LSM-Tree-based Key-Value SSDs

Submission #78

Abstract

Most Key-Value SSDs (KV-SSDs) use hashing because of its $O(1)$ read latency and the simplicity of its implementation. A closer examination, however, reveals that when a KV-SSD holds a large number of items, it suffers from inconsistent I/O latency and reduced throughput. Instead of hashing, we propose to use log-structured merge-trees (LSM-trees) for KV store management, both to improve performance and to support a richer class of KV operations. Our system, which we call *LSM-KVD*, is tightly integrated into an SSD controller on an FPGA-based SSD platform. It improves the read-latency of LSM-trees without sacrificing its throughput over previous LSM-tree implementations. We show that LSM-KVD reduces the average latency by 42% with guaranteed tail latency and improves I/O throughput by 47%, over hash-based KV-SSDs. We also quantify the tradeoff between read-latency and write-throughput for a given amount of DRAM, thus providing a flexible design space for a wide range of KV clients.

1 Introduction

Key-Value store (KVS) has become a necessary infrastructure for many applications, such as caching, web indexing, and storage systems [7, 9, 15, 27]. Owing to its popularity and huge impact on application performance, serious efforts have been made to optimize KVS at various levels: algorithm [5, 12, 13, 37], system [14, 25, 43], and architecture [6, 8, 44].

Another approach is to offload most of the key-value (KV) functionality onto a storage device, the so-called KV-SSD [20, 22, 26, 44]. By making the storage hardware directly serve KV requests via a driver, KV-SSDs enable clients to bypass the deep software stack in the host. For example, by performing most commonly used operations of KV databases, such as RocksDB [17], in KV-SSDs, we not only improve I/O latency and throughput of the operations but also reduce the CPU and DRAM resource requirements on the host-side.

The idea of KV-SSD is promising but the current proposals and products often provide inconsistent tail latency and throughput degradation. Our experiments with a commercial KV-SSD (see §3.2) reveal that such problems arise primarily due to the limitations of hash-based data structures commonly used in KV-SSDs [14, 20, 22, 26, 42, 44].

The hash-based KV-SSD has a simple architecture – it maintains a hash table in the controller DRAM, each entry of which keeps a key (or a signature) and a pointer to the corresponding value in the flash. By looking up the hash table in the DRAM, it can quickly find the desired key-value pair. However, if the DRAM size is not large enough to hold all the hash entries, parts of the hash table must be stored in flash. This inevitably involves expensive flash accesses to look for a key when it is not found in the DRAM-resident hash table. Even worse, if a hash collision occurs then multiple flash accesses may be required, resulting in long tail latency and drop in throughput.

An alternative to hashing is a *log-structured merge tree* (LSM-tree) [28]. The LSM-tree algorithm stores KV pairs in a multi-level sorted tree, and provides fast range queries and scans. However, it is a write-optimized algorithm, and offers slow read speed which dwarfs fast latency of NAND flash. Compaction, an essential requirement of LSM-tree implementation, is also problematic because it requires sorting [5, 25, 37]. CPU cycles for sorting can become a huge burden on embedded CPUs in SSD controllers. Extra I/O operations to keep KV pairs sorted further deteriorate I/O throughput. Due to these drawbacks, LSM-tree has not been considered an attractive solution for KV-SSDs. Nevertheless, a few key-value storage devices, Kinetic HDDs [16] and LightStore [10], have used the LSM-tree. However, they also suffer the same problems as other LSM-tree based designs.

In this paper, we show that, by optimizing the LSM-tree algorithm and tightly integrating it with an SSD controller, LSM-tree-based KV-SSDs can outperform hash-based designs. Our solution is based on three ideas. First, pin all KV indices of the top levels of the tree to DRAM to speedup reads. This level pinning not only guarantees the worst-case read latency, but improves average latency by reducing flash look-ups for retrieving KV indices. Second, run sorting tasks using HW accelerators, in parallel with I/O tasks to eliminate CPU overheads for compaction. Third, combine level-pinning with key-value separation [25] to reduce I/O overheads for compaction. Collectively these techniques remove the compaction bottleneck and provide high write-throughput. We believe that our techniques can also be applied profitably to general LSM-tree implementations, which may not have the same resource constraints as KV-SSDs do.

Based on these ideas, we have designed a new LSM-tree-based KV-SSD, called *LSM-KVD* and implemented it on an FPGA-based SSD platform [21]. Using micro and YCSB [11] benchmarks, we have shown that LSM-KVD outperforms existing KV-SSD designs in several aspects, including tail read-latency, average read-latency, and I/O throughput. It is well known that the LSM-tree algorithm offers an intrinsic trade-off between read-latency and write-throughput, controlled

by the tree height [12, 13]. We have quantified this tradeoff and shown that for a given DRAM capacity an application can configure our implementation to support its KV performance requirements. Alternatively, the manufacturer can supply more DRAM to support KV applications requiring higher performance. For example, for a write-intensive workload, LSM-KVD improves throughput by 31% by sacrificing average read latency by 7.4% but with guaranteed tails.

Paper Organization: In Section 2, we explain background closely related to this study. Section 3 describes the inherent problems of hash-based KV-SSD designs and analyzes the performance of a product KV-SSD. Section 4 presents an overall design of LSM-KVD, along with optimization techniques. In Section 5, we present experimental results. We conclude in Section 6.

2 Background

2.1 NAND Flash-based SSD

A NAND flash-based SSD is a storage device with multiple I/O channels, each connected to NAND flash chips [38]. Each NAND flash chip is a group of flash blocks, each of which comprises 128-256 flash pages. A flash page is a unit of reads or writes, and its size ranges from 4 KB to 16 KB. NAND flash does not support in-place update, and in order to overwrite new data to a pre-written page, the entire block that the page belongs to has to be copied elsewhere and then erased.

SSDs are designed to support the standard block I/O interface. It exposes a linear array of 4 KB logical blocks which are accessed by block I/O primitives (*e.g.,* `READ` and `WRITE`). A flash translation layer (FTL) in the SSD firmware is responsible for providing the block I/O interface [1]. To hide the out-of-place update nature, the FTL writes incoming data to free flash pages in an append-only manner. To redirect 4 KB logical blocks to free pages, the FTL maintains a mapping table indexed by logical block address (LBA), and each entry points to the corresponding flash page. The mapping table is kept in the controller DRAM and its size is approximately 0.1% of the SSD capacity [32, 36]. For example, for a 4 TB SSD, 4 GB DRAM is required. A mapping table has to be persistent (non-volatile) and is protected by batteries to guard against sudden power failures [4]. Similar to other log-structured systems [30], the FTL has to perform garbage collection (GC) to reclaim free space.

2.2 Key-value SSD

A KV-SSD is a new type of SSDs [22, 40] which provides the key-value interface. KV-SSDs look like a container of key-value objects, where each object is labeled by a unique key and contains an associated value, *i.e.,* data. In contrast to a block-addressed system, both the key and the associated value are of variable sizes. A key can be as long as 255 bytes [40]) or even be a character string, and a value can be as big as 2 MB [40]. In addition to `GET()` and `SET()`, the basic operations to access KV objects, KV-SSDs support a rich set of operations like iterations, range queries, and transactions [22, 23]. A more detailed description can be found in SNIA's KV-SSD specification [40].

Making SSDs support the KV interface requires us to redesign the FTL because the existing table-based translation is not suitable for managing KV objects. A variety of KV-SSD designs have been proposed both in academia (*e.g.,* NVMKV [26], KAML [20], and BlueCache [44]), and commercially (*e.g.,* Samsung's KV-SSD [22]). While software KVS use both hash-based and LSM-tree-based approaches, KV-SSDs with a few exceptions are based primarily on hashing. In this paper, we will describe and analyze both hash-based (§3) and LSM-tree-based (§4) KV-SSDs.

3 Hash-based KV-SSD

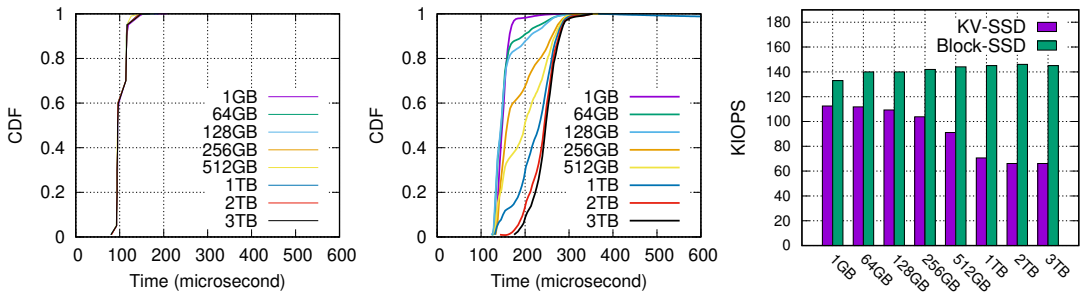
Hash-based KV-SSDs are easy-to-implement and provide $O(1)$ complexity for reading and writing indices. However, as the number of objects stored in KV-SSDs increases, they suffer from inconsistent I/O response times and throughput degradation. In this section, we first explain inherent problems of hash-based designs, and the offer empirical evidence to support our analysis using a commercial hash-based KV-SSD.

