Design of LSM-tree-based Key-value SSDs with Bounded Tails

JUNSU IM and JINWOOK BAE, Daegu Gyeongbuk Institute of Science and Technology, Republic of Korea
CHANWOO CHUNG and ARVIND, Massachusetts Institute of Technology, USA
SUNGJIN LEE, Daegu Gyeongbuk Institute of Science and Technology, Republic of Korea

Key-value store based on a log-structured merge-tree (LSM-tree) is preferable to hash-based key-value store, because an LSM-tree can support a wider variety of operations and show better performance, especially for writes. However, LSM-tree is 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 99th 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. We also find that PinK is able to provide a flexible design space for a wide range of KV workloads by leveraging the read-write tradeoff in LSM-trees.

CCS Concepts: • **Computer systems organization → Embedded systems**; • **Information systems → Database management system engines**;

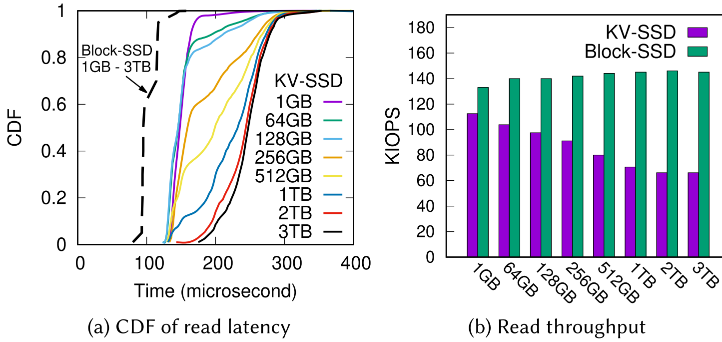
Additional Key Words and Phrases: Log-structured merge-tree, key-value store, key-value SSD, tail latency

ACM Reference format:
Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. 2021. Design of LSM-tree-based Key-value SSDs with Bounded Tails. *ACM Trans. Storage* 17, 2, Article 10 (May 2021), 27 pages.
https://doi.org/10.1145/3452846

1 INTRODUCTION

Offloading the **key-value (KV)** functionality onto a storage device has received a lot of attention recently from both academia and industry [11, 22, 25, 33, 50]. A representative device in this class is

An earlier version of this article was presented at the 2020 USENIX Annual Technical Conference, July 15-17, 2020 [21]. This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1701-11. We also thank Samsung Electronics for providing KV-SSD prototypes. The DGIST team was supported by the National Research Foundation (NRF) of Korea (NRF-2018R1A5A1060031). Arvind and Chanwoo Chung was partially funded by NSF (CCF-1725303) and Samsung Semiconductor (GRO grants).
Authors' addresses: J. Im, J. Bae, and S. Lee (corresponding author), Daegu Gyeongbuk Institute of Science and Technology, 333, Techno Jungang-daero, Dalseong-gun, Daegu, Republic of Korea; emails: {junsu_im, jinwook.bae, sungjin.lee}@dgist.ac.kr; C. Chung and Arvind, Massachusetts Institute of Technology, 77 Massachuestts Ave, Cambridge, MA; emails: {cwchung, arvind}@csail.mit.edu.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2021 Association for Computing Machinery.
1553-3077/2021/05-ART10 $15.00
https://doi.org/10.1145/3452846



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

Fig. 1. Performance comparison of KV-SSD and Block-SSD depending on the total amount of data stored (1GB~3TB).

Samsung's **key-value SSD (KV-SSD)** [25], which directly serves KV requests. By offloading most commonly used operations of KV databases (e.g., RocksDB [17]), 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.

KV-SSDs are promising, but the current proposals and devices often provide inconsistent tail latency and throughput. This is because most KV-SSDs are hash-based [16, 22, 25, 33, 47, 50], 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, then multiple flash accesses are required, resulting in long and unpredictable tail latencies and a drop in throughput.

To understand the behavior of hash-based KV-SSDs, we conducted a set of experiments on a 4-TB KV-SSD prototype (KV-PM983 [41]). We created KV pools ranging in size from 1 GB to 3 TB and chose the average key and value sizes to be 32 B and 1 KB, respectively [5]. Thus, a 3-TB KV pool would hold 3 billion KV pairs. We ran random `GET()` requests on these KV pools for 10 minutes using KVBench [39]. No **garbage collection (GC)** occurred during our experiments.

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 (1-GB pool) to 245.31 μs (3-TB pool). We also observed long tail latency: For the 99.99th percentile, the tail latency increased from 323 to 1,020 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 further supported by another experiment, where we used the same setup to run FIO [6] on a 4-TB Block-SSD [42], which loads its **flash translation layer (FTL)** table in DRAM for 4-KB 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 hash is **log-structured merge tree (LSM-tree)** [35]. The tail latency in such a system is bounded by the number of levels in the tree. Since an LSM-tree indices KV pairs in a

multi-level *sorted* tree, it might also require a much smaller DRAM for indexing KV pairs. LSM-tree also supports range-queries and scans efficiently, without any extra bookkeeping or support from KV clients [25]. Our experiments, however, revealed that a conventional implementation of LSM-tree on an SSD controller failed to deliver the promised benefits. In fact, it showed worse performance than hash 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 hash-based KV-SSD. The second problem was high write-amplification. Even if we use key-value separation like Wisckey [31], compaction incurs many extra storage accesses as it sorts KV indices. Moreover, this LSM-tree compaction cost exacerbates the FTL's 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 article, 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 bounded 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 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.

In addition, we show PinK extends the tradeoff of the LSM-tree to make use of the number of pinned levels. The LSM-tree can provide the various design space by using tradeoff between read latency and write throughput. The knob of the tradeoff is the height of the LSM-tree. As we increase the height of LSM-tree, it provides a better write throughput at the cost of the read latency. However, PinK has another tradeoff knob that is the number of pinned levels. The level-pinning can improve both read latency and write throughput by using DRAM for pinning instead of keeping bloom filters. By configuring the height of the LSM-tree and the number of pinned-levels, PinK can be designed more flexibly than the original LSM-tree.

To demonstrate our idea on PinK, we have implemented PinK on MIT's FPGA-based SSD platform [23] and used the LSM-tree implementation of LightStore [11] as our starting point, because its source code is publicly available. Using YCSB [13] benchmarks, we have shown that PinK outperforms existing KV-SSD designs in several aspects. Compared to a hash-based KV-SSD, PinK reduces 99th percentile tail latency by 73%, improves average read latency by 42%, and shows 37% higher throughput. Furthermore, compared to LightStore, PinK reduces 99th percentile tail latency by 22%, improves average read latency by 22%, and shows 44% higher throughput. We also show the flexible design space of PinK. For example, for a read-sensitive workload, PinK saves about 2x memory usage while supporting same read tail-latency but sacrificing only 9% 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. Section 5 analyzes the tradeoff in PinK and its wide design space. Section 6 presents experimental results. We conclude in Section 7.

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 4-KB logical blocks that are accessed by block I/O primitives (e.g., `READ` and `WRITE`). A FTL in the SSD firmware is responsible for providing the block I/O interface [3]. 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 [40, 44]. 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 built-in capacitors to guard against sudden power failures [7]. Similarly to other log-structured systems [37], the FTL has to perform GC to reclaim free space.

2.2 KV-SSD

A KV-SSD is a new type of SSDs [25, 46] that 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 [46] or even be a character string, and a value can be as big as 2 MB [46]. 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 [25, 26]. A more detailed description can be found in SNIA's KV-SSD specification [46].

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 [33], KAML [22], and BlueCache [50]) and industry (e.g., Samsung's KV-SSD prototype [25, 41]). All these KV-SSDs are based on 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 [16, 22, 33, 50]. A primary design issue of 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 32 B and 1 KB, respectively [5]. If the number of buckets is 2^{32} ($= 2^{42}/2^{10}$) and the bucket size is 36 B (32 B for a key and 4 B for a pointer), then 144 GB of DRAM is required to hold the complete hash table. If KV-SSDs have large-enough DRAM to hold the entire hash table, in addition to the $O(1)$ time complexity for calculating an index, then a KV access only takes $O(1)$ flash access to read/write the KV pair [45]. However, as mentioned previously, SSDs do not have as much DRAM.

