RH-Skiplist : Range-Hash Skiplist for DRAM-PM Memory Systems

Heejin Yoon[*]
20195376
heejin5178@unist.ac.kr

Jin Yang[*]
20195369
yangjin@unist.ac.kr

ABSTRACT

With fast development of non volatile memory, there are many data structures for applying this new hardware. The new hardware can persist data faster than traditional persistent storage such as SSD or HDD. Moreover, unlike volatile memory such as DRAM, it won't lose data because it can persist data. Along with development persistent storage, key-value store design is also influenced. For large key-value store design with PM, we suggest RH-skiplist which is combination of skiplist and hash index. For memory component in key-value store, skiplist is usually used as data structure in DRAM. However, its performance isn't better than other data structure such as B+ tree or hash table. Even so, just using hash table or B+ tree on memory component in key-value store is not answer. Because conventional data structure of key-value store must support sequential write to reduce overhead during writing SSD or HDD. However, PM, which is kinds of PM, can take advantage of random writes. Considering this point, we can suggest new scheme for memory component in key-value store, range-hash skiplist. Performance of range-hash skiplist is better than traditional skiplist in DRAM. In contrast, performance of our scheme is worse than traditional hashmap in PM. The reason of poor performance will be descirbed in detail at 4. Evaluation part.

1 INTRODUCTION

The release of next-generation non-volatile random access memory (NVRAM) such as Intel Optane[5], STT-MRAM[4] has prompted attempts to make good use of its characteristics such as low read and write latencies and persistency in the storage system. For example, it has been proposed to single-level NVM storage systems like NV-Tree[18] where memory with low read/write latency and non-volatile secondary storage are merged. However, unlike DRAM-like read latency, the microsecond-level write latency in the NVM is still higher than DRAM. Therefore, traditional DRAM-based index structures that require a lot of write overhead are not appropriate to fully utilize NVM characteristics. As a way to solve this problem, a hybrid memory system has been proposed that uses NVM to maintain persistence and increase indexing performance using DRAM[1, 11, 17].

On the one hand, many KV stores operate based on Log-Structured Merge-tree using the two main storage types of DRAM and disk. The process of compaction taking place in disk, especially the frequent and expensive compaction between Level 0 and Level 1 caused by the duplication key between each memtable, causes high CPU utilization and IO cost [9]. The NVM, which incorporates both the fast performance of DRAM and the non-volatile characteristics of disk, can be used as a buffer to offset the trade-off of extreme performance in the KV store environment where these DRAMs and disks are used together. As a result, a study has recently emerged to reduce processing overhead by using NVMs[7–9].

In this paper, we propose a new range-Hashtable-Skiplist named RH-Skiplist that is properly modified the existing skiplist index structure to the characteristics of DRAM and NVM. The RH-Skiplist improves search and input performance by placing the index node and leaf node in DRAM and stores hash table of each leaf node in NVM to maintain its persistence. In addition, the leaf node of the existing skiplist was grouped in key range to reduce the splits causing many write overhead. Each leaf node, covering a specific key range, consists of a hash table to utilize the NVM's fast random write and reduce sorting overhead. Finally, we suggest the direction that the RH-Skiplist can contribute to make more optimized key-value store design in the future.

The main contributions of this work are as follows:

- Maintaining index nodes can support partial sorting and reduce hash collisions accordingly.
- By maintaining each hash table in leaf node, we can reduce L_0 - L_1 compaction overhead in Key-value store.
- By keeping the appropriate data structure for each hardware device, we can support efficient storage system.

2 BACKGROUND AND MOTIVATION

2.1 Key value store

Most Key-value store design uses Log-Structured Merge-tree as data structure[10]. LevelDB[3] and RocksDB[2]

[*]Authors contributed equally to this research.

