Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

NEHASH: high-concurrency extendible hashing for non-volatile memory*

Tao CAI†‡, Pengfei GAO†‡, Dejiao NIU, Yueming MA, Tianle LEI, Jianfei DAI

School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang 212013, China

†E-mail: caitao@ujs.edu.cn; 1306943800@qq.com

Received Oct. 14, 2022; Revision accepted Jan. 4, 2023; Crosschecked Apr. 24, 2023

Abstract: Extendible hashing is an effective way to manage increasingly large file system metadata, but it suffers from low concurrency and lack of optimization for non-volatile memory (NVM). In this paper, a multilevel hash directory based on lazy expansion is designed to improve the concurrency and efficiency of extendible hashing, and a hash bucket management algorithm based on groups is presented to improve the efficiency of hash key management by reducing the size of the hash bucket, thereby improving the performance of extendible hashing. Meanwhile, a hierarchical storage strategy of extendible hashing for NVM is given to take advantage of dynamic random access memory (DRAM) and NVM. Furthermore, on the basis of the device driver for Intel Optane DC Persistent Memory, the prototype of high-concurrency extendible hashing named NEHASH is implemented. Yahoo cloud serving benchmark (YCSB) is used to test and compare with CCEH, level hashing, and cuckoo hashing. The results show that NEHASH can improve read throughput by up to 16.5% and write throughput by 19.3%.

Key words: Extendible hashing; Non-volatile memory (NVM); High concurrency
https://doi.org/10.1631/FITEE.2200462 **CLC number:** TP333

1 Introduction

While the file system stores and manages massive data, the amount of its own metadata is increasing, and the number of applications that need to be served at the same time is also increasing. Efficient management and indexing of file metadata has become an important factor affecting file system performance. Extendible hashing is a very effective data indexing method. Compared with B-tree and other tree-based indexing algorithms, it has the advantages of less time overhead and relative stability. It has been applied to many file systems, such as Zettabyte file system (ZFS), general parallel file system (GPFS), Google file system (GFS), and GFS2. It has become a hot spot of current research.

Current extendible hashing still has many limitations that affect performance. First, when the extendible hashing is running, after the data in a hash bucket are full, it needs to be dynamically expanded. At this time, the entire hash directory will be locked, and then expand the size of the hash directory to twice the original size, and the hash bucket also needs to be locked and split. This makes the entire scalable hash inaccessible to other processes, causing all read and write access to be suspended. Second, a single hash bucket is used to store hash keys under each hash directory entry in the extendible hashing. When writing a hash key, the corresponding hash bucket needs to be locked so that the hash bucket cannot be accessed by other processes. Third, the hash key is managed in an unordered way in the hash bucket.

‡ Corresponding authors
* Project supported by the National Natural Science Foundation of China (No. 61806086) and the National Key R&D Program of China (No. 2018YFB0804204)
ORCID: Tao CAI, https://orcid.org/0000-0003-1423-2710; Pengfei GAO, https://orcid.org/0009-0004-8535-4437

Although sorting and other overhead can be reduced, the search time overhead is large, making the size of the hash bucket an important factor affecting extendible hashing performance. Finally, with the increase in the number of hash keys in extendible hashing, it is difficult to store all hash keys in dynamic random access memory (DRAM), which makes the input/output (I/O) performance of external memory devices an important factor affecting extendible hashing performance. Therefore, improving the concurrency and access performance of extendible hashing is an important issue in current extendible hashing research.

Non-volatile memory (NVM), such as phase change memory (PCM) (Li and Lam, 2011), shared transistor technology random access memory (STT-RAM) (Kuan and Adegbija, 2019), the latest technology Intel 3D X point (Dadmal et al., 2017), and commercial Intel Optane DC Persistent Memory (Intel, 2019), has the advantages of read and write speeds close to DRAM and NVM, which provides good support for improving the access speed of data stored in external memory devices. However, compared with DRAM, NVM still has limitations in terms of read and write speed and concurrency; at the same time, NVM has the advantage of read and write speed compared to solid state drive (SSD) and hard disk drive (HDD), making access concurrency an important factor in data access performance. Therefore, how to use the characteristics of NVM, improve the concurrency of hash key management and access, and study new extendible hashing have become an urgent problem to be solved.

Therefore, in the hybrid storage composed of DRAM and NVM, a high-concurrency extendible hashing for NVM (NEHASH) is given. By improving the management strategy of hash directories and hash buckets, the locking granularity in extendible hashing management is reduced to support the high-concurrency access requirements for extendible hashing.

The main contributions of our work can be summarized as follows:

1. The hash directory based on lazy expansion is designed for all access pause problems caused by the expansion of the extendible hashing directory when it is expanded. A three-level hash directory is used instead of a single-level one. The concurrency of extendible hashing is improved by reducing the granularity of locking to a single hash directory entry that causes the expansion. The expansion rate calculation algorithm is designed to dynamically determine the size of the two-time expansion hash directory and can distinguish the difference in the number of hash keys and growth rates of each part of the extendible hashing. In addition, while delaying the expansion of the main hash directory, the space waste of the extendible hashing directory can be reduced, and the access and management efficiency of the extendible hashing directory can be improved.

2. To improve the concurrency of accessing hash keys, first, the single hash bucket mounted under the existing extendible hashing directory is decomposed into multiple buckets, then the bucket directory is added, and the management methods of hash keys in the hash bucket are changed. A hash bucket management algorithm based on groups is designed to reduce the locking granularity when inserting hash keys, the split probability of hash buckets is reduced, and the concurrency of extendible hashing is improved.

3. In view of the frequent access to the hash directory and the large number of layers, a hierarchical storage strategy is given, which stores the hash directory in DRAM and the bucket group in NVM. It can take advantage of the respective advantages of DRAM and NVM to meet the management requirements of massive metadata in the file system. In addition, a hash directory recovery algorithm is designed. After the system restarts, it relies on the hash bucket and other information in the NVM to reconstruct the hash directory to ensure the reliability of the extendible hashing.

4. On the basis of using Intel Optane DC Persistent Memory, its device driver is modified to implement the high-concurrency extendible hashing prototype NEHASH embedded in the device driver. Using the Yahoo cloud serving benchmark (YCSB) test tool for testing and analysis, the test results show that NEHASH has higher concurrency than existing hashing methods, such as CCEH, level hashing, and cuckoo hashing. In a multithreaded environment, the read throughput can be increased by 16.5%, and the write throughput can be increased by 19.3%.

