A Write-efficient and Consistent Hashing Scheme for Non-Volatile Memory

Xiaoyi Zhang, Dan Feng✉, Yu Hua, Jianxi Chen and Mandi Fu
Wuhan National Lab for Optoelectronics, Key Lab of Information Storage System (School of Computer Science and Technology, Huazhong University of Science and Technology), Ministry of Education of China, Wuhan, China
✉Corresponding author: Dan Feng (dfeng@hust.edu.cn)
Email: {zhangxiaoyi, dfeng, csyhua, chenjx, mandi_fu}@hust.edu.cn

ABSTRACT

The development of non-volatile memory technologies (NVMs) has attracted interest in designing data structures that are efficiently adapted to NVMs. In this context, several NVM-friendly hashing schemes have been proposed to reduce extra writes to NVMs, which have asymmetric properties of reads and writes and limited write endurance compared with traditional DRAM. However, these works neither consider the cost of cacheline flush and memory fence nor provide mechanisms to maintain data consistency in case of unexpected system failures. In this paper, we propose a write-efficient and consistent hashing scheme, called group hashing. The basic idea behind group hashing is to reduce the consistency cost while guaranteeing data consistency in case of unexpected system failures. Our group hashing consists of two major contributions: (1) We use 8-byte failure-atomic write to guarantee the data consistency, which eliminates the duplicate copy writes to NVMs, thus reducing the consistency cost of the hash table structure. (2) In order to improve CPU cache efficiency, our group hashing leverages a novel technique, i.e., group sharing, which divides the hash table into groups and deploys a contiguous memory space in each group to deal with hash collisions, thus reducing CPU cache misses to obtain higher performance in terms of request latency. We have implemented group hashing and evaluated the performance by using three real-world traces. Extensive experimental results demonstrate that our group hashing achieves low request latency as well as high CPU cache efficiency, compared with state-of-the-art NVM-based hashing schemes.

CCS CONCEPTS

• **Information systems → Data structures**; **Storage class memory**; • **Software and its engineering → Consistency**; **Software performance**;

KEYWORDS

Non-Volatile Memory, Hashing Scheme, Data Consistency, Performance

ICPP 2018, August 13–16, 2018, Eugene, OR, USA

ACM ISBN 978-1-4503-6510-9/18/08…$15.00
https://doi.org/10.1145/3225058.3225109

ACM Reference Format:
Xiaoyi Zhang, Dan Feng✉, Yu Hua, Jianxi Chen and Mandi Fu. 2018. A Write-efficient and Consistent Hashing Scheme for Non-Volatile Memory. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3225058.3225109

1 INTRODUCTION

Over the past few decades, DRAM has been used as the main memory of computer systems. However, it is becoming insufficient due to its increasing leakage power dissipation and limited scalability [31]. To address this problem, several non-volatile memory (NVM) technologies emerge, such as phase-change memory (PCM) [14], resistive random access memory (ReRAM) [18], spin-transfer torque magnetic RAM (STT-MRAM) [12] and 3D-XPoint [11]. These new types of memory combine the non-volatility property of traditional HDDs with the low access latency and byte-addressability of DRAM. These desirable characteristics allow NVMs to be directly placed on the processor's memory bus along with traditional DRAM [30], or even replace DRAM in the future [16]. With NVM, applications can directly manipulate persistent data in main memory by using fast *load*/*store* instructions without the need to access time-consuming block-based storage devices [30].

The rapid development of NVMs has attracted interest in designing data structures that are efficiently adapted to NVMs. Although NVMs naturally provide non-volatility property, it is challenging for data structures in NVM to ensure consistency in case of unexpected system crashes or power failures [23]. Specifically, if a system crash occurs during an update to a data structure stored in NVM, the data structure may be left in a corrupted state due to partially updating. Different from the traditional block-based storage device, the failure atomicity unit of NVM is generally expected to be 8 bytes [6]. For updates with larger size than the failure atomicity unit, duplicate copy techniques, such as logging and copy-on-write (CoW) are usually employed to guarantee consistency [29]. However, these techniques incur a large amount of extra writes and require the memory writes to be in a correct order [30]. Unfortunately, modern processors and their caching hierarchies usually reorder memory write operations for performance reasons. In order to maintain memory writes to NVM in a certain order, *mfence* and *clflush* instructions are provided by modern CPUs. *Mfence* is used to enforce the order between memory writes and *clflush* is used to explicitly flush a CPU cacheline to memory devices. However, these instructions have been proved to be a major reason of performance degradation [28, 32], and the overhead of these instructions is proportional to the amount of NVM writes [33].

Hashing-based data structures are widely used in applications [1, 7, 8] due to their constant-scale lookup time complexity. Recently, several hashing-based structures for NVM such as PFHT [5] and path hashing [34] have been proposed. These structures focus on reducing the extra writes to NVM, which has longer write latency than read [19] and limited write endurance [15]. However, these NVM-friendly hashing structures do not consider the cost of cacheline flush and memory fence when writing to NVM through memory bus. More importantly, these hashing structures do not provide mechanisms to maintain data consistency in case of unexpected system failures.

To this end, we present a write-efficient and consistent hashing scheme, called group hashing. The basic idea behind group hashing is to reduce the consistency cost while guaranteeing data consistency in case of unexpected system failures. In order to reduce the consistency cost, we use 8-byte failure-atomic write to guarantee the consistency of the hashing structure, which eliminates duplicate copy writes to NVM during insert and delete operations.

Furthermore, our group hashing includes a novel solution, i.e., group sharing, to deal with hash collisions. Specifically, group hashing decouples storage cells into two levels. The cells in the first level are addressable by the hash function. The cells in the second level are non-addressable and used to deal with collisions. Group hashing divides the cells in both levels into many groups, the cells in each group are stored in the contiguous memory space. The total group amount in each level is equal. Each group in the first level matches a group in the second level with the same group number. The collision resolution cells in the group of the second level are shared by the matched group in the first level. When hash collisions occur in the first level, the conflicting items can be stored in the empty cells of the matched group in the second level. As the cells in a group are stored in the contiguous memory address, group hashing can improve the CPU cache efficiency, thus obtaining higher performance in terms of request latency. The main contributions of this paper can be summarized as follows:

- We quantify the consistency cost of existing hashing structures used in NVMs, and present two insightful observations: (1) duplicate copy techniques (e.g., logging) incur a large amount of extra writes which significantly deteriorate the insertion and deletion performance; (2) keeping the memory space continuity of hashing cells for dealing with collisions can reduce CPU cache misses and improve the performance in terms of request latency.
- Based on the observations, we present our group hashing, (1) uses 8-byte failure-atomic write to guarantee the consistency of the hashing structure, which does not require any duplicate copies for logging or CoW; (2) leverages group sharing scheme, which divides the hash table into groups and deploys a contiguous memory space in each group to deal with hash collisions, thus reducing CPU cache misses to improve the performance in terms of request latency.
- We have implemented group hashing and evaluated the performance under three real-world traces. The experimental results demonstrate that our group hashing achieves low request latency as well as high CPU cache efficiency, compared with state-of-the-art NVM-based hashing schemes.

