Cluster Computing (2019) 22:361–371
https://doi.org/10.1007/s10586-019-02925-1

Design and implementation of skiplist-based key-value store on non-volatile memory

Qichen Chen[1] · Hyojeong Lee[1] · Yoonhee Kim[2] · Heon Young Yeom[1] · Yongseok Son[3]

Received: 2 January 2019 / Accepted: 9 March 2019 / Published online: 22 March 2019

Abstract
Non-volatile random access memory (NVRAM) is a promising approach to persistent data storage with outstanding advantages over traditional storage devices, such as hard disk drives (HDDs) and solid state drives (SSDs). Some of its biggest advantages are its DRAM-like read latency and microsecond-level write latency, which are several hundred times faster than those in the original block device. However, one of the issues with using NVRAM as a storage device is designing an indexing system for its data stores to fully utilize NVRAM characteristics. The state-of-the-art indexing systems of non-volatile key-value stores are usually based on B+-trees or their variants, which were originally designed for block-based storage devices with better sequential performance than random performance. The semantics of B+-tree require data being sorted into leaf nodes and inner nodes and frequent splitting and merging to keep balanced. However, all the sorting, splitting, and merging operations cause extra write to NVRAM, which decreases its performance. In this article, we propose NV-Skiplist, a skiplist-based indexing system for key-value stores on NVRAM that fully uses the features of both NVRAM and DRAM. NV-Skiplist constructs its bottom layer in non-volatile memory to maintain data persistence and support range scans. It builds its upper layers in DRAM to retain rapid index searching and prevent consistently large overhead. We also propose a multiranged variant of NV-Skiplist to increase its search performance and scalability. We evaluate the performance of NV-Skiplist and wB+-tree which is a state-of-art scheme on an NVRAM emulator on a server with an Intel Xeon E5-2620 v2 processor. The results show that our design outperforms the original tree-based, non-volatile key-value stores up to 48%.

Keywords NVRAM · Indexing · Skiplist · Key-value · Memory

1 Introduction

The rise of next-generation non-volatile random access memory (NVRAM), including phase-change memory [2], memristors [3], and spin-transfer torque magnetic random access memory [4], has drawn great attention from storage system researchers [5–8]. NVRAM provides different characteristics such as low read and write latencies, the non-volatility properties, and byte-addressable access, demonstrating their potential to replace both DRAM and block storage devices. As a result of these features, key-value stores have recently been re-designed for NVRAM, benefiting from its fast access and persistent features.

A preliminary version [1] of this article was presented at the 3rd IEEE International Workshops on Foundations and Applications of Self* Systems, Trento, Italy, Sep. 2018.

✉ Yongseok Son
sysganda@cau.ac.kr

Qichen Chen
charliecqc@dcslab.snu.ac.kr

Hyojeong Lee
hjlee@dcslab.snu.ac.kr

Yoonhee Kim
yulan@sookmyung.ac.kr

Heon Young Yeom
yeom@dcslab.snu.ac.kr

1 Department of Computer Science and Engineering, Seoul National University, Seoul, South Korea

2 Department of Computer Science, Sookmyung Women's University, Seoul, South Korea

3 School of Computer Science and Engineering, Chung-Ang University, Seoul, South Korea

However, the majority of the latest NVRAM key-value stores derive from the conventional persistent key-value stores that generally store data in block storage devices. Due to the performance gap between random access and sequential access on block storage devices, B+-tree-based indexing systems were introduced [9]. Since the semantics of a B+-tree aggregate adjacent data into the same group with a fixed order, random performance could be improved by pre-fetching adjacent data. In spite of these advantages, B+-tree-based indexing systems are not naturally suited to key-value stores that run on NVRAM. Since there is almost no observable random or sequential performance gap, merged sequential access cannot be benefit; additionally, the cost of keeping data organized would introduce extra data write that decreases overall performance.

Previous several studies have optimized key-value stores on NVRAM by re-designing the existing B+ tree-based systems, such as CDDS B+-tree [7], wB+-tree [5], and NV-tree [8] to reduce persistent costs. However, these systems inherited the nature of B+-trees, meaning that maintaining data orders and merging/splitting operations are still necessary, which may cause additional write amplification or extra cost to maintain the semantics when running on NVRAM. Our study is in line with these studies [5, 7, 8] in terms of optimizing key-value store on NVRAM. In contrast, we focus on investigating the effectiveness of skiplist on NVRAM instead of tree structures.

In this article, we propose a skiplist-based key-value store design called NV-Skiplist instead of a traditional B+-tree. NV-Skiplist runs on both NVRAM and DRAM and aims to exploit the simplicity of skiplist insertion while maintaining a high search performance due to its $O(\log N)$ search time complexity. NV-Skiplist stores keys in the leaf node out of order to reduce the overhead of extra write to NVRAM. To satisfy the semantics of a skiplist, corresponding index nodes are stored and sorted by their ranges in DRAM. For further reducing writes to NVRAM, we use the minimum and maximum keys to represent the keys stored in corresponding leaf node. The original skiplist is balanced by consulting random number generators [10] and leads to simpler algorithms. On this basis, NV-Skiplist chooses a deterministic algorithm [11] to keep itself balanced while achieving a better search performance. In addition, NV-Skiplist also applies multi-range algorithm, which distributes the whole key space into several ranges, on itself for getting a better scalability. The simplicity of the skiplist algorithm makes it easier to implement and provides significant constant factor speed improvements over balanced trees.

We implement NV-Skiplist and the state-of-the-art key-value store, wB+-tree. We evaluate these two key-value stores through the micro-benchmark and YCSB. NV-Skiplist can improve the performance up to 48%, compared with that of wB+-tree. In our previous work [1], we focused on the study of a single-threaded system, meanwhile, this article extends and applies our design to a multi-thread system.