To reduce DRAM usage, some use signatures [9, 16, 27, 47, 50]. 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, 24 GB of DRAM is required. But it causes *signature collision*, which happens when different keys have the same signature. The 24-GB size of the hash table is still huge to keep in 4 GB of DRAM in a typical 4-TB SSD. The DRAM requirement can be further reduced by keeping only popular buckets in a fixed-size DRAM (e.g., 4 GB) while storing the rest in the flash [18]. This, however, causes extra flash reads.

If a designated bucket is not available in DRAM, i.e., *hash table miss*, we have to fetch the bucket from the flash to find the location of a desired KV object. Consequently, the table miss increases flash read costs from $O(1)$ to $O(1+\alpha)$ where α is a miss ratio. Even worse, signature collisions add an unpredictable number of flash reads until the collision resolves, resulting in unbounded read tail latency in the worst case. As shown in Figure 1, this instability of the hash-based KV-SSD deteriorates as the hash table grows.

This inconsistent performance may be due to inefficient collision resolution policies. There are advanced hash strategies, such as Cuckoo [36] and Hopscotch [19, 27], 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. Hash algorithms also cannot efficiently support range and scan operations [25].

2.4 LSM-Tree-based KV-SSD

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 [17] and Cassandra [28]). It is also used in big all-flash array systems such as Purity [12]. Because of its increasing popularity across a variety of systems, many LSM-tree variants have been proposed [4, 24, 48]. Recently LSM-tree has also been used in some implementations of KV-SSDs like LightStore [11] and iLSM-SSD [29].

LSM-tree is a hierarchical structure that consists of multiple trees, each called a *level*. Each level is sorted and behaves as a write buffer for the next level, which has a larger size. Since LSM-tree keeps only the highest level in DRAM, its memory requirement is much smaller than hash. Also, a KV access in LSM-tree requires at most $O(h-1)$ flash accesses owing to its sorted nature, where h is the number of levels, and thus the worst-case latency of LSM-tree is bounded. Many LSM-tree implementations use Bloom filters to improve the average read cost to $O(1)$ flash access [14].

However, LSM-tree-based KV-SSDs have suffered from a lack of CPU power for the compaction process, which is required to keep LSM-tree balanced. In addition, Bloom filters, which is used to skip some levels probabilistically, cannot reduce the worst-case tail latencies to a single read. The design of PinK was motivated by these and some additional inefficiencies in running the conventional LSM-tree on resource-constrained KV-SSDs; we discuss these inefficiencies in Section 3.

2.5 Hash vs. LSM-Tree

NAND flash scales faster than DRAM. According to Reference [20], the capacity of NAND flash has increased 1.43 times per year, while that of DRAM has increased 1.13 times. This requires us to take into account a DRAM scalability issue in choosing algorithms for KV-SSDs. Assuming the same trend, the hash suffers from more degradation in read performance, since the table miss rate becomes higher as NAND flash scales further. The LSM-tree also experiences degradation, since it uses fewer bits per KV pair for Bloom filters. However, the read performance of the LSM-tree is more scalable due to the space-efficient structure of Bloom filters. Monkey has almost the same read performance when the ratio between the size of DRAM and total dataset decreases from 0.16% to 0.02% (see Figure 11(a) in Reference [14]). Conversely, the hash suffers from notable performance degradation when the DRAM size does not grow relative to the dataset as shown in Figure 1. The LSM-tree also exhibits lower update costs than the hash when the entire indices do not fit in DRAM. In the hash, indices in the map have to be updated in place, which in turn involves expensive read-modify-writes in the flash. However, the LSM-tree shows cheaper update costs, since it appends new or updated entries to the flash thanks to its leveled structure. As a result, the LSM-tree offers better write performance when DRAM becomes less sufficient.

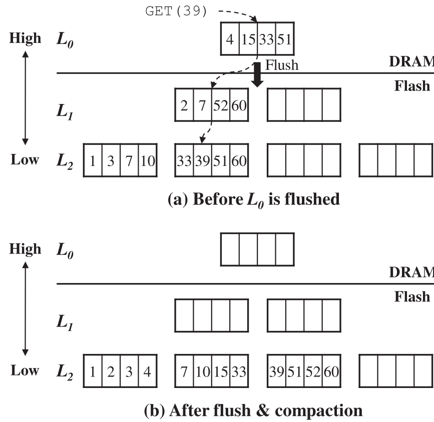


Fig. 2. LSM-tree organization ($h = 3$, $T = 2$). A rectangle represents a KV object, and the number inside is the key.

3 CHALLENGES IN IMPLEMENTING LSM-TREE IN A KV-SSD

In this section, we analyze the performance and present key technical challenges when an LSM-tree is implemented in a resource constrained environment of an SSD controller. We have used LightStore [11] as the baseline for an LSM-tree-based KV-SSD that separates keys and values to speed up writes and uses Bloom filters to speed up reads. We expect iLSM-SSD [29] to exhibit similar read and write performance as LightStore, because it follows same concepts.

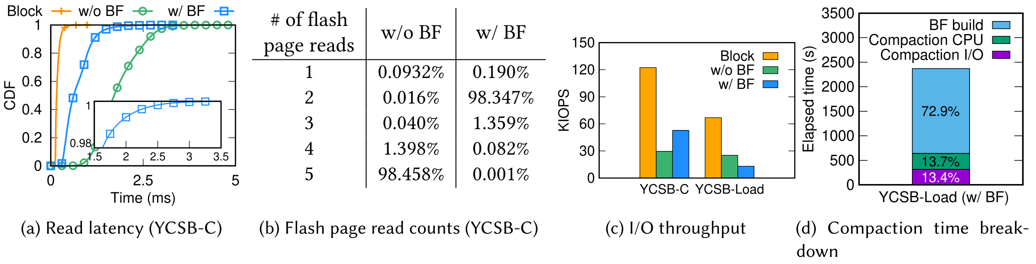
3.1 LSM-Tree Structure

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 (see Figure 2). 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 the LSM-tree, 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 that are not affected by compaction (see Wisckey [31]).

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



# 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%

(a) Read latency (YCSB-C) (b) Flash page read counts (YCSB-C) (c) I/O throughput (d) Compaction time breakdown

Fig. 3. Experimental results of the conventional implementation of LSM-tree on an SSD controller.

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 [14, 15]. Usually, each level or run in the tree has its own filter.

3.2 Performance Analysis

Our PinK implementation uses the same FPGA-based hardware platform as LightStore [11], which has quad-core ARM Cortex A53 running at 1.2 GHz and 4 GB DRAM. This controller specification is similar to those of latest SSDs with in-storage computation capability [34, 38]. PinK is equipped with a 256-GB NAND flash card that provides 1.1 GB/s read and 600 MB/s write throughputs, respectively, and offers 122,349 IOPS for 4 KB reads and 66,843 IOPS for 4 KB writes, respectively.

To understand the weaknesses of the conventional LSM-tree implementations, we first improved the Bloom filter implementation in LightStore [11] by replacing the original one with Monkey [14]. We leveraged AArch64 SIMD instructions in implementing Monkey. Key-value separation [31] was employed by default.

For fast evaluation, we reduce the SSD capacity to 64 GB. 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 64 GB) 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 32 B and 1 KB. We first run YCSB-Load with 44 million (44 GB) uniformly random KV pairs, and then run YCSB-C with 10 million Zipfian requests. Results with other workloads can be found in Section 6.

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, then 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 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 Wisckey, 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 Section 6.2), 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 CPU time does not include the Bloom filter reconstruction time, which will be discussed soon. The compaction overhead has been addressed by KVell [30], but it requires a huge DRAM to hold all indices, which is not available in KV-SSDs.

