed for normal recovery). Figure 14 shows that WALSH takes only about 3.3s and 7s to recover a 1 billion dataset under normal shutdown and crash conditions respectively, while Viper needs about 42.7s in the same situation. This is because WALSH only needs to rebuild the directories in DRAM but Viper needs to rebuild the entire table, and the concurrent crash recovery design of WALSH also contributes to the short recovery time. The crash recovery overhead of WALSH-WAL is slightly higher than that of WALSH, because in the WAL mode, the recovery process also needs to redo the write-ahead operation logs; but it is still much lower than Viper in each case.

4.9 Parameter Sensitivity

We analyze the tunable parameters in WALSH:

(1) Length of the chained lists. The maximum average length of the chained lists in a log area determines the threshold of resizing and GC, which affects the DRAM footprint and performance of WALSH. We reuse the results in Figure 9. When the threshold increases from 4 to 8, the DRAM footprint reduces by 50%, and the insert performance reduces by less than 20%. The main reason of this result is that longer chained lists reduce the number of directories in DRAM, but increase the object traversing overhead and reduce the bloom filter efficiency. Increasing this threshold also affects search performance: when it grows from 4 to 8, the performance of positive and negative search (in the 1 billion objects case) drops by about 32.9% and 45.2%, respectively.

(2) Size of log. It decides the size of data buffered in DRAM and the write granularity in PM, which affects data durability and write performance. We show its effect by comparing the insert throughput (with 40 threads) under a different maximum size of data loss (depending on the total size of logs buffered in DRAM). We normalize the throughput to WALSH-WAL which is the mode that guarantees data durability. Figure 15 shows that as the maximum size of data loss increases from 0 to 6MB, the performance increases up to 1.95X that of WALSH-WAL because the efficiency of write aggregation is improved, but further increase does not help the performance.

(3) Number of log areas. The number of log areas mainly affects the concurrency. Increasing this number from 1 to 512 can bring about 30X increase in WALSH's write performance, and continuing to increase the number will no longer have much impact on performance.

4.10 Load Factor and PM Utilization

Load factor is the ratio of used objects to allocated objects. The load factor of WALSH is nearly 1 because only a fixed number of objects in DRAM logs (by default 256×512) are pre-allocated, and this number is insignificant compared to the total size of a large dataset. Due to the log-structured design of WALSH, there may be multiple versions of objects and directories on PM. Fortunately, WALSH can maintain a good PM utilization by collecting expired logs and directories. For a dataset containing 500 million objects, WALSH requires about 14GB of PM space (after collecting expired directories), while CCEH and PCLHT require about 16GB and 49GB (calculated from their DRAM versions), respectively.

5 Conclusion

This paper presents WALSH to simultaneously achieve the goals of high performance, low DRAM footprint, and efficient recovery for hash tables on hybrid memory. WALSH is a novel log-structured separate chaining hash and it addresses the challenges in resizing and GC. A comprehensive evaluation shows that WALSH makes good tradeoffs between multiple important dimensions: it substantially improves the performance of insert, update, and delete operations, provides a comparable search performance, and significantly saves the DRAM footprint and recovery time better than the state of the art.

Acknowledgments

We appreciate the insightful comments from the anonymous reviewers. The work was done while Dr. Yubo Liu was a visiting scholar at Arizona State University.

References

[1] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. In *Proceedings of USENIX Technical Conference (ATC'17)*. 363–375.

[2] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing latency spikes in log-structured merge key-value stores. In *Proceedings of USENIX Technical Conference (ATC'19)*. 753–766.

[3] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. 2007. Cache-oblivious streaming B-trees. In *Proceedings of ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'07)*. 81–92.

[4] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. 2021. Viper: An efficient hybrid PMem-DRAM key-value store. In *Proceedings of the VLDB Endowment*, Vol. 14. 1544–1556.

[5] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. 2017. Black-box concurrent data structures for NUMA architectures. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. 207–221.

[6] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, modeling, and benchmarking RocksDB Key-ValueWorkloads at Facebook. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST'20)*. 209–223.

[7] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. SpanDB: A fast, cost-effective LSM-tree based KV store on hybrid storage. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST'21)*. 17–32.

[8] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. 1077–1091.

[9] Zhangyu Chen, Yu Hua, Bo Ding, and Pengfei Zuo. 2020. Lock-free concurrent level hashing for persistent memory. In *Proceedings of USENIX Technical Conference (ATC'20)*. 799–812.

