PinK: High-speed In-storage Key-value Store with Guaranteed Tails

Submission #452

Abstract

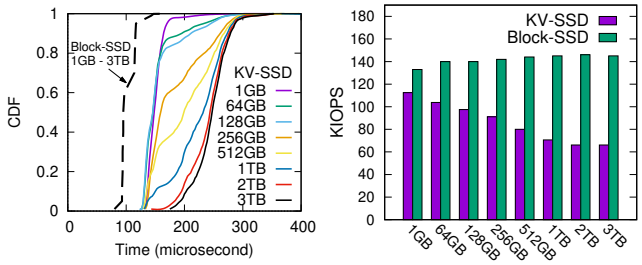
Key-value store based on a log-structured merge-tree (LSM-tree) is preferable to hash-based KV store because an LSM-tree can support a wider variety of operations and show better performance, especially for writes. However, LSM-trees are difficult to implement in the resource constrained environment of a key-value SSD (KV-SSD) and consequently, KV-SSDs typically use hash-based schemes. We present *PinK*, a design and implementation of an LSM-tree-based KV-SSD, which compared to a hash-based KV-SSD, reduces 99^{th} percentile tail latency by 73%, improves average read latency by 42% and shows 37% higher throughput. The key idea in improving the performance of an LSM-tree in a resource constrained environment is to avoid the use of Bloom filters and instead, use a small amount of DRAM to *keep/pin* the top levels of the LSM-tree.

1 Introduction

Offloading the key-value (KV) functionality onto a storage device has received a lot of attention recently from both academia and industry [7, 17, 20, 27, 43]. A representative device in this class is Samsung's key-value SSD (KV-SSD) [20], which directly serves KV requests. By offloading most commonly used operations of KV databases (e.g., RocksDB [14]), KV-SSDs not only improve I/O latency and throughput of KV clients, 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 devices often provide inconsistent tail-latency and throughput. This is because most of KV-SSDs are based on hashing [12, 17, 20, 27, 40, 43], which is attractive because it is rather simple to implement but has some inherent limitations. A hash-based KV-SSD maintains a hash table in the controller DRAM, each entry of which typically contains a key (or the signature of a key) and a pointer to the corresponding KV pair in the flash. The hash table is used to quickly index key-value pairs by simple table lookups. However, when the DRAM size is not large enough to accommodate all the hash table entries, parts of the hash table must be stored in flash. This inevitably involves expensive flash accesses and complex hash table management when accessing entries that are not in memory. Even worse, if a hash collision occurs, multiple flash accesses are required, resulting in long and unpredictable tail-latency and drop in throughput.

To understand the behavior of hash-based KV-SSDs, we conducted a set of experiments on a 4TB KV-SSD prototype (KV-PM983 [35]). We created KV pools ranging in size from 1GB to 3TB, and chose the average key and value sizes to be 32B and 1KB, respectively [3]. Thus, a 3TB KV pool would hold 3 billion KV pairs. We ran random `GET()` requests on these KV pools for 10 minutes using KVBench [33]. No GC occurred during our experiments.



(a) CDF of read latency (b) Read throughput

Figure 1: Performance comparison of KV-SSD & Block-SSD depending on the total amount of data stored (1GB∼3TB)

Figure 1 shows that the KV-SSD suffered from inconsistent read latency, and its throughput dropped as the number of objects stored increased. The average read latency increased from 149.49 μs (1GB pool) to 245.31 μs (3TB pool). We also observed long tail latency: for the 99.99^{th} percentile, the tail-latency increased from 323 μs to 1020 μs. Even worse, the read throughput dropped to 64 KIOPS from 112 KIOPS. Although we did not have access to any of the internal details of the KV SSD design (*e.g.,* the hash function), it is easy to conclude that the performance and tail latency get worse in a hash-based implementation as the total number of stored KV pairs increases. This hypothesis was given further support by another experiment, where we used the same setup to run FIO [4], a 4TB Block-SSD [36], which loads an entire FTL table in DRAM for 4KB page mapping. FIO exhibited stable latency and throughput, regardless of the amount of data stored. Such severe performance variability and unpredictable I/O behaviors make KV-SSDs less attractive than normal SSDs.

An alternative to hashing is *log-structured merge trees* (LSM-trees) [29]. The tail-latency in such a system is bounded by the number of levels in the tree. Since an LSM-tree indexes KV pairs in a multi-level *sorted* tree, it might also require a much smaller DRAM for indexing KV pairs. The LSM-tree also supports range-queries and scans efficiently, without any extra bookkeeping or support from KV clients [20]. Our experiments, however, revealed that a conventional implemen-

tation of the LSM-tree on an SSD controller failed to deliver the promised benefits. In fact, it showed worse performance than hashing in some cases.

The first problem we discovered was the tail latency. Most LSM-tree implementations use Bloom filters to skip lookups in a tree level to improve *average* read latency. Owing to the probabilistic nature of Bloom filters, however, one cannot ensure the *worst-case* read latency; indeed, we observed long tails as in the hash-based KV-SSD. The second problem was high write-amplification. Even if we use key-value separation like Wisckey [25], compaction, an essential task of the LSM-tree to sort KV indices and balance its indexing trees, involves many extra storage accesses. Moreover, this LSM-tree compaction cost exacerbates the FTL's garbage collection (GC) cost. The third problem was that rebuilding Bloom filters and sorting KV pairs for compaction requires lots of CPU cycles, which overburden embedded-class microprocessors found in SSD controllers. This lack of processor performance deteriorates the I/O performance dramatically.

In this paper, we propose an LSM-tree-based in-storage key-value engine, called *PinK*, which overcomes all the problems mentioned above. The novelty of PinK design stems from four specific techniques it uses. At the heart of PinK is *level pinning*. Instead of keeping probabilistic Bloom filters in DRAM, PinK pins exact key-value indices of the top levels of the tree to DRAM. This removes unnecessary flash lookups on the pinned levels in a deterministic manner, thereby enabling us to provide predictable read latency with guaranteed tails. Elimination of Bloom filters also reduces the resource requirement for computing them. Second, the level pinning helps us reduce flash I/Os caused by compaction. Since KV indices are kept in DRAM, PinK can sort them in DRAM without any I/Os. The pinned indices are protected by built-in capacitors, so flushing out up-to-date indices to flash is not necessary. (This idea is feasible only for small amount of DRAM). Third, we discovered that the majority of GC I/Os are induced by updating indices of the LSM-tree. By delaying index updates until the compaction phase, PinK reduces GC I/Os greatly. Finally, by adding hardware comparators in between the SSD controller and NAND chips, and performing KV sorting while reading KV pairs, PinK completely eliminates CPU costs for compaction.

We have implemented PinK on MIT's FPGA-based SSD platform [18], and used the LSM-tree implementation of LightStore [7] as our starting point, because its source code is publicly available. Using YCSB [9] benchmarks, we have shown that PinK outperforms existing KV-SSD designs in several aspects. Compared to a hash-based KV-SSD, PinK reduces 99^{th} percentile tail latency by 73%, improves average read latency by 42% and shows 37% higher throughput. Furthermore, compared to LightStore, PinK reduces 99^{th} percentile tail latency by 22%, improves average read latency by 22% and shows 44% higher throughput.