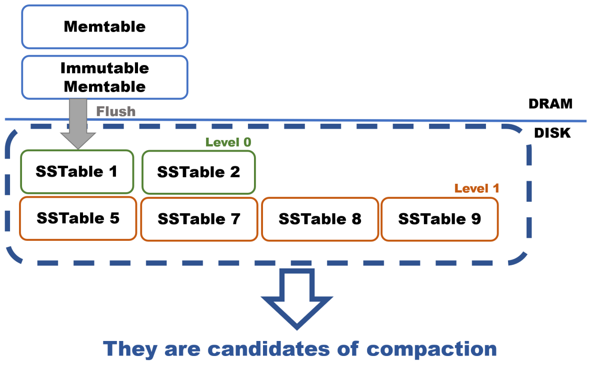

Figure 1: Compaction of LSM tree

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

Figure 2: Memory latency [17]

which is representative persistent key value store also use log-structured merge tree(shortly, LSM tree). As shown in figure 1, LSM tree consists of 2 components, memory and disk. Memory component stores memtable and immutable memtable. Disk component stores files which is called SSTable(Sorted String Table). When key-value pair is inserted, this item will be stored in memtable. Since memtable uses skiplist, the item could be sorted. After memtable size is over constant threshold, the memtable will be changed as immutable memtable. This immutable memtable doesn't allow any mutation including insertion. And immutable memtable is flushed to disk by scheduler. This immutable is modified to SSTable which is unit of file in disk component. This SSTable will reside in L_0 at first time. When size of L_0 is larger than constant threshold, SSTable in L_0 and one in L_1 will be merge-sorted by using compaction works. Compaction takes 2 steps, 1) check whether there is overlapped range between SSTable in upper level and one in lower level. 2) merge sort SSTables which shares overlapped range. Using this method, number of level will be grows with larger dataset size. Figure 1 describes a more detailed. However, as we can predict easily, compaction is expensive operation in perspective of CPU utilization and IO cost. Because computation for merge sort during compaction work can cause high cpu utilization. Moreover, file which resides in disk component has to be loaded on memory component during compaction and this can cause expensive IO cost. Its leveling strategy can also cause higher write amplification. Especially, compaction between L_0 and L_1 is remarkable write stall. Flushed memtable could share frequent overlapped range. This requires frequent and expensive compaction between L_0 and L_1. In other words, compaction between L_0 and L_1 can lead higher overhead. Therefore, we suggest data structure which can optimize compaction works between L_0 and L_1. Optimization will be focused on reducing duplicate key and overlapped range between multiple SSTable. Then we have to focus on data structure which will be used in persistent memory. In conventional key-value store, memtable uses skiplist. This design is considering that secondary storage such as SSD or HDD is beneficial to sequential write. However, as shown in Figure 1, PM can take advantage of random writes. It means that we don't have to waste computation overhead. Therefore, we decide to use hashmap scheme on PM. In this part, hashmap will be sorted partially. If we just keep one large hashmap in PM, file after flushing is totally unsorted. Moreover, this scheme cannot still solve frequent overlapped range between multiple SSTables which reside in L_0 and L_1. Therefore, the hashmap will support partially sorting and this can solve essential reason of compaction overhead which we pointed out. Partially sorting is supported by DRAM and this merge sorting will use range-skiplist. Index node of range-skiplist will keep one leaf node. And this leaf node will store corresponding hashmap which stores real key and value. And index node will store its own min value, then we can know range of each hashmap. Keeping hashmap for one leaf node can say that there is no overlapped range between each hashmap of leaf node. By using this fine-grained SSTable scheme, we can reduce overhead caused by compaction between L_0 and L_1.

2.2 Non-volatile Memory

The recent advent of fast storage systems such as byte-addressable nonvolatile memory, such as STT-MRAM [4] and the now commercially available Intel Optane DC Persistent Memory [6], along with ultra-low latency SSDs such as the Samsung Z-SSD [15] and Intel Optane SSD [5] are changing the landscape. For these fast storage systems, the performance of the software layer such as the efficiency of the index structure will become critical, more strongly affecting the final latency of the system. Non-volatile memories are one type of fast storage media. Such persistent memories (PM) is projected to have latency comparable to DRAM, non-valatile property and high capacity.

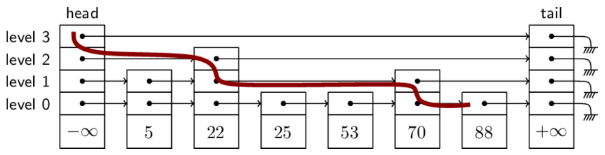

Figure 3: A skiplist with 4 levels, and the traversal searching 88 [16]

2.3 Skiplist

Skiplist[14] is a data structure in the form of an aligned linked list with time complexity of O(log n) for insertion and search operation. Conceptually, the skiplist is a sequence of singly-linked lists $L_0, \ldots, L_h$. Each list L_r contains a subset of the items in L_{r-1}. We start with the input list L_0 that contains n items and construct L_1 from L_0, L_2 from L_1, and so on. The items in L_r are obtained with some fixed probability p. Figure 3 shows an example of the traversal searching 88 in a skiplist with 4 levels.

