A Write-friendly and Cache-optimized Hashing Scheme for Non-volatile Memory Systems

Pengfei Zuo, *Student Member, IEEE,* and Yu Hua, *Senior Member, IEEE*

Abstract—Non-volatile memory technologies (NVMs) are promising candidates for building future memory systems, due to their advantages of high density, high scalability, and requiring near-zero standby power, while suffering from the limited endurance and asymmetric properties of reads and writes, compared with traditional memory technologies including DRAM and SRAM. The significant changes of low-level memory devices cause nontrivial challenges to high-level in-memory and in-cache data structure design due to overlooking the NVM device properties. In this paper, we study an important and common data structure, hash table, which is ubiquitous and widely used to construct the index and lookup table in main memory and caches. Based on the observations that existing hashing schemes cause many extra writes to NVMs, we propose a cost-efficient write-friendly hashing scheme, called path hashing, which incurs no extra writes to NVMs while delivers high performance. The basic idea of path hashing is to leverage a novel hash-collision resolution method, i.e., position sharing, which meets the needs of insertion and deletion requests without extra writes to NVMs. By further exploiting double-path hashing and path shortening techniques, path hashing delivers high performance of hash tables in terms of space utilization and request latency. Nevertheless, the original path hashing has low utilization of each cache line for small items, causing low cache efficiency. We hence propose a cache-optimized path hashing to pack multiple cells in the same path together and store them into one cache line, thus improving the cache line utilization to obtain higher performance. We have implemented path hashing and used a gem5 full system simulator with NVMain to evaluate its performance in the context of NVMs. Extensive experimental results demonstrate that path hashing incurs no extra writes to NVMs, and achieves up to 95% space utilization ratio as well as low request latency, compared with existing state-of-the-art hashing schemes. We have released the source code of path hashing for public use at github.

Index Terms—Non-volatile memory, hashing scheme, write endurance, collision resolution.

1 Introduction

Over the past few decades, traditional memory technologies including DRAM and SRAM, have been used as the main memory and on-chip caches in the memory hierarchy, which however suffers from the increasing leakage power dissipation and limited scalability [1], [2]. To address this problem, non-volatile memory (NVM) technologies, e.g., phase-change memory (PCM), resistive random access memory (ReRAM), and spin-transfer torque RAM (STT-RAM), are considered as promising candidates of next-generation memory [3], [4], due to their advantages of high density, high scalability, and requiring near-zero standby power [5], [6], [7]. NVMs can be directly accessed through the memory bus by using CPU load and store instructions due to the byte-addressable property, avoiding the overhead of block-based interfaces [2], [8], [9], [10], [11]. NVMs however have the limitations in terms of write endurance and performance. NVMs typically have limited write endurance, e.g., $10^7 - 10^8$ writes for PCM [12]. The writes on NVMs not only consume the limited endurance, but also cause higher latency and energy than reads [13].

With the significant changes of memory characteristics in computer architecture, an important problem arises [2], [8], [9], [10], [11], i.e., *how could in-memory and in-cache data structures be modified to efficiently adapt to NVMs?* This paper focuses on the hashing-based data structures. The reason is that hashing-based data structures are able to provide fast query response with the constant-scale lookup time complexity [14], which are much faster than tree-based data structures with the average $O(log(N))$ lookup time complexity where N is the size of data structures. Hence, hashing-based data structures are ubiquitous and widely used to construct the index and lookup table in main memory applications, such as main memory databases [15], [16], [17], [18], [19] and key-value stores [20], [21], [22], e.g., Redis [23] and Memcached [24]. As NVMs are expected to replace DRAM as main memory on which these main memory applications are built, it is important to design NVM-friendly hashing schemes. Designing hashing schemes on traditional DRAM memory mainly considers two performance parameters, including space utilization and request latency [25]. Compared with DRAM, one main challenge in designing NVM-friendly hashing schemes is to cope with the limited write endurance and intrinsic asymmetric properties of reads and writes. NVM writes incur much higher latency (i.e., $3-8X$ [26]) and energy than reads, as well as harm the limited endurance. Hence, one important design goal of NVM-friendly hashing schemes is to reduce NVM writes, while delivering high performance.

Hash collisions are difficult to be fully avoided in hashing-based data structures due to probabilistic property [27]. However, the methods for dealing with hash collisions in traditional hashing techniques [25], [28], [29], [30], [31] usually result in many extra writes , i.e., a single

- *P. Zuo and Y. Hua are with Wuhan National Laboratory for Optoelectronics, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074 China.*
E-mail: {pfzuo, csyhua}@hust.edu.cn.

The preliminary version appears in the Proceedings of the 33rd International Conference on Massive Storage Systems and Technology (MSST), 2017.

insertion request to hash table writes multiple items in NVMs, which are not friendly to the NVM write endurance. For example, chained hashing [28] deals with hash collisions via storing the conflicting items in the linked lists, which not only writes the inserted items themselves but also modifies the pointers of other items. Cuckoo hashing [25] deals with hash collisions via evicting the items in the conflicting positions, causing many data movements.

In this paper, we present a write-friendly hashing scheme, called path hashing [32], for NVMs to minimize the writes while efficiently dealing with the hash collisions. Path hashing leverages a novel solution, i.e., position sharing, for dealing with hash collisions, which is not used in any previous hashing schemes. Storage cells in the path hashing are logically organized as an inverted complete binary tree. The last level of the inverted binary tree, i.e., all leaf nodes, is addressable by the hash functions. All nodes in the remaining levels are non-addressable and considered as the shared standby positions of the leaf nodes to deal with hash collisions. When hash collisions occur in a leaf node, the empty standby positions of the leaf node are used to store the conflicting items. Thus insertion and deletion requests in path hashing only need to probe the leaf node and its standby positions for finding an empty position or the target item, resulting in no extra writes. In summary, the main contributions of this paper include:

- We investigate the influence of existing hashing schemes on the writes to NVMs based on both empirical analysis and experimental evaluation. Our main insights include most of existing hashing schemes usually result in many extra writes to NVMs. It is necessary to improve existing hashing schemes to efficiently adapt to NVMs.
- We propose a novel write-friendly hashing scheme, i.e., path hashing, which leverages position sharing technique to deal with hash collisions, allowing insertion and deletion requests to incur no extra NVM writes. By further exploiting double-path hashing and path shortening techniques, path hashing delivers high performance of hash tables in terms of space utilization ratio and request latency.
- We propose a cache-optimized path hashing to improve the cache line utilization of memory accesses in path hashing. Cache-optimized path hashing divides the binary tree into many subtrees and then packs the cells in each subtree together and stores them in the contiguous memory space. Hence, a single memory access can prefetch multiple cells belonging to the same path, which reduces the number of memory accesses to obtain higher performance.
- We have implemented path hashing and evaluated it using the gem5 full system simulator [33] with NVMain [34]. Experimental results show that path hashing incurs no extra writes to NVMs, and achieves up to 95% space utilization ratio as well as low request latency, compared with existing state-of-the-art hashing schemes. We have released the source code of path hashing at github [1].

The rest of this paper is organized as follows. Section 2 presents the background and motivation. Sections 3 and 4 respectively describe the design of path hashing and cache-optimized path hashing. Section 5 presents the evaluation methodology and results. Section 6 discusses the related work and Section 7 concludes our paper.

2 Background and Motivation

In this section, we introduce the background of non-volatile memory technologies and existing hashing schemes. We then analyze the influence of existing hashing schemes on the number of NVM writes.

2.1 Non-volatile Memory Technologies

Unlike traditional DRAM and SRAM using electric charges, emerging non-volatile memory technologies, e.g., PCM, ReRAM, and STT-RAM, use resistive memory to store information, which have higher cell density and near-zero leakage power. Hence, NVMs have been considered as promising candidates of next-generation main memory and caches [5], [6], [7]. Moreover, since NVMs have lower read/write latency and higher endurance than flash, some work [35], [36], [37] considers using NVMs in the external storage to replace/complement the flash-based SSDs or to store the metadata in SSDs.

PCM exploits the resistance difference between amorphous and crystalline states in phase change materials to store binary data, e.g., the high-resistance state represents '0', and the low-resistance state represents '1'. The cell of ReRAM typically has the Metal-Insulator-Metal structure which can be changed between a high-resistance state (representing '0') and a low-resistance state (representing '1'). STT-RAM is a magnetic RAM which switches the memory states using spin-transfer torque.

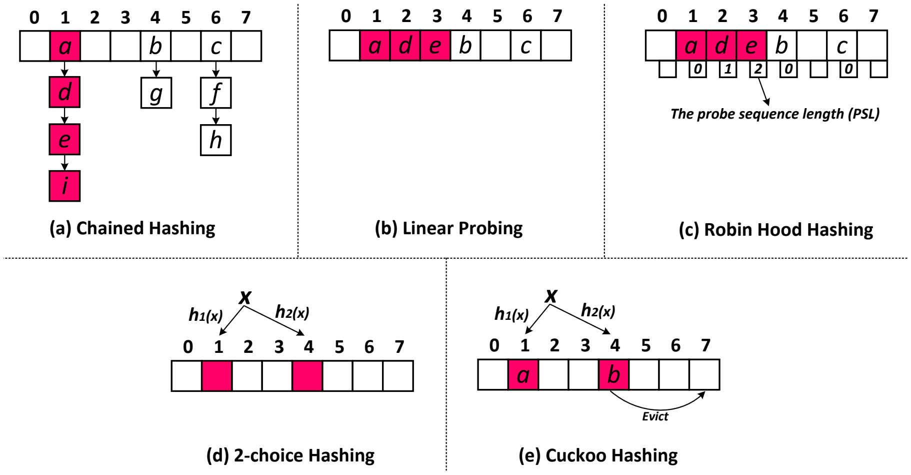
Although different materials and technologies are used in these NVMs, some common limitations are shared. First, they all have the intrinsic asymmetric properties of reads and writes. Write latency is much higher than read latency (i.e., $3-8X$) [5], [6], [26], and writes also consume higher energy than reads. Second, NVMs generally have the limited write endurance, e.g., 10^7-10^8 writes for PCM [12]. Therefore, NVM systems are designed to reduce writes.