Paper Organization: In Section 2, we explain background closely related to this study. Section 3 analyzes the performance of the LSM-tree algorithm in KV-SSD. Section 4 presents an overall design of PinK, 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 conventional Block-SSD is designed to support the standard block I/O interface. It exposes a linear array of 4KB 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 [34, 38]. For example, for a 4TB SSD, 4GB DRAM is required. A mapping table has to be persistent (non-volatile) and is protected by batteries to guard against sudden power failures [5]. Similar to other log-structured systems [31], the FTL has to perform garbage collection (GC) to reclaim free space.

2.2 KV-SSD

A KV-SSD is a new type of SSDs [20, 39] 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 SSD, both the key and the associated value are of variable sizes. A key can be as long as 255 bytes [39] or even be a character string, and a value can be as big as 2MB [39]. 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 [20, 21]. A more detailed description can be found in SNIA's KV-SSD specification [39].

Making SSDs support the KV interface requires a redesign of 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 [27], KAML [17], and BlueCache [43]) and industry (*e.g.,* Samsung's KV-SSD prototype [20, 35]). All these KV-SSDs are based on the hash-based data structure, which we discuss next.

2.3 Hash-based KV-SSD

A hash-based KV-SSD maintains a hash table with many buckets in DRAM, where each bucket holds metadata (*i.e.,* a

key and a pointer) for a specific KV object in flash [12, 17, 27, 43]. A primary design issue of the hash-based KV-SSD is the management of a huge hash table requiring large amounts of DRAM. Suppose that the SSD capacity is 4 TB and the key and value sizes are on average 32B and 1KB, respectively [3]. If the number of buckets is 2^{32} ($= 2^{42}/2^{10}$) and the bucket size is 36B (32B for a key and 4B for a pointer), 144GB of DRAM is required to hold the complete hash table.

To reduce DRAM, some use signatures [6, 12, 22, 40, 43]. Instead of an exact key, a short signature of the key is kept in the bucket. The exact key and its value are stored in the flash. Using signatures reduces the hash table size greatly – if a 16-bit signature is used, 24GB of DRAM is required. However, it causes *signature collision* which happens when different keys have the same signature. 24GB DRAM is still huge for an SSD. The DRAM size can be further reduced by keeping only popular buckets in a fixed-size DRAM (*e.g.,* 4GB) while storing the rest in the flash [15]. This, however, causes extra flash reads. If a designated bucket is not available in DRAM (*i.e., hash table miss* occurs), we have to fetch the bucket from the flash to find the location of a desired KV object. Consequently, owing to signature collision and hash table miss, the hash-based KV-SSD exhibits unstable performance as depicted in Figure 1.

This inconsistent performance may be due to inefficient collision resolution policies. There are advanced hashing strategies, such as Cuckoo [30] and Hopscotch [16, 22], which provide constant worst-case lookups and may avoid the tail latency. But, this benefit comes at the cost of degraded write speed and/or frequent rehashing. Hashing algorithms also cannot efficiently support range and scan operations [20].

2.4 LSM-Tree versus Hashing

An LSM-tree is another data structure that is used widely to implement persistent key-value stores. It is usually implemented purely in host software and can support a wider set of KV operations (*e.g.,* RocksDB [14] and Cassandra [23]). It is also used in big all-flash array (AFA) systems such as Purity [8]. Because of its increasing popularity across a variety of systems, many LSM-tree variants have been proposed [2, 19, 41].

LSM-trees generally have better write performance than hash-based KV-Stores but are more difficult to implement and require more processing and DRAM resources. This is not surprising because LSM-tree algorithms have been designed mostly for server-class x86 host machines, which have plenty of computing resources. Nevertheless, recently LSM-trees have also been used in some implementations of KV-SSDs like LightStore [7], iLSM-SSD [24] and Kinetic HDD [13]. The design of PinK was motivated by some inefficiencies identified in LightStore, which we discuss in Section 3. We expect iLSM-SSD to exhibit similar behavior as LightStore because it also relies on Bloom filters to speed up reads.

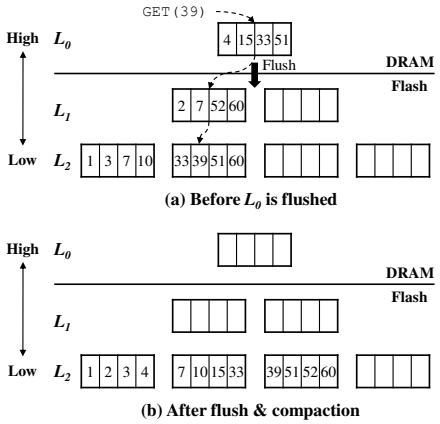


Figure 2: LSM-tree organization ($h = 3$, $T = 2$). A rectangle represents a KV object and the number inside is the key.

3 Challenges in implementing LSM-tree in a KV-SSD

In this section, we analyze the performance and present key technical challenges an LSM-tree is implemented in a resource constrained environment of an SSD controller.

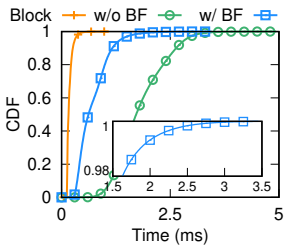
3.1 Basic of LSM-Tree Algorithm

The LSM-tree maintains multiple levels of sorted KV indices, $L_0, L_1, ...,$ and L_{h-1}, where h is the height of an LSM-tree. The level 0, L_0, is kept in DRAM as a write buffer, whereas the rest are stored in persistent media (*e.g.,* flash). In LSM-trees, the levels are organized so that a lower level is T times larger (*i.e.,* the size factor T) than a higher one. Each level is divided into fixed-size runs, where the size of each run is usually the same as that of L_0.

The LSM-tree has two unique properties: **#1.** for each level, KV objects are unique and kept sorted by their keys; and **#2.** the key range of one level may overlap the key range of other levels due to overwrites (see Figure 2).

When a `SET()` request comes, a KV object is first buffered in L_0. Once L_0 becomes full, buffered KV objects are flushed out to L_1. All the objects in L_0 are written to L_1 in an append-only manner. 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 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 objects from two adjacent levels, sorts them in the memory, and writes the sorted objects to next lower level as shown in Figure 2(b). Compaction incurs a huge I/O overhead. This overhead can be mitigated by separating keys from values and by avoiding moving values which are not affected by compaction (see Wisckey [25]).

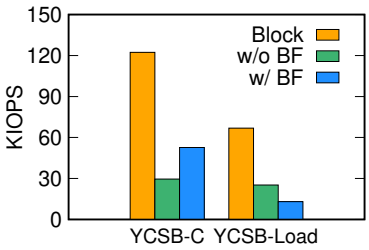
The LSM-tree maintains an in-memory data structure that points to runs of levels in the flash. Each run contains a header



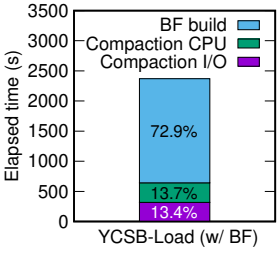
(a) Read latency (YCSB-C)

# of flash page reads	w/o BF	w/ BF
1	0.0932%	0.190%
2	0.016%	98.347%
3	0.040%	1.359%
4	1.398%	0.082%
5	98.458%	0.001%

(b) Flash page read counts (YCSB-C)



(c) I/O throughput



(d) Compaction time breakdown

Figure 3: Experimental results of the conventional implementation of LSM-tree on an SSD controller

that holds the locations of KV objects (KV indices) in the flash. Searching for a key at a specific level is fast. Once a header is read from the flash, the location of a desired KV object can be quickly found since they are sorted by key. However, finding the desired key in the entire tree 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). The number of the worst-case flash lookups is $O(h-1)$ (Note: L_0 is excluded since it stays in DRAM). Bloom filters are often used to avoid useless lookups on levels that do not have desired keys [10, 11]. Usually, each level or run in the tree has each own filter.

