Write-Optimized Persistent Hash Index for Non-Volatile Memory

Renzhi Xiao[†], Dan Feng[*‡§], Yuchong Hu[*§‖], Yucheng Zhang[‡], Lanlan Cui[¶], Lin Wang[§]
[†]*School of Software Engineering, Jiangxi University of Science and Technology, Nanchang, China*
[‡]*Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, China*
[§]*School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China*
[¶]*School of Computer Science and Engineering, XI'AN University of Technology, Xi'an, China*
[‖]*Shenzhen Huazhong University of Science and Technology Research Institute*
[*]*Correspongding Author: Dan Feng (dfeng@hust.edu.cn) and Yuchong Hu (yuchonghu@hust.edu.cn)*
renzhixiaohust@gmail.com, zhangyc_hust@126.com, cuilanlan@xaut.edu.cn, wanglin2021@hust.edu.cn

***Abstract*—A hashing index provides rapid search performance by swiftly locating key-value items. Non-volatile memory (NVM) technologies have driven research into hashing indexes for NVM, combining hard disk persistence with DRAM-level performance. Nevertheless, current NVM-based hashing indexes must tackle data inconsistency challenges caused by NVM write reordering or partial writes, and mitigate rapid local wear due to frequent updates, considering NVM's limited endurance. The temporary allocation of buckets in NVM-based chained hashing to resolve hash collisions prolongs the critical path for writing, thus hampering write performance. This paper presents WOPHI, a write-optimized persistent hash index scheme for NVM. By utilizing log-free failure-atomic writes, WOPHI minimizes data consistency overhead and addresses hash conflicts with bucket pre-allocation. Experimental results underscore WOPHI's significant performance enhancements, with insertion latency slashed by up to 88.2% and deletion latency boosted by up to 82.6% compared to existing state-of-the-art schemes. Moreover, WOPHI substantially mitigates data consistency overhead, reducing cache line flushes by 59.3%, while maintaining robust write throughput for insert and delete operations.**

***Index Terms*—non-volatile memory, chained hashing, write endurance, data consistency**

I. Introduction

Non-Volatile Memory (NVM) technologies, such as STT-RAM [1] and Intel Optane DC Persistent Memory Module (DCPMM) [2], offer DRAM-level performance and hard disk durability. Consequently, researchers are increasingly investigating NVM-based data structures. However, NVM has limited write endurance [2], and its write latency exceeds that of reading [3]. Therefore, NVM-based data structures need to minimize the number of writes to NVM. Moreover, to enhance performance, the CPU and memory controller might reorder NVM writes, which could lead to data inconsistency issues in data structures for NVM due to reordering or partial writes in case of system failure.

Due to their constant O(1) query performance, hashing data structures are widely used in in-memory key-value databases like Memcached and Redis. Despite the fast point query performance offered by hashing, hash collisions can strain system resources, including CPU and memory. Current approaches often utilize chained hashing to manage hash collisions. However, the temporary bucket allocation process in chained hashing during collisions can extend the write critical path, resulting in heightened write latency, especially in NVM environments.

Various hashing-based structures for NVM, including PFHT [4], Group Hashing [5], Path Hashing [6], and Level Hashing [7], have been proposed in academic research over recent years. While Path Hashing and PFHT primarily address write issues, they overlook data inconsistency problems. On the other hand, Group Hashing and Level Hashing tackle both write and data inconsistency issues for NVM, but their writing performance suffers due to frequent updates in NVM.

Therefore, to address consistency and write concerns, we propose WOPHI, a write-optimized persistent hash index scheme for NVM. WOPHI employs log-free failure-atomic writes to reduce data consistency overhead and optimizes bucket pre-allocation to efficiently resolve hash conflicts. We evaluate WOPHI using two micro-benchmarks and two other macro-benchmarks derived from the YCSB framework [8]. Due to the discontinuation of Intel Optane DCPMM and the necessity to model various write latencies in NVM, we employ the widely-used Quartz simulator [9] to simulate NVM characteristics. Experimental results demonstrate that WOPHI achieves up to 88.2% improvement in insertion latency and 82.6% improvement in deletion latency over state-of-the-art schemes.