[10] Zhangyu Chen, Yu Hua, Luochangqi Ding, Bo Ding, Pengfei Zuo, and Xue Liu. 2022. Lock-free high-performance hashing for persistent memory via PM-aware holistic optimization. *ACM Transactions on Architecture and Code Optimization (TACO)* (2022).

[11] Alex Conway, Martin Farach-Colton, and Philip Shilane. 2018. Optimal hashing in external memory. In *Proceedings of International Colloquium on Automata, Languages, and Programming (ICALP'18)*. 1–14.

[12] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. 2020. SplinterDB: Closing the bandwidth gap for NVMe key-value stores. In *Proceedings of USENIX Technical Conference (ATC'20)*. 49–63.

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of ACM Symposium on Cloud Computing (SoCC'10)*. 143–154.

[14] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. 631–644.

[15] Tudor Alexandru David, Rachid Guerraoui, Tong Che, and Vasileios Trigonakis. 2014. *Designing ASCY-compliant Concurrent Search Data Structures*. Technical Report.

[16] Biplob Debnath, Alireza Haghdoost, Asim Kadav, Mohammed G. Khatib, and Cristian Ungureanu. 2016. Revisiting hash table design for phase change memory. *ACM SIGOPS Operating Systems Review* 49, 2 (2016), 18–26.

[17] Anthony Demeri, Wook-Hee Kim, R. Madhava Krishnan, Jaeho Kim, Mohannad Ismail, and Changwoo Min. 2020. Poseidon: Safe, fast and scalable persistent memory allocator. In *Proceedings of International Middleware Conference (Middleware'20)*. 207–220.

[18] Facebook. 2021. RocksDB. http://rocksdb.org

[19] Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. In *Proceedings of Symposium on Networked Systems Design and Implementation (NSDI'13)*. 371–384.

[20] Daokun Hu, Zhiwen Chen, Wenkui Che, Jianhua Sun, and Hao Chen. 2022. Halo: A hybrid PMEM-DRAM persistent hash index with fast recovery. In *Proceedings of International Conference on Management of Data (SIGMOD'22)*. 1049–1063.

[21] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. 2018. Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs. In *Proceedings of USENIX Technical Conference (ATC'18)*. 967–979.

[22] Intel. 2020. Intel Threading Building Blocks Documents. https://software.intel.com/content/www/us/en/develop/documentation/tbb-documentation

[23] Intel. 2020. Persistent Memory Development Kit. http://pmem.io/pmdk

[24] Intel. 2021. 3D XPoint DCPMM. https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-dc-persistent-memory

[25] Myoungsoo Jung. 2022. Hello bytes, bye blocks: PCIe storage meets compute express link for memory expansion (CXL-SSD). In *Proceedings of Conference on Hot Topics in Storage and File Systems (HotStorage'22)*. 45–51.

[26] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young-ri Choi. 2019. SLM-DB: Single-level key-value store with persistent memory. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST'19)*. 191–205.

[27] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for nonvolatile memory with NoveLSM. In *Proceedings of USENIX Technical Conference (ATC'18)*. 993–1005.

[28] R. Madhava Krishnan, Wook-Hee Kim, Xinwei Fu, Sumit Kumar Monga, Hee Won Lee, Minsung Jang, Ajit Mathew, and Changwoo Min. 2021. TIPS: Making volatile index structures persistent with DRAM-NVMM tiering. In *Proceedings of USENIX Technical Conference (ATC'21)*. 805–820.

[29] Miryeong Kwon, Sangwon Lee, and Myoungsoo Jung. 2023. Cache in hand: Expander-driven CXL prefetcher for next generation CXL-SSD. In *Proceedings of Conference on Hot Topics in Storage and File Systems (HotStorage'23)*. 24–30.

[30] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. RECIPE: Converting concurrent DRAM indexes to persistent-memory indexes. In *Proceedings of Symposium on Operating Systems Principles (SOSP'19)*. 462–477.

[31] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: The design and implementation of a fast persistent key-value store. In *Proceedings of Symposium on Operating Systems Principles (SOSP'19)*. 447–461.

[32] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2020. KVell+: Snapshot isolation without snapshots. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. 425–441.