3.2 Performance Analysis

Our PinK implementation uses the same FPGA-based hardware platform as LightStore [7], which has quad-core ARM Cortex A53 running at 1.2GHz and 4GB DRAM. This controller specification is similar to those of latest SSDs with in-storage computation capability [28, 32]. PinK is equipped with a 256GB NAND flash card which provides 1.1 GB/s read and 600 MB/s write throughputs, respectively, and offers 122,349 IOPS for 4KB reads and 66,843 IOPS for 4KB writes, respectively.

To understand the weaknesses of conventional LSM-tree implementations, we first improved the Bloom filter implementation in LightStore [7] by replacing the original one with Monkey [10]. We leveraged AArch64 SIMD instructions in implementing Monkey. Key-value separation [25] was employed by default.

For fast evaluation, we reduce the SSD capacity to 64GB. The number of levels in the tree is set to 5 ($h = 5$) with a size factor of 23. We assume that 64 MB of DRAM (0.1% of 64GB) is available and it is used to keep Bloom filters for levels. We either enable or disable Bloom filters (Monkey) to understand its impact on performance. To characterize basic performance, we run two extreme workloads, YCSB-Load (100% writes) and YCSB-C (100% reads). Average key and value sizes are 32B and 1KB. We first run YCSB-Load with 44 million (44GB) uniformly random KV-pairs, and then run YCSB-C with 10 million Zipfian requests. Results with other workloads can be found in §5.

To understand the impact of the LSM-tree algorithm, we compare the performance of the LSM-tree KV-SSD with that of a Block-SSD implemented on the same platform. The Block-SSD employs a page-level FTL whose flat mapping table, indexed by LBA, can be loaded entirely on DRAM. A physical page mapped to a logical block can be found with only one memory reference.

Figure 3(a) shows the CDF of the read latency of YCSB-C. Without Bloom filters, the LSM-tree KV-SSD shows long read latency over the Block-SSD. Figure 3(b) summarizes the number of flash page reads to service a `GET()` request. If `GET()` is directly served by L_0 (*i.e.,* a write buffer), a page read is not necessary. Otherwise, the LSM-tree looks up lower levels to fetch KV indices from the flash. The majority (98.4%) of `GET()` requests touch up to the last level (L_4), issuing four page reads. This is because almost all of the KV pairs (95%) are stored on L_4. When Bloom filters are enabled, it offers better read latency, but is affected from long tails. With Bloom filter, on average, one flash lookup is required for retrieving a KV object as in Figure 3(b). Owing to its probabilistic nature, however, 1.4% of the total `GET()`s still require more than one flash lookup, which are large enough to cause long tails (see the zoom-in figure in Figure 3(a)).

Figure 3(c) illustrates the I/O throughput. The read throughput of the LSM-tree with Bloom filter in YCSB-C is about half of the throughput that the Block-SSD provides. This is expected because Monkey requires two flash reads, on average, for retrieving KV indices to serve `GET()`.

As we can see in Figure 3(c), in YCSB-Load, we observe serious drops in the write throughput, compared to the Block-SSD. Even with Wiskey, compaction I/Os account for 75.5% of the total I/Os (both reads and writes). While not included in Figure 3(c), I/Os for GC also badly affect the write throughput. According to our analysis (see §5.2), the write amplification factor (WAF), which is 2.52 when only compaction I/Os occur, increases to 5.02 once GC starts to trigger. We find that moving valid pages for GC involves cascade updates of KV indices maintained by the LSM-tree.

The high CPU overheads of the LSM-tree also slow down the write throughput. Due to slow speed of ARM CPUs, sorting KV pairs for compaction, which involves string comparisons, becomes a bottleneck. As shown in Figure 3(d), it takes almost the same time as performing compaction I/Os. The

cost of rebuilding Bloom filters is high. Bloom filters should be rebuilt for newly-created levels after compaction, which requires expensive hash computations and lots of memory accesses. Even though a hash computation is accelerated by SIMD instructions, its negative impact is still huge. Be advised that, our LSM-tree is carefully designed so that I/Os and computation are maximally overlapped. However, this cannot completely hide high computation costs.

The problems we have observed can be summarized as follows: **#1.** LSM-trees exhibit higher average-latency because of multi-level search, and also exhibit unpredictable tail-latency because of Bloom filters; **#2.** Bloom filters require lots of computational power to reconstruct. They have to be reconstructed after each compaction; **#3.** Level compaction (excluding Bloom filter reconstruction) also requires a lot of computation and I/O bandwidth; **#4.** Compaction I/Os may trigger GC which in turn generates more I/Os, resulting in high write amplification.

4 Design of PinK

Bloom filters are used to reduce the average read latency. Another way of reducing the read latency would be to keep popular KV indices in DRAM. The LSM-tree by nature keeps the recently written indices in the top levels. In PinK, we eliminate the Bloom filters and mitigate the increased read latency by pinning top-K levels (§4.2 and §4.3). We will show that level-pinning requires only a small amount of DRAM. Tail latency is already bounded to the height of the tree. Another benefit of level-pinning is that it eliminates the flash I/Os required for compaction of two levels which are already pinned in DRAM. The throughput can be further improved by using hardware accelerators that performs compaction for pinned and flash-resident levels (§4.4). Finally, to alleviate the GC costs associated with compaction, we delay GC by putting updated KV indices in L_0 (§4.5). This reduces the write amplification which affects lifetime of SSDs.

4.1 Overall Architecture

PinK supports variable-sized keys (16B∼128B) and values (1KB∼2MB), along with a rich set of KV operations (*i.e.,* `GET()`, `SET()`, `DELETE()`, `SCAN()`, and `ITERATOR()`), except for a few features like namespaces. Like KV-SSDs, PinK is able to guarantee durability and atomicity of KV operations [5, 20, 37]. While the lack of space does not permit us to describe the details of all the operations, we focus on explaining key data structures and operations which are different from conventional LSM-tree-based KVSs.

Data Structures. Figure 4 illustrates four types of data structures of PinK: a *skiplist* and *level lists*, which all reside in DRAM, and *meta segments* and *data segments*, which all reside in flash. Overall, the design of PinK is not much different from LSM-tree-based KVS combined with Wiskey [25], but it is optimized to maintain compact data structures in the

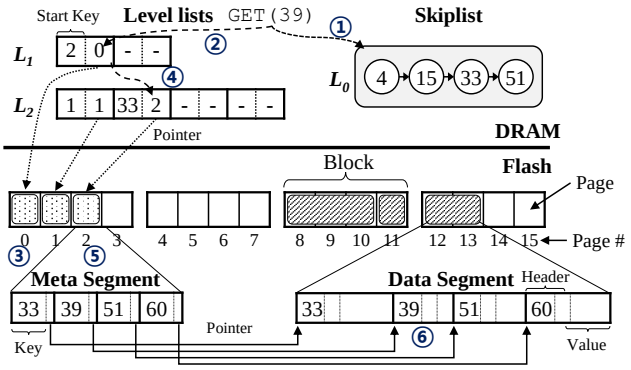


Figure 4: An overall architecture of PinK with its key data structures in DRAM and flash. The tree hierarchy and KV objects are identical to those of Figure 2(a). Given `GET(39)`, ① PinK first looks up the skiplist (L_0). Since a matched one is not found, ② it goes down to L_1, ③ reading a meta segment from the page 0. It does not have a desired key, so ④ PinK visits L_2 and reads ⑤ the page 2 to get a meta segment. Finally, ⑥ it can find the location of the value for the key '39'. Three flash reads are required to serve `GET(39)`.

controller DRAM for better performance in storage devices. Also, the headers of each data structure are designed to be handled easily by HW accelerators. PinK directly deals with NAND chips to perform indexing, GC, and wear-leveling obviating any need for a costly FTL found in most SSDs.