2.2 Existing Main Hashing Schemes

Hashing-based data structures have been widely used to construct the index or lookup table in the main memory [15] and caches [38], due to fast query response and constant-scale addressing complexity. Hash collisions, i.e., two or more keys being hashed to the same cell, are practically unavoidable in hashing-based data structures. Typical examples of existing hashing schemes to deal with hash collisions are described as follows.

Chained hashing stores the conflicting items in a linked list and links the list to the conflicting cell [28]. Chained hashing is popular due to the simple data structure and algorithms. However, when querying an item, the chained hashing needs to scan the list that is linked with the cell. The querying performance is poor when the linked lists are too long. Moreover, the chained hashing also inherits

[1] *https://github.com/Pfzuo/Path-Hashing*



(a) Chained Hashing

(b) Linear Probing

(c) Robin Hood Hashing

(d) 2-choice Hashing

(e) Cuckoo Hashing

Fig. 1. Existing main hashing schemes.

the weaknesses of linked lists in heavy space overhead of pointers when storing small keys and values.

Linear probing needs to probe the hash table for the closest following empty cell when the hash computation results in a collision in a cell. To search a key x, linear probing searches the cell at index $h(x)$ and the following cells, $h(x)+1$, $h(x)+2$, ..., until observing either the cell with the key x or an empty cell. Deleting an item in linear probing is complicated, which needs to rehash/move multiple items to fill the empty positions in the lookup sequence [30].

Robin Hood hashing [31] is an interesting extension of linear probing. The insertion operation in Robin Hood hashing is different from linear probing. A new item can displace an already inserted item, if its probe sequence length is larger than that of the item at the current position. The probe sequence length (PSL) is defined as the length of the probed cell sequence when inserting or searching an item. The insertion operation with displacements can reduce the worst-case search time. The query and deletion operations are the same as those of linear probing.

2-choice hashing uses two different hash functions $h_1(x)$ and $h_2(x)$ to compute two positions for each item [28]. An inserted item is stored in any one empty position between the two positions. The insertion fails when both positions are occupied. The query and deletion are simple, which only need to probe two positions. However, the space utilization is usually low due to only two positions used to deal with hash collisions for an inserted item which are easily both occupied.

Cuckoo hashing uses d $(d \geq 2)$ hash functions to compute d positions for each item. The inserted item is stored in any one empty position among the d positions. If all the d positions are occupied, cuckoo hashing randomly evicts the item in one of d positions. The evicted item further searches the empty position in its d positions. Cuckoo hashing has higher space utilization than 2-choice hashing due to evictions, and achieves constant lookup time, i.e., probing d positions. However, the frequent evictions for inserting an item usually result in high insertion latency and possible endless loop [25], [39], [40]. In practice, $d = 2$ is most often used due to sufficient flexibility when using only two hash functions [20], [41].

2.3 The Influence of Existing Hashing Schemes on NVMs

Designing hashing schemes on traditional memory technologies, i.e., DRAM and SRAM, mainly considers two performance parameters, including space utilization and request latency [25]. When designing hashing schemes on NVMs, the third important parameter, i.e., the number of NVM writes, should be also considered due to the intrinsic asymmetric properties and the limited write endurance of NVMs. We analyze the influence of existing hashing schemes on the number of NVM writes.

In the **chained hashing**, when inserting and deleting an item in the linked list, besides changing the item itself, the pointers of other items also need to change, which results in extra writes to NVMs. For example, as shown in Figure 1(a), when deleting the item d, the pointer of a should point to e.

In the **linear probing**, when removing an item, the following multiple items move forward, which results in multiple NVM writes. In Figure 1(b), a, d, and e have the same hashing positions, i.e., $h(a) = h(d) = h(e) = 1$. When deleting item a, d moves to the position 1 and e moves to the position 2.

In the **Robin Hood hashing**, inserting a new item may displace multiple items, which results in multiple NVM writes. For example, as shown in Figure 1(c), when inserting a new item x whose hash position is 1 into the hash table, Robin Hood hashing probes the items in the position 1 and following positions until finding an item b whose PSL is smaller than that of x. Robin Hood hashing displaces b with a and further inserts b. Moreover, the deletion operation in the Robin Hood hashing also needs to move multiple items, causing multiple NVM writes, like linear probing.

2-choice hashing does not cause extra NVM writes, due to only probing two locations for inserting/deleting an item as shown in Figure 1(d).

In the **cuckoo hashing**, during inserting an item, multiple items are evicted and rewritten to new positions. When the hash table has a high load factor, e.g., $> 50\%$, an insertion usually causes tens of eviction operations, which results in the corresponding number of NVM writes. As shown in Figure 1(e), when inserting the item x, both hashing positions, i.e., 1 and 4, are occupied. Cuckoo

TABLE 1
Comparisons of different hashing schemes

Hashing Schemes	Extra Writes	Load Factor	Cache Utilization
Chained Hashing	Yes	High	Low
Linear Probing	Yes	High	High
Robin Hood Hashing	Yes	High	High
2-choice Hashing	No	Low	Low
Cuckoo Hashing	Yes	Medium	Low
Path Hashing (PH)	No	High	Low
Cache-optimized PH	No	High	High

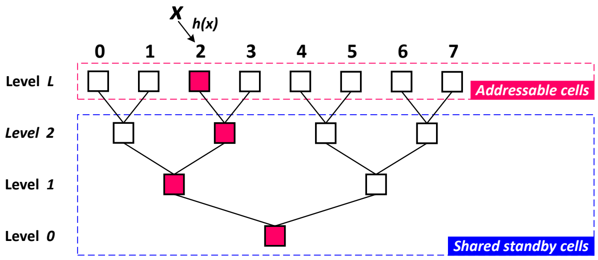


Fig. 2. An illustration of path hashing architecture with L = 3.

hashing randomly evicts the item in one of the two positions, e.g., b. b further searches its another hashing position, 7. If the position 7 is also occupied, the data item in 7 will be evicted. The eviction operations may be endless, called endless loop. If endless loop occurs, the size of hash table needs to be extended and all stored items are rehashed. We also evaluate these hashing schemes in experimental evaluation in terms of the number of NVM writes as shown in Section 5.2.1.

Table 1 shows a high-level comparison among existing hashing schemes and path hashing in terms of extra NVM writes, load factor and CPU cache utilization. In summary, most existing hashing schemes incur extra writes which are not friendly to the NVM write performance and endurance. Even though not causing extra NVM writes, 2-choice hashing has extremely low space utilization as evaluated in Section 5.2.2, since only two positions for an item are used to deal with hash collisions. The space utilization (i.e., the achieved maximum load factor) is a very important parameter especially in the context of space-limited NVM caches and main memory. Hence, it is important for designing a hashing scheme to minimize the NVM writes while ensuring the high performance in terms of space utilization and request latency.

3 The Design of Path Hashing

In this section, we present the path hashing, which leverages position sharing technique to deal with hash collisions without extra NVM writes, and double-path hashing and path shortening techniques to deliver high performance in terms of space utilization and request latency.

Path hashing leverages *position sharing* to allocate several standby cells for each addressable cell in the hash table to deal with hash collisions. The addressable cells in the hash table are addressable by the hash functions and the standby cells are not addressable. When the hash collisions occur in an addressable cell in the hash table, the conflicting items can be stored in its standby cells. An insertion/deletion request only needs to search an addressable cell in the hash table and its standby cells for an empty position or the target item, without extra writes. The standby cells of each addressable cell in the hash table are shared by other addressable cells, which prevents uneven hashing to produce lots of empty standby cells, thus improving the space utilization. An addressable cell and all its standby cells are likely to be occupied, which results in insertion failure. Path hashing leverages *double-path hashing* to compute two addressable cells for each item by using two hash functions, which further alleviates hash collisions and improves space utilization. Moreover, a read request needs to probe multiple standby cells in two paths to find the target item. *Path shortening* is proposed to reduce the number of probed cells in a request. We present the physical storage structure of path hashing in Section 3.4, which allows that all nodes in a read path can be accessed in parallel with the constant-scale time complexity.

3.1 Position Sharing

Path hashing leverages a novel collision-resolution scheme to deal with hash collisions, i.e., *position sharing*. Storage cells in the path hashing is logically organized as an inverted complete binary tree. As shown in Figure 2, the binary tree has $L+1$ levels ranging from the root level 0 to the leaf level L. Only the leaf nodes in the level L can be addressable by the hash functions, i.e., **addressable cells**. The nodes in the remaining levels ranging from level 0 to level $L-1$ are the shared standby positions to deal with hash collisions, i.e., **standby cells**. When an item is inserted into an occupied leaf node ℓ, path hashing searches for an empty standby cell in the path-ℓ. **Path-ℓ** is defined as the path descending from the occupied leaf node ℓ to the root.