II. Background and Motivation

A. Non-Volatile Memory

Emerging non-volatile memory technologies, including STT-RAM [1] and Intel Optane DC persistent memory [2], deliver near-zero leakage power and significantly higher memory capacity, overcoming the high power consumption and scalability limitations of traditional DRAM and SRAM technologies. In addition, NVM can provide both the durability of HDD and the read and write latency near DRAM and byte addressing properties. Therefore, NVM is a promising candidate for the next generation of main memory.

978-3-9826741-0-0/DATE25/© 2025 EDAA

However, due to the inherent limitations of NVM, it cannot be directly utilized as the main memory. Firstly, the write latency is significantly higher than the read latency (by a factor of 3-8) [3], and writing also consumes more energy consumption than reading. Secondly, NVM typically has limited write endurance, such as 10^7-10^8 writes [2]. Consequently, reducing the amount of data written to NVM decreases total system latency and extends NVM system lifespan. Many prior studies have focused on enhancing NVM durability by reducing writes or mitigating uneven wear. Additionally, using NVM as the main memory shifts the volatility-persistence boundary to the interface between volatile cache and persistent NVM. In NVM-oriented systems, it is crucial to ensure data consistency, meaning maintaining data correctness in NVM following a system failure [5]. Unlike traditional secondary storage devices, NVM typically has an 8-byte failure atomic unit [5]. For write units larger than 8 bytes, the order of memory writes must be carefully managed [10]. Therefore, hashing data structures for NVM must address these NVM limitations.

B. Challenges

*1) **Data Inconsistency Upon System Failure**:* Writing a key-value pair into an NVM-based hashing may result in data inconsistency [5], [11]. The failure-atomic write size of NVM is 8 bytes for volatile CPU cache and NVM systems. In contrast, CPU cache flushes are written from volatile caches in 64-byte cache lines back into NVM. Therefore, for an update unit bigger than 8 bytes, due to the write reordering effect of the CPU cache, the data written back to the NVM may be partially updated or reordered in case of system failure. That's data inconsistency. The traditional methods of ensuring consistency, such as redo and undo log mechanisms, will bring double the write overhead, which increases the wear on the NVM and prolongs the write time.

*2) **The Bucket Allocation Operation in the Critical Write Path Increases Write Latency in the Presence of Hash Collisions**:* The traditional chained hash structure is to allocate a bucket when the hash collides temporarily, and the operation time of allocating the bucket is on the critical path of the write operation. Since NVM has a higher write latency, it will reduce the write performance further.

C. Motivation

WOPHI minimizes data consistency overhead by replacing traditional redo/undo logs with log-free failure-atomic writes. Moreover, WOPHI utilizes bucket pre-allocation to segregate bucket allocation operations from the critical write path during hash collisions, effectively reducing write latency.

III. The Design of WOPHI

In this section, we present the design of WOPHI, a write-optimized persistent hash index specifically designed for NVM.

A. The Overview of WOPHI

Figure 1 illustrates the architecture of WOPHI, comprising two components: (1) **Main Buckets (MBs)**, and (2) **Pre-Allocation Buckets for Chain (PAB)**. WOPHI primarily resides in persistent NVM, with volatile and persistent boundaries situated between the cache and NVM. The MBs consist of an addressable bucket array, with each bucket containing four token and four key-value slots, along with a pointer to the next bucket. The PAB comprises a preassigned unaddressable bucket array, where each bucket is utilized in the main buckets chain to store key-value pairs in case of hash conflicts. The bucket pre-allocation mechanism expedites the insertion of new key-value pairs while resolving hash collisions. By mitigating rapid local wear, the module helps prolong the service life of NVM, thus preventing the NVM device from deteriorating too quickly.

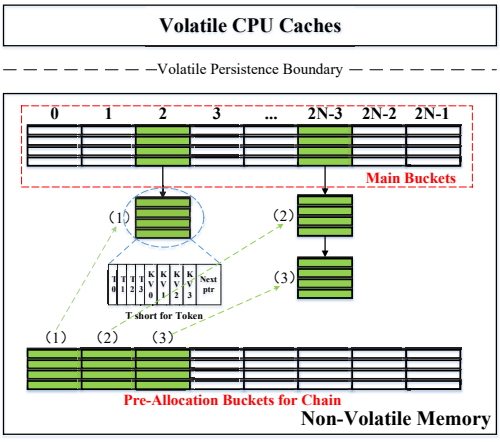

Fig. 1. The architecture of WOPHI.