The rest of this paper is organized as follows. Section 2 provides the background and motivation. Section 3 describes the design of group hashing. Experimental results are presented in Section 4. Related work is discussed Section 5 and we conclude the paper in Section 6.

2 BACKGROUND AND MOTIVATION

2.1 Non-volatile Memory Technologies

As traditional DRAM has the problem of increasing leakage power dissipation and limited scalability, the emerging non-volatile memory technologies such as PCM, STT-MRAM, ReRAM and 3D-XPoint, have attracted more and more attentions in both academia and industry [19]. NVMs synergize the characteristics of non-volatility as HDDs, and low access latency and byte-addressability as DRAM. Such characteristics allow NVMs to complement or substitute DRAM as the main memory in the future computer systems [30].

Table 1: Characteristics of Different Memory Techniques

Techniques	DRAM	PCM	ReRAM	STT-MRAM
Read speed(ns)	10	20~85	10~20	5~15
Write speed(ns)	10	150~1000	100	10~30
Scalability(nm)	40	5	11	32
Endurance	10^{18}	10^{8}	10^{10}	10^{12}~10^{15}

Table 1 shows the key attributes and features of different memory technologies. From the table we observe that, NVMs have longer write latency than read and limited write endurance. Reducing the amount of writes to NVMs can alleviate these two limitations at the same time. To extend the lifetime of NVMs, wear-leveling schemes are used in NVM-based systems [9]. As most of the wear-leveling schemes are built on device level, we assume such wear leveling schemes exist and do not address it in our group hashing. Actually, our design of eliminating duplicate copy writes to NVMs can be combined with wear-leveling schemes to further lengthen NVM's lifetime.

2.2 Data Consistency for Hashing Schemes in NVM

When NVM is directly attached to memory bus, the volatility-persistence boundary has moved to the interface between the volatile CPU cache and persistent NVM [17]. Data consistency, i.e., data correctness after recovering from unexpected failures, must be ensured at the memory level in NVM-based storage systems [29]. Different from block-based storage devices, the failure atomicity unit of NVM is generally expected to be 8 bytes [6, 10, 13]. For updates with larger size, the order of the memory writes must be executed carefully [30]. Unfortunately, memory write operations may be reordered by CPU or memory controller for performance reasons. In order to maintain memory writes to NVM in a certain order, CPUs provide instructions such as *mfence* and *clflush*. However, these instructions have been proved to be a major reason of performance degradation [28, 32].

Hashing-based data structures are widely used in main memory applications [1, 7, 8] due to their constant-scale lookup time complexity. When designing hash-based data structures for NVMs, data

consistency is another key factor apart from efficiently addressing hash collisions. Figure 1 gives an example of potential inconsistencies when a system failure[1] occurs during an insertion to a hashing table in NVM. According to the pseudo-codes in the figure, the insertion first inserts the *key-value* pair to an empty hashing entry, then increments the *count* field. The figure shows three possible inconsistent cases. In the first case, the system failure occurs after the insertion of the *key-value* pair and before the increment of the *count* field. In the second case, the increment of *count* reaches NVM first due to the reordering memory writes, and the system failure occurs after the increment of *count* and before the insertion of the *key-value* pair. The value of *count* is incorrect in these two cases. In the third case, the system failure occurs in the middle of the insertion of the *key-value* pair, the *value* field is partially inserted. The above three cases lead to inconsistent states after recovering from the system failure.

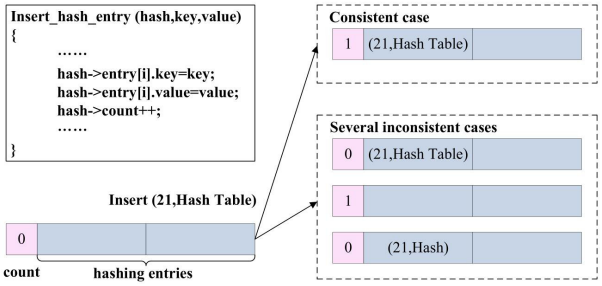


Figure 1: An example of potential inconsistencies upon system failure.

To avoid the second inconsistent case, current processors provide instructions such as *mfence* and *clflush* to ensure the order of memory writes. However, these instructions incur significant overhead, which is proportional to the amount of NVM writes [28, 32]. To avoid the first and the third inconsistent case, duplicate copy techniques such as logging and copy-on-write (CoW) are required. However, these techniques incur a large amount of extra writes, which significantly degrade the performance [29, 32]. Therefore, it is important to reduce the consistency cost while designing NVM-based data structures.

2.3 Motivation

In order to quantify the consistency cost, we use the random integer trace [26, 34] to measure the average request latency and CPU cache miss number in (a) linear probing [24], a traditional hashing scheme used in DRAM, with and without logging (referred as Linear-L and Linear), (b) PFHT [5], a cuckoo hashing [22] variant with larger buckets and at most one displacement during insert operations, with and without logging (referred as PFHT-L and PFHT), (c) path hashing [34], a recently proposed NVM-friendly hashing scheme with the technique of path sharing, with and without logging (referred as Path-L and Path). The load factor of the hashing schemes is set to 0.5. The size of a hash cell in random integer trace is 16 bytes. The details about the experimental environment are described in Section 4.1. Note that only insertion and deletion requests have NVM writes. As shown in Figure 2 (a), for insert and delete operations, Linear-L, PFHT-L and Path-L are 1.95X slower on average than their no logging versions. The reason is that logging mechanism requires duplicate copy writes, which incur a large amount of extra cacheline flush (i.e., *clflush*) operations.

Note that *clflush* instruction flushes a cacheline by explicitly invalidating it, which will incur a cache miss when reading the same memory address later. We use the performance counters in the modern processors, through the PAPI library [20], to count the L3 cache misses during insert, query and delete operations. As shown in Figure 2 (b), for insert and delete operations, Linear-L, PFHT-L and Path-L produce 2.16X more L3 cache misses on average than their no logging versions, which can explain the performance drop in Figure 2 (a).