A *skiplist* corresponds to L_0 in the LSM-tree algorithm and works like a write buffer which buffers incoming KV objects temporarily. The size of L_0 is configured to be large enough (*e.g.,* 8MB∼64MB) to fully utilize the parallelism of multiple NAND channels when KV objects are flushed out to the flash. Each skiplist entry has four fields: <key size, key, value size, value>, and all the entries are sorted by key.

Once the skiplist becomes full, buffered objects are materialized to L_1 as the forms of *meta segments* and *data segments*. In L_1 (and all the lower levels), keys and values are separated into meta and data segments, respectively. A meta segment contains keys and pointers to its associated values in data segments. In addition to values, a data segment stores keys and their sizes to support GC (see §4.5). The size of a meta segment is fixed to a flash page size (*e.g.,* 8KB ∼ 16KB), but a data segment can be of any size – it is like a huge log containing KV objects pointed to by meta segments.

Since meta segments are referenced by the software to look for a KV object and by the hardware accelerators for compaction, they are organized to be manipulated by both of them. A meta segment is composed of an array of <key, pointer> pairs sorted by key, plus a header. A pointer is a 4B integer, but a key size varies from 16B to 128B. To quickly find a variable-size key using binary search, a meta segment header maintains the start locations (2B each) of <key, pointer> pairs. If a meta segment is 16KB, it contains up to 1024 <key, pointer> pairs

where at most 2KB is used as a header. For HW accelerators, a header and <key, pointer> pairs are aligned to 16B for simple implementation. We discuss this in §4.4 in detail.

PinK maintains another in-memory data structure, *level lists*, which keep track of meta segments at every level in the flash. If the tree has five levels (*i.e.,* $h = 5$), there are four level lists except for L_0. Each level list is organized as an array of pairs of fixed-sized pointers (4B each, 8B total); the first one points to the physical location of a meta segment in the flash; the second one points to a start key of that meta segment. Note that start keys of meta segments are stored separately in DRAM to support variable-sized keys (16-128B). This facilitates us to implement binary search to find a desired meta segment in a level list.

Two in-memory data structures, L_0 and the level lists, are protected by capacitors. This provides enough time for PinK to safely flush out them to the flash in the event of power failures or when a system is turned off. PinK also does not need to use a write-ahead log (WAL) to provide atomicity and durability of data.

Data Structure Size. Compared to the hash, PinK requires much smaller DRAM for indexing KV objects. Assume that an SSD capacity is 4TB and each meta segment is 16KB. As in §3, the average sizes of keys and values are 32B and 1KB, respectively [3]. Each entry in a meta segment is 36B (32B key and 4B pointers). A 16KB meta segment can hold 398 <key, pointer> pairs. In a 4TB SSD, there exist 2^{32} 1KB objects, and thus the number of meta segments in the flash is about 10.8M (= $2^{32}/398$). Each of these must be pointed to by some level lists. Each level list entry is 8 B, and each entry has a corresponding start key whose average size is 32B. Thus, only 432MB (= 10.8M×(8B + 32B)) DRAM is needed to hold all the level lists.

4.2 Improving I/O Speed with Level Pinning

Eliminating Read Tails. Retrieving a KV object from PinK requires multiple flash lookups. In the worst case, $O(h-1)$ flash lookups are required to access a KV object. Bloom filters is typically used to avoid useless lookups on levels that do not have desired keys [10, 11]. As pointed out earlier, however, it cannot avoid long tails and causes high CPU costs.

In order to guarantee worst-case latency and to get rid of Bloom filters, PinK adopts *level pinning*. The idea of the level pinning is straightforward. If the LSM-tree has h levels, PinK keeps meta segments for top-k levels ($k \leq h-1$) in DRAM. This simple technique greatly reduces read tails. To process `GET()`, it first searches for a key in top-k levels in DRAM. Only when a key is not found in memory, it looks up the rest of levels resident in the flash. With the level pinning, the number of the worst-case flash lookups is reduced to $O(h-k-1)$.

Level-pinning Memory Requirement. One might think that the level pinning would require large amounts of DRAM, but this is not the case. In the LSM-tree, a upper level (L_i) is

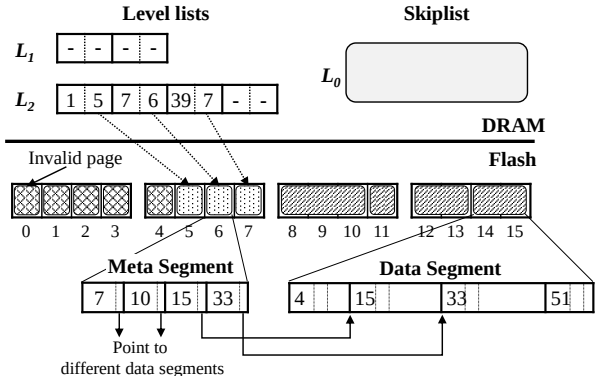


Figure 5: The DRAM and flash layouts of PinK after L_0 (in Figure 4) is flushed out with compaction. The tree hierarchy and KV objects are identical to those of Figure 2(b).

T times smaller than a lower level (L_{i+1}), which implies that the level size increases *exponentially* by a factor of T. In the 4TB 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 0.91MB, 50.86MB, 2.83GB, and 161.63GB, respectively. Meta segments for L_1, L_2, and even L_3 can be loaded in DRAM, considering a large controller DRAM of an SSD (*e.g.,* 4GB DRAM for 4TB SSD). The data structures of PinK do not require large amounts of DRAM (*e.g.,* 432MB), which enables us to pin more levels.

Reducing Compaction I/Os. Another benefit of the level pinning is that it eliminates flash I/Os involved in compaction. The level pinning maintains the meta segments of specific levels in DRAM. Thus, PinK does not need to issue any I/Os since pinned meta segments can be updated in DRAM directly. Dirty segments do not need to be written back to the flash because they are protected by capacitors.

To understand its benefit, let's consider how PinK performs compaction using the examples in Figures 4, and 5. Figure 5 is the data layout after the compaction. We assume that L_1 is pinned to DRAM. Before flushing out L_0, PinK fetches the corresponding meta segments from L_1 (*i.e.,* the page 0 in Figures 4 and 5) and sorts KV indices of L_0 and L_1, 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 5). The level lists are updated accordingly. PinK recognizes that L_1 becomes full, and thus flushes out L_1 to L_2. To do this, PinK reads two meta segments from each of L_1 and L_2 (the pages 1∼4 in Figure 5), sorts them, and finally writes three sorted segments to L_2 (*i.e.,* the pages 5∼7). Since L_1 is pinned, PinK eliminates 3 reads and 2 writes out of 5 reads and 5 writes which occurs while conducting the compaction.