The cost of rebuilding Bloom filters is also high. Bloom filters should be rebuilt for newly created levels after compaction, which requires expensive hash computations and lots of memory accesses. In our experiment, a hash computation is accelerated by SIMD instructions, but its negative impact is still huge. The rebuilding overhead for Bloom filters can be optimized as was done in Reference [8]. Even if we assume the reconstruction time improves significantly, say, 8×-11×, as in Reference [8], the Bloom filter reconstruction still takes 20–25% of total compaction time. Note that it is unclear whether such huge improvement is achievable in ARM-based SSD controllers. 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-tree exhibits higher average-latency because of multi-level search and also exhibits 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. 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 (Section 4.2 and Section 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 that are already pinned to DRAM. The throughput can be further improved by using hardware accelerators that performs compaction for pinned and flash-resident levels (Section 4.4). Finally, to alleviate the GC costs associated with compaction, we delay GC by putting updated KV indices in L_0 (Section 4.5). This reduces the write amplification, which affects lifetime of SSDs.

4.1 Overall Architecture

PinK supports variable-sized keys (∼16 B–128 B) and values (∼1 KB–2 MB), 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

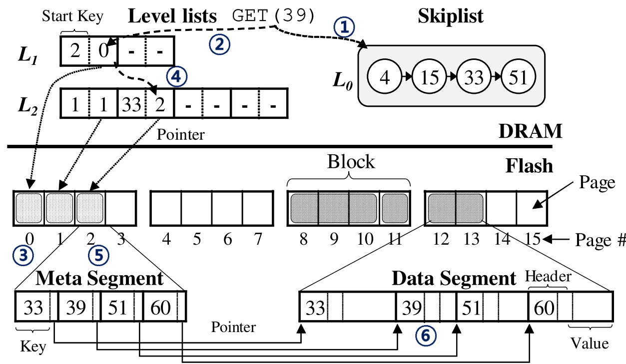


Fig. 4. An overall architecture of PinK with its key data structures in DRAM and flash. The tree hierarchy and KV objects are identical to those of Figure 2(a). Given `GET(39)`, ① PinK first looks up the skiplist (L_0). Since a matched one is not found, ② it goes down to L_1, ③ reading a meta segment 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)`.

operations [7, 25, 43]. 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 that are different from the 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 the LSM-tree-based KVS combined with Wisckey [31], but it is optimized to maintain compact data structures in the 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 that buffers incoming KV objects temporarily. The size of L_0 is configured to be large enough (e.g., ~8 MB–64 MB) 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 Section 4.5). The size of a meta segment is fixed to a flash page size (e.g., ~8 KB–16 KB), 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 4-B integer, but a key size varies from 16 B to 128 B. 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 16 KB, then it contains up to 1,024 <key, pointer> pairs where at most 2 KB is used as a header. For HW accelerators, a header and <key, pointer> pairs are aligned to 16 B for simple implementation. We discuss this in Section 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., $h = 5$, then there are four level lists except for L_0. Each level list is organized as an array of pairs of fixed-sized pointers (4 B each, 8 B 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 (16–128 B). 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 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 4 TB and each meta segment is 16 KB. As in Section 3, the average sizes of keys and values are 32 B and 1 KB, respectively [5]. Each entry in a meta segment is 36 B (32-B key and 4-B pointers). A 16KB meta segment can hold 398 <key, pointer> pairs. In a 4-TB SSD, there exist 2^{32} 1-KB objects, and thus the number of meta segments in the flash is about 10.8 M (= $2^{32}/398$). 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 32 B. Thus, only 432 MB (= 10.8M×(8 B + 32 B)) 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 are typically used to avoid useless lookups on levels that do not have desired keys [14, 15]. As pointed out earlier, however, it cannot avoid long tails and causes high CPU costs.

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, then 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 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 five levels, the amount of DRAM required to pin meta segments for L_1, L_2, L_3, and L_4 are 0.91 MB, 50.86 MB, 2.83 GB, and 161.63 GB, 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., 4 GB DRAM for 4 TB SSD). The data structures of PinK do not require large amounts of DRAM (e.g., 432 MB), 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 us 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

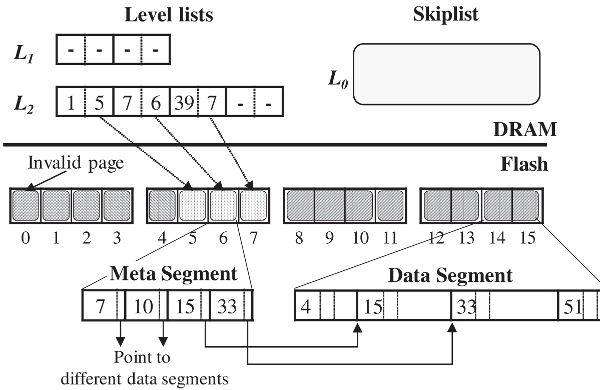


Fig. 5. The DRAM and flash layouts of PinK after L_0 (in Figure 4) is flushed out with compaction. The tree hierarchy and KV objects are identical to those of Figure 2(b).

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 of 5 reads and 5 writes that occurs while conducting the compaction.

Owing to the built-in capacitor, dirty meta segments pinned to DRAM do not need to flush out to flash. However, unlike high-end enterprise SSDs, some SSDs do have not enough capacitors to protect all the meta segments in DRAM. Running PinK on such devices results in a meta durability issue. PinK can address this issue by flushing out dirty pinned meta segments to flash after each compaction. Thanks to the key-value separation technique, this has little overhead. We demonstrate this in Section 6.4. While the capacitor-less setup does not reduce much flash writes for compaction, it still keeps and makes use of pinned levels, saving flash reads for KV operations and compaction.

4.3 Optimizing Search Path

When the LSM-tree retrieves a KV pair, it has to perform binary search on level lists until it finds a matching meta segment for a given key. This does not cause a serious overhead for higher levels (e.g., L_1 and L_2) whose level lists have few entries. However, 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 the LSM-tree 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 of computation is thus expressed as $O(h^2 \cdot \log(T))$. The conventional LSM-tree with Bloom filters offers much lower computation time on average, because it is able to skip binary search operations on unnecessary levels by performing membership tests first. PinK does not use Bloom filters and thus cannot exploit this benefit.

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

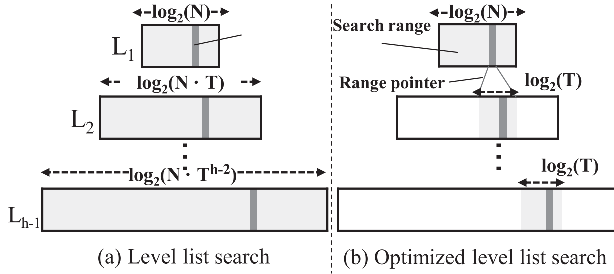


Fig. 6. Search path optimization with range pointers.

prefix that 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 by borrowing fractional cascading technique [10]. Each entry of a level list now has another 4 bytes 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, then 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 can be applied for lower levels, $L_3, \ldots, L_{h-1}$. Thus, the average 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 43.9 MB in the same setting as in Section 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.

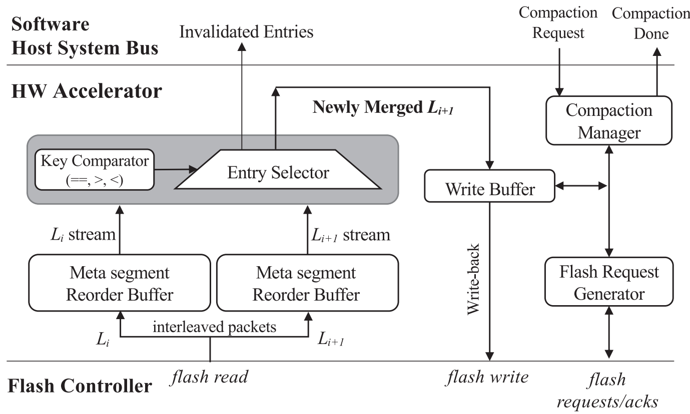


Fig. 7. Compaction accelerator for flash-resident levels.