The contributions of this article are as follows:

1. We propose a skiplist-based index system for key-value stores on NVRAM. To fully exploit the performance of the hybrid memory system, its last level is located in NVRAM and other parts are stored in DRAM.
2. We design the consistent protocol effectively to maintain the consistency under any circumstances.
3. We choose a deterministic mechanism to improve its search performance. We also introduced a multi-header architecture to reduce node traversals during search operations and increase the scalability.
4. We extend our previous work [1] to enable our scheme in a multi-thread system and evaluated it via YCSB.

The rest of this article is organized as follows: Sect. 2 discusses background and motivation. Section 3 presents design and implementation of NV-Skiplist. Section 4 shows the experimental results. Section 5 reviews related work. Finally, Sect. 6 concludes the article.

2 Background and motivation

2.1 NVRAM

NVRAM provides persistence and a lower latency compared with original block-based devices, such as hard disk drives (HDDs) and solid state drives (SSDs). Table 1 shows the read and write latencies, the endurance times, and the random access of each type of NVRAM technology. The read latency of NVRAM is similar to that of DRAM, and the write latency of NVRAM is also in the same order of magnitude with that of DRAM. However, its write endurance is not as high as that of DRAM (especially for the PCM), thus reducing its write count, which must be considered when designing the system software. Finally, NVRAM offers a high random access performance like that of DRAM, which is different with the NAND Flash.

2.2 Index efficiency

With the emergence of NVRAM, its advantages—such as low latency in persistent access and byte addressability—have attracted the attention of storage researchers. Attempts have been made to store key-value data in NVRAM. However, the characteristics of its relatively long

Table 1 Characteristics comparison of different memory technologies [12, 13, 14, 15, 8]

Category	Read latency	Write latency	Write endurance	Random access
DRAM	60 ns	60 ns	10^{16}	High
PCM	50–70 ns	150–1000 ns	10^{9}	High
ReRAM	25 ns	500 ns	10^{12}	High
NAND Flash	35 us	350 us	10^{5}	Low

write latency and indispensable corresponding atomic costs, such as those of memory barriers and CLFUSH [16], must be taken into account when designing an index system for key-value store on NVRAM. Most current researches have focused on reducing write amplification. In Venkataraman et al. [7], a version-based mechanism for maintaining data consistency during modification was introduced. Yang et al. [8] proposed NV-Tree, a variant of B+-tree, in which only the leaf nodes are stored in NVRAM and the size of its internal nodes are fixed to improve the cache performance. Chen et al. [5] proposed wB+-tree in which data stored in the leaf nodes are out of order to prevent additional write caused by sorting.

Though related work achieved remarkable results through putting key-value storage into NVRAM, most of these projects focused on single-thread performances. Further, all the current research is based on B+-trees or their variants, which do not naturally fit NVRAM because they were originally designed for eliminating the performance gap between random and sequential accesses on traditional storage devices, which no longer exist in NVRAM. With motivation by above reasons, we proposed a skiplist-based index system for NVRAM.

3 Design and implementation

In this section, we introduce the design and implementation of NV-Skiplist. We first present an overview of our design, and then we describe the issues that we considered throughout the designing process, such as how to update the index, guarantee system consistency, and make searching more efficient. Last, we outline how the system recovers from normal shutdown and system failure. The goals of our design are as follows:

1. *Persistency* NV-Skiplist should recover to a consistent state after both a regular shutdown and a system failure.
2. *High write performance* NV-Skiplist should compromise the higher NVRAM write latency than DRAM write latency.
3. *DRAM-like search performance* NV-Skiplist should provide a search performance as efficient as its counterpart in DRAM.

Next, we will describe the design details that allow NV-Skiplist to achieve its goals.

3.1 NV-Skiplist overview

NV-Skiplist supports most of basic key-value operations, including *put*, *get*, *delete*, and *scan*. All operations but *scan* require a single key to locate the key-value pair. The *get* operation returns the corresponding value after locating the key-value pair, while *put*, *update*, and *delete* must modify the key-value pair, update the index, and make both persistent. As a result, the efficiency of locating a key-value pair and the persistent procedure can have a significant influence on these operations.

The skiplist inherently supports $O(\log N)$ searching, however, maintaining its persistence and structure while running on NVRAM will introduce various write operations that decrease the performance. In contrast, the *scan* operation takes a key and a count as an input; thus, keeping the index sorted can improve its performance. The last level of a skiplist is fully linked as a linked-list and sorted by keys. Since it is sorted, using a skiplist as index system can potentially provide acceptable scan performance. Updating a skiplist index involves many writes due to its sorted nature, which can significantly reduce the performance due to the expensive NVRAM write (if the whole system is placed in NVRAM). Accordingly, we only place the last level of the skiplist in NVRAM and kept its other parts in DRAM, applying a selective persistence policy on it.

3.2 Selective persistence

Selective persistence, which was first proposed in [8] and [17], refers to keeping the primary dataset in NVRAM and only focusing on its consistency during operations. The non-primary dataset is placed in DRAM and can be rebuilt at any time; its temporary state of inconsistency does not affect the eventual consistency of the entire index system. Due to the cost of the NVRAM write, only placing the last level of the skiplist where the key-value pairs are stored in NVRAM and putting its other parts in DRAM appears to be an effective solution to maximize the performance.

Figure 1 illustrates how selective persistence can be applied to NV-Skiplist. As shown in the figure, the internal levels of NV-Skiplist are stored in DRAM, meanwhile, the last level is stored in NVRAM by forming a persistent linked-list. Thus, NV-Skiplist can be rebuilt as long as the persistence of last level is guaranteed. That is, NV-Skiplist only guarantees the persistency of key-value items; its internal levels are used for improving searching performance.

3.3 Index updating

Figure 2 demonstrates the layout of both the internal levels and last level of NV-Skiplist. As shown in the figure, unlike an original skiplist, we place multiple key-value pairs into a single node to improve search performance by reducing the number of index nodes. However, traveling among index nodes by referencing pointers will certainly cause a significant amount of cache misses because each index node is allocated at a different memory address that may not be physically adjacent. To handle this issue, we group key-value pairs into cache line-sized alignment nodes, and the number of cache misses during search operations should decrease. Meanwhile, the size of last level is aligned with the cache line, which should prevent unnecessary cache line pollution.