4.3 Optimizing Search Path

The level pinning gives us performance benefits by removing Bloom filters, but it creates a challenge that increases the search time of a key in the level lists. The original LSM-tree

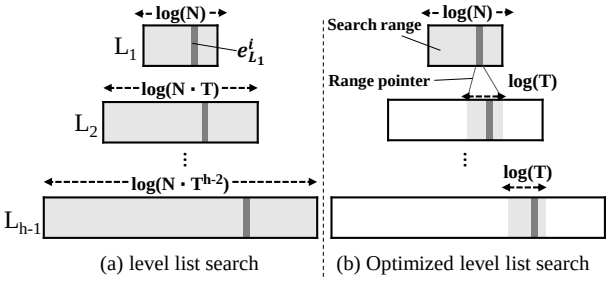


Figure 6: Search path optimization with range pointers

maintains Bloom filters for each level or run. Given a key, it first looks up the Bloom filters for L_i. Only when the Bloom filters return true, it searches for the key in its in-memory data structure (*e.g.,* a level list in PinK) to find the location of a header containing KV indices in the flash (*e.g.,* a meta segment). Otherwise, it skips L_i, visits the next level, L_{i+1}, and looks up L_{i+1}'s filters. As a result, Bloom filters remove useless flash reads and reduces search time in the LSM-tree's in-memory data structures.

Since PinK does not use Bloom filters, it has to perform binary search on level lists until it finds a matching meta segment. This does not cause a serious overhead for higher levels (*e.g.,* L_1 and L_2) whose level lists have few entries. On the other hand, since level lists belonging to lower levels (*e.g.,* L_{h-1}) have many entries, the search overhead becomes huge. Figure 6 (a) shows how PinK performs binary search on the level lists. Suppose L_1 has N entries. Because a level size increases by a factor of T, L_2 has $N \cdot T$ entries, L_3 has $N \cdot T^2$ entries, and finally L_{h-1} has $N \cdot T^{h-2}$ entries. The worst-case time complexity is thus expressed as $O(h^2 \cdot \log(T))$.

To reduce the search overhead, PinK uses two techniques. The first one is to reduce string comparison costs by using a prefix of a key. Recall that each entry of a level list has two pointers, each of which points to a meta segment and a start key string, respectively. We further include a prefix which holds exactly the first four bytes of a start key. During binary search, PinK compares the first four bytes of an input key with a prefix. Only when they match, it performs a full string comparison using the pointer to the key.

The second one is to reduce search ranges of the level lists. Each entry of a level list now has another 4-byte pointer, called a range pointer. It locates the next lower level's entry which has the greatest start key but whose key is less than or equal to that of the upper level entry. Given a key to search, PinK does binary search for L_1 and finds an entry, say $e^i_{L_1}$, in L_1's level list. If a meta segment pointed to by $e^i_{L_1}$ does not have a matched KV index, PinK has to go down to the next level, L_2. The range pointer of $e^i_{L_1}$ becomes the lower search bound for L_2. Then, the range pointer of the next entry $e^{i+1}_{L_1}$ in the same level (*i.e.,* L_1) is the upper search bound. As shown in Figure 6 (b), using two pointers, the number of entries we have to do binary search in L_2 is reduced to T, on average, since a level size increases by a factor of T. This

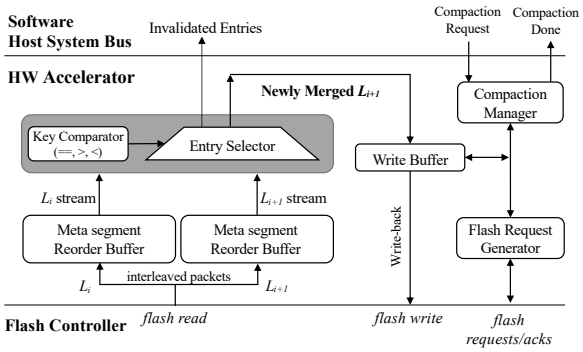


Figure 7: Compaction accelerator for flash-resident levels

can be applied for lower levels, L_3, ... , L_{h-1}. Thus, the time complexity reduces to $O(h \cdot \log(T))$.

With prefixes and range pointers, each entry size in the level lists increases to 16 bytes from 8 bytes. Fortunately, the lowest level L_{h-1}, which has the largest entries, does not maintain range pointers. As a result, additional DRAM is about 43.9MB in the same setting as in §4.1.

4.4 Speeding up Compaction

While the level pinning effectively reduces the number of I/Os for compaction, it does not remove the computation cost for sorting KV pairs. We address this problem by offloading some compaction tasks to a special HW accelerator in the SSD controller. The idea behind this is that compaction is just like merging two sorted lists of KV indices into a single sorted list. The HW accelerator placed between the flash and the host data bus can easily merge two flash-resident levels as meta segments of two levels are streamed from flash at wire speed. The accelerator writes the merged meta segments back to the flash without CPU involvement. By using the HW accelerator, we not only alleviate the computation overhead but improve I/O bus utilization since no to-be-merged and merged segments transferred over system bus. Remaining I/O bandwidth can be utilized by DRAM and flash for other tasks such as searching upper levels or managing pinned levels.

Figure 7 describes the architecture of the HW accelerator. We briefly present how the compaction accelerator for flash-resident levels works. The PinK software requests the accelerator to perform compaction by providing lists of the meta segments' flash addresses of two levels (L_i and L_{i+1}) to be merged and a list of flash addresses to which the merged meta segments (L_{i+1}) are written back. The flash request generator schedules multiple read requests to maximize the flash bandwidth utilization. Since the packets of different flash channels are interleaved, we need to use per-channel reorder buffers for each level to serialize the stream of meta segments.

Once we have sorted meta segment streams from two levels, the compaction engine (gray box in Fig. 7) only needs to keep comparing the keys of two levels and emitting the smaller one. The accelerator generates the output stream at

wire speed without any computation overhead. When two keys match, the entry from the upper level (L_i) supersedes as it is more recent one. Note that the accelerator informs the software the metadata of invalidated entries from L_{i+1} for various purposes such as garbage collection. The generated merged meta segment stream (L_{i+1}) is written back to the flash via small write buffers. Once the operation completes, the accelerator responds with the number of flash pages consumed by the newly generated L_{i+1} meta segments so that the software can reclaim unused flash addresses previously provided to the accelerator.

While not shown in detail, we have a similar accelerator for merging pinned levels that reads from and writes back to host DRAM. DMA engines are used instead of a flash request generator and we do not need reorder buffers.

4.5 Optimizing Garbage Collection

The LSM-tree appends all the data to the flash. As compaction is repeated, obsolete data, which are no longer referenced to by the tree, are accumulated in the flash and must be erased by GC later. There are roughly two types of obsolete data that are created by compaction. The first type is old meta segments. While performing compaction, PinK writes new meta segments that replace old ones. For example, the meta segments stored in the pages 0, 1, and 2 in Figure 5 are not managed by the tree anymore since they contain old indices. The second type is an outdated KV object which was updated with a new one or removed by a client. Outdated KV indices are discarded from the LSM-tree during compaction (see §4.4) so that no meta segments point to them. But, their KV data are still stored somewhere in a data segment(s).

To erase obsolete data and to keep maintaining free space, PinK triggers GC when free space is nearly exhausted. It selects a victim flash block, copies valid data (*i.e.,* pages or KV pairs) to a free block, and erases the victim. For hot-cold separation, meta segments are isolated in different blocks from data segments. PinK should perform GC differently depending on the type of blocks selected as a victim.