Once we have sorted meta segment streams from two levels, the compaction engine (gray box in Figure 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 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 that was updated with a new one or removed by a client. Outdated KV indices are discarded from the LSM-tree during compaction (see Section 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, then 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, then 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,

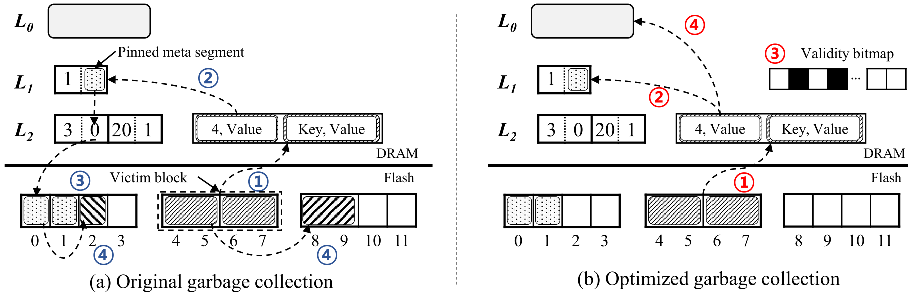


Fig. 8. Garbage collection optimization. (a) PinK original GC (based on Wisckey): ① When the victim block is selected, PinK reads all flash pages in the block. It should search the LSM-tree to figure out the validity of the objects in the victim block. For example, to figure out the validity of the KV pair whose key is "4," ② PinK looks up "4" in the LSM-tree as it would for a GET operation. Since L_1 is pinned level, PinK can check meta segments without flash read I/O. ③ In contrast, L_2 resides in flash. It needs to read the target meta segment. If the index of key "4" in the meta segment has the same physical address as the victim KV pair, then it is considered valid data. ④ The valid kv pair is copied to a new physical address and the meta segment is updated. This GC scheme makes a lot of flash I/Os. (b) PinK optimized GC: It uses a validity bitmap to reduce flash reads, and it delays updates of the meta segments to reduce flash writes. ① and ② are the same as the original one. ③ Instead of reading meta segments in the flash, PinK queries the validity bitmap whether the victim objects are still valid. If some objects are valid ④, then PinK rewrites them into L_0 to avoid directly update the meta segment.

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 Section 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 to DRAM, then 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 as illustrated Figure 8(a) and its caption, 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 to 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 a few values belong to the same meta segment (e.g., 1 to 2 values, on average, in random write workloads). Thus, to move 1 to 2 values, it requires one read and one write flash I/O to check the objects' validity and to update the meta segments. This job makes huge flash I/O traffic due to the relatively small size of a KV pair. For example, one 16-KB flash page stores 14 to 15 KV pairs. If all objects in a victim page are valid, then the KV-SSD should issue up to 30 I/O requests to copy all valid KV pairs.

We adopt two techniques for optimizing GC as shown in Figure 8(b). First, to avoid many read I/Os for checking validity, PinK keeps small amounts of a bitmap for the validity of values. The bitmap maps a select range of physical addresses to their validity. By using this information, KV-SSDs can examine whether the victim physical address is valid or not. Although using a bitmap

for improving GC is not unusual in the FTLs, there is a memory burden to keep the entire validity bitmap for all KV pairs. Instead, PinK keeps only a portion of flash blocks' bitmap in the memory for each flash-resident level. Whenever the compaction to the flash-resident level occurs, PinK picks several physical flash blocks depending on priority decided by GC policy, and the bitmap for that level is built for those blocks. Note that a level is freshly created in PinK during the compaction. Thus, for the select range of physical addresses, the level bitmap can be created by setting valid bits for those KV pairs, whose physical address belongs to the range, in the new level after compaction. To figure out valid KV pairs in each page of the victim block, PinK searches the indices of KV pairs in the victim block on the pinned levels to see if they are invalidated. If not found in pinned levels, instead of reading from flash-resident levels, then PinK queries the validity bitmap in the memory. Since the bitmap of each flash-resident levels contains the validity information, it may search all the bitmaps over the flash-resient levels to filter invalid KV pairs. If the bitmaps do not include the physical address pointed by the victim block, then PinK can do the same procedure as the original GC does. However, GC and compaction policy makes sure that, in most cases, victim KV pairs are mapped by the validity bitmaps.

To reduce write I/O to update meta segment, PinK takes an approach that delays updates of meta segments in 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 bounded by the number of pinned levels.

4.6 Durability and Scalability Issues

Durability with Limited Capacitor: We discussed the durability of pinned levels (meta segments) earlier in Section 4.2. However, PinK has other data structures that reside in DRAM , i.e., level lists and L_0. PinK can keep its durability by regularly writing the data structures into flash. L_0 can be durable by logging incoming write requests into a log area in the flash before processing the requests, just like the conventional LSM-tree does. Hash-based KV-SSDs also need to do the same task if its write buffers are not backed by capacitors. Just like meta segments, after compaction, PinK should also flush out newly updated level lists to the flash to make them persistent. Hash-based KV-SSDs have to write dirty KV indices to in-flash buckets as well whenever they are updated. PinK's corresponding metadata flush operations are less expensive than those of hash-based, thanks to the write-optimized structured of LSM-tree. The validity bitmap for optimized garbage collection does not need to be kept, since it can be recovered by compaction. When the system reboots, PinK reads much less data compared to the conventional LSM-tree for recovery. Both PinK and the conventional LSM-tree read 432 MB of level lists for a 4-TB KV-SSD. The conventional LSM-tree should eventually read the entire levels to regenerate all bloom filters, whereas PinK reads the top-K levels for pinning. As we mentioned in the previous section Section 4, PinK reads about 3 GB of metadata to pin top-4 levels. However, the conventional LSM-tree reads more than 160 GB of data, which is the total size of all five levels.

DRAM Scalability: We have also assumed that the size of DRAM scales as the size of flash in SSDs, i.e., DRAM capacity is kept 0.1% of the total capacity of flash. As mentioned in Section 2.5, the size of DRAM scales slower than that of flash. Thus, SSDs might ship with insufficient DRAM

to pin all the top levels. This problem can be resolved by pinning fewer levels. It increases the worst-case index lookup cost, but PinK offers better worst-case performance than the hash and the conventional LSM-tree. The details are discussed in the last experiment of Section 6.2 Another option we can consider is reducing the height of the tree so that all the levels, except for the last one, can fit into DRAM. This sacrifices write performance owing to increased compaction cost, but bounds $O(1)$ lookup cost in the worst case.

5 DESIGN SPACE OF PINK

We have discussed how the level-pinning technique reduces read latency in Section 4. In this section, we analyze the write performance of the LSM-tree when it adopts the level-pinning technique. Based on the analysis, we explain that PinK is able to extend the tradeoff of the LSM-tree, since it has an additional design parameter, the number of pinned levels (k), giving a broader design space.

5.1 Modeling PinK Write Overhead

Compaction Overhead: We first analyze the compaction write overhead of PinK. While the write amplification of the LSM-tree depends on both the height (h) and size factor (T), the read amplification of the LSM-tree depends only on the height (h) of the LSM-tree. Given the SSD capacity and height of the LSM-tree, the size factor can be derived. Each level in LSM-tree is T times the larger than the previous level. The total size of the LSM-tree is the same as the SSD capacity. When the size of L_0 is S_{L_0}, the relationship between T and h is expressed as Equation (1). When the LSM-tree height (h) gets taller, the size factor (T) becomes smaller,

$$\sum_{i=0}^{h-1}(S_{L_0} \cdot T^i) = \text{SSD capacity}$$
$$T \approx \text{SSD capacity}^{1/h} \tag{1}$$