3.1 Problems of Hash-based Design

A hash-based KV-SSD maintains a hash table with many buckets in the controller DRAM, where each bucket holds metadata, *i.e.,* a key and a pointer, for a specific KV object in flash [14, 20, 26, 44]. The `SET()` operation on a KV object first calculates a bucket index using a hash function, *i.e.,* `index = hash(key)`. If the corresponding bucket is empty, KV-SSDs update the key field of the bucket and writes the object value to flash. The pointer field of the bucket is then updated to point to the physical location of where the value is written. If there is a hash collision, *i.e.,*the bucket is already occupied by another KV pair, KV-SSDs search the hash table until an empty bucket is found by using a collision resolution strategy such as quadratic probing.

Retrieving a KV object using the `GET()` operation is straightforward. KV-SSDs calculate a bucket index and then looks up the target bucket. If the bucket has a matching key, its value is read from the flash using the pointer in the bucket. Otherwise, KV-SSDs searches for a matching one in the hash table using a resolution strategy.

The hash-based KV-SSD has a simple architecture and offers fast response time for both `GET()` and `SET()`. This simplicity and efficiency, however, assume that the SSD has a large enough DRAM in its controller to hold all the hash buckets. In practice, product SSDs do not have enough DRAM. Suppose that the SSD capacity is 4 TB and it has 4 GB DRAM is available for indexing. Further suppose the key and value sizes are on average 32 B and 1 KB, respectively [2]. If the



(a) CDF of read latency (Block-SSD) (b) CDF of read latency (KV-SSD) (c) Read Throughput

Figure 1: Experimental results with Block-SSD and KV-SSD

number of buckets is 2^{32} ($= 2^{42}/2^{10}$) and the bucket size is 36 B (32 B for key and 4 B for pointer), we will need 144 GB of DRAM to hold the complete hash table!

To reduce the DRAM requirement, some use signatures [14, 20, 26, 44]. Given an object key, KV-SSD computes its signature using another hash function and keeps it in the bucket, instead of the key. The exact key and its value are written to the flash. Usually, a 16-bit signature is used, so that the DRAM size reduces to 24 GB, which is still a huge requirement for an SSD. To run KV-SSD with limited DRAM, we can borrow an idea from demand-based FTLs [18] which keep only popular buckets in DRAM, while storing the rest in the flash. This, however, causes extra flash reads while retrieving KV objects. If a designated bucket is not available in DRAM (*i.e., hash-table miss*), we have to fetch the bucket from the flash first to find the location of a desired KV object. In addition, using signatures creates another problem – *signature collision* which happens when different keys have the same signature. If a KV object pointed to by the bucket actually has a different key, we need to lookup the hash table again until a match is found, degrading read latency significantly.

In summary, if a signature size is set large enough (*e.g.,* 32-64 bits) [20, 44], signature collision almost disappears. But, only a small number of buckets are kept in DRAM, and hash-table misses are frequent. With a small signature size (*e.g.,* 0-8 bits) [26], many buckets can be loaded in DRAM, but KV-SSD performance suffers seriously from signature collision.

3.2 Experiments with Product KV-SSD

To confirm our analysis, we carried out experiments using a KV-SSD product (3.84 TB PM983 [33, 34]). The internal architecture of the KV-SSD model is not disclosed to the public, except that it is based on hash-based data structures [22].

Experimental Setup: The number of KV objects stored in a KV-SSD directly affects its performance. This is because, as the number of KV objects increases, more memory is required for the hash-map, which may result in frequent signature collision and/or hash-map misses. Based on this fact, we conduct experiments in two phases: *load* and *run*.

The load phase creates an object pool on an empty SSD by writing a bulk of KV objects. The default key and value sizes were 16 B and 1 KB, respectively. Various sizes of object pools, from 1 GB to 3 TB, were created. For example, a 1GB-pool contained 1M KV objects. During the run phase, we executed KVBench [31] on the object pools. KVBench generated random `GET` requests and ran for 10 minutes. There were no requests to non-existing KV pairs. We measured read latency and throughput. When measuring read latency, the queue depth (QD) was set to 1 to eliminate noises from queueing delays. To measure the maximum throughput the KV-SSD can achieve, QD was set to 64.

In order to assess the impact of the hash-based algorithm on performance objectively, we conducted the same set of experiments on a block SSD. The Block-SSD model is Samsung's 4TB 860EVO. The load phase first made the SSD empty and then wrote different amounts of data – from 1 GB to 3 TB. During the run phase, we executed FIO [3] which issued 1 KB random reads for 10 minutes. Read latency and read throughput were measured with QD=1 and QD=64, respectively.

Result: Figures 1(a) and (c) show the CDF graph of read latency and the read throughput of the Block-SSD, respectively. The Block-SSD exhibited consistent read latency, regardless of the amount of data stored in it. The 99.99^{th} tail latency was around 150∼208 μs and did not change much depending on the space utilization. The Block-SSD also provided stable read throughput for every case, maintaining 140 KIOPS. The Block-SSD has enough DRAM to load all the mapping table in the controller DRAM. Thus, the amount of data (*i.e.,* logical blocks) stored in it didn't affect its performance.

Figures 1(b) and (c) shows the read latency and throughput of the KV-SSD. The KV-SSD showed inconsistent read response times as the number of objects stored got larger. In particular, the average read latency increased from 149.49 μs to 245.31 μs (1.64X), when the object pool-size was increased from 1 GB to 3 TB. We also observed very long tail latency. For the 99.99th percentile, the KV-SSD tail latency increased from 323 μs to 1020 μs when object pool was increased from 1 TB to 3 TB. Even worse, as depicted in Figure 1(c), the read throughput continued to drop as the object pool size got larger, for example, from 112 KIOPS to 64 KIOPS.

Although we could not analyze the internals of the KV-

SSD firmware because it was proprietary, the results strongly suggest that the current hash-based design offers inconsistent latency with long tails and fails to fully utilize the internal throughput of NAND flash as the KV pool gets bigger.

It is possible that the degraded performance of hash-based KV-SSDs is primarily due to their inefficient collision resolution policies. There are more advanced hashing strategies, such as Cuckoo hashing [29] and Hopscotch hashing [19], which provide constant-time worst-case lookups and may avoid the tail latency problem. This benefit, however, comes at the cost of degraded write speed and/or frequent rehashing. It is worth noting that advanced hashing algorithms cannot support range and scan operations as well.

4 Design of LSM-KVD

We first explain some basics of how LSM-trees are used to implement KVS (§4.1). We then introduce a novel LSM-tree-based KV-SSD design, called LSM-KVD. LSM-KVD is designed to meet SNIA's KV-SSD specification [40], and supports variable-size keys (16B∼128B) and values (1KB∼2MB), and a rich set of KV operations (except for a few features like namespaces). Like KV-SSDs, LSM-KVD is able to guarantee durability and atomicity by leveraging built-in capacitors that make the controller DRAM persistent, without a write-ahead log [4, 22, 35]. Lack of space does not permit us to describe the details of all the operations; instead, we focus on explaining our architecture (§4.2), and how it improves average read latency, shortens tail latency (§4.3), and eliminates compaction overheads (§4.5).

4.1 Basics of LSM-Tree

Log-structured merge trees (LSM-trees) are widely used to implement persistent key-value stores [17, 24]. The LSM-tree algorithm maintains multiple levels, L_0, L_1, ..., and, L_{n-1}, where n is the number of levels. The level 0, L_0, is kept in DRAM as a write buffer, whereas the rest are stored in persistent storage media (*e.g.,* flash). In LSM-trees, the levels are organized so that a lower level is T times larger (the size factor) than a higher one. The LSM-tree has two unique properties: **Property #1** which says that for each level, KV objects are unique and kept sorted by their keys, and **Property #2** which says that the key range of one level may overlap the key range of other levels (see Figure 2).

For incoming writes, the KV objects are first buffered in L_0. Once L_0 becomes full, KV objects are flushed out to L_1. All the objects in L_0 are written to L_1 in an append-only manner, maximally utilizing the write performance of SSDs. Similarly, once L_i becomes full, its KV objects are evicted to L_{i+1}. Since the key ranges of adjacent levels may overlap, flushing out KV objects from a higher level to a lower level has to be done in a manner that not to violate Property #1. Therefore, the LSM-tree algorithm performs a process called *compaction* while flushing KV objects to a lower level. Compaction reads

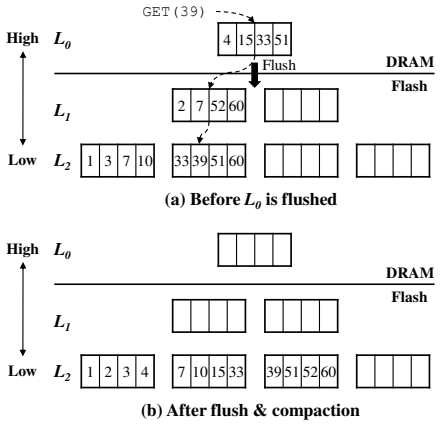


Figure 2: LSM-tree organization ($n = 3$, $T = 2$). A rectangle represents a KV object and the number inside is the key.

all the objects in two adjacent levels and sorts them in the memory. It then writes the fully sorted objects to next lower level as shown in Figure 2(b).