To provide a higher write performance, keys are left unsorted in the last level, and a bitmap is used to track valid entries for decreasing NVRAM write during modification. Since the modern CPU only supports 8-byte atomic write, the bitmap size is fixed to 8 bytes, leading to the storage of no more than 64 key-value pairs in a single node. The next pointer is used to form a linked-list with its siblings for supporting range queries and rebuilding the index system by traveling through the list during failure recovery.

Additionally, the next pointer in the leaf node should also be persisted because the correct sibling node cannot be found after a system failure. The internal levels of the index nodes are placed in DRAM to locate key-value pairs. To reduce the comparison count, we chose the minimum and maximum keys of each corresponding last-level node to represent the range of keys stored in its last-level node; this strategy was first proposed by Lehman et al. [18]. Since the DRAM resident parts of an index node using the minimum and maximum values to represent the range of keys, locating a target node requires fewer compare operations. Our design is intended to outperform the transient skiplist and be similar to a traditional B+-tree. As stated in Sect. 3.1, when serving key-value update operations like Put, Update, and Delete, both the target key-value pairs and the index entries should be persistently updated. We applied selective persistence on NV-Skiplist, where only the last level of the skiplist would be stored in NVRAM; thus, we need only maintain the consistency of its last level.

Maintaining data consistency is the most important thing in an NVRAM key-value store. Since NVRAM write has a relatively higher latency, it is critical to maintain consistency while reducing the NVRAM write latency as much as possible. Ordinary solutions include logging, shadowing, and copy-on-writing, which cause double-write issues while guaranteeing data consistency. Inspired by a log-structured file system, we update the key-value pair out-of-place. Next, the bitmap is updated in-place using the CPU-supported atomic write.

Algorithm 1 $Insert(leaf, new\_entry)$

```
1: pos = leaf.GetUnusedEntrywithBitmap();
2: leaf.entry[pos] = new_entry;
3: clflush(&leaf.entry[pos]);
4: mfence();
5: /*enable the slot */
6: leaf.bitmap[pos] = 1;
7: clflush(&leaf.bitmap);
8: mfence();
```

Algorithm 1 describes how to insert a new entry into the last layer of the index node. Since we use the bitmap to track the usage of each slot, the algorithm begins by finding an unused entry in the bitmap (line 1). Then, it writes a new entry into the free entry (line 2). It stabilizes the new entry in NVRAM with *CLFLUSH* and *MFENCE* (lines 3 and 4) before updating the corresponding bit in the bitmap to indicate that the slot is occupied (line 5). Finally, the

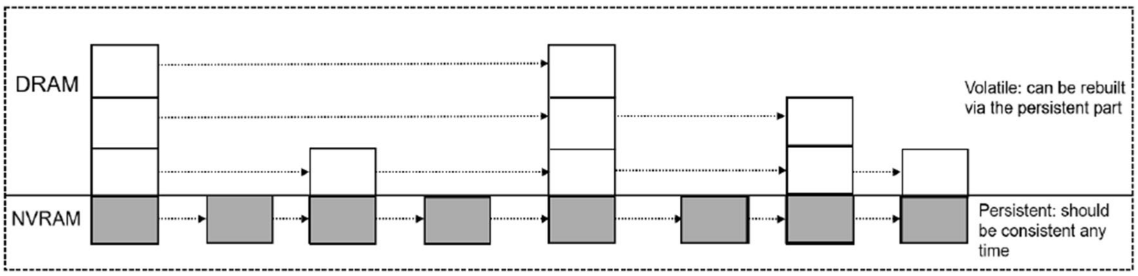

DRAM
NVRAM
Volatile: can be rebuilt via the persistent part
Persistent: should be consistent any time

Fig. 1 Selective persistence applied on NV-Skiplist

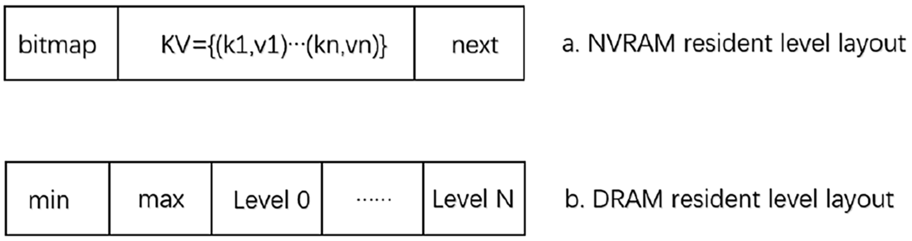


Fig. 2 NV-Skiplist inner level & last level layout

bitmap is made permanent in NVRAM by calling *CLFLUSH* and *MFENCE* (lines 6 and 7).

Data consistency can be guaranteed by following the reasons:

1. If the operation for writing new entry into a free slot fails, the original data in the node is kept intact because we select an unused slot for this insertion.
2. If the operation for updating the bitmap fails, newly inserted entries cannot be detected since the corresponding bit in the bitmap indicates that the slot is still free.

Owing to the fixed size of the last-layer node, splitting is necessary when this node overflows. Most of the B+-tree-based indexing system relies on redo-logging to maintain its consistency during node-splitting.

Figure 3 illustrates how leaf nodes split in wB+-trees. When a leaf node overflows, it copies half of its key-value pairs to the newly created right sibling node (note that all updates should be performed atomically at the same time). In order to solve this problem, a redo-logging is performed; this requires extra cache line flush operations.

Unlike a wB+-tree, inspired by [19], our design creates two new last-level nodes and copies half of the content of the original node to each of them. Then we use an atomic operation, such as CompareAndSwap, to switch the previous node's next pointer to the newly created node.

Figure 4 shows how splitting works in NV-Skiplist in detail. Suppose that the last-level node is larger than a cache line size and flushing this node requires k cache line flushes. Our design requires 2k + 1 cache line flushes (2k for the newly created node and 1 for the previous node's next pointer). In a wB+-tree, a leaf node split requires 2k + 2 cache line flushes (2 for the in-place update on original leaf node, k for the right sibling leaf node, and another k for the redo-logging).