[33] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A holistic approach to fast in-memory key-value storage. In *Proceedings of Symposium on Networked Systems Design and Implementation (NSDI'14)*. 429–444.

[34] Witold Litwin. 1980. Linear Hashing: A new tool for file and table addressing. In *Proceedings of the VLDB Endowment*, Vol. 80. 1–3.

[35] Yubo Liu, Hongbo Li, Yutong Lu, Zhiguang Chen, Nong Xiao, and Ming Zhao. 2020. HasFS: Optimizing file system consistency mechanism on NVM-based hybrid storage architecture. *Cluster Computing* 23 (2020), 2510–2515.

[36] Yubo Liu, Yuxin Ren, Mingrui Liu, Hanjun Guo, Xie Miao, and Xinwei Hu. 2023. Cache or direct access? Revitalizing cache in heterogeneous memory file system. In *Proceedings of the 1st Workshop on Disruptive Memory Systems (DIMES'23)*. 38–44.

[37] Yubo Liu, Yuxin Ren, Mingrui Liu, Hongbo Li, Hanjun Guo, Xie Miao, Xinwei Hu, and Haibo Chen. 2024. Optimizing file systems on heterogeneous memory by integrating DRAM cache with virtual memory management. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST'24)*. 71–87.

[38] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable hashing on persistent memory. In *Proceedings of the VLDB Endowment*, Vol. 13. 1147–1161.

[39] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. 2020. Pronto: Easy and fast persistence for volatile data structures. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. 789–806.

[40] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-optimized dynamic hashing for persistent memory. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST'19)*. 31–44.

[41] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[42] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004), 122–144.

[43] Percona. 2021. PerconaFT. https://github.com/Percona/PerconaFT

[44] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of Symposium on Operating Systems Principles (SOSP'17)*. 497–514.

[45] Lukas Vogel, Alexander van Renen, Satoshi Imamura, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2022. Plush: A write-optimized persistent log-structured hash-table. In *Proceedings of the VLDB Endowment*, Vol. 15.

[46] Chao Wang, Junliang Hu, Tsun-Yu Yang, Yuhong Liang, and Ming-Chang Yang. 2023. SEPH: Scalable, efficient, and predictable hashing on persistent memory. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)*. 479–495.

[47] Jing Wang, Youyou Lu, Qing Wang, Minhui Xie, Keji Huang, and Jiwu Shu. 2022. Pacman: An efficient compaction approach for log-structured key-value store on persistent memory. In *Proceedings of USENIX Technical Conference (ATC'22)*. 773–788.

[48] Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu. 2021. Nap: A black-box approach to NUMA-aware persistent memory indexes. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*. 93–111.

[49] Yongfeng Wang, Yinjin Fu, Yubo Liu, Zhiguang Chen, and Nong Xiao. 2022. Characterizing and optimizing hybrid DRAM-PM main memory system with application awareness. In *Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE'22)*. 879–884.

[50] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. 2020. Characterizing and modeling non-volatile memory systems. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO'20)*.

[51] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A hybrid index key-value store for DRAM-NVM memory systems. In *Proceedings of USENIX Technical Conference (ATC'17)*. 349–362.

[52] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST'20)*. 169–182.

[53] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM. In *Proceedings of USENIX Technical Conference (ATC'20)*. 17–31.

[54] Baoquan Zhang and David H. C. Du. 2021. NVLSM: A persistent memory key-value store using log-structured merge tree with accumulative compaction. *ACM Transactions on Storage* 17, 3 (2021), 1–26.

[55] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. 2021. ChameleonDB: A key-value store for optane persistent memory. In *Proceedings of European Conference on Computer Systems (EuroSys'21)*. 194–209.

[56] Xiaomin Zou, Fang Wang, Dan Feng, Janxi Chen, Chaojie Liu, Fan Li, and Nan Su. 2020. HMEH: Write-optimal extendible hashing for hybrid DRAM-NVM memory. In *Proceedings of International Conference on Massive Storage Systems and Technology (MSST'20).*

[57] Pengfei Zuo and Yu Hua. 2017. A write-friendly hashing scheme for non-volatile memory systems. In *Proceedings of International Conference on Massive Storage Systems and Technology (MSST'17).*

[58] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-optimized and high-performance hashing index scheme for persistent memory. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI'18).* 461–476.

Received 23 February 2024; revised 28 July 2024; accepted 17 October 2024