WOPHI comprises two primary modules: (1) **Log-Free Failure-Atomic Writes**, and (2) **Bucket Pre-Allocation**. The following subsections delve into the functionality of each module. Figure 2 illustrates the write flow of WOPHI. When transferring data from the volatile cache to WOPHI in NVM, the system first checks for available slots in the Main Buckets. If there are available slots, data is written to the Main Buckets using Log-Free Failure-Atomic Writes. Otherwise, WOPHI creates a chain bucket via Bucket Pre-Allocation and writes the data to the PAB using Log-Free Failure-Atomic Writes.

*1) **Log-Free Failure-Atomic Writes**:* For write units larger than 8 bytes, redo/undo logging is employed to ensure data consistency. However, this approach results in double NVM writes, leading to NVM degradation and increased write latency. Copy-On-Write (COW) exacerbates the issue by causing unnecessary reads for unmodified data areas (read amplification) and unnecessary writes for unchanged data areas (write amplification). In contrast, Log-Free Failure-Atomic Writes, abbreviated as Atomic, utilize ordered 8-byte atomic writes with memory fence and cache line flush instructions, eliminating the need for logging. Atomic operations incur no additional NVM writes during insertion and deletion operations. Therefore, Atomic enhances NVM lifespan and boosts write performance.

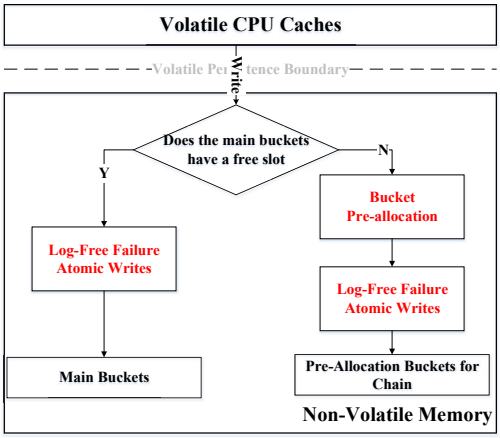

Fig. 2. The write flow of WOPHI.

*2) **Bucket Pre-Allocation**:* The Bucket Pre-Allocation (BPA) technique involves initially allocating an ample number of buckets to handle hash conflicts. By separating bucket allocation operations from critical write paths, BPA effectively reduces write latency. In contrast, traditional chained hashing dynamically allocates buckets from the heap, leading to increased insertion latency as bucket allocation operations occur within the critical write path.

B. The Basic Operation Algorithms of WOPHI

*1) **Log-Free and Consistent Insertion Algorithm Implementation**:* The insertion algorithm of WOPHI is depicted in Algorithm 1. Lines 2-5 compute the hash value of the key using the hash function and locate the corresponding multi-slot hash bucket in the chained hash based on this value. Lines 6-26 employ Log-Free Failure-Atomic Writes to insert a key-value pair into WOPHI if a free slot is available in the designated main bucket chained list. Lines 27-33 illustrate WOPHI's utilization of the PAB to address hash conflicts when a free bucket is present in the pre-allocated bucket array.

*2) **Log-Free and Consistent Deletion Algorithm Implementation**:* The deletion algorithm for WOPHI is illustrated in Algorithm 2. Lines 2-5 determine the hash value for the given key and locate the corresponding bucket chained list in WOPHI. Lines 6-17 remove the key-value pair from WOPHI using Log-Free Failure-Atomic Writes when a key-value slot with a token of 1 and a key matching the searched key is found in the respective bucket chained list. Otherwise, the deletion operation fails, as indicated in Line 18.