When compaction is triggered at each level, the level is merged with the all the KV pairs in the next level and the new lower level is created. Thus, the write amplification of each compaction can be expressed as $O(T)$. It requires $h-1$ time compaction until a KV pair reaches the last level of the LSM-tree. The total write amplification of one SET operation is $O(T \cdot (h-1))$. LSM-tree's compaction I/Os are conducted in a batched manner. For example, each device unit I/O contains multiple KV objects. The write amplification overhead should be divided the number of KV pairs (B) in each I/O unit. As a result, the write amplification can be expressed as Equation (2),

$$O\left(\frac{T}{B} \cdot (h-1)\right). \tag{2}$$

If the built-in capacitor can protect all the in-memory data structures including pinned levels, then the cost of write is Equation (3),

$$O\left(\frac{T}{B} \cdot (h-k-1)\right). \tag{3}$$

Additional Compaction Overhead from GC: When PinK performs the garbage collection, it rewrites valid KV pairs into L_0, delaying the meta segment updates. This implies GC incurs additional compaction overhead. This overhead depends on how many valid KV pairs are in the victim block. The amount of valid data in the victim block is deeply relevant to the over-provisioning ratio (α, $0 < \alpha < 1$), which is the ratio of invisible area to the total area in SSD. Thanks to the over-provisioning area in SSDs, which effectively reduces the number of valid KV pairs in the victim block, the GC copying overhead is alleviated. Given a flash block size (S_B), each flash block has as many as $S_B \cdot (1-\alpha)$ valid bytes. After GC operation, PinK reclaims $S_B \cdot \alpha$ bytes of empty space. PinK can store the number of KV pairs until the reclaimed space becomes full. Thus, given the size

of a KV pair (P), we can calculate the additional number of KV pairs written into L_0 due to GC per each SET operation on SSD steady state as Equation (4). Note that the derivation assumes the worst-case bound under the greedy victim selection policy in GC.

$$\text{The number of rewritten KV pairs} \div \text{The number of KV pairs in reclaimed space} = \frac{\text{Size of valid data in victim block}}{\text{Size of KV pair}} \div \frac{\text{Size of reclaimed space}}{\text{Size of KV pair}} = \frac{(1-\alpha)\cdot S_B}{P} \div \frac{\alpha \cdot S_B}{P} = \frac{1-\alpha}{\alpha}. \tag{4}$$

Consequently, the final write amplification of PinK can be expressed by adding the original cost and the cost of rewriting valid KV pairs as Equation (5). When the built-in capacitors exist, the cost becomes Equation (6),

$$O\left(\frac{T}{B}\cdot(h-1)\cdot\left(1+\frac{1-\alpha}{\alpha}\right)\right) = O\left(\frac{T}{B}\cdot(h-1)\cdot\frac{1}{\alpha}\right), \tag{5}$$

$$O\left(\frac{T}{B}\cdot(h-k-1)\cdot\left(1+\frac{1-\alpha}{\alpha}\right)\right) = O\left(\frac{T}{B}\cdot(h-k-1)\cdot\frac{1}{\alpha}\right). \tag{6}$$

The LSM-tree offers a tradeoff between read latency and write throughput. As previous sections Section 4.2 and Section 5, its worst case read latency is given as Equation (7) and the write throughput is given as Equation (5),

$$O(h-1). \tag{7}$$

The read and write costs of the LSM-tree can be adjust by h. For example, when h gets bigger, the LSM-tree shows worse read latency and better write throughput. In PinK, since the pinned levels absorb flash I/Os in the GET operation and the compaction, the read and write costs have an extra parameter k in addition to h. The cost equations for PinK are given as Equations (8) and (6), which are derived by substituting h with $h-k$ in the LSM-tree's cost equations,

$$O(h-k-1). \tag{8}$$

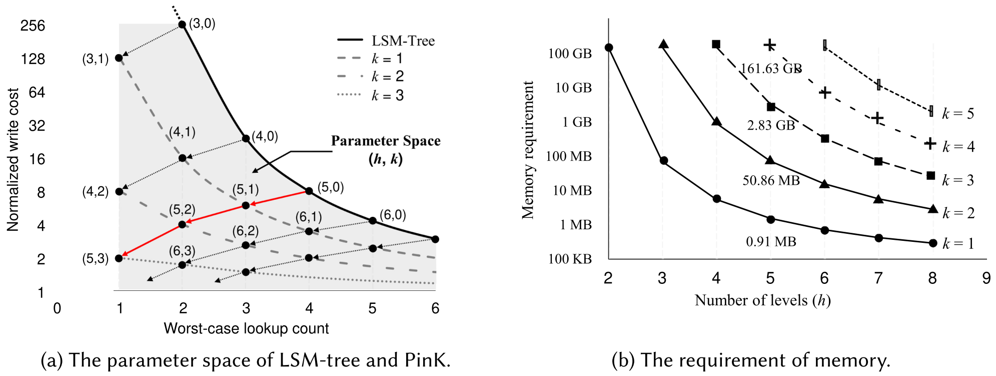
This shows that while the original LSM-tree's design space can be controlled with only h, PinK can provide a wider design space using an additional parameter k.

5.2 Extended Design Space of LSM-tree

The parameters in Equations (1) and (6), such as SSD capacity, B and α, are independent of h. Given the same SSD configuration and the workload, therefore, we can estimate the expected write and read performance depending on h using Equations (6) and (8). Figure 9(a) illustrates the read tail latency, i.e., worst-case lookup count, and write performance, i.e., normalized write cost, given parameters h and k. An outer curve with a bold line in Figure 9(a) shows the original LSM-tree performance tradeoff depending on h ($k = 0$).

The level pinning creates a broader range of the parameter space. In PinK, the read latency and write throughput depend on two parameters h and k. The inner lines in Figure 9(a) illustrate various combinations of the read latency and write throughput of PinK for different ks. As discussed earlier, level pinning requires additional memory. The memory requirement is modeled as $\sum_{i=0}^{k}(S_{L_0}\cdot T^i)$, where size factor T is decided by the h and SSD capacity. Figure 9(b) shows the amount of memory requirement for level pinning depending on h and k.

This broad parameter space makes it much easier for us to satisfy the client's 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., $h = 2$, *or* moderate write



(a) The parameter space of LSM-tree and PinK.

(b) The requirement of memory.

Fig. 9. Parameter space and memory usage of PinK while changing h and k. This figures are based on a 4-TB SSD with a 32-B average key size.

throughput, i.e., $h = 5$, but not both. In PinK, the height of the tree is set to 5 for good write throughput, but by pinning the top 3 levels, L_1, L_2, and L_3, read latency can be bounded (see red lines in Figures 9(a)). Such a decision should be made carefully considering the memory budget. Since the data structures of PinK are memory efficient and modern SSDs are equipped with large DRAM, pinning top 3 levels of the tree is feasible enough as labeled memory requirements in Figure 9(b), i.e., 0.91 MB, 50.68 MB, 2.83 GB, and 161.63 GB.

Furthermore, this parameter space can be used as a hint to determine the size of the capacitor-backed DRAM in the KV-SSD. Given SSD capacity and performance requirements, PinK can derive how many levels should be pinned and how much space the pinned levels take the capacitor-backed DRAM. In terms of the device cost, the tradeoff of PinK may give some choices to the SSD vendors.

6 EXPERIMENTS

We present experimental results on PinK. In particular, we seek to answer the following questions: (i) Does level pinning improve both read latency and write throughput along with shorter tails? (ii) Is the HW sorter able to reduce the compaction cost? (iii) What is the impact of GC on performance? (iv) Can PinK effectively tradeoff between increased performance and memory? and (v) How much does the performance of PinK degrade when capacitors are scarce?

6.1 Experimental Setup

We have implemented PinK on our FPGA-based SSD platform with quad-core ARM Cortex-A53 (Xilinx ZCU102 [49]). The FPGA is used to implement HW accelerators and flash chip controller. The SSD platform has a 256-GB custom flash array card. The size of a page is 8 KB, and the number of pages per block is 256. (See Section 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 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 64 GB, and DRAM for KV indexing structures (e.g., the level lists and pinned meta segments) is set to 64 MB – 0.1% of the SSD capacity.

We evaluate PinK using seven workloads from YCSB, a realistic cloud benchmark [13]. The details of the workloads are described in Table 1. Default key and value sizes are set to 32 B and 1 KB, respectively, which represent averages of common KV workloads [5]. For the YCSB evaluations, we first created a 44-GB KV pool on the 64-GB SSD ("Load" in Table 1)—a total of 44 M

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 [13]	Zipfian	

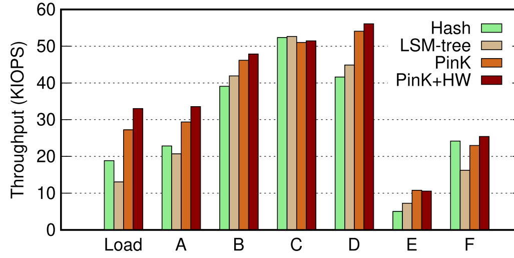


Fig. 10. Overall throughputs of the four KV-SSD setups.

