A Hash-Based Key-Value SSD FTL With Efficient Small-Value Support

Carl Duffy
Dept. Computer Science and Engineering
Seoul National University
cduffy@snu.ac.kr

Sang-Hoon Kim
College of Information Technology
Ajou University
sanghoonkim@ajou.ac.kr

Jin-Soo Kim
Dept. Computer Science and Engineering
Seoul National University
jinsoo.kim@snu.ac.kr

***Abstract*—Key-value SSDs have shown promise in various domains, as their ability to index key-value data inside the disk itself can remove either all or most of the need to maintain and transfer indexing data to and from the host system.**

However, some KVSSDs suffer from an issue that will likely be a sticking point in the road to adoption; they can't efficiently store small values. This work first adapts a previously-existing key-value SSD FTL onto a realistic SSD performance model using NVMeVirt, a software defined SSD emulator. Then, we introduce an improved FTL wherein the key-value SSD can handle much smaller key-value pairs with increased performance and without excessive space amplification.

Our improved FTL, while being able to store significantly smaller key-value pairs, outperforms the existing scheme across a range of tests and metrics. For example, in write-heavy tests our FTL outperforms the original scheme by up to 2×, with a 90% reduction in write amplification.

***Index Terms*—key-value storage, key-value ssd, flash translation layer**

I. Introduction

Key-value SSDs (herein referred to as KVSSDs) index key-value data inside the device itself, receiving key-value commands directly via a key-value interface. This is in contrast to the commonplace architecture where an index is stored in the host system's memory, and it is the user's responsibility to maintain and persist the index on a block interface-based SSD. For example, users can store and retrieve key-value data by sending *store* and *retrieve* commands directly to the KVSSD itself [1].

KVSSDs introduce an attractive prospect; because data for the index no longer needs to be transferred between the disk and the host system, significant amounts of data transfer across the data bus can be avoided, which can lead to increased performance. Moreover, since the device itself accepts key-value commands, it is possible to delegate the expensive task of stale data removal to the device itself with the *delete* command. Indeed, previous work has shown that KVSSDs can outperform block-SSD KV stores [2], [3] and file systems [4].

However, a class of KVSSD that indexes data in-disk using a hash table suffers from the problem of space amplification when KV pairs are small. Inside the KVSSD, flash space is organized into *grains* of data, with a single grain being the smallest write unit. Each grain is given an index into a hash table that is mapped to a physical address on disk. In existing KVSSDs and hash-table KVSSD FTLs, this grain size is between 512 bytes and 1KB. This means that KV pairs of for example 100 bytes will induce roughly 5× to 10× space amplification per pair. The core problem with this scheme is that a *grain bitmap*, which is assumed to be completely memory-resident, is used to determine which KV pairs are invalid during garbage collection (GC). Decreasing the grain size increases the size of the grain bitmap to unacceptable levels.

This work introduces an improved hash-table KVSSD FTL, which we refer to as *Plus*, that puts two insights into practice to enable small KV pair storage without high space amplification. The first insight involves how modern SSDs perform GC. GC is typically performed at the superblock level (large groups of blocks spanning several hundreds of MB), invoking thousands of KV pair reads and writes per iteration. However, hash index to physical address mappings are very small, thus persisting these invalidated mappings to flash during normal operation and reading them before GC introduces a negligible amount of extra read and write overhead. This frees us from relying on a memory-resident grain bitmap to discover invalid KV pair mappings, allowing for much smaller grain sizes. The second insight is that in a grain-based scheme, KV pairs will typically consume many grains worth of data, even when the value size isn't particularly large. This means that significant portions of the logical to physical mappings will go unused, and the KVSSD's internal mapping cache can forego their storage to save space and reduce write amplification.

In this work we implement two FTLs on a software-defined emulator called NVMeVirt [5]; *Original*, a standard hash-based KVSSD FTL, and *Plus*, our improved version. The FTLs run on top of a realistic flash performance model, originally implemented and tested against a real NVMe block SSD. The auxiliary to this work is the first realistic, extensible KVSSD emulator that is available to the general public. The source code for both FTLs and instructions to repeat the experiments in this paper are available at *https://www.github.com/snu-csl/kvvirt*.

Plus, while supporting a 64B grain size, outperforms *Original* with a 512B grain size across all of our tests. For example, on the YCSB A (write-heavy) test, *Plus* outperforms *Original* by 2×, while reducing GC write amplification by 90%.