Algorithm 1: Insert(WOPHI,key,value)

```
1:  //Compute the hash value through the HASH function
2:  hash = HASH(WOPHI,key);
3:  //Calculate the bucket index using the IDX function
4:  index=IDX(hash,WOPHI→init_capacity);
5:  WOPHI_bucket *tb=&WOPHI→buckets[index];
6:  while tb!=NULL do
7:    for i = 0; i < ASSOC_NUM; i++ do
8:      if tb→token[i] == 0 then
9:        //Utilize the log-free failure-atomic writes method
10:       memcpy(tb→slot[i].key,key,KEY_LEN);
11:       memcpy(tb→slot[i].value,value,VALUE_LEN);
12:       pflush(&tb→slot[i].key);
13:       pflush(&tb→slot[i].value);
14:       asm_mfence();
15:       tb→token[i]=1;
16:       pflush(&tb→token[i]);
17:       asm_mfence();
18:       return TRUE;
19:     end if
20:   end for
21:   if tb→next != NULL then
22:     tb=tb→next;
23:   else
24:     break;
25:   end if
26: end while
27: With no available slots in the main bucket area, WOPHI
    must acquire a new bucket, tmp, from PAB
28: if tmp == NULL then
29:   perror("new added temporary bucket from PAB.");
30: end if
31: Apply the log-free failure-atomic writes process to set
    and flush the first slot of tmp with the KV pair and its
    token
32: WOPHI links the newly allocated bucket from PAB to a
    specific chain within the main bucket
33: Allocate a new bucket from the PAB area and
    increment the variable expand_bucket_num by 1
34: return TRUE;
```

IV. Performance Evaluation

This section assesses cache line flushes per insert, micro-benchmark latency, and macro-benchmark throughput for WOPHI, our proposed write-optimized persistent hash index solution for NVM.

A. Experimental Setup

Given the discontinuation of Optane persistent memory and the requirement for configuring four distinct NVM write latencies, we utilize the Quartz simulator [9] to enhance versatility and flexibility in emulating NVM, achieved by introducing additional delays in the CLFLUSH operation. WOPHI is deployed on a server equipped with a 2.4GHz Intel Xeon E5-2630 V3 CPU, featuring 8 cores per CPU. The Last Level Cache size is 20MB, and the total NVM capacity is 100GB. The experiments are conducted on Ubuntu 16.04 with kernel version 4.15.0.

For PFHT and Path Hashing, we introduced a redo-logging mechanism, denoted as PFHT-Log and Path-Log, respectively. Additionally, we implemented a write-friendly and consistent group hashing method using Quartz (Group). Level hashing was simplified to Level. We utilized two micro-benchmarks, Random Number and Sequence Number, to assess the number of CLFLUSH operations required for inserting key-value pairs

Algorithm 2: Delete(WOPHI,key)

```
1: //Compute the hash value through the HASH function
2: hash = HASH(WOPHI,key);
3: //Calculate the bucket index using the IDX function
4: index=IDX(hash,WOPHI→init_capacity);
5: WOPHI_bucket *tb=&WOPHI→buckets[index];
6: while tb!=NULL do
7:    for i = 0; i < ASSOC_NUM; i++ do
8:       if tb→token[i] == 1 && strcmp(tb→slot[i].key,key)
         == 0 then
9:          //When deleting, just reset the slot's token to 0
10:         tb→token[i] = 0;
11:         pflush(&tb→token[i]);
12:         asm_mfence();
13:         return TRUE;
14:      end if
15:   end for
16:   tb = tb→next;
17: end while
18: return FALSE;
```

and the latencies of insert and delete operations. Furthermore, we employed INSERT and DELETE datasets generated by YCSB [8] to evaluate the throughput of various hashing structures.

RandomNum: Generated through a random function, the dataset consists of 2^{24} keys, each comprising 16 bytes. This dataset is a standard choice across previous research works [5], [6], where random keys serve as the identifiers for the hash table's key-value pairs.

SequenceNum: The dataset comprises 2^{24} keys, each sequentially numbered from 0 to $2^{24}-1$. Each key within this dataset can serve as an identifier for hashing entries.

INSERT: In the load phase, the dataset consists of 64 million keys generated in accordance with the Zipfian distribution from the Yahoo! Cloud Serving Benchmark (YCSB) [8] workload-A.

DELETE: Similar to the INSERT dataset, the DELETE dataset encompasses the same key-value pairs. Nonetheless, unlike the INSERT dataset focusing on insertions, the DELETE dataset exclusively executes delete operations for every key-value pair present.

By default, the NVM read and write latencies are assumed to be 200 and 600 nanoseconds, respectively. The reported values are the averages of five separate measurements.