2 Related works

At present, much research has been carried out on extendible hashing, NVM storage devices, and hashing

for NVM at home and abroad. The main research contents are as follows.

2.1 Extendible hashing

Hash algorithm is a strategy that trades storage space for time, and it is a program algorithm that can achieve fast storage and search. Static hashing is based on an array, and its bucket capacity, which is the size of the array, is determined at the beginning of the definition. However, for most databases, it is difficult to predict the number of data buckets required at the beginning, and the database generally grows dynamically. In this case, the static hashing index structure will occur frequently due to the increase in data. The expansion and rehashing of the hash table seriously affect the performance of the database. Dynamic hashing is an upgrade to normal static hashing. It can grow automatically, allowing the number of buckets to be dynamically changed to accommodate the dynamic growth and shrinkage of the database. Extendible hashing (Fagin et al., 1979) is a kind of dynamic hashing, which is classified by the high (low) bits of the key processed by the hash function, similar to the bucket sort in the data structure. Extendible hashing consists of multiple buckets, each storing a fixed amount of data and a resizable array (called a directory), where each hash directory entry stores a pointer to the bucket. Wang and Wang (2010) proposed a new extendible hashing index for flash-based database management systems (DBMSs) and added a split or merge (SM) factor to make it adaptive. Analytical and experimental results show that our design minimizes the cost of index maintenance and that the SM factor makes it work efficiently at different write/update ratios. Hu et al. (2021) presented a lock-free parallel multipartition extendible hashing (PMEH) scheme, which eliminates lock contention overhead, reduces data movement during rehashing, and guarantees data consistency. RACE (Zuo et al., 2022) is a one-sided remote direct memory access (RDMA) conscious extendible hashing index with lock-free remote concurrency control and efficient remote resizing. RACE hashing enables all indexing operations to be performed efficiently by using only one-sided RDMA verbs without involving any computational resource in the memory pool. Dash (Lu et al., 2020) is a scalable extendible hashing for persistent memory (PM), while its bucket size is 256 bytes (the size of the data center persistent memory module (DCPMM) cache line). It maintains a 1-byte fingerprint for each slot to reduce unnecessary bucket scans and accelerate the search. For concurrency control, Dash employs optimistic bucket-level locking, which is implemented by compare-and-swap instructions and a version number.

The above research on extendible hashing cannot be well adapted to NVM storage devices and cannot support the concurrency of the system under multithreading.

2.2 NVM storage devices

NVM storage devices can form mixed memory with DRAM or mixed external memory with HDD and SSD, thereby effectively improving the performance of data storage and access. Hibachi is a hybrid NVM and DRAM buffering strategy for storage arrays (Fan et al., 2017). Hibachi handles read cache and write cache hits separately, improves the cache hit rate, and dynamically adjusts the size of clean and dirty caches to adapt to changes in workload. Random writes are converted to sequential writes to improve write throughput and optimize read performance based on workload characteristics. FlatStore is a new key-value (KV) storage engine designed for NVM and DRAM hybrid memory (Chen YM et al., 2020). It uses NVM to achieve reliable storage of logs and uses OpLog per core to improve throughput and reduce access latency using a pipeline-level batching mechanism. HasFS is a file system for DRAM, NVM, and disk hybrid storage systems (Liu et al., 2020). It uses NVM storage devices to expand the main memory capacity, builds a persistent page cache to eliminate the overhead of regularly writing dirty data back to disk, and designs a hybrid memory consistency mechanism that reduces the time and space overhead of protecting metadata and data. Ziggurat is a layered file system for NVM and disk (Zheng, 2019) that stores write data into NVM, DRAM, or disk, depending on application access data patterns, write size, and possibility to pause until the write operation completes to combine the performance of NVM and the capacity advantages of disk; at the same time, by calculating the heat of file data, cold data are migrated from NVM to disk, and the write bandwidth of disk is effectively used by merging data blocks. Strata is an NVM hybrid storage

system (Kwon et al., 2017) that uses the byte addressing feature of NVM to merge logs and reduce write amplification when writing to SSD and disk, and uses NVM for file allocation and unidirectional migration of data to SSD and disk. vNVML is a user-mode function library (Chou et al., 2020) that can mix NVM with SSD and disk to build a large-capacity and high-speed storage system, effectively ensure the write order and reliability of data, and use DRAM to build a read cache to reduce the number of writes to NVM. TridentFS (Huang and Chang, 2016) stores hot data in PM to reduce flash memory with slow I/O access, while NVMRA (Chen RH et al., 2016) uses PM to improve the random write performance of flash memory.

None of the above studies on NVM storage devices are optimized in combination with extendible hashing.

2.3 Hashing for NVM

In recent years, a variety of NVM-based hash index structures have been proposed. PFHT (Debnath et al., 2015) is a variant of PCM-friendly cuckoo hashing that allows only one cuckoo shift to avoid cascading writes, thus reducing write accesses to PCM. To improve the insert performance of cuckoo hashing, PFHT uses a hidden block to defer full table rehashing and increase the load factor. Path hashing (Zuo and Hua, 2017) is a write-friendly hashing that logically organizes storage units in the form of an inverted binary tree and uses location sharing to resolve hash collisions. Level hashing (Zuo et al., 2018) is a write-optimized, high-performance hash index scheme that proposes a shared two-level hash table, where the top layer can address entries and the bottom layer is used to handle hash collisions. It achieves constant-scale search, insert, delete, and update time complexity in the worst case and generates few extra NVM writes. To ensure low-overhead consistency, level hashing uses a no-log consistency scheme for insert, delete, and adjust operations and an opportunistic no-log scheme for update operations. To cost-effectively resize this hash table, level hashing uses an in-place resizing scheme that needs only to rehash 1/3 of the buckets instead of the entire table, thus greatly reducing the number of rehashed buckets and improving resize performance. The evaluation results show that level hashing is significantly better than PFHT and path hashing.

The above three hash index structures are all implemented based on static hashing. Their common problem is that the rehashing operation of the entire hash table requires considerable computing resources and storage overhead, which will cause a huge delay and is relatively expensive.