For example, as shown in Figure 2, a new item x is hashed in the position of the leaf node 2. If the leaf node 2 is occupied, path hashing scans the path-2 from the leaf node 2 to the root for an empty position. Path hashing leverages the overlaps among different paths to share the standby positions in the level 0 to level $L-1$. As shown in Figure 2, each node in the level 2 is shared by two leaf nodes to deal with hash collisions. Each node in level 1 is shared by four leaf nodes. The root node is shared by all leaf nodes.

Insertion and deletion requests in path hashing only need to read the nodes in a path for finding an empty position or the target item, which hence do not cause any extra writes. The nodes in the levels from 0 to $L-1$ are shared to deal with hash collisions, which prevents uneven hashing to produce lots of empty standby cells, thus improving the space utilization.

3.2 Double-path Hashing

Since a path in the binary tree only has $L+1$ positions, the use of one path can only deal with at most L hash collisions in an addressable position. The hashing scheme using one path fails when more than $L+1$ items are hashed into the same position. To address this problem, we propose the *double-path hashing* which uses two hash functions for each item in the path hashing. Different from 2-choice hashing which seeks two cells for an item using two hash functions, double-path hashing seeks two paths.

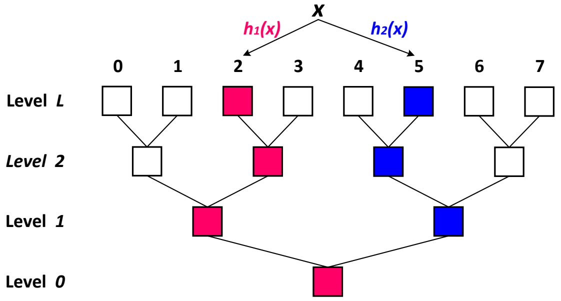


Fig. 3. An illustration of path hashing (L = 3) with two hash functions.

As shown in Figure 3, a new item x has two hashing positions, i.e., 2 and 5, computed by two different hash functions, $h_1(x)$ and $h_2(x)$. The item x is inserted into an empty position between the leaf nodes 2 and 5. If both nodes are occupied, the path hashing simultaneously scans the path-2 and path-5 to find an empty position. It is important that the two hash functions should be independent and not related with each other.

To avoid the overlap of the two paths for a key, we keep its first path (path-ℓ_1) in the first half part of the path hashing and its second path (path-ℓ_2) in the second half part, via the following computation:

$$\begin{aligned} \ell_1 &= h_1(x)\%2^{L-1} \\ \ell_2 &= h_2(x)\%2^{L-1} + 2^{L-1} \end{aligned} \tag{1}$$

Based on the position sharing, double-path hashing can further alleviate the hash collisions via providing more available positions for conflicting items. Moreover, due to the randomization of two independent hash functions, the two paths for an inserted item have no empty position with a low probability, which enables path hashing to maintain a high space utilization, as evaluated in Section 5.2.2.

When there is no empty position in the corresponding two paths for an inserted item, an insertion failure occurs. In that case, the hash table should be resized. We create a new hash table whose size is double of the old one, and then iteratively rehash all key-value items in the old hash table into the new one, like the resizing schemes in existing work [42], [43], [44].

3.3 Path Shortening

For the path hashing with $L+1$ levels, each query request needs to probe two paths with $L+1$ nodes. We observe that the nodes in the bottom levels of the inverted binary tree provide a few standby positions to deal with hash collisions, while increasing the length of the read path. For example, the level 0 only contains 1 position but adds by 1 in the length of the read path, as shown in Figure 4.

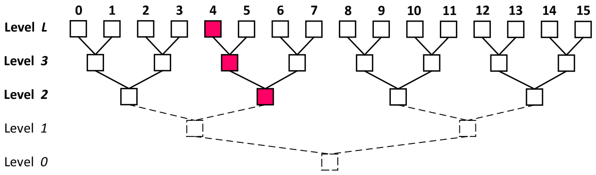


Fig. 4. An illustration of path hashing (L = 4) with path shortening.

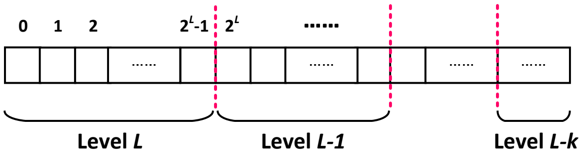


Fig. 5. Physical storage structure of path hashing.

To reduce the length of the read path, we propose the *path shortening* to reduce the number of read nodes in a read path. Path shortening removes multiple levels in the bottom of the inverted binary tree and only reserves several top levels. For a query request which is hashed in the leaf ℓ, path hashing only reads the nodes in the reserved levels in the path-ℓ, which reduces the length of the read path. As shown in Figure 4, the levels 0 and 1 are removed. The levels 2, 3, and 4, are reserved levels. When reading a path, e.g., path-4, we only read the nodes in the reserved levels 2, 3, and 4 in the path. Removing the bottom levels reduces the length of paths, which however reduces the number of positions to deal with hash collisions, thus decreasing the space utilization of hash table. We investigate the influence of the number of reserved levels on space utilization as shown in Section 5.2.3, and observe that reserving a small part of levels can also achieve a high space utilization in path hashing.

3.4 Physical Storage Structure

Even though storage cells in path hashing are logically organized as a binary tree, the physical storage structure of path hashing does not require pointers. Unlike the traditional binary tree built via pointers, the path hashing can be stored in a flat-addressed one-dimensional structure, e.g., a one-dimensional array. Figure 5 shows the physical storage structure of the path hashing with $L+1$ levels and $k+1$ reserved levels ($k \leq L$). The leaf nodes in the level L are stored in the first 2^L positions in the one-dimensional structure. The level $L-1$ is stored in the following 2^{L-1} positions, and so on. The removed bottom levels by path shortening do not need to be stored. In the storage structure, given a leaf node ℓ, it is easy to find all nodes of the path-ℓ in the one-dimensional structure, as described in Algorithm 1.

The flat-addressed storage structure allows all nodes in a path to be read in parallel from NVMs since the node accesses are independent to each other, which has the constant-scale time complexity. The node access pattern

Algorithm 1 Computing the storage locations of all nodes in path-ℓ

Input: The number of levels $L+1$, the number of reserved levels $k+1$, and the leaf node ℓ (stored in the ℓ-th position in the one-dimensional structure);

Output: The storage locations of all nodes in path-ℓ: $P[\,]$

```
$P[0] = \ell$
for ($i = 1; i < k + 1; i++$) do
    $\ell = \lfloor \frac{\ell}{2} \rfloor$
    $P[i] = \ell + 2^{L+1} - 2^{L-i+1}$
return $P[\,]$
```

Algorithm 2 Insert(*Key, Value*)

```
$\ell_1$ = $h_1$(Key)%$2^{L-1}$
$\ell_2$ = $h_2$(Key)%$2^{L-1} + 2^{L-1}$
for ($i = L$; $i > L - k - 1$; $i - -$) do
    if Path-$\ell_1(i)$ != NULL then
        Insert <Key, Value> in Path-$\ell_1(i)$
        Return TRUE
    if Path-$\ell_2(i)$ != NULL then
        Insert <Key, Value> in Path-$\ell_2(i)$
        Return TRUE
Return FALSE
```

of path hashing is different from that of chained hashing in which the nodes in a chain can only be sequentially accessed. For example, as shown in Figure 1, for a query request to position 2, the chained hashing first reads a and then obtains the storage position of d according to the pointer stored in a. The storage position of e can be only obtained by the pointer stored in d. The structure of linked lists in the chained hashing results in low access performance.

3.5 Operations

For a path hashing with $L+1$ levels and $k+1$ reserved levels ($k \leq L$), its physical storage structure is a one-dimensional array with $2^{L+1} - 2^{L-k}$ cells. Each cell stores a *<key, value, token>*, where the *token* denotes whether the cell is empty or not, e.g., *'token == 0'* represents empty and *'token == 1'* represents non-empty. Path hashing determines whether a cell is empty by checking the value of its *token*. We present the practical operations in path hashing including insertion, query and deletion.

3.5.1 Insertion

For inserting a new item *<Key, Value>*, path hashing first computes two locations, ℓ_1 and ℓ_2, by using two different hash functions, $h_1()$ and $h_2()$, as shown in Algorithm 2. If an empty cell exists in the leaf nodes ℓ_1 and ℓ_2 in the level L, path hashing inserts the new item into the empty cell and changes the *token* of the cell to '1'. If both the two leaf nodes are non-empty, path hashing further iteratively checks the nodes of path-ℓ_1 and path-ℓ_2 in the next level until finding an empty cell. In Algorithm 2, *Path-ℓ(i)* denotes the node of path-ℓ in the level i, whose storage location in the one-dimensional array can be computed by Algorithm 1.

3.5.2 Query

In the query operation, path hashing first computes its two paths, path-ℓ_1 and path-ℓ_2, based on the key of the queried item. Path hashing then checks the nodes of two paths from the level L to level $L - k$ until finding the target item, as shown in Algorithm 3. If the item can not be found in the two paths, it means the item does not exist in the hash table.

3.5.3 Deletion

In the deletion operation, path hashing first queries the cell storing the item to be deleted, and then deletes the item from the cell by changing the *token* of the cell to '0', as shown in Algorithm 4.