Compaction incurs a huge I/O overhead because it reads all the data from two levels and then writes back the merged data. Since only keys need to be sorted during compaction, this overhead can be mitigated by separating keys from values, and avoiding moving values which are not affected (see Wisckey [25]). Searching for a key at any specific level is fast since all the KV pairs are sorted. Finding the desired key in the entire tree, however, requires looking in multiple levels because key ranges at different levels may overlap (Property #2). In the worst case, all levels have to be searched as shown by `GET(39)` in Figure 2(a). Since KV indices in the tree are stored in the persistent storage, the number of disk lookups is $O(n-1)$ (Note: L_0 is excluded since it stays in DRAM).

In the following subsections, we introduce our LSM-tree-based KV-SSD design, called LSM-KVD, and show how it mitigates these problems.

4.2 Architecture of LSM-KVD

LSM-KVD has four types of data structures to implement the LSM-tree algorithm (Figure 3): a *skiplist* and *level lists*, which all reside in DRAM, and *meta segments* and *data segments*, which all reside in flash. We describe how these data structures are managed and then estimate its DRAM requirement.

The skiplist corresponds to L_0 in the LSM-tree algorithm, and holds 64 MB of KV objects which is large enough to fully utilize the parallelism of a NAND chip array. Each skiplist entry has four fields: <key size, key, value size, value>, and the skiplist is kept sorted by the keys.

In L_1 (in fact in all the lower levels), following the idea of Wisckey [25], we separate the keys and values into meta segments and data segments, respectively. Each element in the meta segment contains a key and a pointer to its associated value in the data segment. While not shown in Figure 3, the meta segment has a header that keeps track of the size of each

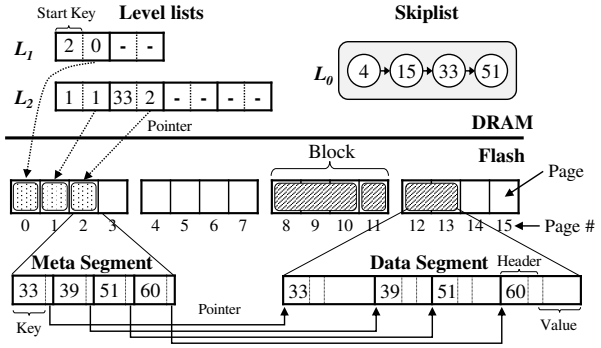


Figure 3: An overall architecture of LSM-KVD with its key data structures in DRAM and flash. The tree hierarchy and KV objects are identical to those of Figure 2(a).

key for fast scanning. The data segment, in addition to the values, keeps keys and their sizes. As we will see in §4.6, this information is needed to support garbage collection (GC). The size of a meta segment is fixed (*e.g.,* 32 KB). A data segment can be of any size – it is like a log containing KV objects pointed to by meta segments.

In our design, each meta segment belongs to one of the levels in the tree. LSM-KVD maintains another in-memory data structure, *level lists*, which keeps track of meta segments at every level in the flash. Each level list is an array of pointers (4 B each) to meta segments belonging to the level. Each array entry also contains the start key (a variable size from 16 B to 128 B) of the meta segment to facilitate searching. Figure 3 illustrates examples of how the level lists point to meta segments which, in turn, point to actual values.

DRAM Requirement for Level Lists: Let's estimate DRAM requirement of LSM-KVD. We assume that an SSD capacity is 4 TB and each meta segment is 32 KB. As we said earlier, the DRAM for the skiplist is fixed to 64 MB, but that for the level lists varies depending on key and value sizes.

First consider the case where the average sizes of keys and values are 32 B and 1 KB, respectively (This is typical for the workloads used for large-scale key-value stores [2]). In a meta segment, each entry is 36 B (32 B key and 4 B pointer). Since the size of a meta segment is 32 KB, each segment can hold 910 <key, value pointer> pairs. In a 4 TB SSD, there can exist 2^{32} 1-KB objects, and thus the number of meta segments in the flash is about 4.7M (= $2^{32}/910$). Each of these must be pointed to by some level list. Level list entries are 36 B each (32 B start key and 4 B pointer). Thus, only 162 MB (= 4.7M×36 B) DRAM is needed to hold all the level lists. For the worst-case scenario, where all the objects have the maximum key size of 128 B, the level lists would require about 2.1 GB DRAM. This can still easily fit in 4 GB DRAM that 4 TB SSD has for mapping. We note is passing that the DRAM requirements to hold level lists is much smaller than 24 GB, the DRAM needed for hash-based designs with 16-bit signatures. LSM-KVD exploits this saved memory space to optimize read latency as we discuss next.

The meta segment size of 32 KB is chosen to minimize DRAM requirement for level lists. Since it is larger than a 16 KB page size of recent NAND devices [39,41], fetching meta segments requires two page reads. This problem can be easily mitigated by writing a meta segment to two pages in different channels. Later, the two pages can be read in parallel from the flash with negligible overhead.

4.3 Optimizing Read Latency

Retrieving a KV object from LSM-KVD requires multiple flash lookups. Given a `GET` request (*e.g.,* `GET(39)`), LSM-KVD first looks up the skiplist (L_0). If a matched one is not found, it has to go down to L_1. LSM-KVD looks up the L_1 level list to get the location of a meta segment that *may* have an index for a desired KV object. After reading it from the flash (*e.g.,* the page 0 in Figure 3), LSM-KVD knows whether the meta segment actually has a KV object to read. If it does not have, LSM-KVD has to keep searching lower levels (L_2, ... , L_{n-1}), fetching meta segments from the flash (*e.g.,* the page 2 in the example of Figure 3). In the worst case, $O(n-1)$ flash lookups are required to access a KV object.

Some suggest using (*i*) bloom filters [12,13] to avoid useless lookups on levels that do not have desired keys or (*ii*) caching popular entries in available DRAM. LSM-KVD partially uses them. But, those approaches are only effective in improving average read latency, not the worst-case latency.

In order to further improve the average and worst-case latency, LSM-KVD adopts a more aggressive strategy: *level pinning*. If the LSM-tree has n levels, LSM-KVD keeps meta segments for top-k levels ($k \leq n-1$) in DRAM. To process a `GET` request, it first searches for a key in top-k levels in DRAM. Only when a key is not found there, it lookups the rest of levels resident in the flash. With the level pinning, the number of the worst-case flash lookups is reduced to $O(n-k-1)$.

DRAM Requirement for Pinning Top-k Levels: One might think that the level pinning would require large amounts of DRAM, but this is not the case. In the LSM-tree, a higher level (L_i) is T times smaller than a lower level (L_{i+1}). This implies that a level size increases exponentially by a factor of T. For example, in the 4 TB SSD organized with 5 levels, the amount of DRAM required to pin meta segments for L_1, L_2, L_3, and L_4 are 1.47 MB, 67.16 MB, 3.04 GB, and 140.84 GB, respectively. Assuming that more than 3 GB DRAM is available, meta segments for L_1, L_2, and L_3 can be entirely loaded in DRAM. Thus, only one flash lookup in the worst-case is needed! Be advised that a more comprehensive analysis of the DRAM requirement for pinning is given in §4.4.

Advantage of Pinning Top-k Levels: To understand the impact of the level pinning, we carried out simulations using two read-dominant workloads, Uniform and Zipfian (`YCSB-D`), which have different distributions. We also compared the level pinning with bloom filters and caching. To reduce the simulation time, we scaled down an SSD capacity to 64 GB.

Table 1: A comparison of flash lookup counts of caching, bloom filter, and level pinning. In the table, the '0' lookup means that an index lookup is hit by L_0 or DRAM.