2.4 Hash Index

Hash index is generally known to provide faster insert and search operations compared to other index structure like B+-tree and skiplist. Most hash table has hash collisions problem where the hash function generates the same index for more than one key. There are various hash index techniques for hash collisions resolving. In this paper, we use cuckoo hashing[12] for DRAM version and hash index for DRAM-PM version using open addressing Hashmap with Robin Hood collision resolution[13] technique for RH-Skiplist.

Algorithm 1: Insert(Key, $Value$)

```
Var $update[1..MaxLevel]$
Insert $Value$ in PM and get $addr\_value$
x := list→header
for $i$ = $list→level$ downto 1 do
    while $x→forward[i]→min \leq searchKey$ do
        x := x→forward[i]
        update[i] := x
    end
end
leaf := update[0]→leafnode
if InsertLeaf(leaf,key,addr_value) then
    break;
else
                                    ▷ need to node split
    average := (leaf.min+leaf→next.min)/2
    new_leaf := make_leafNode(new_min)
    make_indexNode(new_min,new_leaf)
    for (k,v) in leaf do
        if new_min≤key then
            Insert(new_leaf→HT,k,v)
            Delete(leaf→HT,k)
        else
            break;
        end
    end
    if new_min≤key then
        Insert(new_leaf→HT,key,value)
    else
        Insert(leaf→HT,key,value)
    end
end
```

3 DESIGN

Our RH-Skiplist targets the hybrid memory system, which stores index nodes in DRAM and Hash table of leaf nodes in PM. In this case, DRAM storing index nodes serves only as indexing to locate leaf nodes. In the leaf node, there are two main characteristics compared to the traditional skiplist's leaf node. In this section, we will explain each structure one by one.

3.1 Range Skiplist

Original structure of skiplist stores one key in one node. Therefore, we first changed this structure to a range leaf node structure so that one leaf node could have multiple (key, addr_value) pairs. Range skiplist distributes the whole key space into several ranges and the keys of each range are included in one leaf node. In Algorithm 1, you can see how the range skiplist works. If the $InsertLeaf$ function returns true, it means (key,addr_value) has been successfully stored in the hash table of the range to which the key belongs. Otherwise, it means that the hash table of target leaf node($leaf$) is full and there is no room for (key,addr_value). In this case, RH-Skiplist divides the range in half and make $leaf$ cover half of the range and create a new leaf node (new_leaf) to cover the remaining half. At this point, (key,addr_value) that is included in the key range of $newafleaf$ among elements of $leaf$ requires a migration process to insert into new and delete from $leaf$. This series of processes could be called as "split". After the node split process, insert the key (addr_value) again, and insert operation is terminated.

3.2 Leaf node with hash table

Keeping the keys in alignment in each key of the range skiplist causes a lot of write overhead in one insert operation while shifting the keys stored to maintain order. Therefore, we adopted a structure in which keys are stored in a random way without aligning them in hash table of its leaf node. As a result, the keys in leaf node are

Algorithm 2: Search(*Key*)

```
x := list→header
for $i$ = $list→level$ downto 1 do
    while $x→forward[i]→min \le searchKey$ do
        x := x→forward[i]
    end
end
leaf_HT := x→leafnode→HT
return leaf_HT.get(Key)
```

stored in the hash table that ensures fast insert, search performance while storing the keys in the random location. As shown in Algorithm 2, each hash table is accessible from the leaf node of the corresponding key range and returns addr_value through the get function of the hash table. Finally, the returned addr_value provides access to the actual value stored in the PM.

4 EVALUATION

In this section, we will discuss about experimental setup and performance of this design. Moreover, by comparing to hashmap in PM and skiplist in DRAM, we will discuss experimental result in detail.