GC for Meta Segment: If a victim block to GC is a meta-segment block and thus has only meta segments, PinK retrieves a start key of a meta segment by reading its page. Then, it looks up the level lists to see if there is any entry pointing to it. If not, PinK skips it since that segment is obsolete (*e.g.,* the page 4 in Figure 5). Otherwise (*e.g.,* the page 5), it moves the page (*i.e.,* meta segment) to a free page, and then updates the entry so that it locates a new flash page. Cleaning meta segments is cheap because it involves valid page copies and updates of the level lists in DRAM.

GC for Data Segment: Cleaning a data-segment block requires more efforts. Each data segment keeps metadata (*i.e.,* keys and sizes) as noted in §4.1. By scanning a data segment from the victim block, PinK extracts keys for values to move for GC. Using these numbers, PinK looks up the level lists and finds associated meta segments to check the validity (valid or not) of each value. If a meta segment is not pinned in DRAM, it must be read from the flash. In this way, PinK collects a list of valid values in the victim.

The simplest approach to reclaim free space, which is used by Wisckey, is to copy valid values to free pages and to erase the victim block. The meta segments associated with the values should be updated and flushed out to the flash so that they point to the new locations of the values. For meta segments pinned in DRAM, no flash writes are necessary. This approach, however, creates many updates on meta segments in the flash. We observe that many victim values are associated with flash-resident meta segments because they were written long time ago and their meta segments were likely to be demoted to lower levels. Moreover, only few values belong to the same meta segment (*e.g.,* 1∼2 values, on average, in random write workloads). Thus, to move only 1∼2 values, one meta-segment update is required.

To avoid this, PinK takes an approach that delays updates of meta segments in the flash. PinK writes valid KV pairs to L_0 again and then just erases the victim block. Corresponding meta segments now point to wrong flash pages erased by GC, but this is not a problem at all. Read requests to the rewritten KV pairs are served by higher levels, and old entries in the meta segments will eventually be discarded during compaction later. This approach slightly 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.

Note that since KV pairs sitting in lower levels are moved to L_0 during GC, it possibly hurts read latency. However, it does not affect the worst-case read latency, which is one of our design goals, because it is guaranteed by the number of pinned levels.

5 Experiments

We present experimental results on PinK. 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?

5.1 Experimental Setup

We have implemented PinK on our FPGA-based SSD platform with quad-core ARM Cortex-A53 (Xilinx ZCU102 [42]). The FPGA is used to implement HW accelerators and flash chip controller. The SSD platform has a 256GB custom flash array card. The size of a page is 8 KB, and the number of pages per block is 256. (See §3.2 for more detailed performance numbers.) It is connected to a host through 10 GbE (1.25 GB/s) whose bandwidth is high enough to saturate the

Table 1: 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* (*RMW)
Query type	Point					Range	Point
Request distribution	Uniform	Zipfian			Latest [9]	Zipfian	

maximum throughput of the flash array card. The I/O queue depth is set to 64, which is sufficient to fully utilize the parallelism of 8-channel and 8-way in our flash array card. We scale down the SSD capacity to 64GB, and DRAM for KV indexing structures (*e.g.,* the level lists and pinned meta segments) is set to 64MB – 0.1% of the SSD capacity.

We evaluate PinK using seven workloads from YCSB, a realistic cloud benchmark [9]. The details of the workloads are described in Table 1. Default key and value sizes are set to 32B and 1KB, respectively, which represent averages of common KV workloads [3]. For evaluation, we first created a 44GB KV pool on the 64GB SSD ('Load' in Table 1) – total 44M unique KV pairs are written. Then, we ran each workload ('A'∼'F' in Table 1) which sends 10M KV requests to the loaded data set. We initialized the SSD with the Load phase before any other workload executed. On the host, 64 YCSB clients ran simultaneously to maximize throughput. With 44GB data, the storage utilization was 69%. We assigned 10% of the SSD capacity (*i.e.,* 6.4GB), for over-provisioning.

To compare with PinK, we have implemented a hash-based KV-SSD based on what we described in §2.3. The KV-SSD denoted by `Hash` uses a 8-bit signature for each KV pair to balance a hash-table size and a signature collision rate. Note that, in our experimental setup with a relatively small data set, the 8-bit signature is large enough to provide a low collision rate. It requires 320MB of the hash table, which is much larger than the 64MB of DRAM for indexing. Therefore, `Hash` keeps only popular buckets in DRAM using the LRU replacement policy. `Hash` uses additional 1MB DRAM for a write buffer.

We compare `Hash` with two PinK configurations: one with no HW accelerator (`PinK`) and the other with HW accelerators (`PinK+HW`). The conventional LSM-tree implementation based on LightStore [7] (`LSM-tree`) is included for our evaluation. `LSM-tree` is equivalent to PinK, except that it does not employ the optimization techniques explained from §4.2 to §4.5. For `PinK`, `PinK+HW`, and `LSM-tree`, the number of total levels is set to 5. `PinK` and `PinK+HW` pin top-3 levels, $k = 3$. The meta segment size is the same as an 8KB page size. With 8KB meta segments, the amounts of DRAM for the level lists is 10MB (including both prefix and range pointers). The rest of DRAM, 54MB, thus can be used to pin levels. `LSM-tree` uses 9MB for level lists and 55MB of DRAM for bloom filters. As in `Hash`, for L_0 (a write buffer), 1MB DRAM is additionally assigned to `PinK`, `PinK+HW`, and `LSM-tree`.

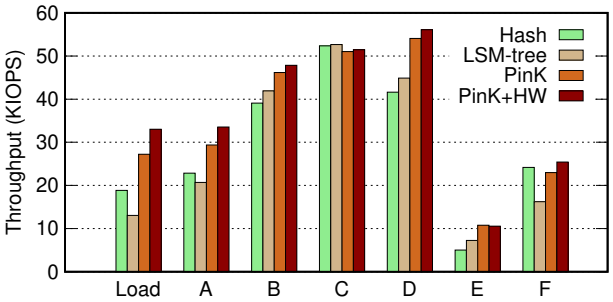


Figure 8: Overall throughputs of the four KV-SSD setups

5.2 Performance Analysis

YCSB Throughput: We measured IOPS of the four KV-SSD setups (`Hash`, `LSM-tree`, `PinK`, and `PinK+HW`) using YCSB. Figure 8 shows the results. `PinK+HW` outperformed `Hash` and `LSM-tree`, providing 37% and 44% higher throughputs, on average, respectively. `LSM-tree` suffered seriously from high CPU overheads caused by rebuilding bloom filers as well as sorting KV pairs. By eliminating bloom filters and reducing compaction I/Os, `PinK` improved IOPS by 34%, on average, over `LSM-tree`. Using the HW accelerators for sorting further improved the performance. As depicted in Figure 8, `PinK+HW` achieved 7.2% higher IOPS than `PinK` on average.

Those benefits of PinK were evident for the workloads with many writes. For Load, YCSB-A, and YCSB-F, we observed that `PinK+HW` improved IOPS by 56∼152% and 10∼21% over `LSM-tree` and `PinK`, respectively. Even with the workloads having relatively small writes (*i.e.,* YCSB-B, D), `PinK+HW` exhibited 14∼24% and 3% higher IOPS than `LSM-tree` and `PinK`, respectively. For the read-only workload, YCSB-C, no performance benefits were observed with `PinK` and `PinK+HW`.