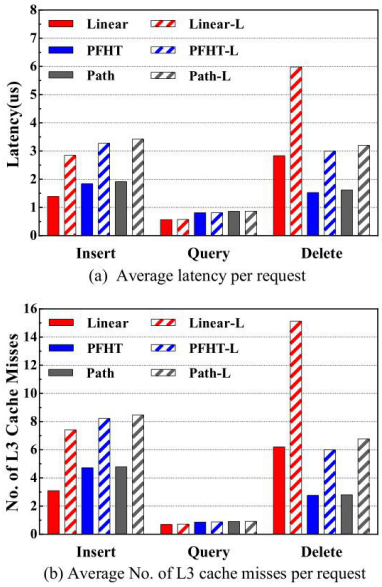


Figure 2: The consistency cost of different hashing schemes.

Besides the extra writes caused by the logging mechanism, CPU cache efficiency also has a significant impact on request performance including query latency. From Figure 2 (a), we observe that, although linear hashing has poor deletion performance due to the extra writes caused by the complicated delete process, linear hashing has better insert and query performance than PFHT and path hashing in both logging and no logging versions. The reason is that when hash collisions occur in linear hashing, the closest contiguous cells are searched and checked, the collision resolution cells are in the contiguous memory address. Actually, most key-value stores, such as memcached and MemC3, are dominated by small items whose sizes are smaller than a cacheline size [1, 7]. For hash cells in the contiguous memory address, a single memory access can prefetch multiple cells belonging to the same cacheline, which reduces the number of memory access and L3 cache miss to obtain higher performance. In path hashing, there are two hashing functions, and the cells in each collision addressing path (i.e., path in the binary tree) are not contiguous in memory space. In this case, each cell in the path requires one memory access, which increases

[1]In this paper, we focus on failures which can be recovered by utilizing data in NVM. Other types of failures such as hardware errors, which require additional mechanisms (e.g., error correction codes), are beyond the scope of this paper.

the number of memory access and L3 cache miss. PFHT also has two hashing functions, but the cells in each bucket are contiguous in memory address, thus PFHT performs better than path hashing but worse than linear hashing.

In summary, we present two insightful observations: **(1) duplicate copy techniques (e.g., logging) incur a large amount of extra writes which significantly deteriorate the insertion and deletion performance; (2) keeping the memory space continuity of hashing cells for dealing with collisions can reduce CPU cache misses and improve the performance in terms of request latency.**

3 THE DESIGN OF GROUP HASHING

In this section, we present group hashing, a consistent and write-efficient hashing scheme for NVMs.

3.1 Design Goals

Based on the above observations, our design goals are below:

Failure-atomic write to commit insert and delete operations. Since duplicate copy techniques incur extra NVM writes and cacheline flush operations which significantly deteriorate the performance, it would be nice to guarantee data consistency without duplicate copy techniques such as logging. Group hashing uses an 8-byte failure-atomic write after an insert or delete operation to commit the update.

High CPU cache efficiency. The CPU cache efficiency has a significant impact on request latency. Group hashing uses group sharing scheme to deal with hash collisions, which organizes the collision resolution cells in the contiguous memory address. In this way, group hashing reduces the number of CPU cache miss and achieves higher performance in terms of request latency.

Fast and efficient recovery after system failure. Group hashing includes an efficient recovery mechanism. When an unexpected system crash or power failure occurs, group hashing can recover to a consistent state in a very short time.

3.2 The Overview of Group Hashing

Our group hashing leverages group sharing scheme to deal with hash collisions. Specifically, group hashing decouples storage cells into two levels. As shown in Figure 3, the cells in the first level are addressable by the hash function, i.e., hash addressable cells. The cells in the second level are non-addressable and used to deal with hash collisions, i.e., collision resolution cells. Group hashing divides the cells in both levels into many groups, the cells in each group are stored in a contiguous memory space. The total group amount in each level is equal. Each group in the first level matches a group in the second level with the same group number. When hash collisions occur, group hashing searches the empty cells in the matched group of the second level. The collision resolution cells in the group of the second level are shared by the matched group in the first level with the same group number.

In the example of Figure 3, the number of cells in a group, i.e., the group size is 4. The total number of hash cells in the hash table is 2^{2n}. A new item *(key,value)* is hashed in the cell index 5 of the first level. If the cell in index 5 is occupied, group hashing searches the empty cells in the group of the second level from index 2^n+4 to 2^n+7. The collision resolution cells in the group with index from 2^n+4 to 2^n+7 are shared by the group with index from 4 to 7.

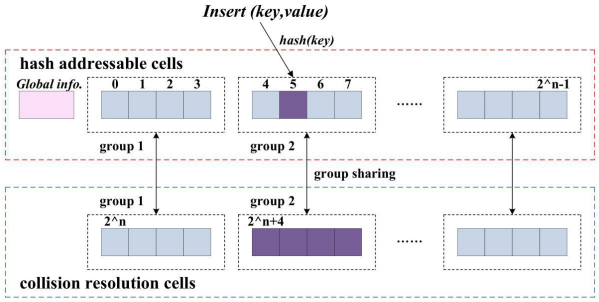


Figure 3: The layout of group hashing.

When hash collisions occur, only the collision resolution cells in a group are needed to be searched and checked. In this way, no extra writes are caused during insert and delete operations in group hashing. As the collision resolution cells inside a group are contiguous in memory address, a single memory access can prefetch the following cells belonging to the same cacheline, which reduces the number of memory access and L3 cache miss, thus obtaining higher performance in terms of request latency.

3.3 Failure Atomic Write in Group Hashing

In this section, we present the data consistency guarantee upon system failure without requiring any kind of duplicate copy techniques as long as the 8-byte failure atomicity assumption is satisfied [6, 10, 13].

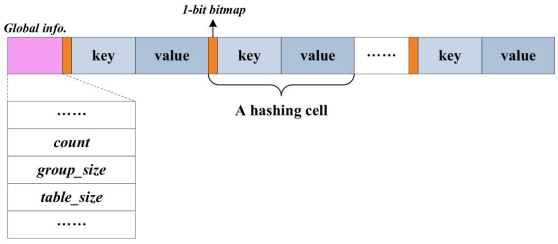


Figure 4: The physical storage structure of group hashing.