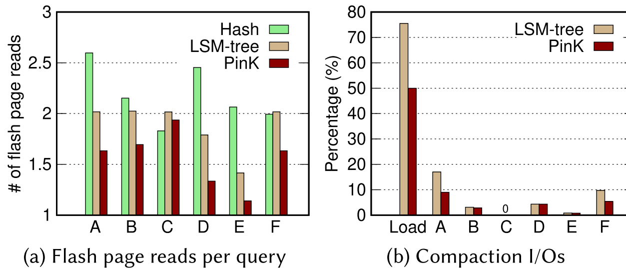
unique KV pairs are written. Then, we ran each workload ("A"~"F" in Table 1) that sends 10 M KV requests to the loaded dataset. We initialized the SSD with the Load phase before any other workload executed. On the host, 64 YCSB clients ran simultaneously to maximize throughput. With 44GB data, the storage utilization was 69%. We assigned 10% of the SSD capacity (i.e., 6.4 GB), for over-provisioning.

To compare with PinK, we have implemented a hash-based KV-SSD based on what we described in Section 2.3. The KV-SSD denoted by `Hash` uses an 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 dataset, the 8-bit signature is large enough to provide a low collision rate. It requires 320 MB of the hash table, which is much larger than the 64 MB of DRAM for indexing. Therefore, `Hash` keeps only popular buckets in DRAM using the LRU replacement policy. `Hash` uses additional 1 MB 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 [11] (`LSM-tree`) is included for our evaluation. `LSM-tree` is equivalent to PinK, except that it does not employ the optimization techniques explained from Section 4.2 to Section 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 8-KB page size. With 8KB meta segments, the amounts of DRAM for the level lists is 10 MB (including both prefix and range pointers). The rest of DRAM, 54 MB, thus can be used to pin levels. `LSM-tree` uses 9 MB for level lists and 55 MB of DRAM for bloom filters. As in `Hash`, for L_0 (a write buffer), 1-MB DRAM is additionally assigned to `PinK`, `PinK+HW`, and `LSM-tree`.

6.2 Performance Analysis

YCSB Throughput: We measured IOPS of the four KV-SSD setups (`Hash`, `LSM-tree`, `PinK`, and `PinK+HW`) using YCSB. Figure 10 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



(a) Flash page reads per query (b) Compaction I/Os

Fig. 11. The impact of the level pinning on flash read I/Os (a) and compaction I/Os (b).

LSM-tree. Using the HW accelerators for sorting further improved the performance. As depicted in Figure 10, 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 11 shows the impact of the level pinning on read and write I/O counts. As shown in Figure 11(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 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 [14] 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 11(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 10, the negative impact of compaction I/O ratio on the throughput was significant.

YCSB Read Latency: Figure 12 shows CDF graphs of read response times of the four KV-SSD setups. Table 2 also lists average, 99th, 99.9th, and 99.99th 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

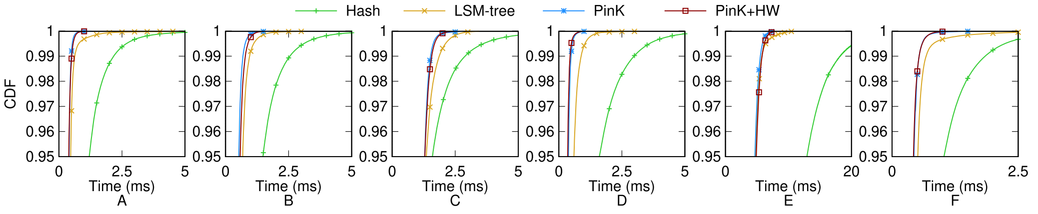


Fig. 12. CDF graphs of read latency of Hash, LSM-tree, PinK, and PinK+HW under YCSB.

Table 2. Comparison of Average and Tail Latency (Unit: μs)

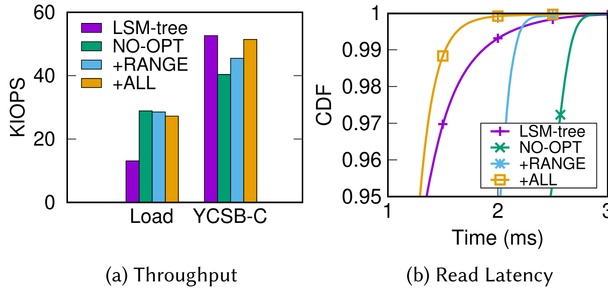
	Percentile	A	B	C	D	E	F
Hash	Average	410	573	592	501	5,628	370
	99th	2,180	2,550	2,900	3,030	17,550	1,850
	99.9th	4,180	4,600	5,710	5,090	25,360	3,260
	99.99th	9,430	9,340	9,830	7,530	34,420	5,180
LSM-tree	Average	302	395	722	294	3,142	329
	99th	640	960	1,870	890	5,790	680
	99.9th	1,700	1,630	2,680	1,370	8,800	1,890
	99.99th	5,250	3,140	3,450	3,210	10,740	3,750
PinK	Average	236	290	732	161	3,027	248
	99th	490	700	1,820	490	5,550	540
	99.9th	670	1,040	2,180	720	6,640	800
	99.99th	1,300	1,800	2,370	1,060	7,590	1,540

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 13(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. While LSM-tree has the same worst case computation time of $O(h^2 \cdot log(T))$ as that of NO-OPT, LSM-tree has better throughput, because its Bloom filter can much improve the average computation overhead by skipping searching of many levels. Figure 13(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 44 M unique KV pairs, and then ran a synthetic workload that issued 100-M SET()s with uniformly random keys to overwrite existing KV pairs.



(a) Throughput (b) Read Latency

Fig. 13. Impact of search path optimizations.

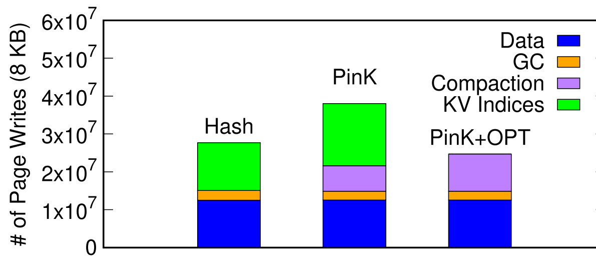


Fig. 14. Analysis of GC cost: "Data" represents pages written/read by `SET()`. "GC" indicates pages written/read to move valid values for GC. "Compaction" represents pages written/read to meta segments during compaction. "KV Indices" indicates pages written/read to update meta segments or in-flash hash indices.

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 14 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 `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 14). Since a bucket size of `Hash` (8-B signatures) is smaller than that of `PinK` (32-B 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+OPT` addresses this problem by rewriting victim KV pairs to L_0, instead of directly updating meta segments (see Section 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.

GC read I/O optimization: As we mentioned in Section 4.5, checking the validity of KV pairs in a victim block degrades GC performance. Even if `PinK+OPT` can reduce write I/Os for "KV Indices" by rewriting KV pairs to L_0, it still needs to read many meta segments to figure out invalidated KV pairs. To address this problem, PinK adopts the validity bitmap. However, in a 64-GB PinK KV-SSD, which has 64 MB DRAM, it requires 16 MB memory to track all the validity bits of physical

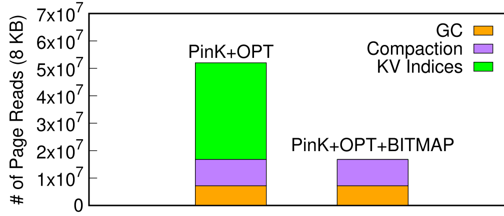


Fig. 15. The number of read I/Os in PinK when the GC is triggered.