Algorithm 3 Query(*Key*)

```
$\ell_1$ = $h_1$(Key)%$2^{L-1}$
$\ell_2$ = $h_2$(Key)%$2^{L-1} + 2^{L-1}$
for ($i = L$; $i > L - k - 1$; $i - -$) do
    if Path-$\ell_1(i)$ != NULL && Path-$\ell_1(i)$.key == Key then
        Return Path-$\ell_1(i)$.value
    if Path-$\ell_2(i)$ != NULL && Path-$\ell_2(i)$.key == Key then
        Return Path-$\ell_2(i)$.value
Return NULL
```

Algorithm 4 Delete(*Key*)

```
$\ell_1$ = $h_1$(Key)%$2^{L-1}$
$\ell_2$ = $h_2$(Key)%$2^{L-1} + 2^{L-1}$
for ($i = L$; $i > L - k - 1$; $i - -$) do
    if Path-$\ell_1(i)$ != NULL && Path-$\ell_1(i)$.key == Key then
        Delete the item in Path-$\ell_1(i)$
        Return TRUE
    if Path-$\ell_2(i)$ != NULL && Path-$\ell_2(i)$.key == Key then
        Delete the item in Path-$\ell_2(i)$
        Return TRUE
Return FALSE
```

3.6 Time Complexity Analysis

For a path hashing with $L + 1$ levels and $k + 1$ reserved levels ($k \leq L$) that has a total of $N = 2^{L+1} - 2^{L-k}$ storage cells, we analyze the average access time complexity and the worst-case access time complexity respectively.

Average Access Time Complexity. When searching, inserting, or deleting a key-value item, path hashing traverses the corresponding two paths for finding the item from the top level to the bottom level. Hence, the access time is different when the item is stored in different levels. For example, when the item is stored in the level L, the access time is at most 2 (memory accesses) due to traversing two cells in the two paths; when the item is stored in the level $L - 1$, the access time is at most 4; when the item is stored in the level $L - 2$, the access time is at most 6; and so on. Hence, for an item stored in level i, the relationship between its access time (T_i) and its storage location (level i) is obtained:

$$T_i = 2(L + 1 - i) \tag{2}$$

Moreover, the probability that an item is stored in the level i is proportional to the number of cells in the level i. For example, the path hashing has a total of $N = 2^{L+1} - 2^{L-k}$ cells and the level L has 2^L cells. Thus the probability that an item is stored in the level L is $\frac{2^L}{N}$. Moreover, it is easy to obtain that the probability ($P(i)$) that an item is stored in the level i is:

$$P(i) = \frac{2^i}{N} = \frac{2^i}{2^{L+1} - 2^{L-k}} \tag{3}$$

From Equations 2 and 3, we obtain the expectation of the average access time (E_T) for searching, inserting, or deleting

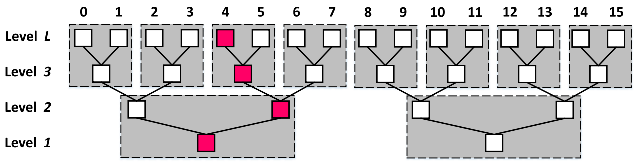


Fig. 6. An illustration of cache-optimized path hashing with $L = 4$, $k = 4$ and $m = 2$.

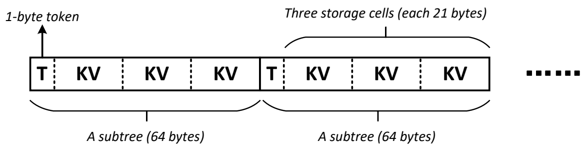


Fig. 7. The physical storage structure of cache-optimized path hashing with $m = 2$.

an item in the path hashing:

$$\begin{aligned} E_T &= \sum_{i=L-k}^{L} T_i P(i) \\ &= \sum_{i=L-k}^{L} 2(L+1-i)\frac{2^i}{N} \\ &= 4 - 2(k+1)\frac{2^{L-k}}{N} \\ &< 4 \end{aligned} \tag{4}$$

From Equation 4, we obtain the average access time complexity is $O(4) \sim O(1)$, i.e., the constant-level time complexity.

Worst-case Access Time Complexity. In the worst case for searching, inserting, or deleting an item, path hashing needs to traverse all cells in the corresponding two paths. Path hashing has $k+1$ levels and thus the two paths contain $2(k+1)$ cells where k is a constant. Hence, the worst-case access time complexity is $O(2(k+1))$, i.e., the constant-level time complexity.

4 Cache-optimized Path Hashing

In this section, we present the cache-optimized path hashing which aims to organize the storage cells of path hashing in the cache-optimized format.

Cache hierarchy is used in computer architectures to bridge the speed gap between CPU and main memory. Due to the high latency of main memory accesses, the CPU cache efficiency is important to the performance of main memory data structures [9], [45]. A cache is organized in cache lines. A cache line is the basic transferring unit between the last level cache and main memory, which is the same as a page in the disk system. The typical size of a cache line is 64 bytes. The CPU accesses a cache line that is available in the cache, called a cache hit. Otherwise, the cache line has to be loaded from main memory, called a cache miss, which is much more expensive than a cache hit in terms of access latency.

Current key-value stores, e.g., memcached and MemC3, are typically dominated by the small items whose sizes are smaller than a cache-line size [20], [46]. Most existing hashing schemes, such as chained hashing [28] and 2-choice hashing [47], suffer from low cache efficiency due to the randomization of hash functions and the low utilization of a cache line for such small items. The original path hashing also has the same problem. For example, in the original storage structure of path hashing as shown in Figure 5, when the CPU accesses the leaf cell '0' with a small item, the following leaf cell '1' or more cells are loaded to the cache due to belonging to the same cache line. However, only the data in leaf cell '0' is useful and the remaining leaf cells in the cache line are not used, which causes a low cache line utilization.

To improve the cache efficiency, we propose a cache-optimized path hashing to pack multiple cells in the same path together and store them in one cache line. To achieve this goal, cache-optimized path hashing divides the binary tree into many subtrees with m levels and then packs the cells in each subtree together and stores them in the contiguous memory space. Hence, a single memory access can prefetch multiple nodes belonging to the same path, which reduces the number of memory accesses to obtain higher performance.

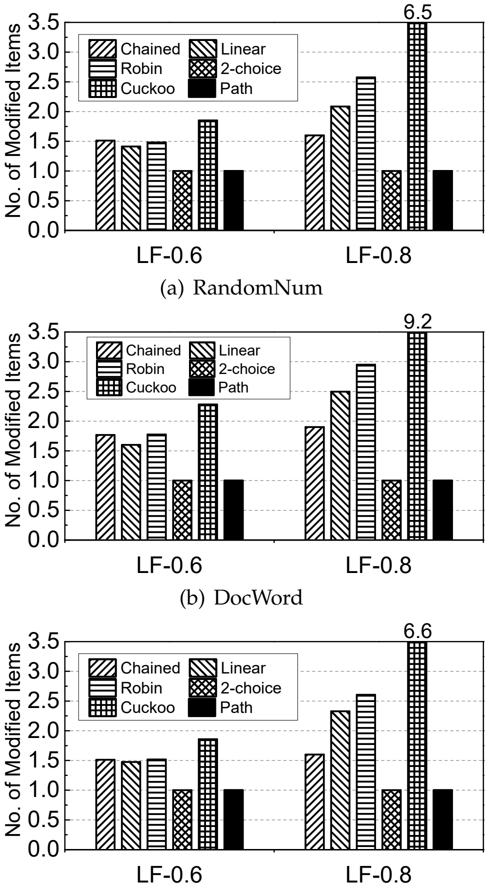
To better present cache-optimized path hashing, we give an illustration as shown in Figure 6. This is a path hashing with $L = 4$ in which $k = 4$ levels are reserved. We pack each subtree with $m = 2$ levels together. Three cells in each subtree are stored in the contiguous memory space and aligned to CPU cache line. Thus a single memory access can always load two nodes belonging to the same path which reduces the number of memory accesses. For example, a query request occurs in Path-4. When the leaf node 4 in Path-4 does not contain the target item, the node of Path-4 in Level 3 is accessed which produces a cache hit without being loaded from main memory, since the node has been prefeched into caches when reading the leaf node 4. The physical storage structure is shown in Figure 7. In a cache line, the first one byte is the *token* in which three bits are used to indicate whether the three cells are empty respectively. The remaining 63 bytes are divided into three equal parts and thus each cell can store up to 21-byte key-value items.

5 Performance Evaluation

In this section, we evaluate our proposed path hashing by being compared with existing hashing schemes in terms of the number of NVM writes, space utilization ratio, and request latency. We also compare path hashing with cache-optimized path hashing to show the benefits obtained from the cache-optimized scheme.