In addition, we choose the average value among all the keys, excluding the maximum and minimum keys in each node as the split pivot. Within the pivot key, a scan is first executed over the entire node where keys smaller than the pivot are moved to one node and keys larger than the pivot are moved to another. Then, the maximum and minimum keys in both index nodes are updated.

3.4 Index searching

The original skiplist supports average $O(\log N)$ search time complexity. Based on the original skiplist, we proposed several optimizations to improve the search performance.

The first optimization uses a deterministic design inspired by [11] to replace the general, non-deterministic design of ordinary skiplists. The design of existing skiplist uses random numbers as the height of each index node. As many index nodes are inserted, their height distribution tends to be similar with that of a balanced binary tree, which leads to an $O(\log N)$ search time complexity. However, we evaluated the non-deterministic design and discovered that it is not search-friendly in this context because we have previously grouped keys, and the total number of index nodes is decreased. As a result, we replaced the non-deterministic design with a deterministic one. The main concept behind a deterministic skiplist is to build a balanced binary tree, like a skiplist, following each insertion.

Algorithm 2 $Deterministic\_Search(key)$

```
1: pred = header;
2: for $level = maxLevel; level >= 0; level --$ do
3:     span = 0;
4:     $item = pred->next[level]$;
5:     while $item != NULL$ do
6:         if $key > item->max$ then
7:             if $span > THRESHOLD$ then
8:                 $item->num\_level++$;
9:                 span = 0;
10:            end if
11:            $pred = item$;
12:            $item = item->next[level]$;
13:            $span++$;
14:        else if $key < item->min$ then
15:            break;
16:        else
17:            return item;
18:        end if
19:    end while
20: end for
```

Algorithm 2 illustrates the deterministic search design in detail. To create it, we defined a SPAN threshold which controls the maximum count of index nodes that passed by on one level before each insertion during the search process. The search begins at the header of the skiplist (line 1) and travels from the uppermost to the bottommost levels to

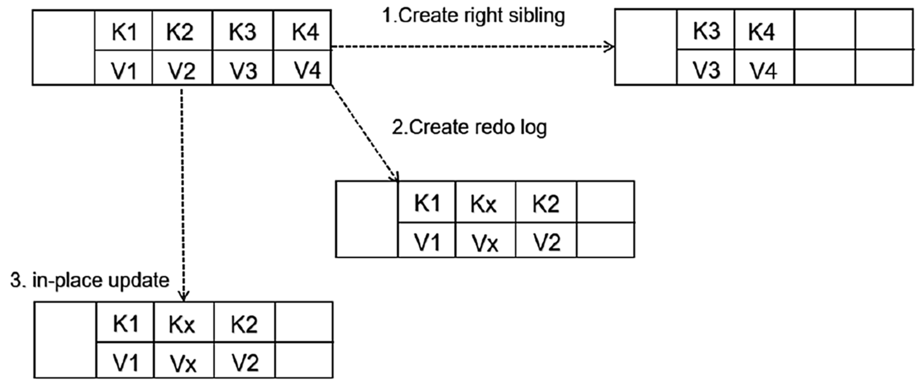


Fig. 3 leaf split in wB+-tree

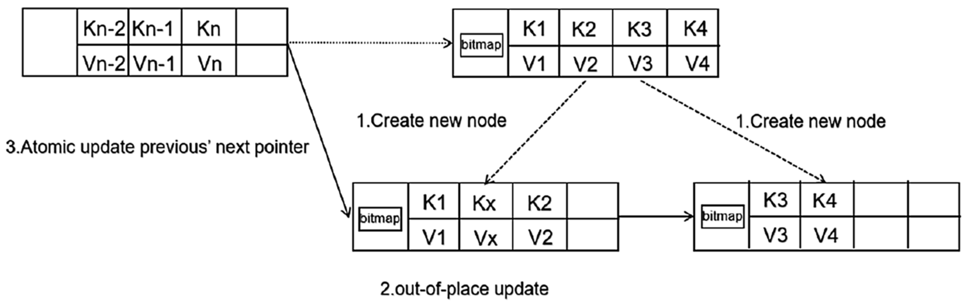


Fig. 4 leaf split in NV-Skiplist

confirm whether the target key belongs to a certain index node (lines 2 to 21). If the target key is larger than the maximum key, the algorithm will move to the next node (lines 11 and 12) and increase the span value (line 13). If the span becomes larger than the prefixed threshold, then the height of the search index nodes increases by 1 (line 8).

As a second optimization, we propose a multi-header design that enhances scalability. In ordinary skiplists, every search procedure starts from its unique header, then travels along the index nodes by following pointers from left to right and top to bottom. To augment potential parallelism and decrease the number of nodes on search paths, we statically divide all key-value pairs and their index nodes into several groups in a process that is initiated by a general skiplist header.

Figure 5 presents the design of the multi-header skiplist. As shown the figure, key-value pairs are distributed among two ranges, and the header information is persistently stored elsewhere. The search procedure is similar to that of general skiplists, but here it indicates the range in which a target key belongs. It is easily implemented according to the global header information. When the range in which the target key belongs is determined, the subsequent steps are as same as those in a general skiplist. Our proposed design is intended to achieve a better multi-thread performance because it distributes contentions into different ranges.

3.5 Index deleting

Algorithm 3 describes how key-value pairs are deleted in NV-Skiplist. First, the algorithm searches the node where the target key-value pair is located (line 1) through the semantics of the skiplist. Next, it scans all entries in the last-level node to locate the key (line 2). When the key is located, the algorithm checks whether the entry is still valid (line 3). If the entry is valid, it is invalidated by an atomic write to set the corresponding bit to 0 (line 6), then using *CLFLUSH* and *MFENCE* to persist the bitmap. Since the bitmap is now atomically updated, a system crash will not destroy its consistency. In addition, we do not reset the key-value pair during delete operations to prevent extra NVRAM write; this is because both the pure search and insert operations begin by checking the validity through the bitmap.