B. CLFLUSH Number per Insertion

We assess the write efficiency of various NVM-based hashing structures by examining the number of CLFLUSH operations required for inserting a key-value item. A higher number of CLFLUSH operations indicates lower insertion efficiency and vice versa. Initially, we record the total number of CLFLUSH operations when inserting key-value entries into the hash table at specific Load Factors (e.g., 0.25, 0.5, and 0.75). The average number of CLFLUSH operations needed to insert a key-value pair is then computed by dividing the total number of CLFLUSH operations by the number of key-value entries inserted.

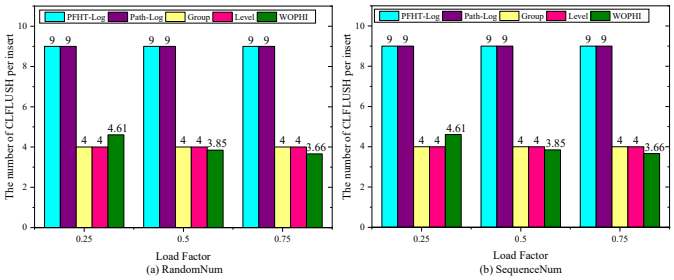

Fig. 3. Average CLFLUSH number per key-value insertion.

Figure 3 illustrates the average number of CLFLUSH operations required for inserting an entry across different persistent hashing indexes. The results in Figures 3(a) and 3(b) are consistent across two distinct workloads, indicating that existing NVM-based hash indexes maintain a consistent number of cache line flushes during write operations. Notably, WOPHI demonstrates a decrease in CLFLUSH number per key-value pair insertion as the load factor increases. In contrast, PFHT-Log and Path-Log exhibit the highest number of CLFLUSH operations due to the logging mechanism's double writes to ensure insertion consistency. However, Group and Level utilize log-free failure-atomic writes, significantly reducing the number of CLFLUSH operations compared to Path-Log. Furthermore, WOPHI adopts the log-free writes method to ensure consistency in NVM insertion and avoids frequent updates present in Group and Level, thereby extending the NVM's lifetime. Consequently, WOPHI demonstrates higher efficiency compared to Group and Level hashing indexes. For instance, at a load factor of 0.75, WOPHI reduces the average number of CLFLUSH operations per insertion by 5.34 times compared to PFHT-Log and Path-Log, representing a 59.3% reduction in the average number of CLFLUSH operations per insertion. WOPHI optimizes cache line flushing operations through log-free failure-atomic writes.

C. Insertion Latency

When comparing the average insertion latency among various persistent hashing indexes, as illustrated in Figure 4, several steps are involved. Firstly, key-value pair entries are inserted into the hash table at specific load factors (e.g., 0.25/0.5/0.75). Then, 1000 key-value entries are inserted, and the latency is recorded. Finally, the latency of inserting 1000 key-value pairs is divided by 1000 to obtain the average latency required for inserting a key-value entry.

For load factors of 0.25 and 0.5, various NVM-based hashing indexes exhibit relatively similar insertion latencies. For instance, at a load factor of 0.5, WOPHI reduces the average insertion latency by 63.0% and 31.9% compared to PFHT-Log and Level, respectively, under the workload RandomNum, and shows similar insertion behavior under the workload SequenceNum. At a load factor of 0.75, WOPHI reduces

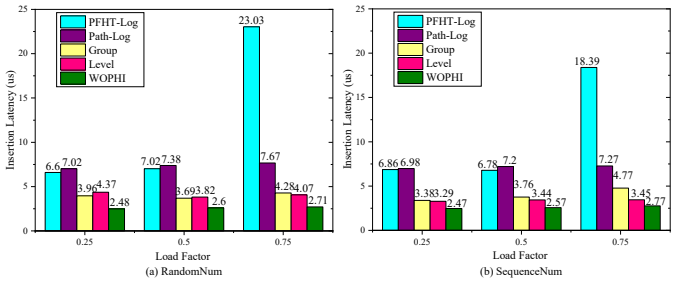

Fig. 4. Average latency when inserting a key-value item.

the average insert latency by 88.2% and 33.4% compared to PFHT-Log and Level, respectively, under the workload RandomNum. Similarly, under the load factor of 0.25 and the workload RandomNum, WOPHI exhibits reduced insertion latency by 62.4% and 43.2% compared to PFHT-Log and Level, respectively. PFHT-Log and Path-Log exhibit relatively higher insertion latencies due to their use of a logging mechanism to ensure data consistency, which incurs a two-fold NVM writes overhead and prolongs the write critical path. Conversely, Group and Level hashing indexes utilize the log-free failure-atomic writes method to ensure data consistency, avoiding the double write overhead of the logging mechanism. WOPHI utilizes log-free failure-atomic writes and a bucket pre-allocation approach to optimize insertion latency, offering a shorter insertion path compared to Level.