Only insert and delete operations have NVM writes, which may cause data consistency problem. Figure 4 shows the physical storage structure of group hashing. The *Global info.* records the global information of the hash table, such as the number of occupied hash cells (*count*), the number of hash cells in a group (*group_size*), and the total number of hash cells (*table_size*). We add an 1-bit *bitmap* in each hashing cell to determine whether the cell is occupied or empty, i.e., *'bitmap == 1'* means the cell is occupied and *'bitmap == 0'* means the cell is empty. The *bitmap* is also used to commit the update in an insert or delete operation. When the *bitmap* is atomic updated from '0' to '1' (or '1' to '0'), it means that the insertion (deletion) has completed. To better present the failure atomic write scheme, let us review the example in Figure 1. After the key-value pair (21,Hash Table) has been written to the target cell following with a *clflush* and *mfence*, *bitmap* is atomic updated from '0' to '1',

then *count* is atomic incremented. If a system failure occurs before the atomic update of *bitmap*, the key-value pair will be cleared during the recovery step. If the system failure occurs between the atomic update of *bitmap* and *count*, group hashing will count the number of occupied cells whose *bitmap* equals to '1' by scanning the whole hash table and modify *count* to the correct value during the recovery step. In both of these two cases, the data consistency is not compromised. The details about the recovery mechanism are described in Section 3.5.

For simplicity, we use *persist* to represent the persistence of the data written to NVM in the rest of this work, which is implemented by executing a *clflush* following with a *mfence*.

3.4 Operations

In this section, we introduce the base operations in group hashing, including insertion, query and deletion.

Insertion: In the insert operation, group hashing first computes the position k in the first level, as shown in Algorithm 1. If the cell in position k is empty, group hashing inserts the key-value pair into this empty cell. Group hashing ensures the persistence of the key-value pair in NVM with a *clflush* and *mfence*. After that, *bitmap* is atomic updated from '0' to '1'. Then the number of occupied hash cells *count* is atomic incremented. If the system failure occurs before the atomic update of *bitmap*, the key-value pair will be cleared during the recovery step. If the system failure occurs between the atomic update of *bitmap* and *count*, group hashing will count the number of occupied cells whose *bitmap* equals to '1' by scanning the whole hash table, and update *count* field to the correct value during the recovery step. In this way, the data consistency can be ensured. If the cell in position k is occupied, group hashing computes the first cell position j of the matched group in the second level and checks the cells in this group until finds an empty cell. Then group hashing inserts the key-value pair into the empty cell following with a *clflush* and *mfence*, and atomic updates *bitmap* and *count*. If there are no empty cells in the matched group, it means that the capacity of the hash table needs to be expanded.

Query: Since a query operation does not modify any persistent data, data consistency will not be compromised during query operations. In the query operation, group hashing first computes the position k in the first level, and checks whether the item in position k is the target item, as shown in Algorithm 2. If the item in position k is not the target item, group hashing computes the first cell position j of the matched group in the second level and checks the cells in this group until finds the target item. If the target item can not be found in the group, it means that the queried item does not exist in the hash table.

Deletion: In the delete operation, group hashing first queries and checks whether the item in the first level is the target item to be deleted, as shown in Algorithm 3. If the item in position k is the target item, group hashing deletes the key-value pair and persists the update. Different from the process in the insertion, the atomic update on *bitmap* is before the deletion of the key-value pair. The reason is that, if the atomic update of *bitmap* is after the deletion of the key-value pair, when a system failure occurs after the deletion of the key-value pair and before the atomic update of *bitmap*, the *bitmap* still equals to '1' but the key-value pair has been deleted, which leads to an inconsistent state. Actually, in the recovery step, group hashing will clear the data in key-value field of the cells whose *bitmap* equals to '0'. Once the *bitmap* is atomic updated to '0', the delete operation has been completed. In this way, the delete operation is atomic like insert operation and the data consistency can be guaranteed. If the item in position k is not the target item, group hashing searches it in the matched group in the second level until finds the target item and deletes it.

When hash collisions occur during insert, query and delete requests, only the cells in a group need to be searched and checked, which has the time complexity of *O(1)*. Furthermore, the cells in a group are contiguous in memory address, less CPU cache misses

Algorithm 1 Insert(group,key,value)

```
1:  k=h(key);
2:  /*Check whether the cell in the first level is empty*/
3:  if group->tab1[k].bitmap == 0 then
4:     Insert(key,value);
5:     Persist(key,value);
6:     Atomic Update group->tab1[k].bitmap to 1;
7:     Persist(group->tab1[k].bitmap);
8:     AtomicInc(group->count);
9:     Persist(group->count);
10:    return TRUE;
11: end if
12: /*Lookup empty cell in the group of the second level*/
13: j=k-k%group->group_size;
14: for i=0; i<group->group_size; i++; do
15:    if group->tab2[j+i].bitmap == 0 then
16:       Insert(key,value);
17:       Persist(key,value);
18:       Atomic Update group->tab2[j+i].bitmap to 1;
19:       Persist(group->tab2[j+i].bitmap);
20:       AtomicInc(group->count);
21:       Persist(group->count);
22:       return TRUE;
23:    end if
24: end for
25: return FALSE;
```

Algorithm 2 Query(group,key)

```
1:  k=h(key);
2:  /*Check whether the item in the first level is the target*/
3:  if group->tab1[k].bitmap == 1 && group->tab1[k].key == key
    then
4:     return group->tab1[k].value;
5:  end if
6:  /*Lookup the target item in the group of the second level*/
7:  j=k-k%group->group_size;
8:  for i=0; i<group->group_size; i++; do
9:     if group->tab2[j+i].key == key then
10:       return group->tab2[j+i].value;
11:    end if
12: end for
13: return NULL;
```

Algorithm 3 Delete(group,key)

```
1: k=h(key);
2: /*Check whether the item in the first level is the target*/
3: if group->tab1[k].key == key then
4:    Atomic Update group->tab1[k].bitmap to 0;
5:    Persist(group->tab1[k].bitmap);
6:    Delete(key,value);
7:    Persist(key,value);
8:    AtomicDec(group->count);
9:    Persist(group->count);
10:   return TRUE;
11: end if
12: /*Lookup the target item in the group of the second level*/
13: j=k-k%group->group_size;
14: for i=0; i<group->group_size; i++; do
15:    if group->tab2[j+i].key == key then
16:       Atomic Update group->tab2[j+i].bitmap to 0;
17:       Persist(group->tab2[j+i].bitmap);
18:       Delete(key,value);
19:       Persist(key,value);
20:       AtomicDec(group->count);
21:       Persist(group->count);
22:       return TRUE;
23:    end if
24: end for
25: return FALSE;
```

would be produced during the requests, thus group hashing can obtain more performance improvements. The group size has a significant impact on performance and space utilization, we will discuss it in Section 4.5.