# of lookups	Caching (LRU)	Bloom filter	Level pinning		
			$k=1$	$k=2$	$k=3$
0	0.003%	0.003%	0.008%	0.248%	4.053%
1	0.018%	95.401%	97.687%	97.067%	95.947%
2	0.368%	4.153%	2.277%	2.685%	-
3	7.773%	0.421%	0.029%	-	-
4	91.838%	0.022%	-	-	-

(a) Uniform distribution

# of lookups	Caching (LRU)	Bloom filter	Level pinning		
			$k=1$	$k=2$	$k=3$
0	27.835%	16.265%	26.908%	47.072%	64.135%
1	20.693%	81.886%	68.948%	44.000%	35.865%
2	14.104%	1.741%	4.084%	8.927%	-
3	20.061%	0.105%	0.060%	-	-
4	17.307%	0.003%	-	-	-

(b) Zipfian distribution (YCSB-D)

64 MB DRAM was assumed to be available for leveling pinning, bloom filtering, or caching. The height of the tree was set to 5. For the level pinning, k varied from 1 to 3. For $k=3$, the level pinning used up 64 MB DRAM. For $k=2$ and $k=1$, 192 KB and 4.34 MB were required, respectively. The rest of DRAM was used as bloom filters for non-pinned levels. The bloom filter and the caching used all of 64 MB DRAM for filtering and caching. Note that the bloom filter is identical to the level pinning with $k=0$.

Table 1 shows our simulation results. The level pinning exhibited the guaranteed number of worst-case flash lookups, regardless of the workloads. Under Uniform, the bloom filter performed fairly well, offering almost $O(1)$ lookups on average. However, 4.57% of the total requests required more than two flash lookups, which increased the tail latency. In the absence of locality, caching showed the worst performance. With the Zipfian distribution, the level pinning with $k=3$ showed the best performance; 64.135% of the requests were served by L_0 or pinned levels. The caching didn't perform well – only 27.835% were hit by DRAM owing to the small cache size. In the bloom filter, 16.265% were hit by L_0, which helped reduce tail latency. But, the level pinning with $k=1$ showed shorter tail latency and might provide better average latency with a higher hit ratio (*e.g.,* 26.9%).

In addition to reducing read latency, the level pinning enables memory-performance tradeoff, allowing users to choose the best option for meeting their performance demands. We discuss this next.

4.4 Speed and Memory Tradeoff

LSM-trees offer an intrinsic tradeoff between read latency and average write throughput. As pointed out before, its worst-case read latency is $O(n-1)$. The average write throughput is mainly decided by a write amplification factor (WA), which is a ratio of the amount of data actually written to the amount

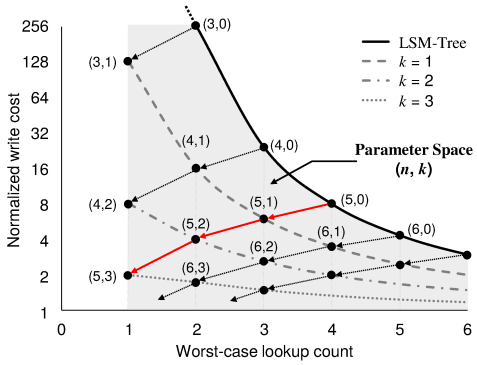


Figure 4: The parameter space of LSM-tree and LSM-KVD depending on n and k.

of data written to the LSM-tree. WA is caused mostly by compaction. According to the published analysis [12, 13], the write throughput is expressed as $O((n-1)\cdot\frac{R^{1/(n-1)}}{B})$[1] where R is the number of meta segments in an SSD and B is the average number of keys in a meta segment. Based on a 4 TB SSD with a 32 B average key size, R and B are 4,719,744 and 910. Unless otherwise stated, the same numbers are used throughout the paper.

Two parameters, B and R, are independent of n because they are decided by the capacity of SSDs and input workloads. Given the same SSD capacity and the workload, therefore, we can estimate expected read and write performance depending on n using the equations. A outer curve with a bold line in Figure 4 is the performance tradeoff depending on n. If n is set 2, the LSM-tree exhibits the best read latency (*i.e.,* $O(1)$), but write throughput degrades. As n gets larger, write throughput improves, but read latency increases.

The level pinning creates a broader range of the parameter space. In LSM-KVD, the read latency and write throughput become $O(n-k-1)$ and $O((n-k-1)\cdot\frac{R^{1/(n-1)}}{B})$, respectively. The inner lines in Figure 4 illustrate various combinations of read latency and write throughput of LSM-KVD, depending on k and n. As noticed, the level pinning improves write throughput as well. This is because, by pinning meta segments in DRAM, the number of I/Os for writing meta segments to the flash is reduced.

As discussed earlier, pinning requires additional memory. The memory requirement is modeled as $\sum_{i=1}^{k}(R^{1/(n-1)})^i\cdot M$, where M is the size of a meta segment. Figure 5 shows the DRAM requirement depending on n and k.

This broad parameter space makes it much easier for us to satisfy clients' diverse performance demands. Suppose that users want extremely short read latency as well as moderate write throughput. The LSM-tree is able to offer *either* short read latency (*i.e.,* $n=2$) *or* moderate write throughput (*i.e.,* $n=5$), but not both. In LSM-KVD, the height of the tree

[1] This equation is from Eq. (1) of [12] that models the tradeoff of the original LSM-Tree.

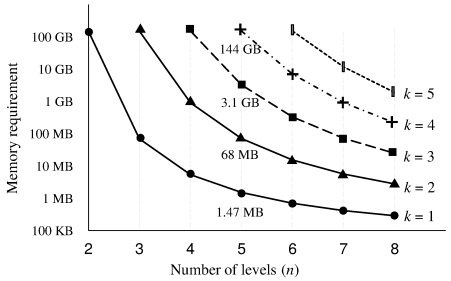


Figure 5: The requirement of DRAM depending on n and k.

is set to 5 for good write throughput, but by pinning top 3 levels, L_1, L_2, and L_3, short read latency can be guaranteed (see red lines in Figure 4). Such a decision should be made carefully considering available memory budget. Since the data structures of LSM-KVD are memory efficient and modern SSDs are equipped with large DRAM, pinning several levels of the tree is feasible enough.

4.5 Optimizing Compaction

To keep each level sorted, LSM-KVD performs compaction, which involves extra I/Os and CPU cycles. We first explain how LSM-KVD performs compaction using the examples in Figures 2, 3, and 6. Figure 6 is the data layout after the compaction. Before flushing out L_0, LSM-KVD fetches the meta segment from L_1 (*i.e.,* the page 0 in Figures 3 and 6) and sorts KV indices of L_0 and L_1 in the memory, which creates two sorted meta segments. The sorted meta segments are then flushed out to L_1 (*i.e.,* the pages 3 and 4 in Figure 6). The level lists are then updated to point to the new meta segments in L_1. LSM-KVD recognizes that L_1 becomes full, and thus flushes out L_1 to L_2. To do this, LSM-KVD reads two meta segments from each of L_1 and L_2 (the pages 1∼4 in Figure 6), sorts them, and finally writes three sorted segments to L_2 (*i.e.,* the pages 5∼7). If the same key exists in two different levels, one that belongs to a lower level is discarded since it was updated by a new one (*e.g.,* '7', '33', '51', and '60' in Figure 2). Note that if L_1 and L_2 are pinned to DRAM, it is unnecessary to read/write their meta segments from flash.

By separating keys from values, LSM-KVD only needs to read and write meta segments for the compaction. Moreover, for meta segments pinned to DRAM, LSM-KVD does not need to issue any I/Os since they can be directly updated in the memory. While it is effective, this compaction strategy moves the bottleneck from I/O to CPU. That is, sorting KV indices on the SSD controller becomes a burden. We conducted an experiment with a 64 GB dataset where uniformly random KV pairs were written. As shown in Figure 7, merge sorting accounted for 70% (416.6 seconds) of the total compaction time (597 seconds). One way to hide CPU overheads is to run CPU and I/O tasks in parallel [17]. However, this resulted in only 10% reduction in compaction time (see Figure 7).

We address this problem by offloading merge-sorting tasks

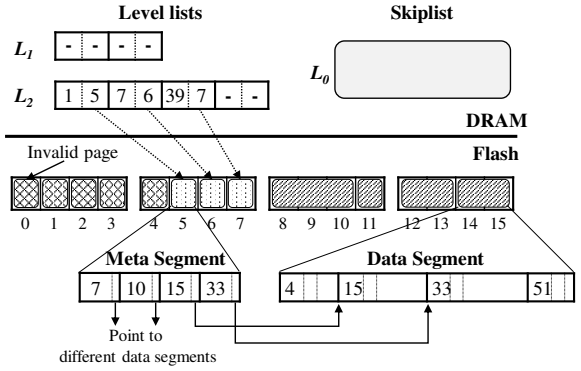


Figure 6: The DRAM and flash layouts of LSM-KVD after L_0 (in Figure 3) is flushed out with compaction. The tree hierarchy and KV objects are identical to those of Figure 2(b).

to a special hardware accelerator in the SSD controller. Compaction is just like merging two sorted lists of KV indices into a single sorted list and a hardware accelerator placed between the flash and the host data bus can easily accomplish this. The compaction is done as we read meta segments from flash without losing any bandwidth as shown in Figure 8.

To support the HW accelerator, LSM-KVD prepares two input lists that contain flash or memory locations of meta segments for L_i and L_{i+1}, which are involved in compaction. It also prepares another list containing locations (flash pages or DRAM addresses) where sorted segments are stored. When the compaction starts, the HW accelerator reads KV indices from L_i and L_{i+1} meta segments, sorts them, and writes sorted indices to designated DRAM addresses or flash pages in a pipelined manner. As Figure 7 shows, the HW accelerator dramatically reduced the compaction time (209 seconds).