4.1 Experimental setup

We performed 2 kinds of experiments. For DRAM experiment, experiments consists of range-skiplist which hashmap of leafnode is implemented on DRAM and conventional skiplist. And for PM experiments, experiments consists of range-skiplist which hashmap of leafnode is impelmented on PM and hashmap on PM. Setup of DRAM experiment is equipped with two Intel Xeon E5-2640. Each CPU which is runnning at 2.6GHz has 8 cores. The memory size is 39GB. Setup of PM experiments is equipped with two Intel Xeon Gold 6242 CPU which is running at 2.80GHz. And each has 16 cores. We use Intel Optane DC Persistent memory 128GB. we set PM as app direct mode. Memory size of this server is 32.6GB. About operating system, we use Ubuntu 18.04 release version for DRAM experiments. For PM experiments, we use Ubuntu 18.10 release version.

4.2 Workloads

As workloads, we used micro-benchmarks which we implemented. Micro-benchmarks consists of sequential write and sequential read.

4.3 DRAM experiment

This section will handle DRAM experiments. This experiments use sequential write and read. Figure 5 an Figure 6 shows normalized latency of sequential write and read

dataset size	range HT insert	SL insert	range HT read	SL read
1	11.178	51.656	3.117	48.566
2	26.619	251.312	6.840	245.685
4	63.1689	1026.43	12.799	1015.33
8	181.226	4123.14	26.359	4093.7

Table 1: Sequential performance on DRAM, SL: Skiplist

dataset size	range HT insert	HM insert	range HT read	HM read
1	52.428509	6.98518	0.729831	0.296202
2	231.641087	14.409046	2.132373	0.75321
4	1030.402723	29.213535	6.469217	1.69313
8	4137.656239	59.144023	23.354112	3.565371

Table 2: Sequential performance on PM, HM: Hashmap

with growing dataset size. The latency is normalized with value when dataset size is 1M for showing scalable performance with growing dataset size. As figure 5 and 6, performance of conventional skiplist is worse than one of RT-Skiplist. When datasize is larger with 0.5M, 1M, 2M and 4M, latency of RT-Skiplist is not extremely larger. However, latency of conventional skiplist is grows linearly. This point means that performance conventional skiplist can be sensitive to dataset size. Moreover, this linearly lowered performance means that this skiplist is not proper design for large key-value store. Through this experiments, we can find various problems about performance of skiplist which is used commonly in key-value store and trial to optimize the skiplist is valid.

4.4 PM experiment

Because we already showed worse performance of conventional skiplist than our scheme in DRAM, we will skip comparison with conventional skiplist in PM. Instead of it, we decide to compare our scheme with hashmap. All key and value stored in hashmap is stored in PM. Like DRAM experiments, for scalability with growing dataset size, figure 7 and 8 shows normalized latency with growing dataset size. The latency is normalized with value when dataset size is 1M. As figure 7 and 8, performance of our design is highly worse than one of hashmap. We think the reason of poor performance is caused by split overhead. By implementation, when split is occurred, if key of entry is higher than new min value, the entry will be migrated to new hashmap and this entry will be deleted in original hashmap. And this split requires loop for entire hashmap to check whether key is smaller than