5.1 Experimental Configurations

Since real NVM devices are not available for us yet, we implement path hashing and existing hashing schemes in the gem5 [33] full-system simulator with NVMain [34] to evaluate their performance in the context of NVMs. NVMain is a timing-accurate main memory simulator for emerging non-volatile memory technologies. The configuration parameters of the system are shown in Table 2. The system has a three-level cache hierarchy. L1 cache is private and L2 cache is shared. L3 cache is DRAM, whose capacity is equally partitioned among all the cores [48]. The size of all cache lines is 64 bytes. Without loss of generality, we model

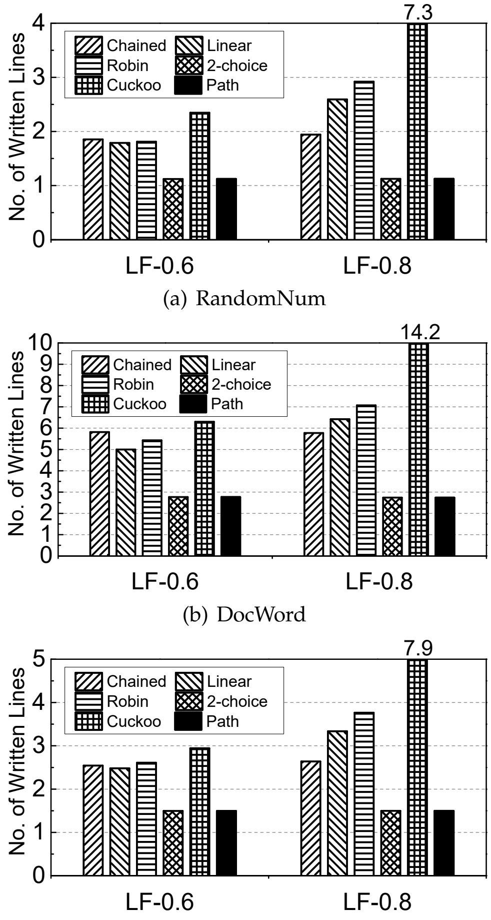


(a) RandomNum

(b) DocWord

(c) Fingerprint

Fig. 8. The normalized number of modified items. ("LF-0.6" and "LF-0.8" mean the load factor (LF) is 0.6 and 0.8 respectively.)



(a) RandomNum

(b) DocWord

(c) Fingerprint

Fig. 9. The average number of written cache lines for each request.

PCM technologies [49] as the main memory to evaluate path hashing that in fact can be also used in other NVMs. The read latency of the PCM is 75 ns and the write latency is 150 ns, like the configurations in [48], [50].

We compare path hashing with existing hashing schemes described in Section 2.2, i.e., chained hashing, linear probing, Robin Hood hashing, 2-choice hashing, and cuckoo hashing. We use three datasets including a random-number dataset and two real-world datasets as follows.

- **RandomNum.** We generate the random integer data ranging from $0 - 2^{26}$ via a pseudo-random number generator. We use the randomly generated numbers as the keys of the items in hash tables. The randomly generated integer is a commonly used dataset for evaluating the performance of hashing schemes [25], [41], [51].
- **DocWord.** The dataset consists of five text collections in the form of bags-of-words [52], in which we use the largest collection, *PubMed abstracts*, for evaluation. *PubMed abstracts* contains 8.2 million documents and about 730 million words in total. We use the combinations of document IDs and word IDs as the keys of the items in hash tables.
- **Fingerprint.** The dataset is extracted from MacOS [53], [54] which contains the daily snapshots of a Mac OS X server running in an academic computer lab collected by File system and Storage Lab at Stony Brook University. We use the MD5 fingerprints of data chunks in MacOS as the keys of the items in hash tables.

TABLE 2
Experimental Configurations

Processor and Cache	
CPU	4 cores, X86-64 processor, 2 GHz
Private L1 cache	32 KB , 2-way, LRU, 2-cycle latency
Shared L2 cache	4 MB, 8-way, LRU, 20-cycle latency
Shared L3 cache	32 MB, 8-way, LRU, 50-cycle latency
Memory Controller	FCFRFS
Main Memory using PCM	
Capacity	16 GB
Read latency	75 ns
Write latency	150 ns

5.2 The Comparisons among Path Hashing and Existing Hashing Schemes

5.2.1 *NVM Writes*

Only insertion and deletion requests cause the writes to NVMs. We first insert n items into a hash table and then delete $0.5n$ items from this hash table, using the five hashing schemes respectively. We use the hash tables with 2^{23} cells for random-number dataset, with 2^{24} cells for DocWord

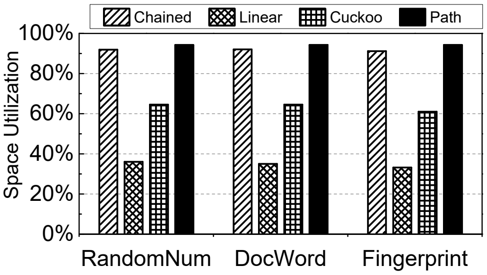


Fig. 10. Space utilization ratios of hashing schemes.

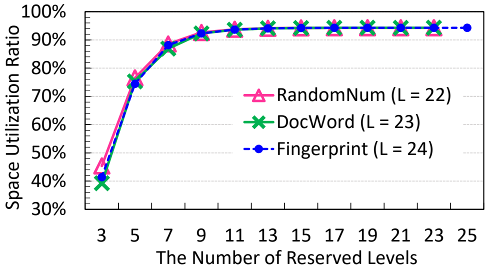


Fig. 11. The number of reserved levels vs. space utilization ratio.

dataset, and with 2^{25} cells for Fingerprint dataset. Load factor in hash table is defined as the ratio of the number of the inserted items to the total number of cells in hash table [25]. We evaluate the performance under two load factors, i.e., 0.6 and 0.8. The higher load factor naturally produces higher hash collision ratio. For 2-choice hashing and cuckoo hashing with a high load factor, many items fail to be inserted into the hash table due to their low space utilizations. We store these insertion-failure items in an extra stash, like ChunkStash [55], and continue to insert other items. The total number of modified items is normalized to the total number of requests (i.e., $1.5n$), as shown in Figure 8.

As shown in Figure 8, chained hashing, linear probing, Robin Hood hashing, and cuckoo hashing modify extra items which naturally incur more writes to NVMs. Higher load factor results in more extra modified items. Among these hashing schemes, cuckoo hashing needs to modify most items, due to frequently evicting and rewriting items during insertion. Linear probing moves many items to deal with deletion requests especially when the hash table is in a relatively high load factor, i.e., 0.8. Robin Hood hashing incurs more writes than linear probing since both its insertion and deletion requests need to move items. Chained hashing needs to modify the pointers of other items when inserting and deleting an item in the linked lists. To execute a deletion/insertion request, path hashing and 2-choice hashing only write the deleted/inserted item without modifying extra items, which are write-friendly for NVMs.

We also evaluate the average number of the written cache lines to NVMs for each request, as shown in Figure 9. The average number of written cache lines is approximatively proportional to the average number of modified items in each scheme. The average number of written cache lines in the DocWord dataset is much larger than those of the RandomNum and Fingerprint datasets due to the larger item size.

5.2.2 *Space Utilization*

Space utilization ratio is defined as the load factor of hash tables when insertion failure occurs. Higher space utilization ratio means that more items can be stored in a given-size hash table, which is a significant parameter in the context of main memory and the caches with limited space. For chained hashing, a half of memory is used for hash table positions, and a half for list positions [25]. If the chained hashing runs out of list positions, the insertion failure occurs. For cuckoo hashing and 2-choice hashing, we respectively allocate a stash with the 1% size of hash table, which is not large. Otherwise, linearly searching the items stored in the stash results in high latency. For cuckoo hashing, when the number of evictions for an item achieves 100 [51], we store the item into the stash. For 2-choice hashing, when both the two positions of an item are occupied, we store the items in the stash. When the space of the stash runs out, the insertion failure occurs. For path hashing, when all nodes in the two paths for an item are occupied, the insertion failure occurs. Their space utilization ratios are shown in Figure 10.

As shown in Figure 10, 2-choice hashing has extremely low space utilization ratio since only two positions for an item are used to deal with hash collisions, which are easily occupied by other items. Cuckoo hashing obtains higher space utilization ratio than 2-choice hashing, due to further evicting one of items in the occupied positions when both positions of an item are occupied. Linear probing and Robin Hood hashing are not shown in the figure, since they do not have a fixed space utilization ratio. Their load factors can be up to 1, while the query performance is close to that of the linear list. Chained hashing has a high space utilization ratio since the conflicting items can always link with the lists until running out of list positions. The space utilization ratio of our proposed path hashing achieves about 95% in the three datasets, which is more than that of chained hashing, due to efficiently dealing with hash collisions via position sharing and double-path hashing.

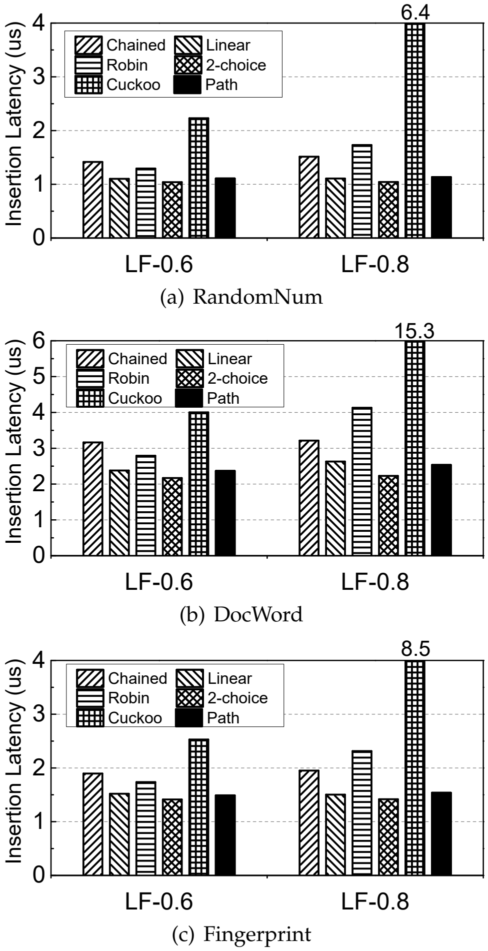
5.2.3 *The Number of Reserved Levels vs. Space Utilization*

As described in Section 3.3, we remove multiple levels in the bottom of path hashing to reduce the length of the read path. However, removing the bottom levels also reduces the number of positions to deal with hash collisions, thus reducing the space utilization ratio. We hence investigate the relationship between the number of the reserved levels and space utilization ratio of path hashing.