Algorithm 3 $Delete(entry)$

```
1: leaf = search_for_leaf($entry$) ;
2: pos = leaf.GetEntry(entry);
3: if $leaf.bitmap[pos] == notvalid$ then
4:     return not existed
5: else
6:     leaf.bitmap[pos] = 0;
7:     clflush(&leaf.bitmap);
8:     mfence();
9: end if
```

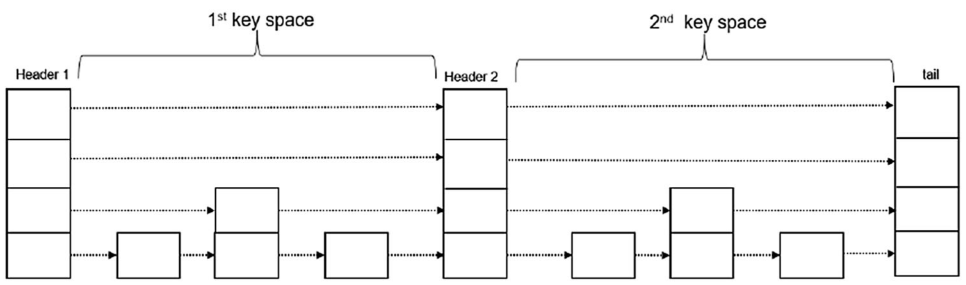


Fig. 5 Multiple header design of NV-skiplist

3.6 Recovery

In this section, we describe how recovery is performed in NV-Skiplist in cases of normal shutdown and system failure.

Recovery after a normal shutdown NV-Skiplist persists the DRAM resident part of the skiplist into a reserved address and automatically indicates that it is a normal shutdown by updating a flag. During recovery, NV-Skiplist checks whether the flag is set; if so, it reads the skiplist index and restores it to DRAM.

Recovery after a system failure In the case of system failure, NV-Skiplist recovers from the consistent linked-list (the last level of NV-Skiplist) by scanning all valid entries. Upon scanning each node, it selects the maximum and minimum keys and inserts them into the skiplist index.

4 Evaluation

4.1 Experiment environment

We run experiments on a server that has following hardware: 4 Intel Xeon E5-2620 CPUs with 6 cores, 12 threads on each, 15 MB L3 cache and a clock speed of 2 GHz. The evaluation system runs Linux and has 64 GB RAM. Both our implementation and the competitor are compiled with GCC 4.8.4 using optimization—O3. We used DRAM-based NVRAM emulator libpmem libraries [20] to simulate NVRAM. By providing low-level persistent memory support for applications using direct access storage, libpmem offers storage that supports load and store access without paging caches from a block storage device.

We compared our design with that of a wB+-tree [5], which is a B+-tree variant that stores all tree nodes in NVRAM. In a wB+-tree, key-value pairs are inserted into the leaf node in an append-only style. Additionally, a wB+-tree employs a small metadata called slot to track the order of keys in the leaf node without sorting them itself. There is also a bitmap used to track valid entries in the leaf node. Due to the metadata, several *CLFLUSH* and *MFENCE* procedures are required to maintain consistency. In addition, a wB+-tree relies on redo-logging to maintain its consistency during node splitting. To make our experiment feasible, we slightly modified a wB+-tree so that only the leaf nodes were stored in NVRAM. In doing so, the interference of a large amount of expensive NVRAM writes was avoided.

4.2 Micro-benchmark

In this section, we focus on the single-thread "insert" and "get" performances of NV-Skiplist. For our micro-benchmark, we use uniformly distributed keys; each key has a value of 8-byte integers. In our following experiments, D-skiplist stands for the deterministic skiplist and M-Skiplist stands for multi-range skiplist.

4.2.1 Insert performance

Figure 6 shows the total latency occurring from inserting 25,600,000 key-value pairs into NV-Skiplist with a deterministic mechanism, a multi-range mechanism, and wB+-tree. Both our optimizations were proven to work. The deterministic design of the skiplist with a threshold of 3 outperforms wB+-tree by 15%, and the multi-range design of 64 headers performed almost 27% better than wB+-tree. Since each insert in a skiplist occurs after the target node is located, insert performance can also benefit from optimized searching processes.

4.2.2 Get performance

Figure 7 shows the performance of *get* operations while searching for 25,600,000 key-value pairs. As shown in the figure, the purely deterministic design performs 45% slower than does wB+-tree. The reason that cache invalidation storm occurs is due to the pointer chasing. In wB+-tree, keys are stored in adjacent physical addresses in both internal and leaf nodes, and loading the first key of the node will pre-fetch all other keys. However, because each index node is linked through pointers in NV-Skiplist, pointer chasing will introduce a significant number of cache misses. The opposite result is expected from the *get* and *insert* performances due to the lesser degree of write amplification in the *insert* operation for the consistent cost

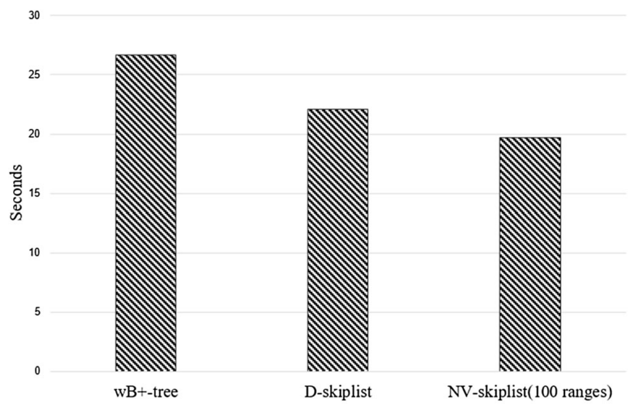


Fig. 6 Latency of inserting 25,600,000 key-value pairs

of our design. Despite this, with the multi-range mechanism, the *get* performance of this operation can be as effective as in wB+-tree.

4.2.3 Analysis of insert & get performance

In this section, we analyze the reasons that NV-Skiplist behaves as it does in these experiments. The target we chose is the multi-range NV-Skiplist, since it has the best insert and get performance above all.