One of the observations we did not expect was that PinK significantly outperformed `LSM-tree` for YCSB-D which issues only a small number of writes. This was due to the somewhat unique I/O behavior of YCSB-D that read recently-written KV pairs frequently. In PinK, recently-written KV pairs were stored in top levels pinned to DRAM. Thus, the majority of `GET()` requests were directly served by pinned levels, avoiding flash I/Os.

`LSM-tree` performed worse than `Hash` for the write-intensive benchmarks (Load, YCSB-A and F) owing to CPU overheads, but exhibited higher IOPS for the read-oriented workloads (YCSB-B, C and D). For YCSB-E with range queries, the LSM-tree-based KV-SSDs showed much higher IOPS than `Hash`, thanks to their sorted indexing structure.

Impact of Level Pinning: Figure 10 shows the impact of the level pinning on read and write I/O counts. As shown in Figure 10(a), `PinK` reduced the number of flash reads per query by 33% and 62% over `LSM-tree` and `Hash`, respectively. Since `PinK` pinned exact KV indices in DRAM, it eliminated many flash reads.

`Hash` was badly affected from hash misses and collisions. `Hash` maintained only signatures in DRAM. Thus, even when

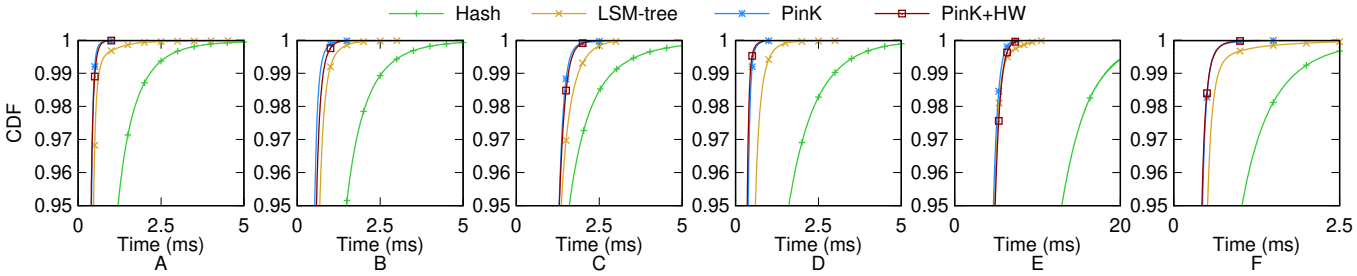


Figure 9: CDF graphs of read latency of `Hash`, `LSM-tree`, `PinK`, and `PinK+HW` under YCSB

it has hits on `SET()` requests, it had to retrieve exact keys from flash unless designated buckets were empty. `LSM-tree` exhibited two flash page reads per query: one for a KV index and the other for a value (1 KB). This is because Monkey bloom filters [10] used in `LSM-tree` requires one read to fetch indices, on average. For YCSB-D and E, the number of reads per query was less than 2. Since YCSB-D tends to read recently written KV pairs, many of `GET()`s were directly served by L_0 or pinned levels. YCSB-E contained range queries, so `LSM-tree` could fetch several desired KV indices by one read.

Figure 10(b) shows the percentage of compaction I/O out of the total I/O for LSM-tree operations (both reads and writes). By absorbing many index updates in pinned levels, it reduced the number of compaction I/Os by 52% over `LSM-tree`. Except for Load and YCSB-A with many writes, compaction I/Os only accounted for less than 20% of the total I/Os. However, as shown in Figure 8, the negative impact of compaction I/O ratio on the throughput was significant.

YCSB Read Latency: Figure 9 shows CDF graphs of read response times of the four KV-SSD setups. Table 2 also lists average, 99^{th}, 99.9^{th}, and 99.99^{th} percentile read latency of `Hash`, `LSM-tree`, and `PinK`. As expected, `PinK` and `PinK+HW` showed better average latency with shorter tails compared to the others. Thanks to bloom filters, `LSM-tree` performed fairly well compared to hash-based one, but had long tails as expected. `Hash` suffered from long tails due to multiple flash I/Os caused by hash misses and collisions. YCSB-E showed longer latency than the others because it issued range queries that carry multiple `GET()` commands.

Impact of Search Path Optimization: To understand the impact of the search path optimization, we carried out experiments with optimization techniques enabled one by one. `NO-OPT` represents PinK with no optimization, `Range` is PinK with range pointers, and `ALL` is with both range pointers and prefix. We used Load and YCSB-C workloads.

Figure 11 (a) shows the throughputs under Load and YCSB-C. For Load, there were slight performance drops as the optimization technique was added. This was due to overheads required for managing additional data structures. These were not significant. For YCSB-C with 100% reads, high throughput improvements were observed. In particular, `ALL` exhibited almost the same read throughput as `LSM-tree`. This means that the search overheads were almost eliminated. Figure 11 (b) presents the CDF of read latency under YCSB-C. We observed similar performance trends. `ALL` showed almost the same read latency as `LSM-tree` but with shorter tails.

Garbage Collection: With all the workloads of YCSB, GC did not involve many valid page copies. This was because almost all of the victim blocks were meta-segment blocks that held invalid KV indices. To simulate a situation where GC severely triggered, we designed another set of experiments. We first created a KV pool with 44M unique KV pairs, and then ran a synthetic workload that issued 100M `SET()`s with uniformly random keys to overwrite existing KV pairs. WAF reached 3.27 and became stable with little fluctuation after 90M `SET()`s were issued. This indirectly confirms that we issued sufficient I/Os to induce heavy GC I/O traffic.

Figure 12 analyzes the number of page writes issued during GC. `Hash` involved a smaller number of page writes for GC than `PinK`. After moving valid flash pages, both `Hash` and

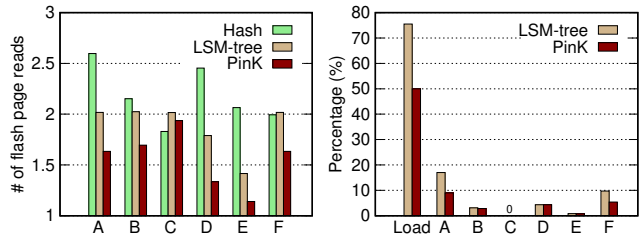
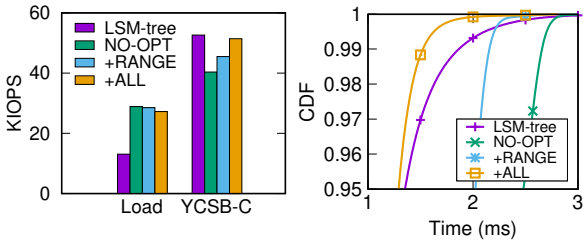


Figure 10: The impact of the level pinning on flash read I/Os (a) and compaction I/Os (b)

Table 2: Comparison of average and tail latency (unit: μs)

	Percentile	A	B	C	D	E	F
`Hash`	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-tree`	Average	302	395	722	294	3,142	329
	99^{th}	640	960	1,870	890	5,790	680
	99.9^{th}	1,700	1,630	2,680	1,370	8,800	1,890
	99.99^{th}	5,250	3,140	3,450	3,210	10,740	3,750
`PinK`	Average	236	290	732	161	3,027	248
	99^{th}	490	700	1,820	490	5,550	540
	99.9^{th}	670	1,040	2,180	720	6,640	800
	99.99^{th}	1,300	1,800	2,370	1,060	7,590	1,540



(a) Throughput (b) Read Latency

Figure 11: Impact of search path optimizations

PinK have to update in-flash hash buckets or meta segments so that they point to the new locations of the moved pages (denoted by 'KV Indices' in Figure 12). Since a bucket size of Hash (8B signatures) is smaller than that of PinK (32B keys), more buckets are packed into a single flash page for Hash. Thus, the number of flash page I/O for updating KV indices becomes smaller than that of PinK. Even worse, PinK suffered from extra compaction I/Os.