3.5 Recovery

Group hashing must recover to a consistent state after an unexpected system crash or power failure. Algorithm 4 shows the pseudo-code of the recovery step in group hashing. Since only insertion and deletion requests may cause data consistency problem, we take a deletion request as an example to elaborate the recovery step in group hashing. We classify the possible cases into two scenarios.

First, a system crash occurs between the atomic update on *bitmap* and the persistence of deleting the key-value pair, i.e., between line 5 and line 6 in Algorithm 3. In this case, *bitmap* has been atomic updated to '0', the deletion of the key-value pair and the atomic decrease of the *count* field have not been completed. Group hashing scans the whole hash table, and resets the data in key-value field of the cells whose *bitmap* equal to '0', thus the possible partially deleted key-value pair data can be cleared. Then group hashing counts the amount of the cells whose *bitmap* equals to '1' and updates the *count* field to the correct value. After that, group hashing recovers to a consistent state.

Second, a system crash occurs between the persistence of deleting the key-value pair and the atomic update on the *count* field, i.e., between line 7 and line 8 in Algorithm 3. In this case, *bitmap* has been atomic updated to '0', and the deletion of the key-value pair has been persisted, but the atomic update of *count* field has not been completed. Since group hashing does not know when the system crash occurs, it still scans the whole hash table, resets the data in key-value field of the cells whose *bitmap* equals to '0', then counts the amount of the cells whose *bitmap* equals to '1' and updates the *count* field to the correct value. In this way, group hashing can also recover to a consistent state after the recovery step.

Algorithm 4 Recover(group)

```
1: count=0;
2: /*Scan the whole hash table*/
3: for i=0; i<group->table_size; i++; do
4:    /*Clear and reset the cells whose bitmap equals to 0*/
5:    if group->tab1[i].bitmap == 0 then
6:       Reset(key,value);
7:       Persist(key,value);
8:    else
9:       count++;
10:   end if
11:   if group->tab2[i].bitmap == 0 then
12:      Reset(key,value);
13:      Persist(key,value);
14:   else
15:      count++;
16:   end if
17: end for
18: /*Update the number of occupied cells to the correct value*/
19: group->count=count;
20: Persist(group->count);
21: return TRUE;
```

4 PERFORMANCE EVALUATION

In this section, we evaluate the performance of our group hashing and answer the following questions:

(1) *How does group hashing perform against existing hashing schemes on NVM?*

(2) *How is the CPU cache efficiency of group hashing compared with other hashing schemes on NVM?*

(3) *How is group hashing sensitive to the group size?*

(4) *How long does it take for group hashing to recover to a consistent state after a system failure occurs?*

We first describe the experimental setup and then evaluate group hashing with three real-world traces.

4.1 Experimental Setup

Since NVM is not yet commercially available for us, we use a portion of the DRAM region as NVM. It is managed by PMFS [6], an open-source NVM-based file system, which gives direct access to the memory region with *mmap*. As NVM has similar read latency to DRAM and emulating read latency is complicated due to CPU features such as speculative execution, memory parallelism, prefetching, etc. [6], we only emulate NVM's slower writes than DRAM by adding extra latency after a *clflush* instruction, like previous research works on NVM [6, 29]. In our experiments, we set the extra latency to 300ns by default [6]. The configurations of the server we use in our experiments are listed in Table 2.

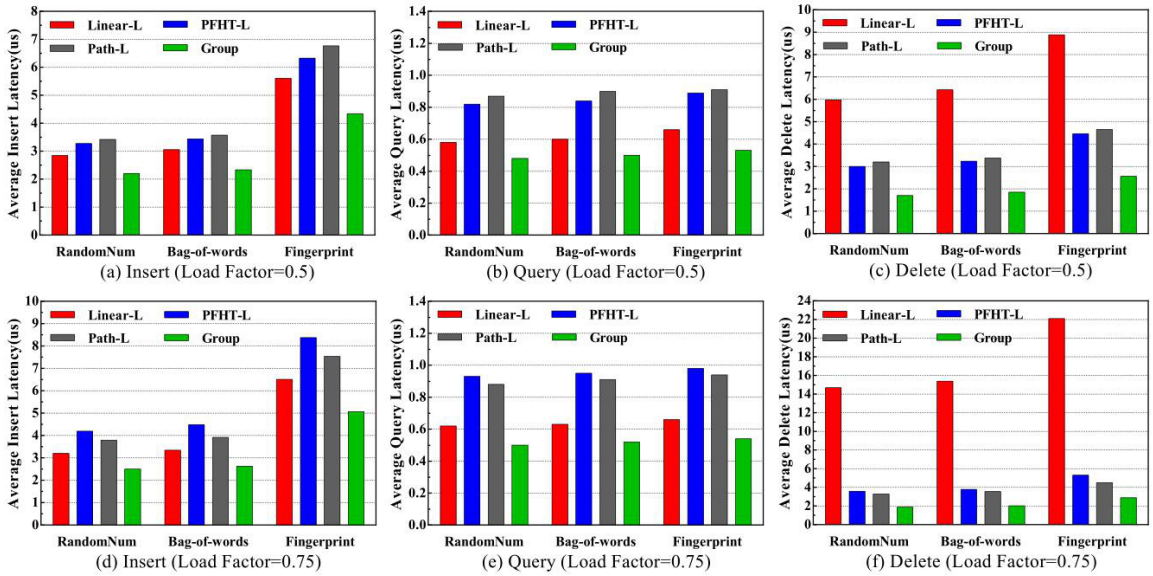


Figure 5: The average latency of requesting an item.

We compare our group hashing with (1) linear probing [24], a traditional hashing scheme used in DRAM, (2) PFHT [5], an NVM-friendly cuckoo hashing [22] variant with larger buckets and at most one displacement during insert operations, and (3) path hashing [34], a recently proposed NVM-friendly hashing scheme with the technique of path sharing and path shortening. There are some other hashing schemes used in traditional DRAM, such as chained hashing [2], 2-choice hashing [2] and cuckoo hashing [22]. However, chained hashing performs poorly under memory pressure due to frequent memory allocation and free calls, 2-choice hashing has too low space utilization ratio, PFHT is an NVM optimized variant of cuckoo hashing, thus we do not take them into the comparison. We use three real-world traces in the experiments.