PFHT-Log experiences insertion performance degradation at a load factor of 0.75 due to its utilization of cuckoo hashing and sequential stash to handle hash collisions. When the cuckoo hashing main table of PFHT-Log cannot accommodate new key values for insertion, they are inserted into the sequential stash. As illustrated in Figure 4, WOPHI achieves reductions in insertion latency of up to 43.2% and 88.2% compared to Level and PFHT-Log, respectively, demonstrating the lowest insertion latency under RandomNum and SequenceNum workloads.

D. Delete Latency

Figure 5 illustrates the deletion latency across various persistent hashing indexes. Similar to insertion latency, the average latency per deletion of a key-value entry is computed by deleting 1000 key-value entries at specific load factors (e.g., 0.25/0.5/0.75).

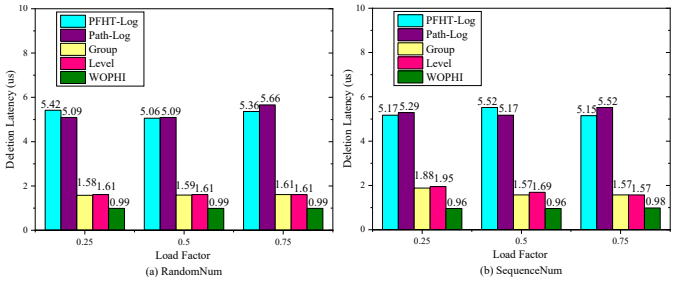

Fig. 5. Average deletion latency for a key-value item.

Under a load factor of 0.5 and the SequenceNum workload, WOPHI reduces the average deletion latency by 82.6% and 43.2% compared to PFHT-Log and Level, respectively, with similar deletion behavior observed under the RandomNum workload. At a load factor of 0.75 and the SequenceNum workload, WOPHI decreases the average deletion latency by 81.0% and 37.6% compared to PFHT-Log and Level. Under a load factor of 0.25 and the SequenceNum workload, WOPHI reduces the average deletion latency by 81.43% and 50.8% compared to PFHT-Log and Level. PFHT-Log and Path-Log exhibit the highest deletion latency due to significant deletion overhead caused by the logging mechanism. Although Group and Level reduce this overhead using log-free failure-atomic writes, there is still room for further optimization of the write overhead caused by frequent local updates. WOPHI achieves the lowest deletion delay by utilizing the Log-Free Failure-Atomic Writes mechanism. As depicted in Figure 5, WOPHI reduces deletion latency by up to 50.8% and up to 82.6% compared to Level and PFHT-Log, respectively.

E. Influence of Diverse NVM Write Latency

Figure 6 illustrates the average insertion and deletion latencies of key-value pair requests across different NVM write latencies. With rising NVM write latency, the latencies for insertion and deletion in various NVM-based persistent hashing structures also escalate.

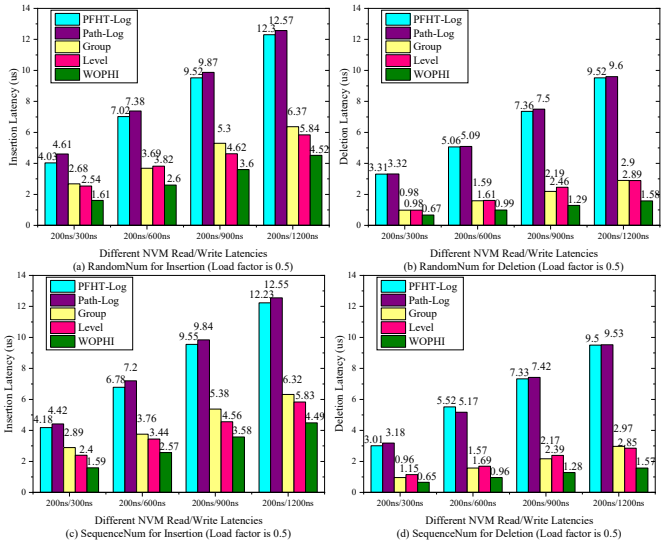

Fig. 6. Insertion and deletion average latencies for key-value pair requests with varying NVM write latencies.