CCEH (Nam et al., 2019) is a variant of extendible hashing that dynamically splits and merges hash buckets as needed to overcome the disadvantage of full table rehashing. By introducing the three-layer structure of the middle segment, the size of the directory and the size of the bucket can be reduced to achieve constant-level search, that is, only two cache line accesses. Atomic write optimization delays delete operations to reduce I/O overhead. HMEH (Zou et al., 2020) is an optimization scheme based on CCEH, which proposes a hybrid RAM-NVM write-optimal and high-performance extendible hashing scheme, where KV items persist in NVM, while directories are placed in DRAM for faster access. To rebuild the directory, HMEH also maintains a radixtree-structured catalog in NVM with negligible overhead. Furthermore, HMEH proposes a cross-KV strategy by naturally eviction write-back items, which ensures data consistency without performance degradation due to persistent barriers.

Both CCEH and HMEH are implemented based on extendible hashing, which can well solve the problem of full table rehashing. However, in the case of multithreading, the frequency of hash directory expansion will increase, and when the hash directory is expanded, all other threads cannot access, resulting in reduced concurrency.

3 Lazy expansion strategy for hash directory

In the existing extendible hashing, when expanding the hash directory, the entire hash table needs to be locked, which seriously affects the concurrency of the extendible hashing. We modify the structure of the hash directory in extendible hashing, design a lazy expansion algorithm for the hash directory, and improve the concurrency of extendible hashing.

The original single-layer hash directory in extendible hashing is extended to build a three-layer extendible hashing directory, i.e., the main hash directory, the

one-time expansion hash directory, and the two-time expansion hash directory. As shown in Fig. 1, the 0^{th}, 2^{nd}, and 6^{th} entries in the main hash directory have the one-time expansion hash directory, and the 0^{th} and 12^{th} entries in the one-time expansion hash directory have the two-time expansion hash directory. The structure of the two-time expansion hash directory is the same as the existing extendible hashing directory structure, and each hash directory entry is represented by HL(HL_value, point), where HL_value is the hash directory value and point is a pointer to a hash bucket. The structures of the main hash directory and one-time expansion hash directory are modified. Each hash directory entry is represented by HL_E(HL_value, time_value, point), where time_value is a time-related value and point is a pointer to the next level hash extension directory or hash bucket. On extendible hashing initialization, only the main hash directory is created and initialized.

On this basis, a lazy expansion algorithm for hash directories is given. First, when the main hash directory needs to be expanded, it is not expanded first, but only a one-time expansion hash directory is added under the hash directory entry that needs to be extended, and each one-time expansion hash directory contains two hash directory entries in the form of HL_E. The other parts of the main hash directory are unchanged. Second, when the one-time expansion hash directory is full and needs to be expanded, the main hash directory is delayed again through the two-time expansion hash directory. Spread_rate is defined to represent the expansion rate of the two-time expansion hash directory, which corresponds to the one-time expansion hash directory entry that needs to be extended. On this basis, the rules for constructing the two-time expansion hash directory are as follows:

Rule 1: If the value of Spread_rate is 1, the extended two-time expansion hash directory contains four sets of hash directory entries in the form of HL.

Rule 2: If the value of Spread_rate is 0, the extended two-time expansion hash directory contains two sets of hash directory entries in the form of HL.

When building a two-time expansion hash directory, use Spread_rate to dynamically adjust the size of the generated two-time expansion hash directory to adapt to the uneven distribution and growth rate of hash values under each hash directory entry in the extendible hashing. The expansion rate calculation algorithm is used to obtain the corresponding Spread_rate value when constructing each two-time expansion hash directory. The main process of the expansion rate calculation algorithm is as follows:

Step 1: When the actual hash value is stored under the i^{th} hash directory entry in the main hash directory, save system time to the time_value of the main hash directory entry HL_E, which is recorded as $t_{0\text{-}i}$;

Step 2: When the i^{th} hash directory entry in the main hash directory needs to be extended to build a one-time expansion hash directory, compare system time with $t_{0\text{-}i}$, save the two time differences into the time_value of the main hash directory entry HL_E, and overwrite the previous $t_{0\text{-}i}$, recorded as Δt_i;

Step 3: In the one-time expansion hash directory corresponding to the i^{th} hash directory entry in the main hash directory, when the actual hash value starts to be stored under the j^{th} hash directory entry, save its system time into time_value of the one-time expansion hash directory entry HL_E, recorded as $t_{1\text{-}i\text{-}j}$;

Step 4: When the j^{th} hash directory entry in the one-time expansion hash directory needs to be expanded to build the two-time expansion hash directory, compare the system time with $t_{1\text{-}i\text{-}j}$ and record the two time differences as $\Delta t_{i\text{-}j}$;

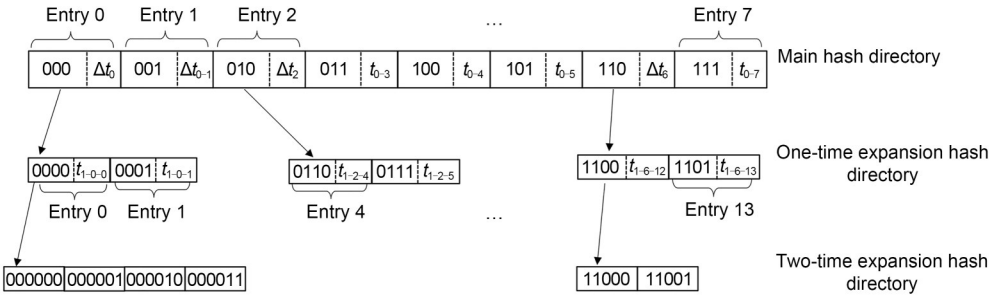

Fig. 1 Structure of the extendible hashing directory for high-concurrency extendible hashing

Step 5: Calculate the value of Spread_rate as follows:

$$\begin{cases} \Delta t_i \geq \Delta t_{i-j}, \ \text{Spread\_rate} = 0, \\ \Delta t_i < \Delta t_{i-j}, \ \text{Spread\_rate} = 1. \end{cases} \tag{1}$$

When $\Delta t_i \geq \Delta t_{i-j}$, that is, when Spread_rate=0, the time it takes for the main hash directory to start storing hash values until it is full and to create a one-time expansion hash directory is greater than or equal to the time it takes for the one-time expansion hash directory to start storing hash values until it is full and to create a two-time expansion hash directory. At this point, it is considered that the number of hash keys belonging to this hash directory entry begins to decrease. To reduce the waste of space for the hash directory, the size of the two-time expansion hash directory is set to 2. Similarly, when $\Delta t_i < \Delta t_{i-j}$, that is, when Spread_rate=1, the time it takes for the main hash directory to start storing hash values until it is full and to create a one-time expansion hash directory is less than the time it takes for the one-time expansion hash directory to start storing hash values until it is full and to create a two-time expansion hash directory. At this point, it is considered that the number of hash keys belonging to this hash directory entry begins to increase. To reduce the waste of space for the hash directory, the size of the two-time expansion hash directory is set to 4.