Table 2: Server Configurations

CPU	Intel Xeon E5-2620, 2.0 GHz
CPU cores	12
Processor cache	384KB/1.5MB/15MB L1/L2/L3 cache
DRAM	8GB
NVM	8GB, emulated with slowdown, the write latency is 300 ns
Operating system	CentOS 6.5, kernel version 3.11.0

RandomNum: This trace is widely used for evaluating the performance of hashing schemes in previous research works [26, 34]. We generate the random integer ranging from 0 to 2^{26} and use the generated integers as the keys of the hash items to be inserted into the hash table. The size of an item in this trace is 16 bytes.

Bag-of-Words: There are five text collections in the form of bags-of-words in this trace [25]. We choose PubMed abstracts, which is the largest collection and contains about 82 million items, for evaluation. The combinations of DocID and WordID are used as the keys of the hash items. The size of an item in this trace is 16 bytes.

Fingerprint: This trace is collected from the daily snapshots of a Mac OS X server [27]. We use the 16-byte MD5 fingerprints of the files as the keys of the hash items. The size of an item in this trace is 32 bytes.

In our experiments, we use 2^{23} hash table cells in *RandomNum* trace, 2^{24} cells in *Bag-of-Words* trace, and 2^{25} cells in *Fingerprint* trace. We evaluate the performance under two load factors, 0.5 and 0.75. In our group hashing, the group size, i.e., the number of cells in a group, is set to 256 by default. In PFHT, each bucket contains 4 hash cells and we store the insertion-failure cells in an extra stash with 3% size of the hash table. In path hashing, we set the reserved levels to 20. To ensure data consistency like group hashing, we add a logging scheme in linear probing, PFHT and path hashing, referred as linear-L, PFHT-L and path-L. For all the experiments, each result is the average of five independent executions.

4.2 Request Latency

In all the experiments, we first insert items into the hash table until the load factor reaches the predefined value. After that, we insert 1000 items into the hash table, then query and delete 1000 items from the hash table. At last, we calculate the average latency of requesting an item. Figure 5 shows the average latency of requesting an item in different hashing schemes. As shown in the figures, our group hashing outperforms other hashing schemes in insert, query and delete operations under two load factors due to the low consistency cost and high CPU cache efficiency. Linear hashing has better insert and query performance than PFHT and path hashing. This is because when hash collisions occur in linear hashing, the closest contiguous cells are searched and checked, the collision resolution cells are in a contiguous memory address. For small items whose sizes are smaller than a cacheline size, a single memory access can prefetch multiple cells belonging to the same cacheline, which reduces the number of memory access and L3 cache miss, thus

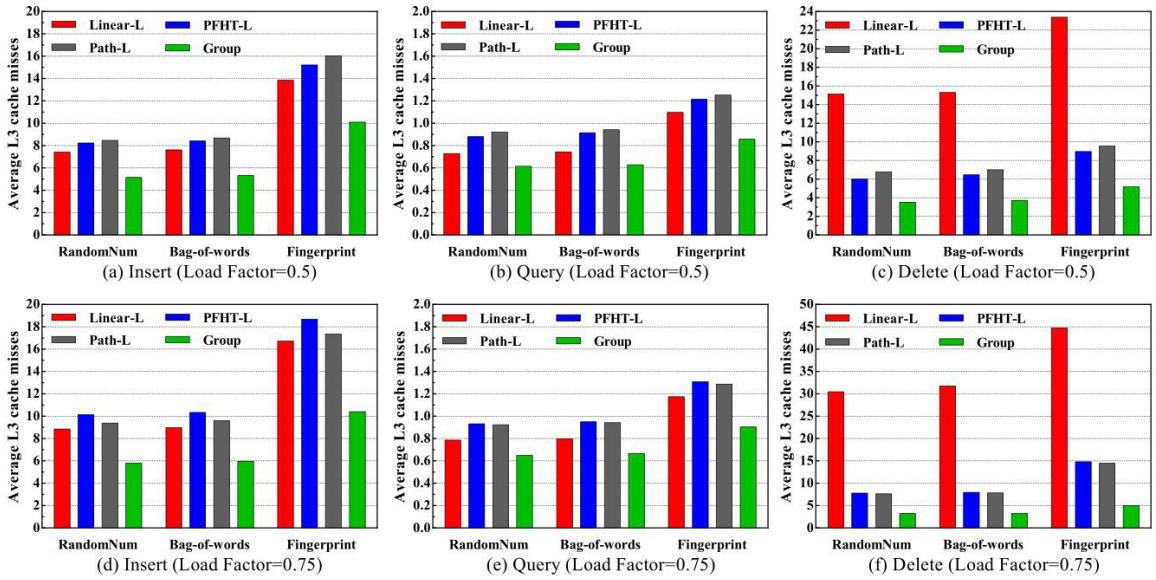


Figure 6: The average L3 cache miss number of requesting an item.

obtaining higher insert and query performance. However, linear hashing has poor delete performance due to the extra writes caused by its complicated delete process, and the problem becomes worse as the load factor increases to 0.75. We notice that PFHT performs better than path hashing under load factor 0.5, but performs worse than path hashing under load factor 0.75. The reason is that, under load factor 0.75, more items are stored in the extra stash. In this case, PFHT needs to spend more time to linearly search hash items in the stash. In the case of load factor 0.5, as multiple cells in a bucket of PFHT is in the continuous memory space, and less items are stored in the stash, while the cells in the collision addressing path (i.e., path in the binary tree) of path hashing are not in contiguous memory space, PFHT performs better than path hashing. We also observe that the insert and delete performance in *Fingerprint* trace is much slower than that in the other two traces. This is because *Fingerprint* has larger item size than the other two traces.

4.3 CPU Cache Efficiency

We use the PAPI library [20] to count the number of L3 cache miss during insert, query and delete operations under two load factors. Figure 6 shows the average L3 cache miss number of requesting an item in different hashing schemes. As shown in the figures, our group hashing produces least CPU cache misses. There are two reasons. First, our group hashing does not include any duplicate copy techniques, less NVM writes and cacheline flush operations are produced. Second, contiguous cells inside a group are searched when hash collision occurs in group hashing. A single memory access in group hashing can prefetch the following cells belonging to the same cacheline, thus reducing the number of memory access and L3 cache miss. For linear hashing, it produces less L3 cache misses than PFHT and path hashing in insert and query operations because the collision resolution cells are in a contiguous memory address. But linear hashing produces much more L3 cache misses in delete operation due to the extra writes caused by the complicated delete process. Like the request latency, PFHT produces less L3 cache misses than path hashing under load factor 0.5, but produces more L3 cache misses than path hashing under load factor 0.75. The reason is similar. In the case of load factor 0.75, more items are stored in the extra stash and PFHT needs to spend more time to linearly search hash items in the stash.