Figure 8 demonstrates the data reference counts measured by perf tool after inserting 25,600,000 key-value pairs. From the figure, the data reference count of NV-Skiplist is much lower than that of wB+-tree. We believe that this is one of the reasons that our design outperforms wB+-tree in insertion. Since we group key-value pairs into one node and use ranges to represent each index node, the number of keys that are accessed during a search are significantly reduced. In addition, the multi-ranged structure distributes the search procedure to different ranges, thus decreasing the total count of accessed index nodes.

Figure 9 presents the number of L1 cache misses while the system was getting 25,600,000 key-value pairs. This result shows that cache misses seem to be inevitable due to the structure of linked pointers. We believe that cache misses are the core reason that the get performance in deterministic skiplist is inferior to the insert performance compared with wB+-tree despite our optimizations to reduce write amplifications.

4.3 Macro-benchmark

4.3.1 YCSB

The Yahoo! Cloud Serving Benchmark (YCSB) is an open-source specification and program suite used for evaluating the retrieval and maintenance capabilities of computer programs. It is often used to compare the relative performance of NoSQL database management systems.

In our experiment, we updated 2 million records with 2 million operations through various numbers of threads. The key distribution conforms to the uniform distribution. Figure 10 illustrates the performance of our system. It is evident that our proposed NV-Skiplist scales better than does wB+-tree; this is because our multi-ranged design disperses the contentions among different ranges. In addition, we implemented the insert operation of NV-Skiplist with CompareAndSwap operations in the last level and avoided using any atomic operations in upper levels. This was practical because the data consistency of the last level is maintained; even missing keys could finally be found in the last level. This mechanism may increase latency when certain keys cannot immediately be found (in the upper levels), but due to its lock-free structure, our design was able to scale well.

As mentioned above, the entire key range was statically divided into several ranges during NV-Skiplist initialization. The static separation works fairly well when keys are uniformly distributed; however, if some hot zones exist, keys may aggregate in certain ranges and maximize the height of each index node in those ranges, causing the search time complexity to degrade to O(n).

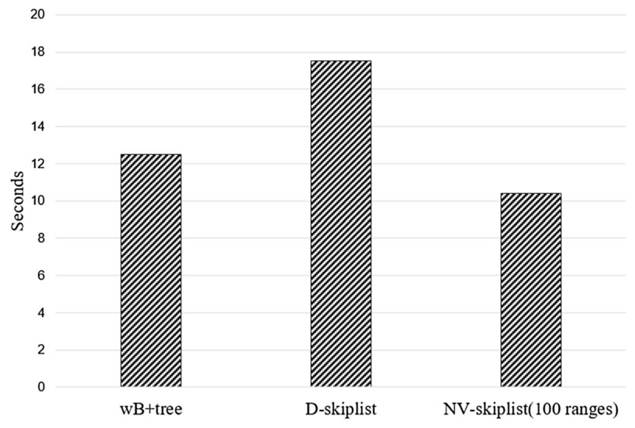


Fig. 7 Latency of getting 25,600,000 key-value pairs

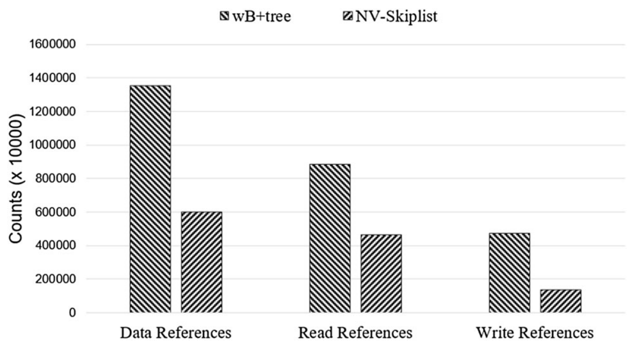


Fig. 8 Data reference count during insert operation

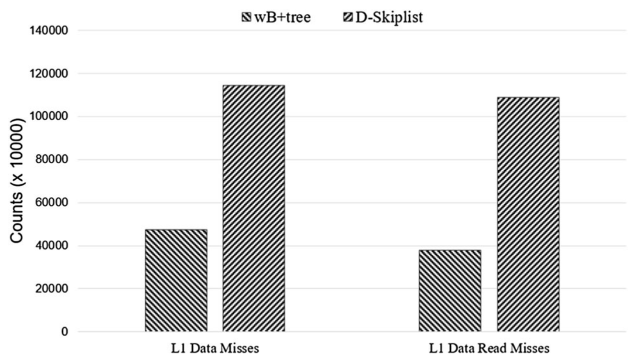


Fig. 9 Cache miss count during get operation

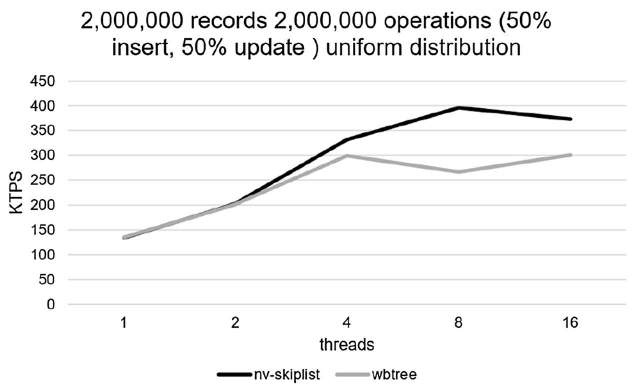


Fig. 10 YCSB evaluation on Insert & Update operations

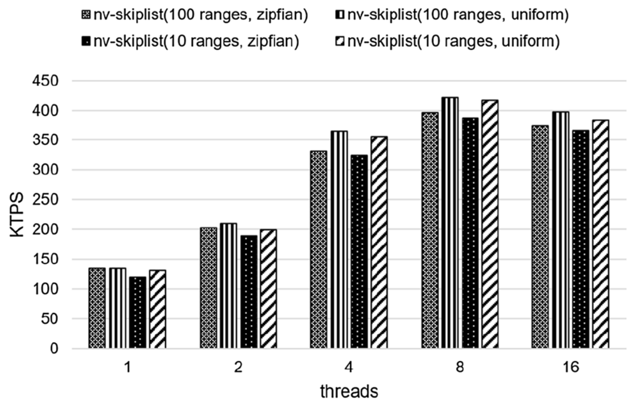


Fig. 11 YCSB evaluation for insert on different key distribution with different ranges with fixed number of threads