Under the RandomNum workload and a load factor of 0.5, WOPHI reduces the average latency required to insert a key-value pair by 60.0%-63.3% and 22.1%-36.6% compared to PFHT-Log and Level, respectively. Similarly, the average latency to delete a key-value entry with WOPHI is 79.8%-83.4% lower than PFHT-Log and 31.6%-45.3% lower than Level. Both PFHT-Log and Path-Log experience high insertion and deletion latencies, resulting from the double write overhead inherent in their logging mechanisms. In contrast, Group and Level hashing achieve lower insertion and deletion latencies by ensuring data consistency through log-free failure-atomic writes, thus avoiding the double write overhead. Utilizing Log-Free Failure-Atomic Writes mechanism, WOPHI efficiently reduces write overhead, resulting in the lowest insertion and

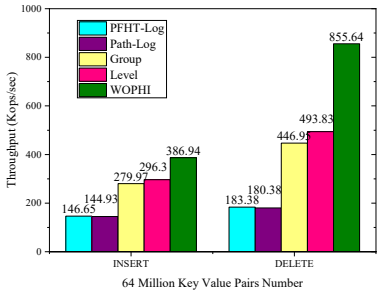

Fig. 7. Average throughput when requesting 64 million items.

deletion latencies across a range of NVM write latencies (e.g., 300/600/900/1200 nanoseconds).

F. Throughput under YCSB Workloads

Figure 7 displays the average throughput of various persistent hashing indexes under YCSB-generated insert and delete workloads. Two datasets, INSERT and DELETE, are generated using the YCSB workload. To assess the insertion and deletion throughput of different hash tables, we initially insert 64 million key-value entries into each hash table and record the insertion time. Subsequently, we delete the 64 million key-value entries from each hash table and record the deletion time. The throughput of inserting and deleting 64 million key-value entries is then calculated by dividing the number of entries by the corresponding time.

Under the INSERT workload with 64 million key-value pairs, WOPHI exhibits insertion throughputs 2.64, 2.67, 1.38, and 1.31 times greater than PFHT-Log, Path-Log, Group, and Level hashing, respectively. Similarly, for the DELETE workload, WOPHI achieves deletion throughputs 4.67, 4.74, 1.91, and 1.73 times higher than PFHT-Log, Path-Log, Group, and Level hashing, respectively. PFHT-Log and Path-Log demonstrate the lowest throughput due to their logging mechanism, which necessitates two NVM writes. In contrast, Group and Level hashing achieve higher throughput by avoiding the multi-write overhead of the logging mechanism through log-free failure-atomic writes. WOPHI exhibits the highest insertion and deletion throughput with log-free failure-atomic writes. Additionally, WOPHI employs bucket pre-allocation to manage hash collisions, detaching the allocation process from the write critical path to reduce insertion latency and enhance throughput.

V. Related Work

Hashing Indexes for NVM. Previous research on NVM-centric hashing structures, exemplified by Path hashing [6] and PFHT [4], primarily aims to minimize NVM writes without fully addressing NVM consistency concerns. Path hashing [6] utilizes an inverted binary tree structure and employs a path-shortening mechanism to enhance lookup efficiency. PFHT [4] restricts cuckoo displacements to one at most, reducing NVM writes compared to conventional cuckoo hashing. On the other hand, Group hashing [5], Level hashing [7], and CCEH [11] prioritize NVM consistency. Group hashing [5] ensures data consistency using 8-byte failure-atomic writes without duplicate copies for copy-on-write (COW) or logging. It employs group sharing techniques to handle hash collisions, improving CPU cache efficiency. Level hashing [7] maintains hash table consistency through opportunistic log-free approaches, avoiding expensive logging or COW mechanisms in most scenarios, resulting in high performance with minimal additional NVM writes. Additionally, it suggests a cost-efficient in-place resizing strategy to substantially minimize NVM writes and mitigate performance decline during resizing tasks. Clevel [12], a derivative of Level hashing, introduces lockless concurrency and asynchronous resizing without compromising read performance. CCEH [11] ensures failure-atomicity without explicit logging and reduces dynamic memory block management overhead. Dash [13], SmartHT [14], ROPHI [15], and SEPH [16] are variations of hashing indexes tailored for DCPMM.