When the two-time expansion hash directory is full, the entire hash directory needs to be expanded. At this point, only the main hash directory needs to be expanded, and the number of bits of HL_value in the main hash directory entry is the maximum number of bits of all HL_values in the original two-time expansion hash directory. In addition, the bucket groups are pointed to by the original one-time expansion hash directory and the original two-time expansion hash directory are pointed to by the corresponding hash directory entries in the new main hash directory.

By constructing a three-layer hash directory, the global expansion of the hash directory can be delayed. When building the one-time and two-time expansion hash directories, only the entries that cause expansion in the main hash directory or the one-time expansion hash directory need to be added. The lock avoids locking the entire hash directory so that access to other parts of the extendible hashing is not affected, which can improve the concurrency of the extendible hashing; at the same time, the expansion rate calculation algorithm can be used to perceive the extendible hashing directory. The difference between the number of hash values and the growth rate under different directory entries adaptively adjusts the size of the two-time extendible hashing directory, reduces the space waste of the extendible hashing directory, and improves the access and management efficiency of the extendible hashing directory.

4 Hash bucket management based on group

First, change the way that only a single hash bucket is used to store hash keys under each hash directory entry, and build a bucket group for storing hash keys. As shown in Fig. 2, each bucket group contains multiple hash buckets, a conflicting hash bucket, and a bucket directory. The hash bucket is used to store hash values, with a size of 256 B, which can adapt to the read and write characteristics of NVM.

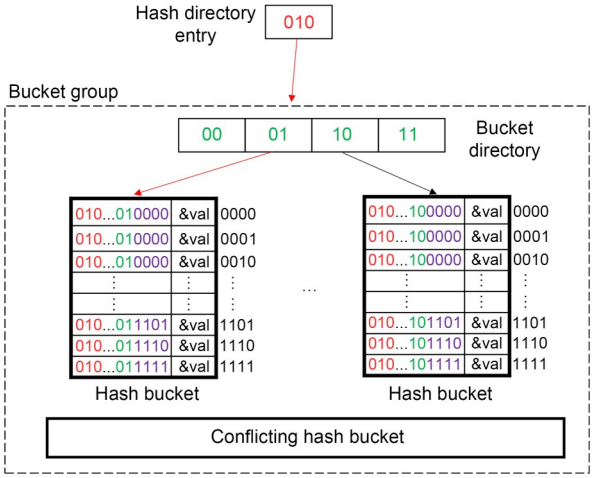

Fig. 2 Hash value organization based on group

At the same time, the way that hash keys are stored out of order in the hash bucket in the existing extendible hashing is changed. In the hash bucket, the storage location is determined according to the last bit of the hash key. To improve the concurrency of access in the bucket, we use slot-level locks to lock only the slots in the bucket.

When a hash conflict occurs, the conflicting record is directly inserted into the conflicting hash bucket in the bucket group. This is because we have numbered the slots in each bucket, so there is no need for linear probing, avoiding the overhead of additional access to NVM. The size of the conflicting hash bucket is 256 B, which is the size of the NVM access granularity, and the conflicting hash bucket is used to store the conflicting hash key and to reduce the read and write amplification problem. Therefore, the records in the conflicting hash bucket are not sorted, and the records are queried by traversal.

Therefore, the role of each bit in the hash key in extendible hashing management is shown in Fig. 3. The highest red bits are the bits corresponding to the extendible hashing directory, the lowest purple bits are the index bits in the hash bucket, and the green bits are the corresponding bits of the bucket directory. Extendible hashing directory bits and bucket directory bits can be dynamically expanded inwards.

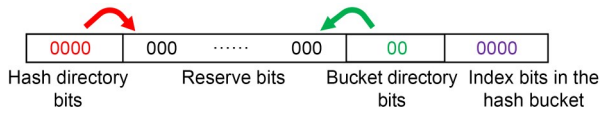

Fig. 3 The role of bits in hash key
References to color refer to the online version of this figure

The pseudocode for inserting records in NEHASH is given in Algorithm 1.

Algorithm 1 Insertion algorithm of NEHASH

```
1: obtain the subscript
2: if the main hash directory entry is empty
3:     create bucket group
4:     obtain the corresponding bit of the bucket directory
5:     obtain the index bit in the bucket
6:     insert
7:     lock the corresponding slot
8: else
9:     if hash conflict
10:        insert record into the conflicting hash bucket
11:    else
12:        insert
13:        lock the corresponding slot
```

The pseudocode of search records in NEHASH is given in Algorithm 2.

Algorithm 2 Search algorithm of NEHASH

```
1: obtain the subscript
2: if the main hash directory entry is empty
3:     return false
4: else
5:     find bucket group
6:     obtain the corresponding bit of the bucket directory
7:     obtain the index bit in the bucket
8:     if the search is correct
9:         return val
10:    else
11:        search collision bucket
```

In the group-based hash bucket management algorithm, multiple 256 B hash buckets that conform to NVM read–write characteristics are used instead of a single hash bucket, and a bucket directory is added to build a bucket group based on the hash bucket. The concurrency of extendible hashing is improved by reducing the scope of locking when writing hash keys. Using the last bit of the hash key to determine the storage location in the hash bucket can effectively improve the efficiency of search and management and can also use the lock on a single hash record to replace the lock on the hash bucket. This improves the concurrency of extendible hashing.

5 Hierarchical storage strategy

In Section 3, a hash directory based on lazy expansion and a three-layer hash directory are used to replace the original single-layer hash directory, which can not only improve the concurrency of the hash directory but also increase the time overhead of accessing the hash directory. Although NVM storage devices have higher read and write speeds, there is still some gap compared to DRAM. Simply using NVM storage devices to store all data in NEHASH cannot guarantee high access performance.

A hierarchical storage strategy is given, and NEHASH's hash directory and bucket groups are distributed to DRAM and NVM storage devices. Fig. 4 shows NEHASH's main hash directory, one-time expansion hash directory, and two-time expansion hash directory storage in DRAM, while all bucket groups are stored in NVM storage devices.