As shown in Figure 11, we observe that reserving a small part of levels can also achieve a high space utilization ratio in path hashing. For example, reserving 9 levels achieves over 92% space utilization ratio for a binary tree with 25 levels in the Fingerprint dataset. Reserving 11 levels can achieve the space utilization ratio close to that of a full binary tree.

5.2.4 *Insertion Latency*

We insert the same number of items in the five kinds of hash tables and store the insertion-failure items in the stashes for 2-choice hashing and cuckoo hashing. We compare the average insertion latency of different hashing schemes, as shown in Figure 12. Cuckoo hashing has the



(a) RandomNum

(b) DocWord

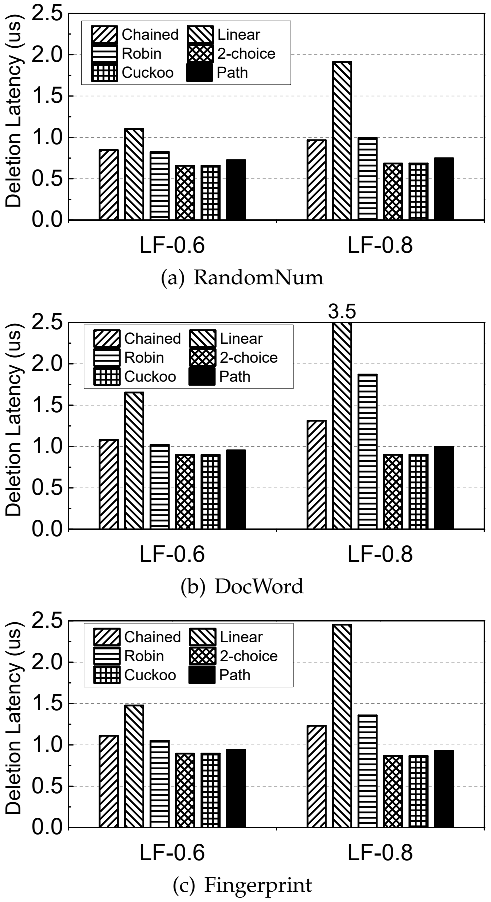
(c) Fingerprint

Fig. 12. Average latency of inserting an item.

highest insertion latency, due to frequently evicting and rewriting items. Its insertion latency dramatically increases with increasing the load factor from 0.6 to 0.8, since the higher hash collision ratio causes much more evictions. Robin Hood hashing also displaces and rewrites multiple items during insertions which results in high insertion latency. Chained hashing incurs high insertion latency due to modifying extra items during insertion. 2-choice hashing has the lowest insertion latency due to only probing two positions for each insertion. Path hashing and linear hashing have the low latency close to 2-choice hashing, due to only probing empty positions for insertion.

5.2.5 Deletion Latency

We compare the average deletion latency of different hashing schemes, as shown in Figure 13. We observe that linear probing has the highest deletion latency due to moving multiple items when deleting an item, which dramatically increases with the growing load factor of hash tables. Robin Hood hashing has much lower deletion latency than linear probing even though also needing to move items during deletions. It is because Robin Hood hashing reduces the average probe sequence length of each item during insertions, which results in less items to be moved during deletions. Chained hashing incurs high deletion latency due to traversing the linked lists and modifying other items. 2-choice hashing and cuckoo hashing have the low deletion latency due to only probing two positions. Path hashing has the slightly higher latency than 2-choice hashing due to probing multiple positions in several levels. Note that for cuckoo hashing and 2-choice hashing, we do not evaluate their delete/query latency to the stash due to only focusing on the delete/query latency to hash table.



(a) RandomNum

(b) DocWord

(c) Fingerprint

Fig. 13. Average latency of deleting an item.

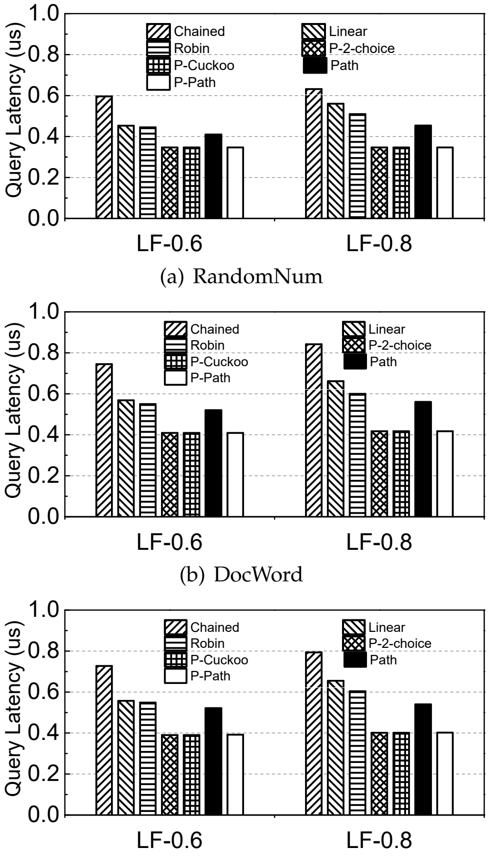
5.2.6 Query Latency

Cuckoo hashing and 2-choice hashing require two memory accesses for querying an item. The two memory accesses are independent and can be executed in parallel. To evaluate their parallel query performance, i.e., P-Cuckoo and P-2-choice, we only evaluate the second hash query after the first hash query fails, like the method in [25]. For path hashing, the node accesses in the read paths are also independent and can be executed in parallel as descried in Section 3.4. We also evaluate the parallel query performance of path hashing, i.e., P-Path.

We compare the average query latency of different hashing schemes, as shown in Figure 14. We observe that chained hashing causes the highest query latency, due to serially accessing the long linked lists which results in multiple random memory accesses. Comparing the results of the load factors 0.6 and 0.8, higher load factor results in longer linked lists in chained hashing, thus causing higher query latency. Linear probing has high query latency due to scanning the successive table cells, which increases with the growing of the load factor. Robin Hood hashing has lower query latency than linear probing due to reducing

TABLE 3
The number of storage cells in a path hash table with $L = 21$

Levels in path hashing	The total number of storage cells	Percentage
The top 1-4 levels	3,932,160	100%
The top 5-8 levels	245,760	6.25%
The top 9-12 levels	15,360	0.39%
The top 13-16 levels	960	0.0244%



(a) RandomNum

(b) DocWord

(c) Fingerprint

Fig. 14. Average latency of querying an item.

the average probe sequence length of each item during insertions. We observe that P-Cuckoo and P-2-choice have the lowest query latency due to the constant-scale time complexity when executed in parallel. Path hashing without parallelism has the higher query latency than cuckoo hashing due to probing multiple nodes in the read paths, while being still lower than those of linear probing, chained hashing, and Robin Hood hashing. Parallel path hashing (P-Path) has the approximate query latency as P-Cuckoo and P-2-choice.

5.3 The Comparisons between Path Hashing and Cache-optimized Path Hashing

In this subsection, we compare the original path hashing and cache-optimized path hashing in terms of request latency. In the evaluation, we use the RandomNum dataset. Each key-value item is 21 bytes in which the key is 10-byte random integer and the value is 11 bytes. Thus cache-optimized path hashing packs each subtree with 2 levels together as the illustration described in Section 4 and Figure 6. The top level of the hash table used in the evaluation has 2^{21} cells ($L = 21$). We also investigate the influence of the number of the reserved levels on the latency of different kinds of requests.

5.3.1 *Query Latency*

To show the benefits of cache-optimized scheme, we evaluate the performance of both positive and negative queries. For a query, if the target item is found in the hash table, the query is positive. Otherwise, it is negative. For a positive query, path hashing probes the two pathes from the top level to bottom level and stops the probing once finding the target item. For a negative query, path hashing has to go through the two paths before finding the target item not existing in the hash table. The latency of negative query indicates the worst-case search time in path hashing. We first insert items in the hash tables until reaching the maximal load factor, and then respectively perform 1 million positive queries and 1 million negative queries to evaluate the average latency of each query. We also evaluate the query latency when reserving the different numbers of levels in path hashing. The results are shown in Figures 15 and 16.

Figure 15 shows the average latency of negative query. We observe that cache-optimized path hashing reduces the query latency by 43%-49% compared with the original path hashing. The reason is that cache-optimized path hashing packs two cells in the same path into a cache line. When the first cell is loaded into cache, the second cell is prefetched, which reduces half of memory accesses. We also observe that the average latency of negative query increases with the increase of the number of reserved levels for path hashing. It is because negative query needs to probe all cells in the two paths whose lengths are increased when increasing the number of reserved levels.