Figure 11 shows how the key distribution affects the overall insertion performance. In this evaluation, we inserted 2 million key-value pairs into NV-Skiplist, each case containing 10 ranges and 100 ranges in which the keys conform to uniform and Zipfian distributions. From Fig. 11, it is clear that the more ranges the entire key space is divided into, the better its performance. Meanwhile, the performance degraded more in the Zipfian distribution than

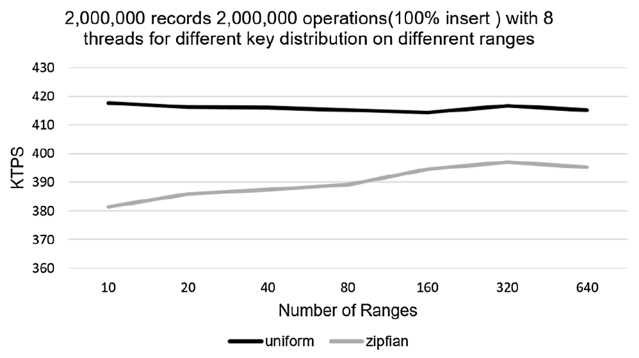


Fig. 12 YCSB evaluation for insert with different key distribution on different ranges with variable number of threads

in the uniform one, and the decreased performance became more significant when only 10 ranges were used. This discovery confirmed our theory that hot zones in the key distribution do influence performance. As a result, we will continue to dynamically divide the keys according to the workload in our future projects.

To see how the number of ranges affects NV-Skiplist's performance with various key distributions, we inserted 2 million key-value pairs with 8 threads and various numbers of ranges and keys conforming to the uniform and Zipfian distributions as shown in Fig. 12. We discovered that the number of ranges does not affect the performance when keys conform to uniform distribution, but when the keys conform to a Zipfian distribution, performance increases as the number of ranges does. This is because a large number of ranges decentralizes hot zones and the index nodes in each range can form similarly to a binary search tree.

5 Related work

The emergence of NVRAM enables novel and intricate techniques for designing data structures. However, logging or shadow-paging are still necessary to maintain its consistency. To reduce associated costs, Yang et al. [8] proposed NV-Tree, a persistent B+-tree based on the CSB+-tree [21]. NV-Tree maintains consistency only in its leaf nodes, relaxing that of its inner nodes. It also stores all inner nodes in a consecutive memory space aligned with the cache line size and locates them through an offset rather than through pointers; this allows it to achieve a more efficient use of space and cache-hit rate. If system failure occurs, its inner nodes can be rebuilt according to the consistent leaf nodes. Additionally, since the size of the inner-nodes is pre-defined, when the number of key-value pairs NV-Tree contains exceeds its capacity, all inner-nodes should also be rebuilt.

Venkataraman et al. [7] proposed CDDS-tree, a persistent B+-tree built by maintaining a limited number of data structure versions with the requirements that updates should not weaken the structural integrity of an older version and that updates are atomic. CDDS-Tree guarantees that failures between operations will never leave data in an inconsistent state. It can recover by reading the latest version and discarding events occurring prior to that version. However, it requires GC to periodically clean its old versions.

Chen et al. [5] a persistent tree that relies on atomic write and redo-logging to ensure consistency. It maintains its keys out of order in the leaf nodes and uses a sorted indirect slot array in each leaf node to improve its search performance. However, since the size of its sorted slot array cannot be larger than 8 bytes—the largest size of atomic write that is supported by a CPU—the size of its leaf nodes is limited and causes frequent splitting where additional NVRAM write occurs. While these projects perform better than many existing persistent data structures, the performance gap between them and their fully transient counterparts is still significant. To address this, we propose NV-Skiplist.

6 Conclusion

In this article, we proposed a skiplist-based, non-volatile indexing system called NV-Skiplist. To accommodate the features of NVRAM, we optimized this system by adding minimum and maximum keys into the nodes to reduce the computation overhead; we also grouped the key-value pairs into a cache line-sized alignment node to reduce its number of cache misses. We further enhanced our system's performance by incorporating deterministic and multi-header designs. The results indicate that both optimizations improve performance in insertion procedures, and the pure deterministic design is proven to reduce the performance gap between a wB+-tree and our design in getting procedures. With the application of the multi-header design, the get performance could become as efficient as in wB+-tree. Furthermore, our multi-header design is proven to scale better than wB+-tree when the key distribution conforms to the uniform distribution, since the contention can be distributed into different ranges.

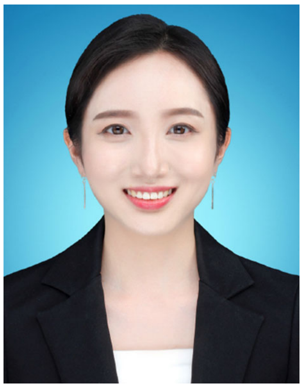
Acknowledgements This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) and funded by the Ministry of Science, ICT and Future Planning (2015M3C4A-7065646, 2016M3C4A7952587). This project was supported by the National Research Foundation of Korea (NRF) and a Grant provided by the Korean Government (MSIT) (No. NRF-2017-R1A2B4005681, 2018R1C1B5085640) (Corresponding Author: Yongseok Son).