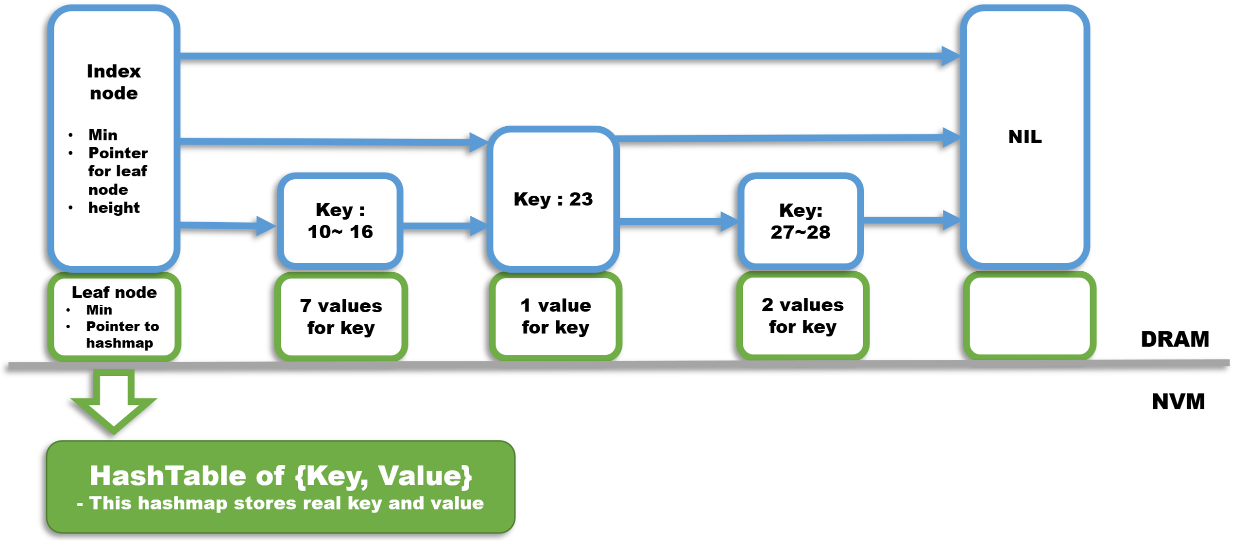

Figure 4: RH-Skiplist

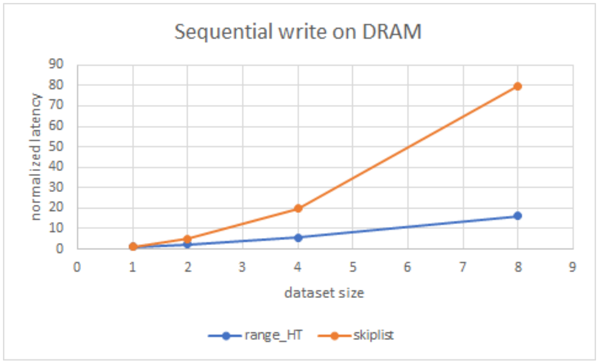

Figure 5: sequential write on DRAM

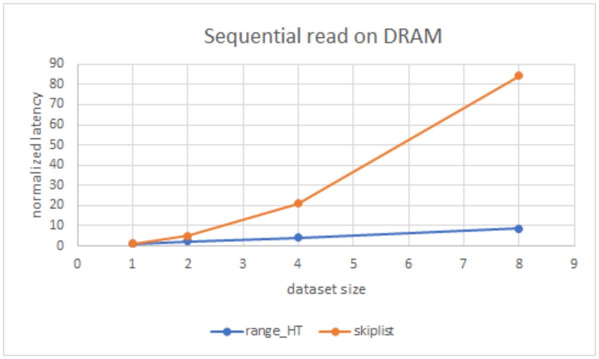

Figure 6: sequential read on DRAM

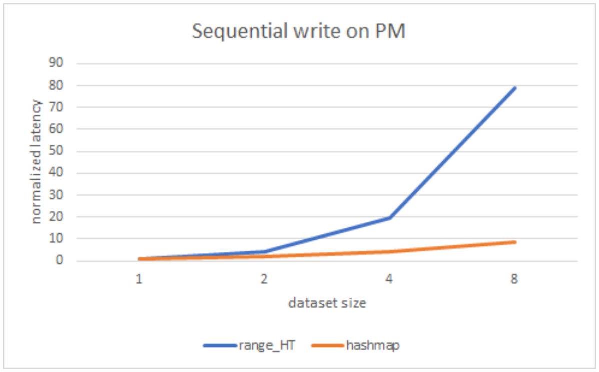

Figure 7: sequential write on PM

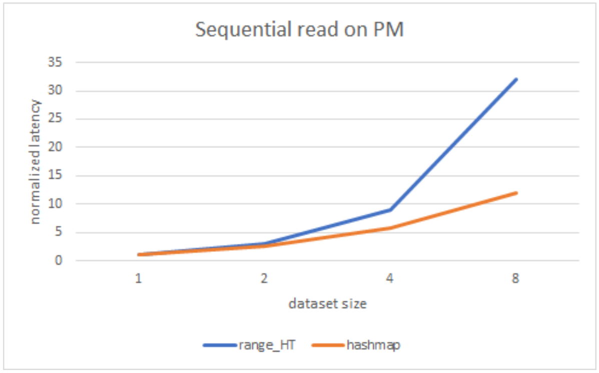

Figure 8: sequential read on PM

new min value or not. We think these points lead poor performance of our scheme. For higher performance, light data structure such as trie will be kept for soring hashmap. By supporting sort for each hashmap, we can make split faster and more efficient. This works remains as future works.