4.6 Garbage Collection

The LSM-tree is designed to append all the data to storage media. KV objects which were deleted or updated by new ones are discarded from the tree as part of compaction process (see §4.5). As compaction is repeated, therefore, obsolete data which are not pointed to by the tree are accumulated in the flash, which must be erased by GC later.

There are three types of obsolete data that are created after compaction. The first type is a meta segment. While performing compaction, LSM-KVD writes new meta segments that replace old ones. For example, the meta segments stored in the pages 0, 1, and 2 in Figure 6 are not managed by the tree any more since they contain old indices. The second type is

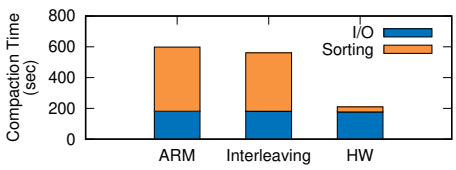


Figure 7: I/O and sorting times for compaction

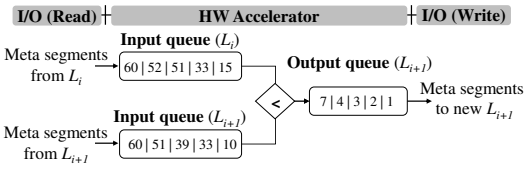


Figure 8: Merge sorting of L_i and L_{i+1}

an outdated KV object which was updated by a new one. It is discarded from the tree during compaction (see §4.5), but their data are still stored in a data segment(s). The final one is a deleted KV object. It is also discarded like a old KV object during compaction, but its value is still in a data segment(s).

To erase obsolete data and to keep supplying free space, LSM-KVD triggers GC when free space is under a certain threshold. It selects a victim block, copies valid pages to a free block, and erases the victim. LSM-KVD should move valid pages differently depending on their types. If a page to move has a meta segment (*e.g.,* the page 5 in Figure 6), LSM-KVD first retrieves the start key from the segment. Using it, LSM-KVD finds a corresponding entry in the level lists. After moving the page to a free page, it updates the entry so that it points to a new flash page.

Moving a page belonging to a data segment requires more efforts. As noted in §4.2, data segments keep metadata, keys and sizes, and thus LSM-KVD can extract key numbers of victim KV pairs to move. Using these numbers, LSM-KVD finds meta segments that point to the KV pairs. After copying valid pages (having several KV pairs) to a free block, it updates associated meta segments in the flash, so that they points to new locations of the values. Updating meta segments inevitably creates another obsolete meta segments. To avoid this, LSM-KVD writes victim KV pairs to L_0 again, just invalidating the pages and erasing the victim block. Corresponding meta segments now point to wrong flash pages erased by GC, but they will be discarded during compaction later. This approach increases compaction costs, but greatly reduces GC costs by reducing meta segment updates. This is because victim KV pairs rewritten to L_0 are coalesced with neighboring KV pairs and then are written to the same meta segment together.

5 Experiments

We present experimental results on LSM-KVD. Particularly, we seek to answer the following questions: (*i*) does the level pinning improves both read latency and write throughput along with shorter tails?, (*ii*) is the HW sorter effective to reduce the compaction cost?, (*iii*) what is the impact of GC on performance? and (*iv*) is the tradeoff nature of LSM-KVD useful to improve performance or save the memory?

5.1 Experimental Setup

We implemented the LSM-KVD software and hardware accelerator on our FPGA-based SSD platform that has a quad-core ARM Cortex-A53 CPU running at 1.2 GHz with 16 GB DRAM. This hardware specification is almost equivalent to those of recent SSDs. The SSD platform has a 256 GB custom flash array card that provides 1.2 GB/s for reads and 500 MB/s for writes. The size of a page is 8 KB, and the number of pages per block is 256. The read and write latencies are 120 μs and 350 μs, respectively. Our SSD platform was connected to a host through 10 GbE (1.25 GB/s) whose bandwidth was high enough to saturate the maximum throughput of the flash array card. The I/O queue depth was set to 64, which was sufficient to fully utilize the parallelism of 8-channel and 8-way in our flash array card. For fast evaluation, we scaled down the SSD capacity to 64 GB. Available DRAM for KV indexing was set to 64 MB – 0.1% of the SSD capacity.

We evaluated LSM-KVD using seven workloads from YCSB, a realistic cloud benchmark [11]. The details of the workloads are described in Table 2. Default key and value sizes were set to 32 B and 1 KB, respectively, representing average KV workload [2]. We first created a 44 GB KV pool on the 64 GB SSD ('Load' in Table 2) – total 44M unique KV pairs were written. Then, we ran each workload ('A'∼'F' in Table 2) on the loaded data set. With 44 GB data, the storage utilization was about 70%. On the host side, 64 YCSB clients ran simultaneously to generate enough KV requests.

To compare with LSM-KVD, we implemented a hash-based KV-SSD based on what we described in §3.1. Three different signature sizes, 0-bit (no signature), 8-bit, and 32-bit, were used, which are denoted by `Hash(0)`, `Hash(8)`, and `Hash(32)`, respectively. Each hash bucket of `Hash(0)` didn't hold any signatures, so its hash table size was small, 256 MB (=2^{26}· 4B). `Hash(8)` and `Hash(32)` had larger signatures, 8-bit and 32-bit, and thus their hash table sizes were 320 MB and 512 MB, respectively. The 64 MB of DRAM for indexing was not large enough to hold the entire hash table. Therefore, the hash-based KV-SSDs keeps only popular buckets in DRAM using the LRU replacement policy.

For LSM-KVD, the number of levels of the tree was set to 5, and k was set to 3 by default. Three LSM-KVD configurations were evaluated: (*i*) one with no leveling pinning (`LSM-KVD(5,0)`), (*ii*) another with $k = 3$ but no HW accelerator (`LSM-KVD(5,3)`), and (*iii*) the other with $k = 3$ and HW accelerator (`LSM-KVD(5,3)+`).

For all the three, the size of a meta segment was the same as an 8 KB page size. This was because our system didn't support the plane-level parallelism. With 8 KB meta segments, the amount of DRAM for the level lists was 10 MB. The rest of DRAM, 54 MB, can be used for `LSM-KVD(5,3)` and `LSM-KVD(5,3)+` to pin levels. Unfortunately, 54 MB is so

Table 2: A summary of YCSB workloads

	Load	A	B	C	D	E	F
R:W ratio	0:100	50:50	95:5	100:0	95:5	95:5	50:50
Query type	Point					Range	Point
KV distribution	Uniform	Zipfian			Latest [11]	Zipfian	

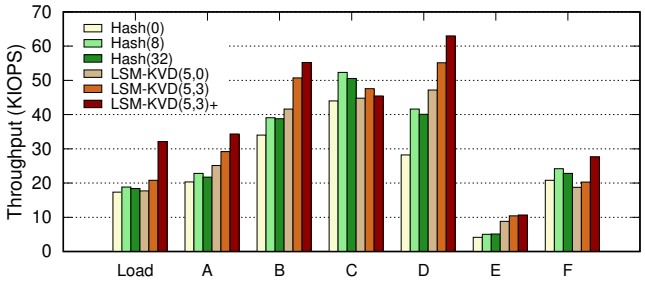
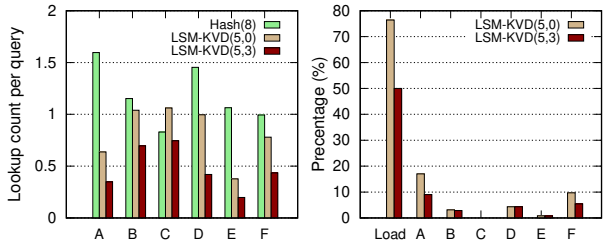


Figure 9: Throughput comparison of KV-SSD and LSM-KVD



(a) Lookup count per query (b) Compaction cost

Figure 10: Flash lookup counts (a) and compaction cost (b)

huge for pinning two levels (4 MB), but too small to pin three levels (100 MB). To accommodate the top-3 levels in 54 MB DRAM, we assigned more meta segments to the last level which are not pinned to DRAM. This slightly increases compaction costs, but does not distort experimental results much. LSM-KVD without the level pinning, `LSM-KVD(5,0)`, used 54 MB of DRAM for bloom filters.

5.2 Performance Analysis

YCSB Throughput: We measured IOPS of the hash-based KV-SSDs and LSM-KVD while running YCSB. Figure 9 shows the results. LSM-KVD outperformed the hash-based ones, providing 47% higher IOPS, on average. Particularly, for YCSB-E with range queries, LSM-KVD showed 71∼111% higher IOPS thanks to LSM-tree's sorted nature.