PinK+GCOPT addresses this problem by rewriting victim KV pairs to L_0, instead of directly updating meta segments (see §4.5). 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 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 tell us that the compaction I/O cost of the LSM-tree, which is considered a major reason that makes people choose the hashing rather than the LSM-tree, is actually not a serious problem in achieving high I/O performance.

Read Latency and LSM-tree Height (h): Until now we have assumed that h and k are fixed to 5 and 3, respectively, and except for the last level, the rest is pinned to DRAM. As explained earlier (§4.2), this is a reasonable setup given that it required DRAM as small as 0.1% of flash storage and modern SSDs have more DRAM than that. However, to improve write performance further [26], one might want to increase the height of the tree. Unfortunately, as the tree gets

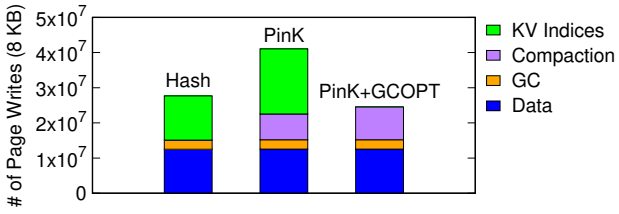
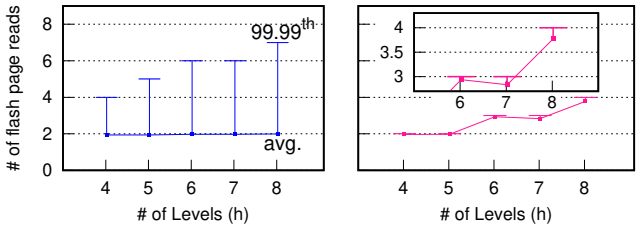


Figure 12: Analysis of GC cost: 'Data' represents pages written by SET(). 'GC' indicates pages written to move valid values for GC. 'Compaction' represents pages written to meta segments during compaction. 'KV Indices' indicates pages written to update meta segments or in-flash hash indices.



(a) Bloom filter (b) PinK

Figure 13: The number of flash page reads with varying h

taller, PinK cannot pin all the higher levels to DRAM. Given 64MB DRAM, for example, for h = 6, 7, and 8, the amount of DRAM required to pin all the levels but the last one are 176, 292, and 437MB, respectively. For h = 6 and 7, PinK cannot pin two lowest levels, and, for h = 8, the last three levels cannot be pinned. Even in such cases, the worst-case read latency can be guaranteed, but it increases to 3 reads (for h = 6 and 7) and 4 reads (for h = 8).

To understand its impact, using YCSB-C (100% reads), we measured the number of flash reads per query with various tree heights (h). Figure 13 shows the average read counts and 99.99th percentile read counts of LSM-tree and PinK. The average read count of LSM-tree was close to 2. Again, regardless of h, Monkey required one flash read for fetching KV indices, on average. However, owing to its probabilistic nature, the tail latency increased greatly, and the gap between the tail and the average got wider as h increased.

Unlike LSM-tree, PinK exhibited stable read counts. While the average read count increased along with h, the worst-case read count was guaranteed as $O(h-k-1)$. For YCSB-C, there were no huge differences between the average and the tail read counts. This is because YCSB-C had low temporal locality and thus the majority of GET() were served by the flash-resident last level This experimental results confirm that PinK can provide more *stable* read latency even when h is set high and all the levels cannot be pinned to DRAM.

6 Conclusion

We have presented a novel LSM-tree-based KV-SSD design, called *PinK*. By pinning KV indices of top levels of the LSM-tree to DRAM, PinK is able to guarantee the worst-case read latency, while improving average read latency. Moreover, by combining the level pinning with hardware accelerators, PinK not only eliminated sorting overheads, but reduced I/O operations related to compaction greatly. Our experimental results show that PinK outperformed existing hash-based KV-SSDs in tail read-latency, average read-latency, and I/O throughput. In future, we plan to explore the idea of the level pinning in general-purpose KVS like RocksDB. We think the main challenge in realizing this idea is providing a small amount of battery-backed DRAM which can be referenced by the host CPU along with the normal system DRAM.

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] ASHKIANI, S., LI, S., FARACH-COLTON, M., AMENTA, N., AND OWENS, J. D. GPU LSM: A Dynamic Dictionary Data Structure for the GPU. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium* (2018), pp. 430–440.

[3] 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.

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

[5] 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.

[6] CHANDRAMOULI, B., PRASAAD, G., KOSSMANN, D., LEVANDOSKI, J., HUNTER, J., AND BARNETT, M. Faster: A Concurrent Key-value Store with In-place Updates. In *Proceedings of the ACM International Conference on Management of Data* (2018), ACM, pp. 275–290.

[7] 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.

[8] COLGROVE, J., DAVIS, J. D., HAYES, J., MILLER, E. L., SANDVIG, C., SEARS, R., TAMCHES, A., VACHHARAJANI, N., AND WANG, F. Purity: Building Fast, Highly-Available Enterprise Flash Storage from Commodity Components. In *Proceedings of the ACM International Conference on Management of Data* (2015), p. 1683–1694.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

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

[17] 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.

[18] 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.

[19] KAI REN, G. G. TABLEFS: Enhancing Metadata Efficiency in the Local File System. In *Proceedings of the USENIX Annual Technical Conference* (2013).

[20] 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.

[21] 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).

[22] KOURTIS, K., IOANNOU, N., AND KOLTSIDAS, I. Reaping the performance of fast NVM storage with uDepot. In *Proceedings of the USENIX Conference on File and Storage Technologies* (2019), pp. 1–15.

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

[24] LEE, C.-G., KANG, H., PARK, D., PARK, S., KIM, Y., NOH, J., CHUNG, W., AND PARK, K. iLSM-SSD: An Intelligent LSM-tree Based Key-Value SSD for Data Analytics. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems* (2019), pp. 384–395.

[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] LUO, C., AND CAREY, M. J. LSM-based Storage Techniques: a Survey. *The VLDB Journal* (2019).

[27] 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.

[28] NGD SYSTEMS, INC. NGD Catalina NVMe SSD. https://www.ngdsystems.com/products/, 2018.

[29] 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.

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

[31] 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.

[32] SAMSUNG ELECTORNICS. Samsung Smart SSD. https://samsungatfirst.com/smartssd-ocp/, 2018.

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

[34] 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.

[35] 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.

[36] SAMSUNG ELECTRONICS. 860EVO SSD Specification. https://www.samsung.com/semiconductor/global.semi.static/Samsung_SSD_860_EVO_Data_Sheet_Rev1.pdf, 2018.

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

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

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

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

[41] WANG, J., ZHANG, Y., GAO, Y., AND XING, C. pLSM: A Highly Efficient LSM-Tree Index Supporting Real-Time Big Data Analysis. In *Proceedings of IEEE Annual Computer Software and Applications Conference* (2013), pp. 240–245.

[42] XILINX. Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit. https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html, 2018.

[43] 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.