5 CONCLUSION

With developments of hardware, we have to design new software scheme. Moreover, there are problems with traditional hardware in key-value store. Therefore, we have to design new scheme to solve existing problems in key-value store. Combination with hashmap and range skiplist will reduce compaction overhead between L_0 and L_1. Moreover, as we showed, its performance is highly better than traditional skiplist. It means this optimization for skiplist could contribute to improve performance of key-value store. However, we found its performance is worse than performance of traditional hashmap in PM. This lower throughput is caused by split overhead. For relieving the split overhead, we will remain optimization to its performance in PM as future works.

6 FUTURE WORKS

When this design is applied to key-value store, we have to consider modified structure of SSTable which is stored in disk component. Because, the SSTable is not sorted, if one hashmap is considered as one SSTable. Then we have 2 things to make this design compatible with key-value store which uses LSM tree design. First, metadata which stores information of data item should be reconstructed. Secondly, we need another compaction mechanism, since structure of SSTable is different. When compaction work is performed, step for checking whether there is overlapped range between multiple SSTable could be processed efficiently due to sorted SSTable. However, design what we suggest is incompatible with conventional metadata scheme and compaction mechanism. Therefore, we need to come up with new design of metadata and elaborate compaction scheme for combination with key-value store and our scheme. And we found split overhead affected experimental result. For reducing split frequency, we could make size of hashmap larger. However, it can cause higher L_0-L_1 compaction overhead which we want to optimize. Therefore, we have to add light data structure which can sort hashmap again such as Trie. The goal of this work is also to reduce efficiently L_0-L_1 compaction overhead.

REFERENCES

[1] BALMAU, O., DINU, F., ZWAENEPOEL, W., GUPTA, K., CHANDHIRAMOORTHI, R., AND DIDONA, D. SILK: Preventing latency spikes in log-structured merge key-value stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association, pp. 753–766.

[2] FACEBOOK. RocksDB: A Persistent Key-Value Store for Flash and RAM Storage. https://github.com/facebook/rocksdb.

[3] GOOGLE. LevelDB. https://github.com/google/leveldb.

[4] HUAI, Y. Spin-transfer torque MRAM (STT-MRAM): Challenges and prospects. *AAPPS bulletin 18*, 6 (Dec. 2008), 33–40.

[5] INTEL. Breakthrough performance for demanding storage workloads. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-ssd-905p-product-brief.pdf.

[6] INTEL. Optane DC persistent memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[7] KAIYRAKHMET, O., LEE, S., NAM, B., NOH, S. H., AND CHOI, Y. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST)* (2019).

[8] KANNAN, S., BHAT, N., GAVRILOVSKA, A., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. Redesigning lsms for nonvolatile memory with novelsm. In *2018 USENIX Annual Technical Conference (USENIX ATC)* (2018).

[9] LEPERS, B., BALMAU, O., GUPTA, K., AND ZWAENEPOEL, W. Kvell: The design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2019), SOSP '19, ACM, pp. 447–461.

[10] O'NEIL, P., CHENG, E., GAWLICK, D., AND O'NEIL, E. The log-structured merge-tree (lsm-tree). *Acta Inf. 33*, 4 (June 1996), 351–385.

[11] OUKID, I., LASPERAS, J., NICA, A., WILLHALM, T., AND LEHNER, W. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data* (New York, NY, USA, 2016), SIGMOD '16, ACM, pp. 371–386.

[12] PAGH, R., AND RODLER, F. F. Cuckoo hashing. In *Algorithms — ESA 2001* (Berlin, Heidelberg, 2001), F. M. auf der Heide, Ed., Springer Berlin Heidelberg, pp. 121–133.

[13] POBLETE, P., AND VIOLA, A. Robin hood hashing really has constant average search cost and variance in full tables.

[14] PUGH, W. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM 33*, 6 (1990).

[15] SAMSUNG. Ultra-low latency with Samsung Z-NAND SSD. https://www.samsung.com/us/labs/pdfs/collateral/Samsung_Z-NAND_Technology_Brief_v5.pdf.

[16] SÁNCHEZ, C., AND SÁNCHEZ, A. A decidable theory of skiplists of unbounded size and arbitrary height. *CoRR abs/1301.4372* (2013).

[17] XIA, F., JIANG, D., XIONG, J., AND SUN, N. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)* (2017).

[18] YANG, J., WEI, Q., CHEN, C., WANG, C., YONG, K. L., AND HE, B. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *13th USENIX Conference on File and Storage Technologies (FAST)* (2015).