Among the three KV-SSDs, `Hash(8)` showed the best throughput. Their performance is decided by a table hit ratio and a collision rate. Thanks to a small table size, `Hash(0)` showed the highest hit ratio, 49%, but there were no huge differences from those of `Hash(8)`, and `Hash(32)` which exhibited 48% and 47% hit ratios, respectively. Depending on the signature size, `Hash(0)`, `Hash(8)`, and `Hash(32)` had different collision ratios, 44%, 1%, and 0.4%. `Hash(8)` had a sufficiently low collision ratio because the number of unique keys in our setup is 2^{26}. Thus, a 8-bit signature is large enough to avoid frequent hash collisions. If the SSD capacity gets larger, a collision rate will be higher. Owing to a large bucket size with 32-bit signatures, `Hash(32)` involved more page writes to update the in-flash hash table, resulting in slightly poorer throughput than `Hash(8)`.

`LSM-KVD(5,0)` achieved similar or slightly higher throughput than the hash-based ones. By pinning three levels, `LSM-KVD(5,3)` exhibited higher IOPS than `LSM-KVD(5,0)`. The level pinning not only reduced flash lookups, but also mitigated compaction costs by updating meta segments in DRAM. Hence, across all the workloads, its benefit was noticeable. Figure 10 provides the detailed results of the reductions in flash lookups and compaction I/Os. As shown in Figure 10(a), `LSM-KVD(5,3)` reduced the number of flash lookups per query by 57% and 63% compared to `LSM-KVD(5,0)` and `Hash(8)`, respectively. `Hash(8)` was badly affected from hash table misses and signature collision, so it required many flash lookups. Using bloom filters, `LSM-KVD(5,0)` reduced the number of lookups but it was less effective than pinning levels. Figures 10(b) shows the percentage of compaction I/O count in the total I/O count. For write-intensive workloads (*i.e.,* Load, YCSB-A, and F), the compaction I/Os of `LSM-KVD(5,0)` accounted for about 23.28% of the total I/Os. `LSM-KVD(5,3)` reduced it to 13.4% with the level pinning.

The number of compaction I/Os did not take a great portion of the total I/Os. However, because of its high CPU utilization, it degraded overall I/O throughput. As we can see in Figure 9, by eliminating high CPU overhead for sorting, `LSM-KVD(5,3)+` further improved throughputs of Load, YCSB-A, and F by 54%, 18%, and 37% in comparison to `LSM-KVD(5,3)`. Except for YCSB-C with 100% reads, we also observed 2%∼14% improvements with the HW sorter in YCSB-B, D, and E since non-trivial writes were issued.

For YCSB-C, the hash-based KV-SSDs performed slightly better than LSM-KVD. YCSB-C is a 100% read workload with moderate locality. By caching popular indices in available DRAM, they benefited from the locality of KV requests, showing a 53.5 hash-table hit ratio. LSM-KVD is not aware of the presence of locality. It just keeps KV pairs in levels according to their arrival times. Thus, even if KV pairs have high temporal locality, they could stay in the lowest level all the times unless they are written again. Similarly, YCSB-F had high temporal locality, so the hash-based KV-SSDs achieved a high hit ratio, 67.14%, showing fairly good throughput.

YCSB Read Latency: Figure 11 shows CDF graphs of read response times of the hash KV-SSD and LSM-KVD. Table 3 also lists average, 99^{th}, 99.9^{th}, and 99.99^{th} read latency

Table 3: A comparison of average and tail latency (unit: μs)

	Percentile	A	B	C	D	E	F
`Hash(8)`	Average	410	573	592	501	5,628	370
	99^{th}	2,180	2,550	2,900	3,030	17,550	1,850
	99.9^{th}	4,180	4,600	5,710	5,090	25,360	3,260
	99.99^{th}	9,430	9,340	9,830	7,530	34,420	5,180
`LSM-KVD(5,0)`	Average	327	488	835	407	3,095	365
	99^{th}	710	1,190	2,050	1,310	5,800	760
	99.9^{th}	1,340	1,620	3,020	1,620	7,070	1,220
	99.99^{th}	2,370	2,260	4,550	2,160	8,460	2,240
`LSM-KVD(5,3)`	Average	247	400	653	233	2,910	283
	99^{th}	510	970	1,400	850	5,380	560
	99.9^{th}	750	1,290	2,170	1,280	6,440	850
	99.99^{th}	1,670	1,860	2,730	1,670	7,410	1,600

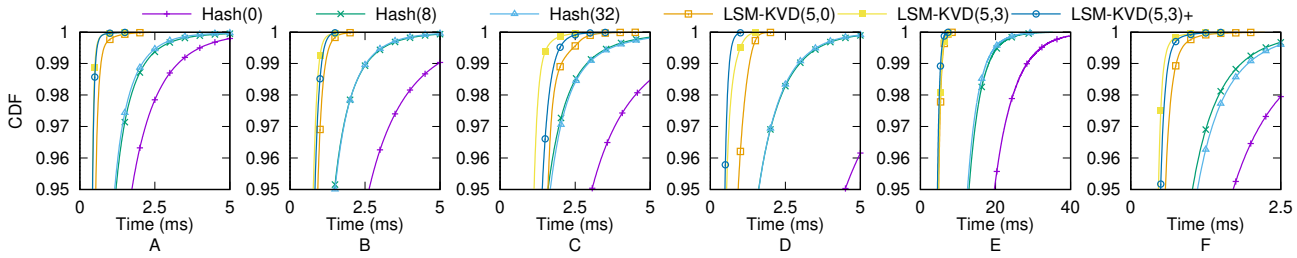


Figure 11: CDF graphs of read latency of the hash-based KV-SSD and LSM-KVD

of `Hash(8)`, `LSM-KVD(5,0)`, and `LSM-KVD(5,3)`. As expected, `LSM-KVD(5,3)` and `LSM-KVD(5,3)+` showed shorter average latency with short tails compared to the others. Thanks to bloom filters, `LSM-KVD(5,0)` performed fairly well compared to hash-based ones, but had long tails since it couldn't benefit from the level pinning. All the hash-based KV-SSDs suffered from long tails. `Hash(0)` showed the worst latency because of frequent hash collision. YCSB-E showed longer latency than the others because it issued range queries that carry multiple `GET()` commands.

Impact of KV Pool Size: The performance of KV-SSD varies depending on the number of KV objects stored. To understand its impact, we conducted experiments, varying storage utilization from 4%, 30%, 50% to 70%. For each, 2.5M, 20M, 32M, and 44M KV pairs were written, respectively, during the Load phase.

Figure 12 shows IOPS of `Hash(8)` and `LSM-KVD(5,3)+`. `Hash(8)` exhibited better throughput with lower storage utilization. With the smaller number of KV objects stored, more hash buckets could be cached in DRAM and a hash collision rate was reduced. When the storage utilization was 4%, `Hash(8)` was able to load all the buckets in DRAM. There were no hash table misses, and a collision rate was almost zero. Except for YCSB-C issuing only reads, however, `Hash(8)` still suffered from extra flash lookups while handling writes. When an existing KV pair was updated with new data, even if a corresponding bucket with the same signature existed in DRAM, `Hash(8)` had to read the flash to confirm whether that bucket actually points to the same KV pair. These flash lookups cannot be avoided with signature-based hashing and become a burden on the hash-based ones.

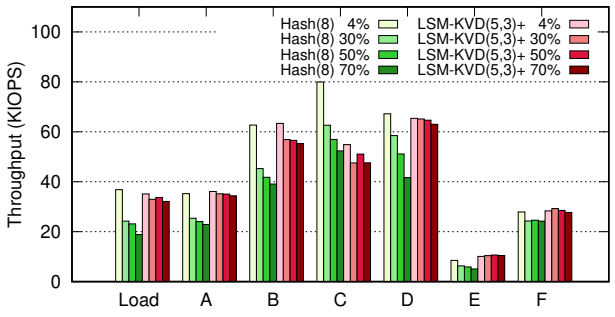


Figure 12: I/O throughput depending on KV pool sizes

`LSM-KVD(5,3)+` showed consistent throughput, regardless of the KV pool size. It offered slightly improved IOPS with smaller KV pools. This is because, as the KV pool size got smaller (*i.e.,* fewer KV indices), a larger fraction of the total KV indices were kept in pinned meta segments.

Garbage Collection: We evaluated the impact of GC on performance. To simulate the worst-case scenario where GC frequently happens, we created a KV pool with 44M unique KV pairs and ran a synthetic workload that issued uniformly random 100M `SET()` commands to update KV pairs. The over-provisioning space was 30% of the SSD capacity.