Figure 16 shows the average latency of positive query. We observe that cache-optimized path hashing reduces the query latency by about 23% compared with the original

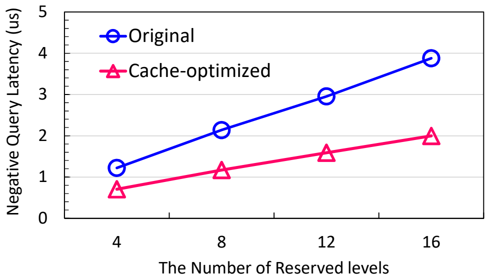


Fig. 15. The average latency of negative query. (“Original” indicates the original path hashing without the cache-optimized scheme. “Cache-optimized” indicates the cache-optimized path hashing.)

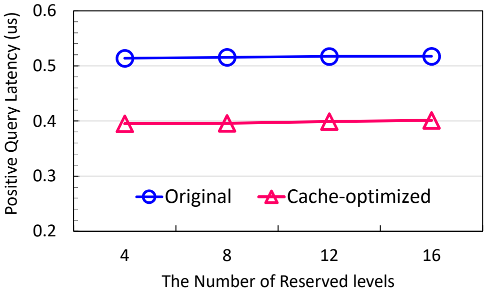


Fig. 16. The average latency of positive query.

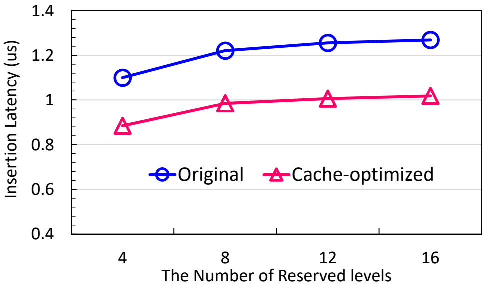


Fig. 17. The average insertion latency.

path hashing. There are two differences between the experimental results of positive query and negative query.

First, cache-optimized scheme reduces about 23% positive query latency while reducing near 50% negative query latency. The reason is that cache-optimized scheme reduces the latency only when the target item exists in the second-level cell in a subtree with 2 levels for a positive query. In this case, to find the target item, cache-optimized path hashing loads only one cache line while the original path hashing needs to load two cache lines. For a negative query, both the original and cache-optimized path hashing need to access the two cells in the same subtree. Thus the cache-optimized path hashing always reduces half of memory accesses for a negative query.

Second, the negative query latency increases with the increase of the number of reserved levels, while the positive query latency does not change. We have presented the reason for the negative query above. We below analyze the reason why the positive query latency does not change. In the path hashing, most of storage cells are located in the upper levels. For example, the total number of storage cells in the top 5-8 levels is only 6.25% of that in the top 1-4 levels, as shown in Table 3. Thus for one million queried items in the positive query test, most of them can be found in the upper levels and only a few items are in the bottom levels. Hence, the average latency of positive query is nearly identical when path hashing has the different numbers of reserved levels.

5.3.2 Insertion Latency

To compare their insertion latency, we insert key-value items into the path hash tables with the different number of reserved levels until reaching their maximum load factors. The number of inserted items and maximum load factor are shown in Table 4. The average insertion latency is shown in Figure 17. We can observe that cache-optimized path hashing reduces the insertion latency by about 20% compared with the original path hashing. The reason is that when accessing a cell in a path, cache-optimized path hashing prefecthes the next cell in the same path. When the first cell is not empty, cache-optimized path hashing accesses the next cell without being loaded from main memory, thus reducing the insertion latency. With the increase of the number of reserved levels, the average insertion query does not significantly increase, since most of items are inserted in the upper levels.

5.3.3 Deletion Latency

To compare their deletion latency, we first insert key-value items into the path hash tables until reaching their maximum load factors. We then remove one million items existing in the hash tables. The average deletion latency is shown in Figure 18. We can observe that cache-optimized path hashing reduces the deletion latency by about 23% compared with the original path hashing. The deletion latency is nearly identical to the positive query latency shown in Figure 16, since the deletion operation is similar to the query operation, which first queries the location of the cell storing the target item and then removes the item by setting the token of the cell to '0'.

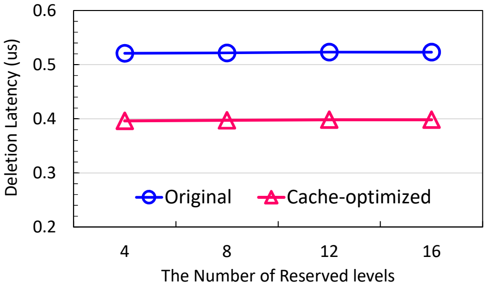


Fig. 18. The average deletion latency.

6 Related Work

As emerging NVMs become promising to play an important role in the memory hierarchy, e.g., main memory and caches. The changes of the memory characteristics bring the challenges to the in-memory or in-cache data structure design. In order to efficiently adapt to the new memory characteristics and support hardware-software co-design in memory systems, data structures have been improved to enhance the endurance and performance of NVM systems.

Existing work has mainly focused on the tree-based data structures stored in NVMs. Chen et al. [8] propose unsorted-node schemes to improve B^+-tree algorithm for PCM. They show that the use of unsorted nodes, instead of sorted nodes in B^+-tree, can reduce PCM writes. Chi et al. [10] observe that using unsorted nodes in B^+-tree suffers from several problems, e.g., CPU-costly for insertion and wasting space for deletion. They further improve B^+-tree algorithm for NVMs via three techniques including the sub-balanced unsorted node, overflow node, and merging factor schemes. CDDS B-Tree [2] and NV-Tree [9] aim to reduce the consistency cost of B^+-tree when maintained in NVMs. Chen et al. [11] propose wB^+-tree to minimize the movement of index entries from insertion and deletion requests by achieving write atomicity, thus reducing the extra NVM writes. Oukid et al. [56] consider B^+-tree in a

TABLE 4
The number of inserted items in a path hash table with $L = 21$

Path hashing	No. of storage cells	No. of inserted items	Max. load factor
Reserved 4 levels	3,932,160	2,764,308	0.703
Reserved 8 levels	4,177,920	3,856,220	0.923
Reserved 12 levels	4,193,280	3,971,036	0.947
Reserved 16 levels	4,194,240	3,988,722	0.951

hybrid NVM-DRAM main memory and propose the FP-tree, in which the leaf nodes of B^+-tree are persisted in NVM while the inner nodes are stored in DRAM to deliver high performance. Lee et al. [57] focus on the radix tree data structure and analyze the limitations of the radix tree for NVMs. They then propose the WORT (Write Optimal Radix Tree) to eliminate the duplicate-copy writes for logging or copy-on-write in the radix tree.

Hashing-based data structures are also popular and widely used to construct the index and lookup table in main memory (e.g., main memory databases) [15], [24], [41] and caches [38], [58]. Debnath et al. [59] propose a PCM-friendly cuckoo hashing variant called PFHB which modifies cuckoo hashing to allow the eviction operations at most once and uses an extra stash to store the insertion-failure items. PFHB reduces the writes of cuckoo hashing to PCM at the expense of significantly reducing the lookup performance. Our work investigates the influence of hashing-based data structures on the writes to NVMs, and proposes a write-friendly hashing scheme, path hashing, which allows insertion and deletion requests of hash table do not cause any extra writes to NVMs while delivers high performance in terms of space utilization and request latency.

7 Conclusion and Future Work

In this paper, we propose a cost-efficient write-friendly hashing scheme, called path hashing, for NVMs to minimize the NVM writes while maintaining high performance of hash tables. Path hashing leverages position sharing technique to deal with hash collisions without extra NVM writes, and double-path hashing and path shortening techniques to deliver high performance in terms of space utilization and request latency. To improve the cache efficiency of path hashing, we propose a cache-optimized path hashing to pack multiple cells in the same path together and store them into one cache line, which reduces the number of memory accesses to obtain higher performance. We have implemented path hashing and evaluated it in gem5 with NVMain using a random-number dataset and two real-world datasets. Extensive experimental results show that path hashing incurs no extra NVM writes, and achieves up to 95% space utilization ratio as well as low request latency, compared with existing state-of-the-art hashing schemes.

With the non-volatility property, NVMs are expected to persist data structures as persistent memory for instantaneous failure recovery. In that case, it is essential to guarantee the consistency of data structures on persistent memory. Our future work will consider to design persistent hashing-based data structures for persistent memory.

References

[1] Y. Xie, “Modeling, Architecture, and Applications for Emerging Memory Technologies,” *IEEE Design & Test of Computers*, vol. 28, no. 1, pp. 44–51, 2011.

[2] S. Venkataraman, N. Tolia, P. Ranganathan, R. H. Campbell *et al.*, “Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory.” in *Proc. USENIX FAST*, 2011.

[3] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, “Pinatubo: a processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories,” in *Proc. ACM DAC*, 2016.

[4] J. Xu and S. Swanson, “NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories,” in *Proc. USENIX FAST*, 2016.

[5] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, “Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling,” in *Proc. MICRO*, 2009.

[6] J. J. Yang, D. B. Strukov, and D. R. Stewart, “Memristive devices for computing,” *Nature nanotechnology*, vol. 8, no. 1, pp. 13–24, 2013.

[7] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, “Scalable high performance main memory system using phase-change memory technology,” *In Proc. ISCA*, 2009.

[8] S. Chen, P. B. Gibbons, and S. Nath, “Rethinking Database Algorithms for Phase Change Memory,” in *Proc. CIDR*, 2011.