4.4 Space Utilization

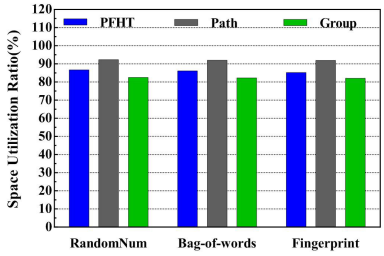


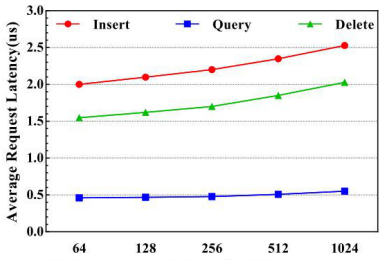
Figure 7: Space utilization ratios of different hashing schemes.

Space utilization is another important parameter for hash tables, which is defined as the load factor when an item fails to insert into the hash table. For PFHT, each bucket contains 4 hash cells and we store the insertion-failure cells in an extra stash with 3% size of the hash table. For path hashing, we set the reserved levels to 20. For group hashing, the group size, i.e., the number of cells in a group, is set to 256. The space utilization ratios of different hashing schemes are shown in Figure 7. Path hashing achieves the highest space utilization ratio due to its path sharing technique and double hashing paths. PFHT also has two hash functions and it has an extra stash, but its space utilization ratio is slightly lower

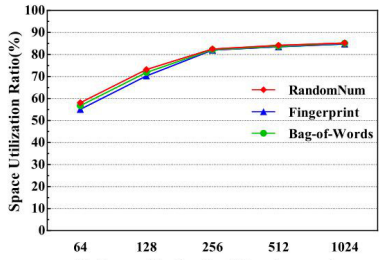
than path hashing because it only allows the eviction operation at most once. The result of linear probing is not shown in the figure, because linear probing does not have a fixed space utilization ratio, and its load factor can be up to 1. Our group hashing achieves about 82% in the three traces, which is lower than PFHT and path hashing. Although two hash functions can be used in our group hashing to improve the space utilization ratio, the continuity of the collision resolution cells is damaged, more L3 cache misses would be produced, which deteriorates the performance in terms of request latency. Considering the performance benefits described above, the space utilization ratio of group hashing is acceptable.

4.5 Effect of the group size

The group size, i.e., the number of cells in a group, can affect the performance and space utilization ratio. We vary the group size from 64 to 1024 and evaluate the request latency and space utilization ratio. Figure 8 (a) shows the request latency in trace *RandomNum* under load factor 0.5. From the figure, we observe that the insert, query and delete latencies increase as the group size grows. This is because larger group sizes incur more search time when hash collision occurs. Figure 8 (b) shows the space utilization ratio in three traces. As shown in the figure, the space utilization ratio also increases as the group size grows. We notice that when the group size reaches 256, the space utilization ratio can achieve over 80%, and the request latency is acceptable. Therefore, we choose 256 as the default group size in our group hashing.



(a) Average request latency for different group sizes



(b) Space utilization for different group sizes

Figure 8: Group size vs. request latency and space utilization.

4.6 Failure Recovery

At last, we evaluate the recovery time of group hashing after a system failure occurs. We vary the hash table size from 128MB to 1GB and evaluate the recovery time in trace *RandomNum*. Compared with the total execution time of inserting the hash table to load factor 0.5, as shown in Table 3, the percentage of recovery time in the total execution time is below 1%, which is totally negligible. The results indicate that group hashing can recover to a consistent state quickly and efficiently after a system failure.

Table 3: Recovery time for different hash table sizes

Hash Table Size	128MB	256MB	512MB	1GB
Recovery Time(ms)	77.8	156.3	314.1	630.2
Execution Time(ms)	8426.2	16854.3	33710.5	67422.8
Percentage	0.92%	0.93%	0.93%	0.93%

5 RELATED WORK

In order to efficiently adapt to the characteristics of the emerging NVMs, designing persistent data structures for NVMs is an interesting topic.

Most previous research works on persistent data structures for NVMs focus on the tree-based index structures, such as B^+-tree and radix tree. CDDS B-tree [28] uses multi-version to guarantee data consistency. However, CDDS B-tree suffers from poor insert and query performance due to numerous dead entries and nodes. wB^+-Tree [3] uses unsorted nodes with bitmaps to reduce the number of expensive NVM writes. Although wB^+-Tree adopts 8-byte atomic update strategy on bitmaps, it still needs logging on more complex operations such as splits for data consistency. Chi *et al.* [4] observes that using unsorted nodes in B^+-Tree suffers from CPU-costly for insertion and wasting space for deletion problems. They propose three techniques including the sub-balanced unsorted node, overflow node, and merging factor schemes to address the problems. NV-Tree [32] and FPTree [21] only keep leaf nodes in persistent NVM while internal nodes are kept in volatile DRAM, which reduces consistency cost significantly. WORT [13] is a recently proposed radix tree data structure for NVM. To avoid duplicate copy writes on NVM, it uses 8-byte atomic write to ensure data consistency. FAST and FAIR B^+-Tree [10] is another recently proposed B^+-Tree for NVM. It makes read operations tolerate transient inconsistency to avoid expensive duplicate copy writes on NVM. However, these tree-based indexing structures only provide *O(log(N))* lookup time complexity on average, which is much slower than hashing-based structures.

Besides the tree-based index structures, hashing-based data structures are also widely used in main memory applications [1, 7, 8] due to their constant-scale lookup time complexity. PFHT [5] is an NVM-friendly cuckoo hashing variant with larger buckets and at most one displacement during insert operations, which reduces cascading writes and improves the CPU cache utilization. Path hashing [34] is a recently proposed NVM-friendly hashing scheme. It organizes storage cells as an inverted complete binary tree and uses a novel hash-collision resolution method called position sharing, which incurs no extra writes to NVMs. However, these two hashing schemes focus on reducing extra writes to NVM and do not provide mechanisms to maintain data consistency in case of unexpected system failures. Furthermore, they do not consider the persistency cost of cacheline flush and memory fence when writing to NVM through memory bus. On the contrary, our group hashing uses 8-byte failure-atomic write to guarantee data consistency without any duplicate copies for logging or CoW. To further improve CPU

cache efficiency, our group hashing adopts group sharing technique to deal with hash collisions, which obtains higher performance in terms of request latency.