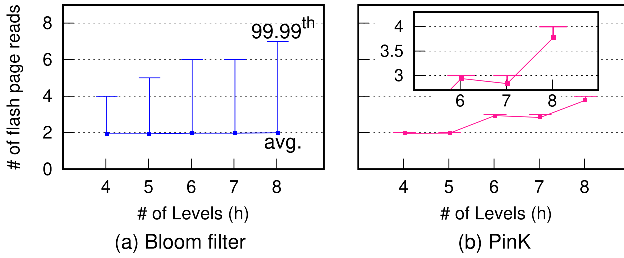


Fig. 16. The number of flash page reads with varying h.

addresses in 512-B granularity, a PinK object unit size. Instead of keeping the entire bitmap, PinK maintains a validity bitmap of select blocks that were prioritized by the GC policy.

To evaluate the effect of using validity bitmaps on GC, we simulate the same condition as the previous garbage collection test. We assigned additional 1 MB memory, which is about 1.5% of the total 64 MB memory to bitmap. Figure 15 shows that `PinK+OPT+BITMAP`, which adopts the bitmap removes all read I/Os when checking the validity of KV pairs. Since the last level has about 95% of total objects, the compaction including the last level occurs whenever 5% of data is inserted to PinK on average. The 1-MB bitmap contains 6.25% of total physical validity information. Thus, `PinK+OPT+BITMAP` avoids reading meta segments for filtering the invalid KV pairs.

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 (Section 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 [32], one might want to increase the height of the tree. Unfortunately, as the tree gets taller, PinK cannot pin all the higher levels to DRAM. Given 64 MB 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 437 MB, 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 bounded, 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 16 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 bounded 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

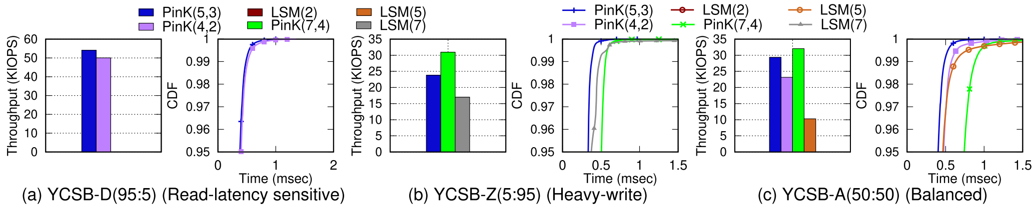


Fig. 17. Experimental results of the tree configurations.

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.

6.3 PinK Design Space Analysis

After evaluating PinK's performance, we carried out experiments to understand its versatility to satisfy KV clients' demands. We selected three scenarios, YCSB-A, YCSB-D, and YCSB-Z, which have different performance requirements. As noted in Table 1, YCSB-A has a R:W ratio of 50:50 and users of YCSB-A workloads should find a balance between the write and read performance. YCSB-D has a R:W ratio of 95:5 and thus PinK configured for better read performance is preferred unless the write throughput starts to deteriorate. However, a manually configured workload YCSB-Z has a R:W ratio of 5:95. The YCSB-Z users prefer higher write throughput as long as a reasonable read latency is provided. Finally, For the workloads described above, we considered three PinK configurations of `PinK(h,k)` (`h` is the number of total levels and `k` is the number of pinned levels), `PinK(5,3)`, `PinK(4,2)`, and `PinK(7,4)`, for YCSB-A, D, and Z, respectively. As shown in Figure 9(a), `PinK(5,3)` has the most balanced performance for reads and writes, even if it consumes large DRAM, i.e., 5/4 MB, to pin three levels. `PinK(4,2)` provides the same level of read latency as `PinK(5,3)`, but write throughput is lower than `PinK(5,3)`, since the tree height is 4. Instead, `PinK(4,2)` requires much less DRAM, i.e., 28 MB, because only two levels are pinned. `PinK(7,4)` requires the same amount, i.e., 54 MB, of DRAM as `PinK(5,3)`. By increasing the tree height, it offers higher write throughput, but sacrifices read latency. Note that the SSD size is different from Figure 9(b), thus the memory usage of level-pinning can be different.

From the conventional LSM-tree, we included three configurations `LSM(h)`, `LSM(5)` for balance performance (YCSB-A), `LSM(2)` for short read latency (YCSB-D), and `LSM(7)` for high write throughput (YCSB-Z). All of them used 54 MB DRAM for Monkey bloom filters.

Figure 17 shows I/O throughput and read latency, respectively, for YCSB-D, Z, and A. For clarity of presentation, we plot the selected tree configurations in the graphs. For YCSB-D (see Figure 17(a)), `PinK(4,2)` showed lower read latency as `PinK(5,3)` with slightly degraded I/O throughput, 7.5%, compared to `PinK(5,3)`. Considering 2× lower DRAM requirement of `PinK(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 would require 5 days to finish the Load phase with at 105 IOPS.

For YCSB-Z (see Figure 17(b)), `PinK(7,4)` exhibited the best throughput, which was 28% higher than that of `PinK(5,3)`. Thus, it is the best for write-heavy workloads. Although `LSM(7)` has tuned for write-heavy workload, it shows 29% lower write throughput than `Pink(5,3)` due to bloom filter and compaction I/Os. It also has inconsistent read latency with long tails.

For YCSB-A (see Figure 17(c)), `PinK(5,3)` performed well, achieving high throughput and short read latency. `PinK(7,4)` exhibited rather higher throughput, but its read latency was much worse than `PinK(5,3)`. `LSM(5)` could not outperform `PinK(5,3)` in terms of read latency and throughput.

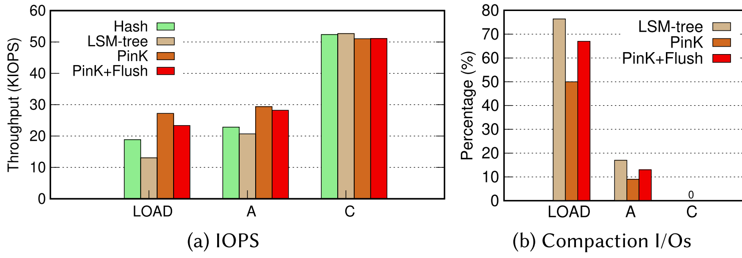


Fig. 18. Experimental results of when PinK flushes all dirty meta segments. (a) The IOPS results of all 4 KV-SSDs. (b) The ratio of compaction I/O to total flash I/Os on LSM-tree-based KV-SSDs.

6.4 The Overhead of Flushing Dirty Meta Segments

We have assumed all of PinK's DRAM-resident data structures (e.g., pinned levels and level lists) can be protected by built-in capacitors. However, some SSDs do not have enough capacitors to protect all of DRAM. We analyze the performance degradation of PinK under this environment as detailed in Section 4.6 and 4.2. The new configuration of PinK (`PinK+Flush`) that flushes all dirty meta segments whenever compaction occurred is compared with `PinK`, `LSM-tree` and `Hash`, which do not flush any metadata. We use three benchmarks, YCSB-LOAD, YCSB-A, and YCSB-C, for the evaluation.

Figure 18(a) presents the throughput (IOPS) measured on four configurations of KV-SSDs. Owing to the write-optimized data structure of LSM-tree, `PinK+Flush` shows 23.9% and 23.4% higher performance than `Hash` in YCSB-LOAD and YCSB-A, respectively. Since `PinK+Flush` has more write overhead in compaction, it processes 15% and 4% slower than `PinK` in YCSB-LOAD and A. However, `PinK+Flush` has almost the same read performance as `PinK` by having the same number of pinned levels. `LSM-tree` has lower performance than `PinK-flush` in write-intensive benchmarks because of the bloom filter and compaction overhead.

To understand the additional compaction overhead of `PinK+Flush`, we compare the ratio of compaction I/O to the total I/O in the three LSM-tree-based configurations (`LSM-tree`, `PinK`, and `PinK+Flush`) as shown in Figure 18(b). Although `PinK+Flush` flushes dirty meta segments, it has a lower compaction overhead than `LSM-tree`, since `PinK+Flush` does not need to read meta segments. However, `PinK+Flush` issues 17% and 4% more compaction I/Os than `PinK` in YCSB-LOAD and YCSB-A due to flushing dirty meta segments.