Figure 13 analyzes the number of page writes that happened while performing GC. `Hash(8)` involved a smaller number of page writes for GC than `LSM-KVD(5,3)`. After moving valid pages, both `Hash(8)` and `LSM-KVD(5,3)` have to update in-flash hash buckets or meta segments to point to the new locations of the pages ('KV Index' in Figure 13). A bucket size of `Hash(8)` is smaller than that of `LSM-KVD(5,3)` because it contains a 8-bit signature, instead of an exact key (*i.e.,* 32 B). Since more buckets are packed into a single flash page, the probability of which victim KV pairs update the same flash page for updating buckets was high. Thus the 'KV Indices' portion of `Hash(8)` was smaller than that of `LSM-KVD(5,3)`. Even worse, `LSM-KVD(5,3)` suffered from extra I/Os for compaction.

`LSM-KVD(5,3)+OPT` addresses this problem by rewriting victim KV pairs to L_0, instead of directly updating meta segments (see §4.6). This removed all flash writes associated with 'KV Indices', but potentially increased compaction costs since the indices for the victim pages in L_0 will be eventually

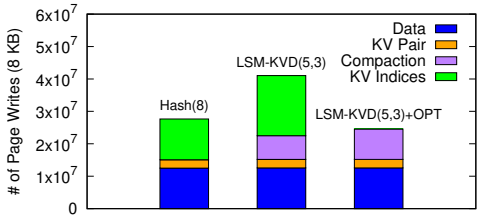


Figure 13: Analysis of GC cost: 'Data' represents pages written by `SET()`; 'KV Pair' indicates pages written to move valid KV pairs. 'Compaction' represents pages written to meta segments during compaction. 'KV Indices' indicates pages written to update meta segments or in-flash hash indices.

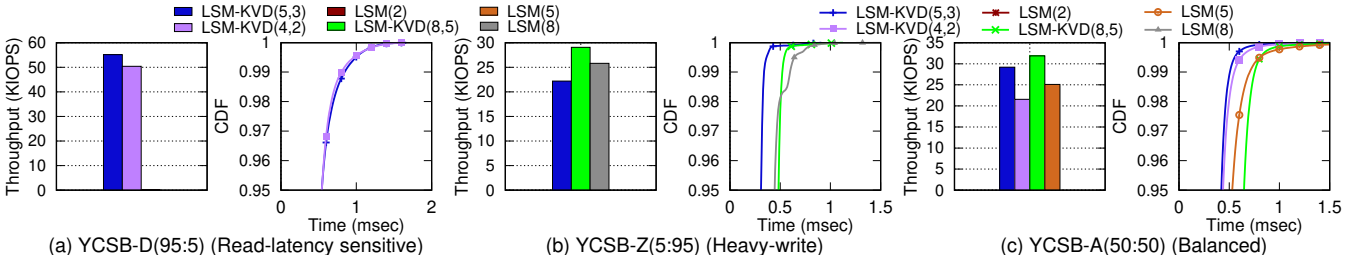


Figure 14: Experimental results of the tree configurations

written to meta segments again. This extra compaction cost was not so high. We observed that victim KV pairs in L_0 were likely to be coalesced with neighboring KV pairs and their indices were written to the same meta segment together.

Our results in Figure 13 confirm that extra I/Os required to keep the tree sorted (*i.e.,* compaction) and to clean up obsolete data (*i.e.,* GC) are similar or lower than those of the hash-based KV-SSDs. Thus, these are not a bottleneck to provide high throughput.

5.3 Trade-off Analysis

After analyzing the performance of LSM-KVD, we carried out experiments to understand its versatility to satisfy KV clients' demands. To show that, we selected three scenarios, YCSB-A, YCSB-D, and YCSB-Z, which have different performance requirements. As noted in Table 2, the read and write (R:W) ratio of YCSB-A is 50:50, so the both are important. YCSB-D has the R:W ratio of 95:5. Thus, improving read performance is preferred, unless the write throughput seriously deteriorates. YCSB-Z is manually synthesized by configuring YCSB parameters, so that it has the R:W ratio of 5:95. For YCSB-Z, users prefer high write throughput so long as reasonable read latency is provided.

For the above workloads, we considered three tree candidates of LSM-KVD, `LSM-KVD(5,3)`, `LSM-KVD(4,2)`, and `LSM-KVD(8,5)`, for YCSB-A, D, and Z, respectively. As shown in Figure 4, `LSM-KVD(5,3)` has the most balanced performance for reads and writes, even if it consumes large DRAM (*i.e.,* 54 MB) to pin three tree levels. `LSM-KVD(4,2)` provides the same level of read latency as `LSM-KVD(5,3)`, but write throughput is lower than `LSM-KVD(5,3)` since the tree height is 4. Instead, `LSM-KVD(4,2)` requires much less DRAM (*i.e.,* 28 MB) because only two levels are pinned. `LSM-KVD(8,5)` requires the same amount (*i.e.,* 54 MB) of DRAM as `LSM-KVD(5,3)`. By increasing the tree height, it offers higher write throughput, but sacrifices read latency.

From the original LSM-tree, we included three configurations, `LSM(5)` for balance performance (YCSB-A), `LSM(2)` for short read latency (YCSB-D), and `LSM(8)` for high write throughput (YCSB-Z). All of them used 54 MB DRAM for bloom filters. `LSM(n)` is equivalent to `LSM-KVD(n,0)`.

Figure 14 shows I/O throughput and read latency, respectively, for YCSB-D, Z, and A. For clear presentation, we plot the selected tree configurations in the graphs. For YCSB-D (see Figure 14(a)), `LSM-KVD(4,2)` showed very short read latency as `LSM-KVD(5,3)` with slightly degraded I/O throughput, 8.6%, compared to `LSM-KVD(5,3)`. Considering 2× lower DRAM requirement of `LSM-KVD(4,2)`, it would be the best option for read-latency sensitive workloads. We couldn't plot the read latency of `LSM(2)`, since it required 5 days to finish the Load phase with 105 IOPS.

For YCSB-Z (see Figure 14(b)), `LSM-KVD(8,5)` exhibited the best throughput, which was 31% higher than that of `LSM-KVD(5,3)`. Thus, it is the best for write-heavy workloads. `LSM(8)` showed fairly good write throughput, but inconsistent read latency with long tails made it less attractive.

For YCSB-A (see (see Figure 14(c)), `LSM-KVD(5,3)` performed well, achieving high throughput and short read latency. `LSM-KVD(8,5)` exhibited rather higher throughput, but its read latency was much worse than `LSM-KVD(5,3)`. `LSM(5)` couldn't outperform `LSM-KVD(5,3)` in terms of read latency and throughput.

6 Conclusion

In this paper, we presented a novel LSM-tree based KV-SSD design, called *LSM-KVD*. By aggressively pinning KV indices of top levels of the tree to DRAM, LSM-KVD was able to guarantee the worst-case read latency, while improving average read latency. Moreover, by combining the level pinning with hardware accelerators, LSM-KVD not only eliminated sorting overheads, but reduced compaction I/Os greatly. Our experimental results showed that LSM-KVD outperformed existing hash-based KV-SSDs in several aspects, including tail read latency, average read latency, and I/O throughput. In particular, by allowing users to flexibly configure the target I/O performance of KV-SSD according to their demands, LSM-KVD reduced overall execution times of KV clients.

As future work, we plan to adopt the idea of the level pinning to general-purpose KVS like RocksDB. A key challenge here may be providing atomicity and durability for meta segments pinned to DRAM which are vulnerable to power failure. This was not a serious issue in LSM-KVD since pinned meta segments are protected by built-in batteries. For LSM-KVD to be used for general-purpose KVS, therefore, techniques like WAL should be properly incorporated.

References

[1] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M. S., AND PANIGRAHY, R. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference* (2008).

[2] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems* (2012), pp. 53–64.

[3] AXBOE, J. FIO: Flexible I/O Tester Synthetic Benchmark. *URL https://github.com/axboe/fio (Accessed: 2015-06-13)* (2005).

[4] BAE, D.-H., JO, I., CHOI, Y. A., HWANG, J.-Y., CHO, S., LEE, D.-G., AND JEONG, J. 2B-SSD: The Case for Dual, Byte- and Block-addressable Solid-state Drives. In *Proceedings of the Annual International Symposium on Computer Architecture* (2018), pp. 425–438.

[5] 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 *Proceedings of the USENIX Annual Technical Conference* (2019), pp. 753–766.

[6] BLOTT, M., KARRAS, K., LIU, L., VISSERS, K., BÄR, J., AND ISTVÁN, Z. Achieving 10Gbps Line-rate Key-value Stores with FPGAs. In *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing* (2013).

[7] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H., ET AL. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the USENIX Annual Technical Conference* (2013), pp. 49–60.

[8] CHALAMALASETTI, S. R., LIM, K., WRIGHT, M., AUYOUNG, A., RANGANATHAN, P., AND MARGALA, M. An FPGA Memcached Appliance. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (2013), pp. 245–254.

[9] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS) 26*, 2 (2008), 4.

[10] CHUNG, C., KOO, J., IM, J., ARVIND, AND LEE, S. LightStore: Software-defined Network-attached Key-value Drives. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2019), pp. 939–953.

[11] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM symposium on Cloud computing* (2010), pp. 143–154.