979-8-3503-8855-8/24/$31.00 ©2024 IEEE
NVMSA.2024

II. NVMeVirt

NVMeVirt [5] is a software-defined NVMe SSD emulator, meaning users write their own code for the SSD's FTL and provide the underlying flash timings. Different from other emulators, NVMeVirt works by registering itself as a pseudo PCIe device on the PCI bus, and thus presents itself as a real NVMe device to the user (e.g. an *nvme list* command will list the NVMeVirt device). User I/O may travel through the entire storage stack before it reaches NVMeVirt, which is loaded as a kernel module.

NVMeVirt works by executing FTL logic in *real-time*, while producing flash read and write timings from said logic. The timings generated from flash IO are used to determine when a request completes. For example, a write from a user will execute FTL mapping update logic in real-time, then generate a flash page write timing. This write's completion time will depend on the user-configured flash speed.

Our reasons for choosing NVMeVirt in this work are three-fold. First, we choose NVMeVirt over a real hardware approach because acquiring real hardware requires high initial effort and/or high cost. Second, we reject using a simulator as simulators typically run much slower than real devices. Finally, NVMeVirt's architecture gives it a property that is very useful for low latency (modern NVMe) device development and testing; it enables a user to bypass the kernel stack entirely using user-space I/O frameworks [6], which is an increasingly popular topic in key-value storage [7]

III. Demand Based KVSSD FTL

This section first introduces the FTL that our improved scheme is based on, *Original*, then details the improvements we make to help it handle small key-value pairs without space amplification in *Plus*.

To the best of our knowledge, the original demand-based KVSSD FTL scheme is the only open-source hash-based KVSSD FTL available [8] and we use it and its grain based scheme as the baseline in this work. Samsung's real KVSSD [9] also has a minimum value size (1KB), which implies that it may also implement a grain based scheme.

A. Original Implementation

Original is based on the well-studied and understood DFTL scheme for block SSDs [10], but adds functionality to support variable sized KV pairs. We first walk through a *store* command.

Overview (1) A user calls the *store* command with a key and value, targeting a KVSSD. Then, the KV pair is assigned a physical *grain* on the disk; each physical page is divided into a fixed number of grains, which represent the smallest unit of write that the FTL will represent. For example, with a grain size of 512 bytes, a key-value pair of 1024B in size will occupy two grains. Likewise, a KV pair 100B in size will occupy 1 grain, wasting 412B of space.

(2) The hash table contains hash index to grain mappings, and is stored on flash, with some sections cached in the KVSSD's DRAM. The key is hashed, and the resulting value is the hash index used to discover which section of the hash table this key belongs to. For example, hash index *H50* represents section 0 of the hash table, which covers hash indexes 0 to 1023[1].

If this section of the hash table is absent from the KVSSD's DRAM, it is read from flash. This read from flash may cause an eviction of a previously modified DRAM-resident hash table section, triggering an extra flash write. Once the hash table section is inside DRAM, the hash index for this key is checked to find the old location of the KV pair on disk (i.e. the old grain). The size of each hash table section is fixed, and contains a fixed set of ordered hash index to grain mappings. Therefore, finding a hash index to grain mapping is a simple check of an offset within the page.

(4) Once the old grain is found, the physical page to which this grain belongs is read into the KVSSD's DRAM. The key at the old grain is checked against the key for this store operation. It is possible that keys do not match; two different keys have hashed to the same hash index (a collision). In this case, the store restarts from (2) with a new hash.

(5) If the keys match, the old grain is marked invalid (set to 0) in a completely DRAM-resident *grain bitmap*. Likewise, the new grain on which the KV pair resides is marked valid (set to 1) inside the same bitmap. The grain bitmap is the central structure used to check the validity of KV pairs in *Original*. The *store* is now finished.

Notes *retrieve* works much in the same way as *store*, except there is no new grain assigned. When the correct old grain is found, data at the grain is copied back to the user. The amount of grains a KV pair consumes (the length) is determined by scanning the grain bitmap from the first grain of this pair to the first valid grain of the next pair.

The hash index of each grain in a page is stored inside the page's out-of-bounds (OOB) area, and is used during GC. GC works by first checking the grain bitmap at each victim page. If all of the grains of a page are invalid, reading the page is skipped. If one or more grains are valid, the page is read into DRAM and the hash indexes are retrieved from the OOB area. Each valid grain is copied to a new page, and the hash index for that grain is used to update the corresponding hash table section.

The Core Problem *Original* relies solely on the grain bitmap to determine the validity of KV pairs, and the size of the grain bitmap is directly tied to the size of the grain. Consider a 4TB KVSSD with 4GB DRAM (a typical amount) and a 512B grain size. This results in a grain bitmap of 1GB, which fits comfortably in DRAM. However, if we reduce the grain size to 64B, the grain bitmap is now 8GB, twice the available DRAM. We use 64B as a grain size representative of a small value in this work based on value size findings in previous KV workload analyses [11].