The hash directory stored in the DRAM will be lost after the system restarts, so in the NVM storage device, it is necessary to save some information of the hash directory so that the hash directory can be rebuilt after the system restarts. Recovery_BG(BG_Id, D_Length, BG_Address) is used to represent the information required for each bucket group during recovery. BG_Id is the bucket group identifier, D_Length is the length of the hash directory corresponding to the bucket group, and BG_Address is the address of the bucket group in NVM. Recovery_BG corresponding to all bucket groups is organized into Rec_BG_TABLE in the form of a linear table and stored in the NVM storage device.

When the system restarts, the rebuilding process of the NEHASH hash directory is as follows:

Step 1: Access the Rec_BG_TABLE stored in the NVM storage device and read all Recovery_BG.

Step 2: Compare the size of D_Length in all Recovery_BG, find the maximum value, and record it as D_L_Max.

Step 3: Build the main hash directory with $2^{D\_L\_Max}$ as the length.

Step 4: Compare the D_Length value of each Recovery_BG in Rec_BG_TABLE in turn; if it is equal to D_L_Max, go to step 5; otherwise, go to step 6.

Step 5: Access the corresponding bucket group according to the BG_Address of Recovery_BG, obtain the value of the header D_Length bit of the hash value in the bucket group, and connect it with the corresponding directory entry in the main hash directory.

Step 6: Split the bucket group pointed to by BG_Address, construct the corresponding bucket group according to the standard that the length of the corresponding directory entry is D_L_Max bits, and connect the corresponding bucket group with the corresponding directory entry in the main hash directory.

For the situation in Fig. 4, the hash directory recovered after the system restarts is shown in Fig. 5.

Using the hierarchical storage strategy, the respective advantages of DRAM and NVM storage devices can be used, and the access and management efficiency of extendible hashing can be improved by distributing hash directories and bucket groups. At the same time, after the hash directory is lost, the hash directory can be restored by relying on the relevant

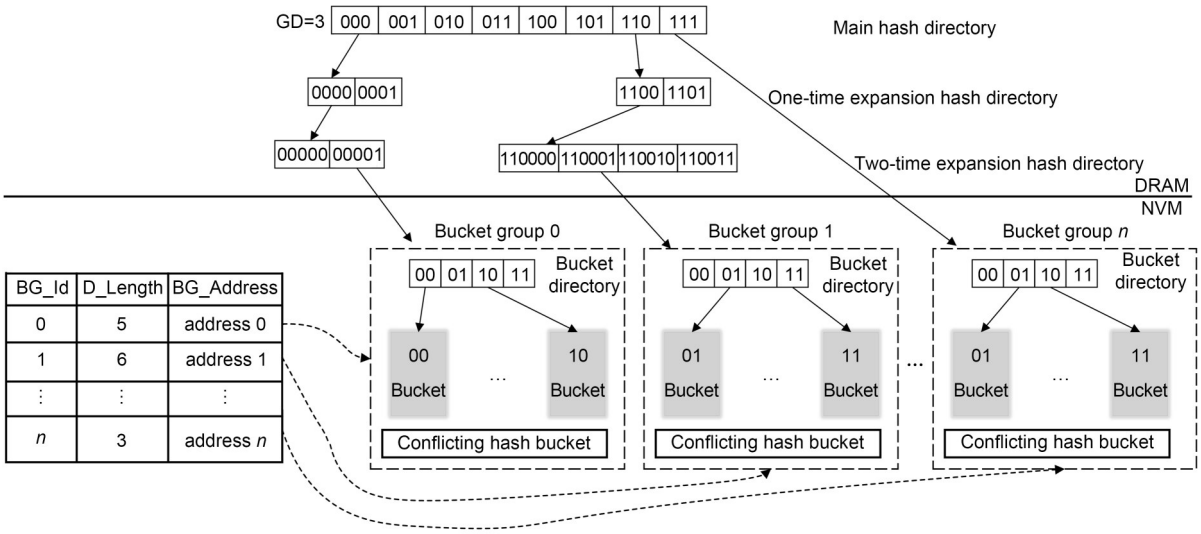

Fig. 4 Hierarchical storage strategy of NEHASH

Fig. 5 Recovered hash directory

information stored in the NVM storage device, thereby effectively ensuring the reliability of NEHASH.

6 Prototype and analysis

We modify the hash directory and hash bucket on the base extendible hashing, and to reduce the extra time overhead required to convert between kernel mode and user mode when accessing data, the prototype system of NEHASH is implemented in the Intel Optane DC Persistent Memory driver PMEM, and new system calls are added to enable user mode programs to perform access.

We use YCSB, which is commonly used in the database field, as a test tool for testing, using 10 million random integers as a workload. Since the failure atomic unit of NVM is 8 bytes (Oukid et al., 2016), both the size of the key and the value are set to 8 bytes to ensure the atomicity of the data. Among them, workload *a* is a read−write balanced type, with 50% read and 50% write; workload *b* is read more and write less, with 95% read and 5% write; workload *g* is read less and write more, with 5% read and 95% write. Meanwhile, we compare NEHASH with existing hashing schemes such as CCEH, level hashing, and cuckoo hashing (Pagh and Rodler, 2004) (called CCEH, LEVL, and Cuckoo). The test environment is built with one server, and the configuration of the server is shown in Table 1.

Table 1 Configuration of the test environment

Part	Configuration
CPU	Two Intel Xeon CPUs E5-2620 v3 @2.40 GHz
Memory	128 GB
NVDIMM	Two 128 GB Intel Optane DC Persistent Memory
Disk	600 GB SAS disk
OS	CentOS 7.0 with kernel 4.4.112

6.1 Concurrent I/O performance

To test the concurrent access performance of NEHASH, we use 10 million random integers as the workload and use the YCSB tool to test the average throughput of NEHASH at 1, 2, 4, 8, and 16 threads.

Fig. 6 presents the average throughput under workload *a* with a mix of read and write. When the number of threads is relatively low, CCEH shows a relatively good average throughput. This is because CCEH's three-layer structure, through cacheline-sized buckets, reduces access to cachelines. However, when the number of threads is relatively high, especially at 16 threads, the average throughput of NEHASH is 13.8% higher than that of CCEH. This is because NEHASH uses a hash directory based on lazy expansion, which can delay the global expansion of the hash directory and avoid frequently locking the entire hash directory, so it can effectively improve concurrency in a multi-threaded environment. LEVL and Cuckoo show relatively poor average throughput, which does not increase with the increase in the number of threads. This is because LEVL and Cuckoo are based on static hashing, and the rehashing of the static hash table will lead to high latency, blocking all insert and search operations in other threads.