VI. Conclusion

We propose WOPHI, a write-optimized persistent hash index for NVM. WOPHI uses a log-free failure-atomic write strategy to minimize the overhead of ensuring data consistency. Furthermore, it adopts a bucket pre-allocation method to bypass traditional allocation, reducing latency from hash collisions. Experimental results show that WOPHI achieves high write performance with minimal data consistency overhead, significantly enhancing overall write efficiency.

VII. Acknowledgment

This work was supported by the National Key Research and Development Program of China No.2023YFB4502801, the National Natural Science Foundation of China No.U22A2027, No.62262042, the Jiangxi Provincial Natural Science Foundation No.20224BAB202017, No.20242BAB25073, Shenzhen Science and Technology Program No.JCYJ20220530161006015, and Key Laboratory of Information Storage System Ministry of Education of China.

References

[1] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "Energy reduction for stt-ram using early write termination," in *Proceedings of the 2009 International Conference on Computer-Aided Design*, 2009, pp. 264–268.

[2] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Berkeley, CA, USA: USENIX, 2020, pp. 169–182.

[3] J. Yue and Y. Zhu, "Accelerating write by exploiting pcm asymmetries," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. Piscataway, NJ, USA: IEEE, 2013, pp. 282–293.

[4] B. Debnath, A. Haghdoost, A. Kadav, M. G. Khatib, and C. Ungureanu, "Revisiting hash table design for phase change memory," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 2, pp. 18–26, 2016.

[5] X. Zhang, D. Feng, Y. Hua, J. Chen, and M. Fu, "A write-efficient and consistent hashing scheme for non-volatile memory," in *ICPP*. New York, NY, USA: ACM, 2018, p. 87.

[6] P. Zuo and Y. Hua, "A write-friendly hashing scheme for non-volatile memory systems," in *MSST*. Piscataway, NJ, USA: IEEE, 2017, pp. 1–10.

[7] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *OSDI*. Berkeley, CA, USA: USENIX, 2018, pp. 461–476.

[8] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*. New York, NY, USA: ACM, 2010, pp. 143–154.

[9] H. Volos, G. Magalhaes, L. Cherkasova, and J. Li, "Quartz: A lightweight performance emulator for persistent memory software," in *Middleware*. New York, NY, USA: ACM, 2015, pp. 37–49.

[10] F. Xia, D. Jiang, J. Xiong, and N. Sun, "Hikv: A hybrid index key-value store for dram-nvm memory systems," in *ATC*. Berkeley, CA, USA: USENIX, 2017, pp. 349–362.

[11] M. Nam, H. Cha, Y.-r. Choi, S. H. Noh, and B. Nam, "Write-optimized dynamic hashing for persistent memory," in *FAST*. Berkeley, CA, USA: USENIX, 2019, pp. 31–44.

[12] Z. Chen, Y. Hua, B. Ding, and P. Zuo, "Lock-free concurrent level hashing for persistent memory," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. Berkeley, CA, USA: USENIX, 2020, pp. 799–812.

[13] D. Hu, Z. Chen, J. Wu, J. Sun, and H. Chen, "Persistent memory hash indexes: an experimental evaluation," *Proceedings of the VLDB Endowment*, vol. 14, no. 5, pp. 785–798, 2021.

[14] R. Xiao, H. Jiang, D. Feng, Y. Hu, W. Tong, K. Liu, Y. Zhang, X. Wei, and Z. Li, "Accelerating persistent hash indexes via reducing negative searches," in *2023 IEEE 41st International Conference on Computer Design (ICCD)*. IEEE, 2023, pp. 174–181.

[15] R. Xiao, D. Feng, Y. Hu, H. Jiang, L. Wang, Y. Zhang, L. Cui, G. Xu, and F. Wang, "Read-optimized persistent hash index for query acceleration through fingerprint filtering and lock-free prefetching," in *2024 IEEE 42nd International Conference on Computer Design (ICCD)*. IEEE, 2024, pp. 223–230.

[16] C. Wang, J. Hu, T.-Y. Yang, Y. Liang, and M.-C. Yang, "Seph: Scalable, efficient, and predictable hashing on persistent memory," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 479–495.