References

1. Qichen, C., Heonyong, Y.: Design of skiplist based key-value store on non-volatile memory. FAS*W Foundations and Applications of Self* Systems (2018)
2. Burr, G.W., Breitwisch, M.J., Franceschini, M., Garetto, D., Gopalakrishnan, K., Jackson, B., Kurdi, B., Lam, C., Lastras, L.A., et al.: Phase change memory technology. J. Vac. Sci. Technol. B **28**, 223–262 (2010)
3. Yang, J.J., Williams, R.S.: Memristive devices in computing system: Promises and challenges. ACM J. Emerg. Technol. Comput. Syst. **9**, 11 (2013)
4. Apalkov, D., Khvalkovskiy, A., Watts, S., Nikitin, V., Tang, X., Lottis, D., Moon, K., Luo, X., Chen, E., Ong, A., Driskill-Smith, A., Krounbi, M.: Spin-transfer torque magnetic random access memory (stt-mram). ACM J. Emerg. Technol. Comput. Syst. **9**(12), 13 (2013)
5. Chen, S., Jin, Q.: Persistent b+-trees in non-volatile main memory. PVLDB **8**, 786–797 (2015)
6. Son, Y., Kang, H., Yeom, H. Y., Han, H.: A log-structured buffer for database systems using non-volatile memory. In: Proceedings of the Symposium on Applied Computing, ACM, pp. 880–886 (2017)
7. Venkataraman, S., Tolia, N., Ranganathan, P., Ampbell, R. H.: Consistent and durable data structures for non-volatile byte-addressable memory. USENIX FAST (2011)
8. Yang, J., Wei, Q., Wang, C., Chen, C., Yong, K., He, B.: Nv-tree: A consistent and workload-adaptive tree structure for non-volatile memory. IEEE Trans. Comput. 65:2169–2183 (2016)
9. Comer, D.: The ubiquitous b-tree. ACM Comput. Surv. **11**, 121–137 (1979)
10. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Commun. ACM **33**, 668–676 (1990)
11. Mei, F., Cao, Q., Wu, F., Li, H.: A concurrent skip list balanced on search. Advanced Parallel Processing Technologies, pp. 117–128 (2017)
12. Chang, M. F., Wu, J. J., Chien, T. F., Liu, Y. C., Yang, T. C., Shen, W. C., King, Y. C., Lin, C. J., Lin, K. F., Chih, Y. D., Natarajan, S., Chang, J.: 19.4 embed- ded 1mb reram in 28nm cmos with 0.27-to-1v read using swing-sample-and-couple sense amplifier and self-boost- write-termination scheme. IEEE International Solid-State Circuits Conference Digest of Technical Papers, ISSCC'14, pp. 332–333 (2014)
13. Lee, B. C., Ipek, E., Mutlu, O., Burger, D.: Architecting phase change memory as a scalable dram alternative. In: Proceedings of the 36th annual international symposium on Computer architecture, ISCA, 2–13 (2009)
14. Micron. Slc nand flash products
15. Department of Computer & Science Engineering Univeristy of California San Diego D.: The non-volatile memory technology database (nvmdb). *Tech Rep* (2015)
16. Intel 64 and ia-32 architectures software developer's manual
17. Ismail, O., Johan, L., Anisoara, N., Thomas, W., Wolfgan, L.: Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. ACM SIGMOD international conference on Management of data, pp. 371–386 (2016)
18. Lehman, T. J., Carrey, M. J.: A study of iondex structures for main memory database management systems. VLDB, pp. 294–303 (1985)
19. Wook-Hee, K., Jihye, S., Jinwooong, K., Beomseok, N.: clfb-tree:cacheline friendly persistent b-tree for nvram. ACM Transactions on Storage(TOS)-Special Issue on NVM and Storage 14 (2018)
20. https://pmem.io/pmdk/

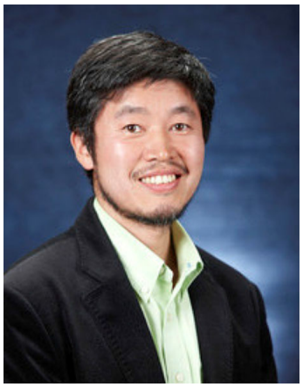
21. Rao, J., Ross, K.A.: Making b+-trees cache consious in main memory. ACM SIGMOD Rec. **29**, 475–486 (2000)

Qichen Chen received his B.S. degree in Department of Computer Science and Infomation from NanJing XiaoZhuang University in 2011. Currently, he is an Ph.D. Candidate in Department of Computer Science and Engineering of Seoul National University. His research interests are operating, distributed, and database systems.



Hyojeong Lee received her B.S. degree in Department of Computer Science from Sookmyung Women's University in 2018. Currently, she is an M.S. student in Department of Computer Science and Engineering at Seoul National University. Her research interests are operating, distributed, and database systems.

Yoonhee Kim she is the professor of Computer Science Department at Sookmyung Women's University. She received her Bachelors degree from Sookmyung Women's University in 1991, her Master degree and Ph.D. from Syracuse University in 1996 and 2001, respectively. She was a Research Staff Member at the Electronics and Telecommunication Research Institute during 1991 and 1994. Before joining the faculty of Sookmyung Women's University in 2001, she was the faculty of Computer Engineering dept. at Rochester Institute of Technology in NY, USA. Her research interests span many aspects of runtime support and management in distributed computing systems. She is a member of IEEE and OGF, and she has served on variety of program committees, advisory boards, and editorial boards.



Heon Young Yeom is a Professor with the School of Computer Science and Engineering, Seoul National University. He received B.S. degree in Computer Science from Seoul National University in 1984 and his M.S. and Ph.D. degrees in Computer Science from Texas A&M University in 1986 and 1992 respectively. From 1986 to 1990, he worked with Texas Transportation Institute as a Systems Analyst, and from 1992 to 1993, he was with Samsung Data Systems as a Research Scientist. He joined the Department of Computer Science, Seoul National University in 1993, where he currently teaches and researches on distributed systems and transaction processing.

Yongseok Son received his B.S. degree in Information and Computer Engineering from Ajou University in 2010, and the M.S. and Ph.D. degrees in Department of Intelligent Convergence Systems and Electronic Engineering and Computer Science at Seoul National University in 2012 and 2018, respectively. He was a postdoctoral research associate in Electrical and Computer Engineering at University of Illinois at Urbana-Champaign. Currently, he is an assistant professor in School of Computer Science and Engineering, Chung-Ang University. His research interests are operating, distributed, and database systems.