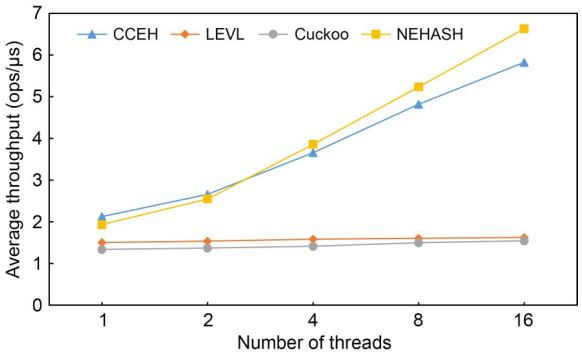

Fig. 6 Average throughput under workload *a*

Fig. 7 is the result of the average throughput under the read more write less workload *b*. When the number of threads is less than 2, the average throughput of NEHASH is slightly lower than that of CCEH, because CCEH has a cacheline-sized bucket that needs only two cacheline accesses to read the data inside the bucket. However, NEHASH needs to access the multi-layer hash directory and bucket directory when accessing the data in the bucket group, which will lead to relatively high access latency. When the number of threads is greater than 4, the average throughput of NEHASH begins to increase significantly, especially in the stage from 8 threads to 16 threads. The average throughput of NEHASH shows a clear trend of improvement, which is 2.71 times that of LEVL and 3.27 times that of Cuckoo. When the number of threads

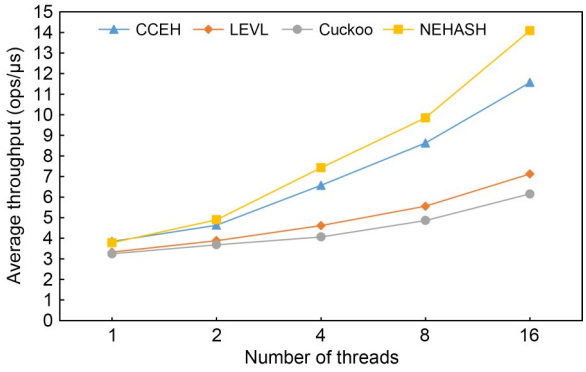

Fig. 7 Average throughput under workload *b*

is 16, the average throughput of NEHASH is 16.5% higher than that of CCEH. This is due to the hash bucket management algorithm based on groups in NEHASH. Through the bucket directory index and the bucket index with the last 4 bits of the hash key, the data in the bucket can be quickly located, improving the read average throughput in a high-concurrency environment.

Fig. 8 shows the average throughput under workload *g* with read less and write more. As seen from Fig. 8, the average throughputs of NEHASH and CCEH are significantly higher than those of LEVL and Cuckoo. This is because NEHASH and CCEH are implemented based on extendible hashing, and the hash table can be dynamically expanded with increasing data. When the number of threads is less than 2, the average throughput of NEHASH is lower than that of CCEH. This is because when the number of threads is relatively small, the concurrency is not high at this time, and when NEHASH inserts data, it is necessary to build structures such as bucket groups and bucket directories first, which results in lower average throughput; however, CCEH avoids the high copy-on-write operation and reduces the overhead of split through dynamic segment split and lazy deletion. When the number of threads reaches 4, the average throughput of NEHASH begins to be higher than that of CCEH. When the number of threads reaches 16, the average throughput of NEHASH still maintains an upwards trend, while CCEH has no obvious upwards trend and begins to level off. The average throughput of NEHASH is 19.3% higher than that of CCEH. This is because NEHASH reduces the range of locks when writing hash values when a large amount of data is written concurrently by 16 threads, which improves the performance of extendible hashing. Concurrency can show relatively high throughput in a 16-thread environment.

6.2 Changing the size of the bucket directory

To determine the proper number of bits in the bucket directory in NEHASH, we perform the following tests. First, we change the number of bits of the bucket directory in NEHASH to 1, 2, 4, 6, 8, and 10, that is, the corresponding bucket group sizes are 512 B, 1 KB, 4 KB, 16 KB, 64 KB, and 256 KB. For comparison, we also set the segment size of CCEH to 512 B, 1 KB, 4 KB, 16 KB, 64 KB, and 256 KB. Using 10 million random integers as the workload, the read and write performances of NEHASH and CCEH are tested by YCSB tools.

Fig. 9 is the result of changing the bucket directory bits of NEHASH and the segment size of CCEH to test the search throughput. When the number of digits in the bucket directory is relatively small, NEHASH shows a relatively low search throughput. This is because when the number of digits in the bucket

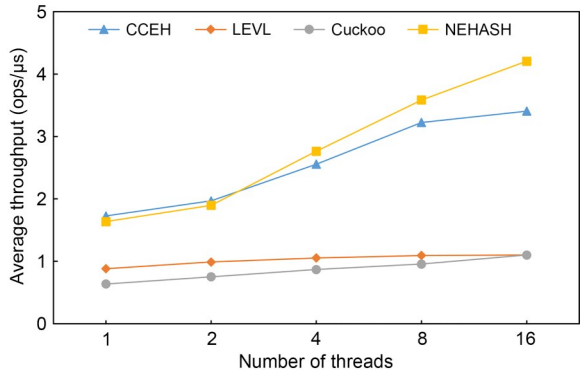

Fig. 8 Average throughput under workload *g*

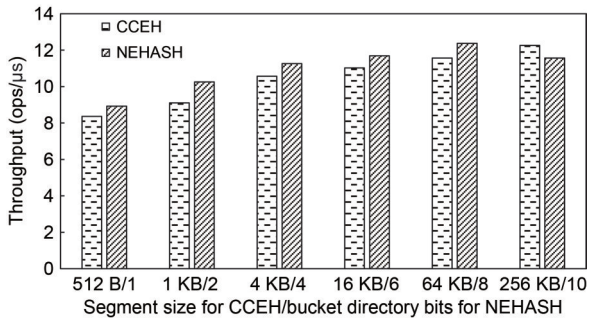

Fig. 9 Search throughput

directory is small, the number of buckets in the bucket group is also very few, which means that there will be more hash collisions. After a hash collision occurs, NEHASH needs to read the data in the hash collision bucket, and the data in the hash collision bucket are out of order, so NEHASH needs to traverse the search in order, resulting in lower search throughput. When the number of bucket directories reaches 8, the search throughput at this time is 1.53 times that when the bucket directory is 1. This is because as the number of bucket directories increases, the number of buckets in the bucket group also increases, resulting in the probability of conflict being greatly reduced, so there is no need to search for data in the hash conflict bucket, and the data can be obtained directly through the index number in the bucket, thus greatly improving the search throughput. Compared with CCEH, when the number of bucket directories is less than or equal to 8, the search throughput of NEHASH is higher than that of CCEH. This is because NEHASH stores the hash directory in the critical path in DRAM, which can effectively reduce search latency. However, when the number of bucket directories continues to increase to 10, the search throughput will decrease slightly. This is because when the number of bucket directories continues to increase, the number of buckets in the bucket group will double, which will cause considerable management overhead.