[12] DAYAN, N., ATHANASSOULIS, M., AND IDREOS, S. Monkey: Optimal Navigable Key-value Store. In *Proceedings of the ACM International Conference on Management of Data* (2017), pp. 79–94.

[13] DAYAN, N., AND IDREOS, S. Dostoevsky: Better Space-time Trade-offs for LSM-tree based Key-value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the ACM International Conference on Management of Data* (2018), pp. 505–520.

[14] DEBNATH, B., SENGUPTA, S., AND LI, J. FlashStore: High Throughput Persistent Key-value Store. *Proceedings of the VLDB Endowment 3*, 1-2 (2010), 1414–1425.

[15] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles* (2007), pp. 205–220.

[16] ELDAKIKY, H., AND DU, D. H. C. Key-Value Pairs Allocation Strategy for Kinetic Drives. In *Proceedings of the IEEE International Conference on Big Data Computing Service and Applications* (2018), pp. 17–24.

[17] FACEBOOK, INC. RocksDB: A Persistent Key-value Store for Fast Storage Environments. https://rocksdb.org.

[18] GUPTA, A., KIM, Y., AND URGAONKAR, B. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2009), pp. 229–240.

[19] HERLIHY, M., SHAVIT, N., AND TZAFRIR, M. Hopscotch Hashing. In *International Symposium on Distributed Computing* (2008), Springer, pp. 350–364.

[20] JIN, Y., TSENG, H.-W., PAPAKONSTANTINOU, Y., AND SWANSON, S. KAML: A Flexible, High-performance Key-value SSD. In *Proceedings of the*

IEEE International Symposium on High Performance Computer Architecture (2017), pp. 373–384.

[21] JUN, S.-W., LIU, M., LEE, S., HICKS, J., ANKCORN, J., KING, M., XU, S., AND ARVIND. BlueDBM: An Appliance for Big Data Analytics. In *Proceedings of the Annual International Symposium on Computer Architecture* (2015), pp. 1–13.

[22] KANG, Y., PITCHUMANI, R., MISHRA, P., KEE, Y.-S., LONDONO, F., OH, S., LEE, J., AND LEE, D. D. G. Towards Building a High-performance, Scale-in Key-value Storage System. In *Proceedings of the ACM International Conference on Systems and Storage* (2019), pp. 144–154.

[23] KIM, S.-H., KIM, J., JEONG, K., AND KIM, J.-S. Transaction Support using Compound Commands in Key-Value SSDs. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems* (July 2019).

[24] LAKSHMAN, A., AND MALIK, P. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review 44*, 2 (2010), 35–40.

[25] LU, L., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies* (2016), pp. 133–148.

[26] MÁRMOL, L., SUNDARARAMAN, S., TALAGALA, N., RANGASWAMI, R., DEVENDRAPPA, S., RAMSUNDAR, B., AND GANESAN, S. NVMKV: A Scalable and Lightweight Flash Aware Key-value Store. In *Proceedings of the USENIX Conference on Hot Topics in Storage and File Systems* (2014), pp. 8–8.

[27] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., ET AL. Scaling Memcache at Facebook. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation* (2013), pp. 385–398.

[28] O'NEIL, P., CHENG, E., GAWLICK, D., AND O'NEIL, E. The Log-structured Merge-tree (LSM-tree). *Acta Informatica 33*, 4 (1996), 351–385.

[29] PAGH, R., AND RODLER, F. F. Cuckoo Hashing. *Journal of Algorithms 51*, 2 (2004), 122–144.

[30] ROSENBLUM, M., AND OUSTERHOUT, J. K. The Design and Implementation of a Log-structured File System. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (1992), 26–52.

[31] SAMSUNG ELECTRONICS. KV SSD Host Software Package. https://github.com/OpenMPDK/KVSSD.

[32] SAMSUNG ELECTRONICS. Samsung Introduces World's Largest Capacity (15.36TB) SSD for Enterprise Storage Systems. https://news.samsung.com/global/samsung-now-introducing-worlds-largest-capacity-15-36 2016.

[33] SAMSUNG ELECTRONICS. Samsung Key Value SSD enables High Performance Scaling. https://www.samsung.com/semiconductor/global.semi.static/Samsung_Key_Value_SSD_enables_High_Performance_Scaling-0.pdf, 2017.

[34] SAMSUNG ELECTRONICS. Samsung PM983 NVMe PCIe SSD. https://samsungsemiconductor-us.com/labs/pdfs/Samsung_PM983_Product_Brief.pdf, 2017.

[35] SAMSUNG ELECTRONICS. KV SSD Firmware Introduction. https://github.com/OpenMPDK/KVSSD/wiki/presentation/kvssd_seminar_2018/kvssd_seminar_2018_fw_introduction.pdf, 2018.

[36] SAMSUNG ELECTRONICS. 960PRO SSD Specification. https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/ssd960/, 2019.

[37] SEARS, R., AND RAMAKRISHNAN, R. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the ACM International Conference on Management of Data* (2012), pp. 217–228.

[38] SEONG, Y. J., NAM, E. H., YOON, J. H., KIM, H., CHOI, J.-Y., LEE, S., BAE, Y. H., LEE, J., CHO, Y., AND MIN, S. L. Hydra: A Block-mapped Parallel Flash Memory Solid-state Disk Architecture. *IEEE Transactions on Computers 59*, 7 (2010), 905–921.

[39] SIAU, C., KIM, K., LEE, S., ISOBE, K., SHIBATA, N., VERMA, K., ARIKI, T., LI, J., YUH, J., AMARNATH, A., NGUYEN, Q., KWON, O., JEONG, S., LI, H., HSU, H., TSENG, T., CHOI, S., DARNE, S., ANANTULA, P., YAP, A., CHIBVONGODZE, H., MIWA, H., YAMASHITA, M., WATANABE, M., HAYASHI, K., KATO, Y., MIWA, T., KANG, J. Y., OKUMURA, M., OOKUMA, N., BALAGA, M., RAMACHANDRA, V., MATSUDA, A., KULKANI, S., RACHINENI, R., MANJUNATH, P. K., TAKEHARA, M., PAI, A., RAJENDRA, S., HISADA, T., FUKUDA, R., TOKIWA, N., KAWAGUCHI, K., YAMAOKA, M., KOMAI, H., MINAMOTO, T., UNNO, M., OZAWA, S., NAKAMURA, H., HISHIDA, T., KAJITANI, Y., AND LIN, L. 13.5 A 512Gb 3-bit/Cell 3D Flash Memory on 128-Wordline-Layer with 132MB/s Write

Performance Featuring Circuit-Under-Array Technology. In *Proceedings of the IEEE International Solid-State Circuits Conference - (ISSCC)* (Feb 2019), pp. 218–220.

[40] SNIA. Key Value Storage API Specification Version 1.0. https://www.snia.org/tech_activities/standards/curr_standards/kvsapi.

[41] TANAKA, T., HELM, M., VALI, T., GHODSI, R., KAWAI, K., PARK, J., YAMADA, S., PAN, F., EINAGA, Y., GHALAM, A., TANZAWA, T., GUO, J., ICHIKAWA, T., YU, E., TAMADA, S., MANABE, T., KISHIMOTO, J., OIKAWA, Y., TAKASHIMA, Y., KUGE, H., MOROOKA, M., MOHAMMADZADEH, A., KANG, J., TSAI, J., SIRIZOTTI, E., LEE, E., VU, L., LIU, Y., CHOI, H., CHEON, K., SONG, D., SHIN, D., YUN, J. H., PICCARDI, M., CHAN, K., LUTHRA, Y., SRINIVASAN, D., DESHMUKH, S., KAVALIPURAPU, K., NGUYEN, D., GALLO, G., RAMPRASAD, S., LUO, M., TANG, Q., INCARNATI, M., MACEROLA, A., PILOLLI, L., DE SANTIS, L., ROSSINI, M., MOSCHIANO, V., SANTIN, G., TRONCA, B., LEE, H., PATEL, V., PEKNY, T., YIP, A., PRABHU, N., SULE, P., BEMALKHEDKAR, T., UPADHYAYULA, K., AND JARAMILLO, C. 7.7 A 768Gb 3b/Cell 3D-floating-gate NAND Flash Memory. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)* (Jan 2016), pp. 142–144.

[42] TWITTER INC. Fatcache: Memcache on SSD. https://github.com/twitter/fatcache.

[43] WANG, P., SUN, G., JIANG, S., OUYANG, J., LIN, S., ZHANG, C., AND CONG, J. An Efficient Design and Implementation of LSM-tree based Key-value Store on Open-channel SSD. In *Proceedings of the European Conference on Computer Systems* (2014), p. 16.

[44] XU, S., LEE, S., JUN, S.-W., LIU, M., HICKS, J., ET AL. Bluecache: A Scalable Distributed Flash-based Key-value Store. *Proceedings of the VLDB Endowment 10*, 4 (2016), 301–312.