[9] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, “NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems,” in *Proc. USENIX FAST*, 2015.

[10] P. Chi, W. C. Lee, and Y. Xie, “Adapting B+-Tree for Emerging Nov-volatile Memory Based Main Memory,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 35, no. 9, pp. 1461–1474, 2016.

[11] S. Chen and Q. Jin, “Persistent b+-trees in non-volatile main memory,” *Proceedings of the VLDB Endowment*, vol. 8, no. 7, pp. 786–797, 2015.

[12] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, “A durable and energy efficient main memory using phase change memory technology,” in *Proc. ISCA*, 2009.

[13] S. Schechter, G. H. Loh, K. Strauss, and D. Burger, “Use ECP, not ECC, for hard failures in resistive memories,” in *Proc. ISCA*, 2010.

[14] W. D. Maurer and T. G. Lewis, “Hash table methods,” *ACM Computing Surveys (CSUR)*, vol. 7, no. 1, pp. 5–19, 1975.

[15] H. Garcia-Molina and K. Salem, “Main Memory Database Systems: An Overview,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 4, no. 6, pp. 509–516, 1992.

[16] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, “Meet the walkers: Accelerating index traversals for in-memory databases,” in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.

[17] H. Lim, M. Kaminsky, and D. G. Andersen, “Cicada: Dependably fast multi-core in-memory transactions,” in *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*, 2017.

[18] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker, “Staring into the abyss: An evaluation of concurrency control with one thousand cores,” *Proceedings of the VLDB Endowment*, vol. 8, no. 3, pp. 209–220, 2014.

[19] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood, *Implementation techniques for main memory database systems*. ACM, 1984, vol. 14, no. 2.

[20] B. Fan, D. G. Andersen, and M. Kaminsky, “MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing.” in *Proc. USENIX NSDI*, 2013.

[21] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey, “Architecting to achieve a billion requests per second throughput on a single

key-value store server platform," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[22] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang, "Mega-KV: A case for GPUs to maximize the throughput of in-memory key-value stores," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1226–1237, 2015.

[23] "Redis," https://redis.io/.

[24] "Memcached," https://memcached.org/.

[25] R. Pagh and F. F. Rodler, "Cuckoo Hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[26] J. Yue and Y. Zhu, "Accelerating write by exploiting PCM asymmetries," *Proc. IEEE HPCA*, 2013.

[27] M. Ajtai, "The complexity of the pigeonhole principle," in *Proc. IEEE FOCS*, 1988.

[28] J. L. Carter and M. N. Wegman, "Universal Classes of Hash Functions," *Journal of Computer and System Sciences*, vol. 18, no. 2, pp. 143–154, 1979.

[29] M. Patrascu and M. Thorup, "On the k -Independence Required by Linear Probing and Minwise Independence," *Acm Transactions on Algorithms*, vol. 12, no. 1, pp. 715–726, 2016.

[30] B. Pittel, "Linear probing: the probable largest search time grows logarithmically with the number of records," *Journal of Algorithms*, vol. 8, no. 2, pp. 236–249, 1987.

[31] P. Celis, "Robin hood hashing," *PhD thesis, University of Waterloo*, 1986.

[32] P. Zuo and Y. Hua, "A write-friendly hashing scheme for non-volatile memory systems," in *Proc. MSST*, 2017.

[33] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[34] M. Poremba and Y. Xie, "Nvmain: An architectural-level main memory simulator for emerging non-volatile memories," in *IEEE Computer Society Annual Symposium on VLSI*, 2012.

[35] D. Liu, K. Zhong, T. Wang, Y. Wang, Z. Shao, E. H.-M. Sha, and J. Xue, "Durable address translation in pcm-based flash storage systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 2, pp. 475–490, 2017.

[36] H.-S. Chang, Y.-H. Chang, Y.-H. Kuan, X.-Z. Huang, T.-W. Kuo, and H.-P. Li, "Pattern-aware write-back strategy to minimize energy consumption of pcm-based storage systems," in *the 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, 2016.

[37] J. K. Kim, H. G. Lee, S. Choi, and K. I. Bahng, "A pram and nand flash hybrid architecture for high-performance embedded storage subsystems," in *Proceedings of the 8th ACM international conference on Embedded software*, 2008.

[38] J. S. Miguel, J. Albericio, A. Moshovos, and N. E. Jerger, "Doppelgänger: a cache for approximate computing," in *Proc. MICRO*, 2015.

[39] Y. Hua, H. Jiang, and D. Feng, "Fast: Near real-time searchable data analytics for the cloud," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.

[40] Y. Sun, Y. Hua, S. Jiang, Q. Li, S. Cao, and P. Zuo, "SmartCuckoo: A Fast and Cost-Efficient Hashing Index Scheme for Cloud Storage Systems," in *Proc. USENIX ATC*, 2017.

[41] A. D. Breslow, D. P. Zhang, J. L. Greathouse, N. Jayasena, and D. M. Tullsen, "Horton tables: fast hash tables for in-memory data-intensive computing," in *USENIX ATC*, 2016.

[42] N. Piggin, "ddds: "dynamic dynamic data structure" algorithm, for adaptive dcache hash table sizing. linux kernel mailing list," https://lwn.net/Articles/302132/.

[43] J. Triplett, P. E. McKenney, and J. Walpole, "Resizable, scalable, concurrent hash tables via relativistic programming." in *USENIX Annual Technical Conference (USENIX ATC)*, 2011.

[44] H. Gao, J. F. Groote, and W. H. Hesselink, "Almost wait-free resizable hashtables," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[45] J. Rao and K. A. Ross, "Making b+-trees cache conscious in main memory," in *Proc. ACM SIGMOD*, vol. 29, no. 2. ACM, 2000, pp. 475–486.

[46] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1. ACM, 2012, pp. 53–64.

[47] M. Mitzenmacher, A. W. Richa, and R. Sitaraman, "The Power of Two Random Choices: A Survey of Techniques and Results," *Handbook of Randomized Computing*, vol. 11, pp. 255–312, 2000.

[48] V. Young, P. J. Nair, and M. K. Qureshi, "DEUCE: Write-efficient encryption for non-volatile memories," in *Proc. ACM ASPLOS*, 2015.

[49] Y. Choi, I. Song, M.-H. Park, H. Chung, S. Chang, B. Cho, J. Kim, Y. Oh, D. Kwon, J. Sunwoo *et al.*, "A 20nm 1.8 v 8gb pram with 40mb/s program bandwidth," in *Proc. ISSCC*, 2012.

[50] A. Awad, P. Manadhata, S. Haber, Y. Solihin, and W. Horne, "Silent shredder: Zero-cost shredding for secure non-volatile main memory controllers," in *Proc. ACM ASPLOS*, 2016.

[51] Y. Sun, Y. Hua, D. Feng, L. Yang, P. Zuo, and S. Cao, "MinCounter: An efficient cuckoo hashing scheme for cloud storage systems," in *Proc. IEEE MSST*, 2015.

[52] "Bags-of-Words data set," http://archive.ics.uci.edu/ml/datasets /Bag+of+Words, 2008.

[53] V. Tarasov, A. Mudrankit, W. Buik, P. Shilane, G. Kuenning, and E. Zadok, "Generating realistic datasets for deduplication analysis," in *Proc. USENIX ATC*, 2012.

[54] Z. Sun, G. Kuenning, S. Mandal, P. Shilane, V. Tarasov, N. Xiao, and E. Zadok, "A long-term user-centric analysis of deduplication patterns," *Proc. IEEE MSST*, 2016.

[55] B. Debnath, S. Sengupta, and J. Li, "ChunkStash: speeding up inline storage deduplication using flash memory," in *Proc. USENIX ATC*, 2010.

[56] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, "FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory," in *Proc. SIGMOD*, 2016.

[57] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, "WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems," in *Proc. USENIX FAST*, 2017.

[58] Y. Tian, S. M. Khan, D. A. Jiménez, and G. H. Loh, "Last-level cache deduplication," in *Proceedings of the 28th ACM international conference on Supercomputing (ICS)*, 2014.

[59] B. Debnath, A. Haghdoost, A. Kadav, M. G. Khatib, and C. Ungureanu, "Revisiting hash table design for phase change memory," in *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads (INFLOW)*, 2015.

Pengfei Zuo received the BE degree in computer science and technology from Huazhong University of Science and Technology (HUST), China, in 2014. He is currently a PhD student majoring in computer science and technology at HUST. His current research interests include data deduplication, non-volatile memory, and key-value store. He publishes several papers in major conferences including USENIX ATC, SoCC, ICDCS, MSST, DATE, etc. He is a student member of IEEE.

Yu Hua received the BE and PhD degrees in computer science from the Wuhan University, China, in 2001 and 2005, respectively. He is a full professor at the Huazhong University of Science and Technology, China. His research interests include computer architecture, cloud computing, and network storage. He has more than 100 papers to his credit in major journals and international conferences including IEEE Transactions on Computers (TC), IEEE Transactions on Parallel and Distributed Systems (TPDS), USENIX ATC, USENIX FAST, INFOCOM, SC, ICDCS and MSST. He has been on the program committees of multiple international conferences, including USENIX ATC, RTSS, INFOCOM, ICDCS, MSST, ICNP and IPDPS. He is a senior member of the IEEE, ACM and CCF, and a member of USENIX.