Fig. 10 shows the result of changing the bucket directory bits of NEHASH and the segment size of CCEH to test the insertion throughput. It can be found that when the number of bucket directories is only 1, the write throughput of NEHASH is very poor and is lower than that of CCEH. This is because when the number of bucket directories is 1, there are only two 256-byte buckets in the bucket group, and the capacity of the bucket group is too small. At this time, bucket group splitting and expansion of the hash directory often occur when writing data, which will greatly reduce the insertion throughput of NEHASH. As the number of bucket directories increases, the number of buckets in the bucket group will increase exponentially, and the frequency of bucket splitting and hash directory expansion will decrease when writing data, so the insertion throughput of NEHASH will also increase accordingly. When the number of bucket directory bits is 6, the write throughput of NEHASH reaches the maximum value, which is 1.93 times that when the number of bucket directory bits is 1. When the segment size and bucket directory bits continue to increase, the insertion throughputs of both CCEH and NEHASH show a downward trend, and the downward trend of NEHASH is more obvious than that of CCEH because when the CCEH segment is larger than 16 KB, segment splitting will result in a large number of NVM writes, which will reduce the CCEH insertion throughput. When the number of bucket directories in NEHASH is greater than 6, that is, when the bucket group capacity is greater than 16 KB, multiple hash buckets and additional bucket directories need to be created to split the bucket group, which will generate more NVM write operations than CCEH. Therefore, insertion throughput there will be a clear downward trend in throughput.

6.3 Tail latency

To evaluate the overhead problem in the hash scheme, we use 16 threads to test the latency distribution of each hash scheme under write- and read-intensive workloads, and their cumulative distribution functions (CDFs) are shown in Figs. 11 and 12.

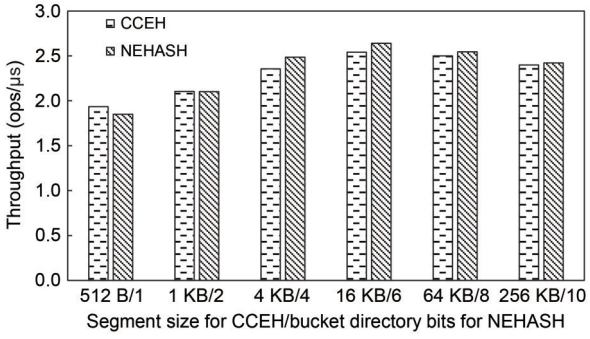

Fig. 10 Insertion throughput

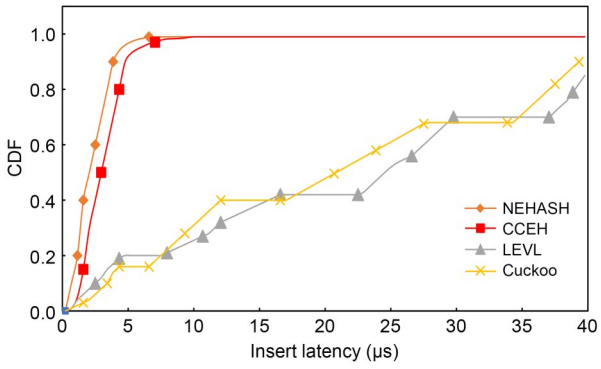

Fig. 11 Latency distribution of insertion

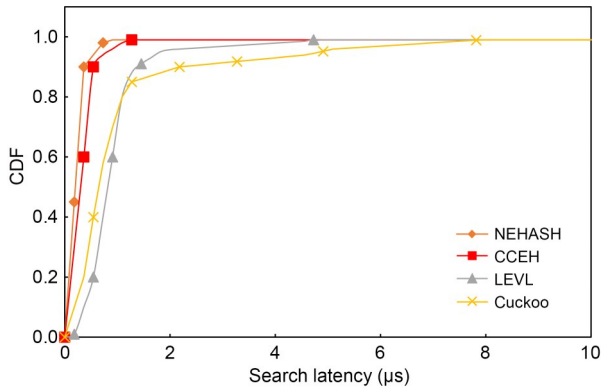

Fig. 12 Latency distribution of search

Fig. 11 shows the distribution of tail latency for a write-intensive workload. LEVL and Cuckoo show relatively poor tail latency because they are based on static hashing schemes. Static hashing schemes require locking the entire hash table during rehashing, which blocks substantial concurrent queries and incurs dramatic tail latency. However, NEHASH and CCEH are based on dynamic hashing schemes and therefore exhibit good tail latency. CCEH restricts its 99th percentile latency to 10.2 μs. However, NEHASH shows the lowest tail latency and restricts its 99th percentile latency to 6.5 μs. This is because CCEH will perform linear probing when there is a confliction in the hash bucket, which results in additional access to NVM and incurs overhead, while NEHASH avoids the overhead of linear probing through the index in the hash bucket.

Fig. 12 shows the distribution of tail latency for a read-intensive workload. From this, it can be seen that all four hashing schemes show similar tail delays. Cuckoo shows the worst tail latency because Cuckoo needs to handle hash collisions through two hash functions and will access the hash bucket multiple times, so it will cause serious search overhead. However, NEHASH still shows good tail latency, restricting its 99th percentile latency to within 1.2 μs. This is due to the bucket directory index in the bucket group.