6 CONCLUSION

In this paper, we propose a write-efficient and consistent hashing scheme, called group hashing, for NVMs to improve the performance in terms of request latency while guaranteeing data consistency in case of unexpected system failures. Group hashing uses 8-byte failure-atomic write to guarantee the data consistency, which eliminates the duplicate copy writes to NVMs, thus reducing the consistency cost of the hash table structure. To improve CPU cache efficiency, our group hashing leverages group sharing to deal with hash collisions, which divides the hash table into groups and organizes the collision resolution cells in the contiguous memory address, thus reducing CPU cache misses to obtain higher performance in terms of request latency. Extensive experimental results show that our group hashing achieves low request latency as well as high CPU cache utilization, compared with state-of-the-art NVM-based hashing schemes.

ACKNOWLEDGMENT

This work was supported by the National High Technology Research and Development Program (863 Program) No.2015AA015301, NSFC No.61772222, No.61772212, No.61472153, No.61502191; the National Key Research and Development Program of China under Grant 2016YFB1000202; State Key Laboratory of Computer Architecture, No.CARCH201505; This work was also supported by Engineering Research Center of data storage systems and Technology, Ministry of Education, China.

REFERENCES

[1] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. *ACM Sigmetrics Performance Evaluation Review* 40, 1 (2012), 53–64.

[2] J Lawrence Carter and Mark N Wegman. 1979. Universal classes of hash functions. *J. Comput. System Sci.* 18, 2 (1979), 143–154.

[3] Shimin Chen and Qin Jin. 2015. Persistent B+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment* 8, 7 (2015), 786–797.

[4] Ping Chi, Wang Chien Lee, and Yuan Xie. 2016. Adapting B^+-Tree for Emerging Nonvolatile Memory-Based Main Memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 9 (2016), 1461–1474.

[5] 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.

[6] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*.

[7] Bin Fan, David G Andersen, and Michael Kaminsky. 2013. MemC3: compact and concurrent MemCache with dumber caching and smarter hashing. In *in Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. 371–384.

[8] Yu Hua, Bin Xiao, Dan Feng, and Bo Yu. 2008. Bounded LSH for Similarity Search in Peer-to-Peer File Systems. In *Proceedings of the 37th International Conference on Parallel Processing (ICPP)*. 644–651.

[9] Fangting Huang, Dan Feng, Yu Hua, and Wen Zhou. 2017. A wear-leveling-aware counter mode for data encryption in non-volatile memories. In *Proceedings of the IEEE 2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 910–913.

[10] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable transient inconsistency in byte-addressable persistent B^+-tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies(FAST)*. 187–200.

[11] Intel and Micron. 2015. Intel and Micron produce breakthrough memory technology. https://newsroom.intel.com/news-releases/. (2015).

[12] Emre Kültürsay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. 2013. Evaluating STT-RAM as an energy-efficient main memory alternative. In *Proceedings of the IEEE 2013 International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 256–267.

[13] Se Kwon Lee, K Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H Noh. 2017. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems.. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*. 257–270.

[14] Zheng Li, Fang Wang, Dan Feng, Yu Hua, Wei Tong, Jingning Liu, and Xiang Liu. 2016. Tetris Write: Exploring More Write Parallelism Considering PCM Asymmetries. In *Proceedings of the 45th International Conference on Parallel Processing (ICPP)*. 159–168.

[15] Yi Lin, Po Chun Huang, Duo Liu, Xiao Zhu, and Liang Liang. 2016. Making In-Memory Frequent Pattern Mining Durable and Energy Efficient. In *Proceedings of the 45th International Conference on Parallel Processing (ICPP)*. 47–56.

[16] Ren Shuo Liu, De Yu Shen, Chia Lin Yang, Shun Chih Yu, and Cheng Yuan Michael Wang. 2014. NVM Duet: unified working memory and persistent store architecture. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 455–470.

[17] Youyou Lu, Jiwu Shu, and Long Sun. 2015. Blurred persistence in transactional persistent memory. In *Proceedings of the IEEE 31st Symposium on Mass Storage Systems and Technologies (MSST)*.

[18] Manqing Mao, Yu Cao, Shimeng Yu, and Chaitali Chakrabarti. 2015. Optimizing latency, energy, and reliability of 1T1R ReRAM through appropriate voltage settings. In *Proceedings of the IEEE 33rd International Conference on Computer Design (ICCD)*. 359–366.

[19] Sparsh Mittal and Jeffrey S Vetter. 2016. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Transactions on Parallel and Distributed Systems* 27, 5 (2016), 1537–1550.

[20] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. 1999. PAPI: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, Vol. 710.

[21] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 371–386.

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

[23] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory persistency. In *Proceeding of the 41st International Symposium on Computer Architecuture (ISCA)*. 265–276.

[24] Boris Pittel. 1987. Linear probing: the probable largest search time grows logarithmically with the number of records. *Journal of Algorithms* 8, 2 (1987), 236–249.

[25] UC Irvine Machine Learning Repository. 2008. Bags of Words data set. http://archive.ics.uci.edu/ml/datasets/Bag+of+Words. (2008).

[26] Yuanyuan Sun, Yu Hua, Song Jiang, Qiuyu Li, Shunde Cao, and Pengfei Zuo. 2017. SmartCuckoo: a fast and cost-efficient hashing index scheme for cloud storage systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*. 553–565.

[27] Vasily Tarasov, Amar Mudrankit, Will Buik, Philip Shilane, Geoff Kuenning, and Erez Zadok. 2012. Generating Realistic Datasets for Deduplication Analysis.. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*. 261–272.

[28] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, Roy H Campbell, et al. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*. 61–75.

[29] Haris Volos, Andres Jaan Tack, and Michael M Swift. 2011. Mnemosyne: Lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 91–104.

[30] 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 the 2017 USENIX Annual Technical Conference (ATC)*. 349–362.

[31] Yuan Xie. 2011. Modeling, architecture, and applications for emerging memory technologies. *IEEE Design & Test of Computers* 28, 1 (2011), 44–51.

[32] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing consistency cost for nvm-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, Vol. 15. 167–181.

[33] Yiying Zhang and Steven Swanson. 2015. A study of application performance with non-volatile main memory. In *Proceedings of the IEEE 31st Symposium on Mass Storage Systems and Technologies (MSST)*.

[34] Pengfei Zuo and Yu Hua. 2017. A write-friendly hashing scheme for non-volatile memory systems. In *Proceedings of the IEEE 33rd Symposium on Mass Storage Systems and Technologies (MSST)*.