7 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 particular, by allowing users to flexibly configure the target I/O performance of KV-SSD according to their demands, PinK reduced overall execution times of KV clients. 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 durability for pinned levels in the host system. Using emerging technologies such as persistent memory may offer a solution.

ACKNOWLEDGMENTS

PinK's source code can be found at Reference [1] and Reference [2].

REFERENCES

[1] 2020. PinK HW Source Code. Retrieved from https://github.com/chanwooc/lightstore-platform/tree/pink-hw.

[2] 2020. PinK SW Source Code. Retrieved from https://github.com/dgist-datalab/PinK.

[3] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark S. Manasse, and Rina Panigrahy. 2008. Design tradeoffs for SSD performance. In *Proceedings of the USENIX Annual Technical Conference*.

[4] S. Ashkiani, S. Li, M. Farach-Colton, N. Amenta, and J. D. Owens. 2018. GPU LSM: A dynamic dictionary data structure for the GPU. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium*. 430–440.

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. 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*. 53–64.

[6] Jens Axboe. 2005. FIO: Flexible I/O tester synthetic benchmark. Retrieved June 13, 2015 from https://github.com/axboe/fio.

[7] Duck-Ho Bae, Insoon Jo, Youra Adel Choi, Joo-Young Hwang, Sangyeun Cho, Dong-Gi Lee, and Jaeheon Jeong. 2018. 2B-SSD: The case for dual, byte- and block-addressable solid-state drives. In *Proceedings of the Annual International Symposium on Computer Architecture*. 425–438.

[8] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. 2012. Don't thrash: How to cache your hash on flash. *Proc. VLDB Endow.* 5, 11 (2012).

[9] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. Faster: A concurrent key-value store with in-place updates. In *Proceedings of the ACM International Conference on Management of Data*. ACM, 275–290.

[10] Bernard Chazelle and Leonidas J. Guibas. 1986. Fractional cascading: I. A data structuring technique. *Algorithmica* 1, 1, 1 (1986), 133–162. DOI: https://doi.org/10.1007/BF01840440

[11] Chanwoo Chung, Jinhyung Koo, Junsu Im, Arvind, and Sungjin Lee. 2019. LightStore: Software-defined network-attached key-value drives. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 939–953.

[12] John Colgrove, John D. Davis, John Hayes, Ethan L. Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. 2015. Purity: Building fast, highly-available enterprise flash storage from commodity components. In *Proceedings of the ACM International Conference on Management of Data*. 1683–1694.

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing*. 143–154.

[14] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. In *Proceedings of the ACM International Conference on Management of Data*. 79–94.

[15] Niv Dayan and Stratos Idreos. 2018. 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*. 505–520.

[16] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2010. FlashStore: High throughput persistent key-value store. *Proc. VLDB Endow.* 3, 1–2 (2010), pp. 1414–1425.

[17] Facebook, Inc.[n.d.]. RocksDB: A Persistent Key-value Store for Fast Storage Environments. Retrieved from https://rocksdb.org.

[18] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. 2009. 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*. 229–240.

[19] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. 2008. Hopscotch hashing. In *Proceedings of the International Symposium on Distributed Computing*. Springer, 350–364.

[20] IC Knowledge LLC.2020. Lithovision-2020: Economics in the 3D Era. Retrieved from https://semiwiki.com/wp-content/uploads/2020/03/Lithovision-2020.pdf.

[21] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. 2020. PinK: High-speed in-storage key-value store with bounded tails. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 173–187.

[22] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. 2017. KAML: A flexible, high-performance key-value SSD. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*. 373–384.

[23] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. 2015. BlueDBM: An appliance for big data analytics. In *Proceedings of the Annual International Symposium on Computer Architecture*. 1–13.

[24] Garth Gibson Kai Ren. 2013. TABLEFS: Enhancing metadata efficiency in the local file system. In *Proceedings of the USENIX Annual Technical Conference*.

[25] Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang-suk Kee, Francisco Londono, Sangyoon Oh, Jongyeol Lee, and Daniel D. G. Lee. 2019. Towards building a high-performance, scale-in key-value storage system. In *Proceedings of the ACM International Conference on Systems and Storage*. 144–154.

[26] Sang-Hoon Kim, Jinhong Kim, Kisik Jeong, and Jin-Soo Kim. 2019. Transaction support using compound commands in key-value SSDs. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems*.

[27] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. 2019. Reaping the performance of fast NVM storage with uDepot. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 1–15.

[28] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operat. Syst. Rev.* 44, 2 (2010), pp. 35–40.

[29] Chang-Gyu Lee, Hyeongu Kang, Donggyu Park, Sungyong Park, Youngjae Kim, Jungki Noh, Woosuk Chung, and Kyoung Park. 2019. 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*. 384–395.

[30] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: The design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 447–461.

[31] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating keys from values in SSD-conscious storage. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 133–148.

[32] Chen Luo and Michael J. Carey. 2020. LSM-based storage techniques: A survey. *VLDB J.* 29, 1 (2020), 393–418. DOI: 10.1007/s00778-019-00555-y

[33] Leonardo Mármol, Swaminathan Sundararaman, Nisha Talagala, Raju Rangaswami, Sushma Devendrappa, Bharath Ramsundar, and Sriram Ganesan. 2014. NVMKV: A scalable and lightweight flash aware key-value store. In *Proceedings of the USENIX Conference on Hot Topics in Storage and File Systems*. 8–8.

[34] NGD Systems, Inc. 2018. NGD Catalina NVMe SSD. Retrieved from https://www.ngdsystems.com/products/.

[35] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Inf.* 33, 4 (1996), pp. 351–385.

[36] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *J. Algor.* 51, 2 (2004), 122–144.

[37] Mendel Rosenblum and John K Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.* 10, 1 (1992), pp. 26–52.

[38] Samsung Electornics. 2018. Samsung Smart SSD. Retrieved from https://samsungatfirst.com/smartssd-ocp/.

[39] Samsung Electronics. [n.d.]. KV SSD Host Software Package. Retrieved from https://github.com/OpenMPDK/KVSSD.

[40] Samsung Electronics. 2016. Samsung Introduces World's Largest Capacity (15.36TB) SSD for Enterprise Storage Systems. Retrieved from https://news.samsung.com/global/samsung-now-introducing-worlds-largest-capacity-15-36tb-ssd-for-enterprise-storage-systems.

[41] Samsung Electronics. 2017. Samsung Key Value SSD enables High Performance Scaling. Retrieved from https://www.samsung.com/semiconductor/global.semi.static/Samsung_Key_Value_SSD_enables_High_Performance_Scaling-0.pdf.

[42] Samsung Electronics. 2018. 860EVO SSD Specification. Retrieved from https://www.samsung.com/semiconductor/global.semi.static/Samsung_SSD_860_EVO_Data_Sheet_Rev1.pdf.

[43] Samsung Electronics. 2018. KV SSD Firmware Introduction. Retrieved from https://github.com/OpenMPDK/KVSSD/wiki/presentation/kvssd_seminar_2018/kvssd_seminar_2018_fw_introduction.pdf.

[44] Samsung Electronics. 2019. 960PRO SSD Specification. Retrieved from https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/ssd960/.

[45] Justin Sheehy and David Smith. 2010. Bitcask: A log-structured hash table for fast key/value data. *Basho White Paper* (2010).

[46] SNIA. [n.d.]. Key Value Storage API Specification Version 1.0. Retrieved from https://www.snia.org/tech_activities/standards/curr_standards/kvsapi.

[47] Twitter Inc. [n.d.]. Fatcache: Memcache on SSD. Retrieved from https://github.com/twitter/fatcache.

[48] J. Wang, Y. Zhang, Y. Gao, and C. Xing. 2013. pLSM: A highly efficient LSM-tree index supporting real-time big data analysis. In *Proceedings of IEEE Annual Computer Software and Applications Conference*. 240–245.

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

[50] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, et al. 2016. Bluecache: A scalable distributed flash-based key-value store. *Proc. VLDB Endow.* 10, 4 (2016), 301–312.

Received December 2020; accepted February 2021