7 Conclusions

This paper designs a hash directory based on lazy expansion, uses a three-layer hash directory instead of a single-layer hash directory, perceives and adapts to the growth rate of hash keys under the hash directory item, delays the expansion of the hash directory, and dynamically adjusts the expansion rate of the hash directory. The group-based hash bucket management algorithm is designed, and the bucket group is used instead of a single hash bucket, which can reduce the locking granularity when managing hash keys and improve the concurrency of hash key management. The hierarchical storage strategy and the hash directory recovery algorithm are designed, which can take advantage of respective advantages of DRAM and NVM, so that extendible hashing can effectively manage the massive metadata in the file system. Using Intel Optane DC Persistent Memory, on the basis of its driver, the prototype for high-concurrency extendible hashing for NVM (NEHASH) is realized, and YCSB is used to compare with CCEH, level hashing, and cuckoo hashing. It is verified that NEHASH can be improved by up to 16.5% read throughput and 19.3% write throughput.

In future work, we plan to closely integrate NEHASH with the file system to effectively improve the performance of the NVM file system.

Contributors

Tao CAI designed the research. Tao CAI and Pengfei GAO processed the data. Tao CAI drafted the paper. Dejiao NIU, Yueming MA, Tianle LEI, and Jianfei DAI helped organize the paper. Tao CAI and Pengfei GAO revised and finalized the paper.

Compliance with ethics guidelines

Tao CAI, Pengfei GAO, Dejiao NIU, Yueming MA, Tianle LEI, and Jianfei DAI declare that they have no conflict of interest.

Data availability

The data that support the findings of this study are available from the corresponding authors upon reasonable request.

References

Chen RH, Shen ZY, Ma CL, et al., 2016. NVMRA: utilizing NVM to improve the random write operations for NAND-flash-based mobile devices. *Softw Pract Exper*, 46(9): 1263-1284. https://doi.org/10.1002/spe.2378

Chen YM, Lu YY, Yang F, et al., 2020. FlatStore: an efficient log-structured key-value storage engine for persistent memory. Proc 25th Int Conf on Architectural Support for Programming Languages and Operating Systems, p.1077-1091. https://doi.org/10.1145/3373376.3378515

Chou CC, Jung J, Reddy ALN, et al., 2020. Virtualize and share non-volatile memories in user space. *CCF Trans High Perform Comput*, 2(1):16-35.

https://doi.org/10.1007/s42514-020-00019-8

Dadmal UD, Vinkare RS, Kaushik PG, et al., 2017. 3D X point technology. IETE Zonal Seminar "Recent Trends in Engineering & Technology, p.13-17.

Debnath B, Haghdoost A, Kadav A, et al., 2015. Revisiting hash table design for phase change memory. *ACM SIGOPS Oper Syst Rev*, 49(2):18-26. https://doi.org/10.1145/2883591.2883597

Fagin R, Nievergelt J, Pippenger N, et al., 1979. Extendible hashing—a fast access method for dynamic files. *ACM Trans Database Syst*, 4(3):315-344. https://doi.org/10.1145/320083.320092

Fan ZQ, Wu FG, Park D, et al., 2017. Hibachi: a cooperative hybrid cache with NVRAM and DRAM for storage arrays. Proc 33rd Int on Conf Massive Storage Systems and Technology.

Hu J, Chen JX, Zhu YF, et al., 2021. Parallel multi-split extendible hashing for persistent memory. Proc 50th Int Conf on Parallel Processing, Article 48. https://doi.org/10.1145/3472456.3472500

Huang TC, Chang DW, 2016. TridentFS: a hybrid file system for non-volatile RAM, flash memory and magnetic disk. *Softw Pract Exper*, 46(3):291-318. https://doi.org/10.1002/spe.2299

Intel, 2019. Intel® Optane™ Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html [Accessed on Mar. 5, 2022].

Kuan K, Adegbija T, 2019. MirrorCache: an energy-efficient relaxed retention L1 STTRAM cache. Great Lakes Symp on VLSI, p.299-302. https://doi.org/10.1145/3299874.3318022

Kwon Y, Fingler H, Hunt T, et al., 2017. Strata: a cross media file system. Proc 26th ACM Symp on Operating Systems Principles, p.460-477. https://doi.org/10.1145/3132747.3132770

Li J, Lam C, 2011. Phase change memory. *Sci China Inform Sci*, 54(5):1061-1072. https://doi.org/10.1007/s11432-011-4223-x

Liu YB, Li HB, Lu YT, et al., 2020. HasFS: optimizing file system consistency mechanism on NVM-based hybrid storage architecture. *Cluster Comput*, 23(10):2501-2515. https://doi.org/10.1007/s10586-019-03023-y

Lu BT, Hao XP, Wang TZ, et al., 2020. Dash: scalable hashing on persistent memory. *Proc VLDB Endow*, 13(8):1147-1161. https://doi.org/10.14778/3389133.3389134

Nam M, Cha H, Choi YR, et al., 2019. Write-optimized dynamic hashing for persistent memory. Proc 17th USENIX Conf on File and Storage Technologies, p.31-44.

Oukid I, Lasperas J, Nica A, et al., 2016. FPTree: a hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. Int Conf on Management of Data, p.371-386. https://doi.org/10.1145/2882903.2915251

Pagh R, Rodler FF, 2004. Cuckoo hashing. *J Algorithms*, 51(2):122-144. https://doi.org/10.1016/j.jalgor.2003.12.002

Wang L, Wang HH, 2010. A new self-adaptive extendible hash index for flash-based DBMS. IEEE Int Conf on Information and Automation, p.2519-2524. https://doi.org/10.1109/ICINFA.2010.5512045

Zheng SA, 2019. Ziggurat: a tiered file system for non-volatile main memories and disks. Proc 17th USENIX Conf on File and Storage Technologies, p.207-219.

Zou XM, Wang F, Feng D, et al., 2020. HMEH: write-optimal extendible hashing for hybrid DRAM-NVM memory. Proc 36th Int Conf on Mass Storage Systems and Technologies.

Zuo PF, Hua Y, 2017. A write-friendly hashing scheme for non-volatile memory systems. Proc 33rd Int Conf on Massive Storage Systems and Technology.

Zuo PF, Hua Y, Wu J, 2018. Write-optimized and high-performance hashing index scheme for persistent memory. Proc 13th USENIX Symp on Operating Systems Design and Implementation, p.461-476.

Zuo PF, Zhou QH, Sun JZ, et al., 2022. RACE: one-sided RDMA-conscious extendible hashing. *ACM Trans Storage*, 18(2):11. https://doi.org/10.1145/3511895