[1]A hash index to grain mapping is 8B. Thus 4KB / 8B for 1024 indexes per page.

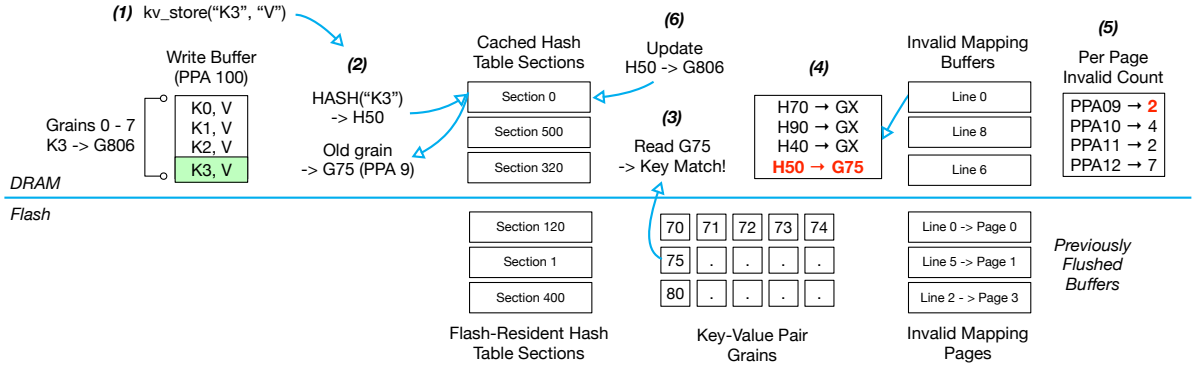

Fig. 1. An overwrite *store* command in *Plus*. *H* refers to a hash table index. *G* refers to a grain.

IV. Plus

The goal of *Plus* is to enable efficient small-value handling (i.e. small grains). Achieving this goal consists of two parts; the removal of the grain bitmap and the introduction of *invalid mapping pages*, and a change in the caching scheme to reduce the size of the mapping cache when possible.

A. Plus Part One - Enabling Smaller Grains

Plus introduces the idea of *invalid mapping pages*, which allow us to check grain validity without the grain bitmap. We step through a *store* command in Figure 1 and explain *Plus* along the way. We assume 8 grains per page, and that each hash table section (a page) holds a maximum of 480 entries. Each hash table section in *Plus* holds less than those of *Original*, explained later in Section IV-B.

(1) A user calls *store* with key *K3* and value *V*. The KV pair is given a spot in PPA 100. Three two-grain pairs were already assigned space in this page, and thus *K3* gets grain *806* (PPA 100 * 8 grains per page + 6).

(2) *K3* is hashed to get hash index *H50*. *H50* belongs to hash table section 0, which is already cached in DRAM. We check section 0 for hash index 50, and find that it has already been written to grain *G75*.

(3) *G75*, which is PPA 9, is read from disk. We compare the key at *G75* to *K3* and find a match, meaning *G75* is the current location of *K3*.

(4) *H50* $\rightarrow$ *G75* is recorded in an *invalid mapping buffer* for superblock 0, of which *G75* belongs.

An *invalid mapping buffer* is a per-superblock, page-sized, in-memory buffer of invalid hash index to grain mappings for that superblock. *Invalid mapping buffers* eventually become *invalid mapping pages* on flash when they are full. Multiple *invalid mapping pages* typically exist per-superblock, and a simple per-superblock list of the physical locations of these pages is maintained in DRAM. Invalid mapping pages are scattered around the flash space; they don't necessarily reside on the same superblock as the KV data.

Invalid mapping pages are how *Plus* checks the validity of grains during GC. At the beginning of GC, the in-memory list of invalid mapping pages for the victim superblock is scanned and each invalid mapping page is read into DRAM. A transient hash table of (invalid) hash index to grain mappings is built based on the contents of the pages. When GC reads a page containing KV data, the hash index to grain mappings inside that page are first discovered from the OOB area, as in *Original*. Next, for each hash index to grain mapping in the page, we check the previously constructed invalid mapping hash table. If there is a match in the table, this KV pair was previously invalidated, and the copy is skipped. If there is no match, this KV pair was never invalidated (i.e. it is valid), and we need to copy it. At the end of GC, the DRAM list of invalid mapping pages for the victim superblock is cleared, and the invalid mapping hash table deallocated.

(5) The old version of *K3* was 2 grains in size (calculated via the grains in the OOB area) and belonged to PPA 9. We add *2* to a *per-page invalid count* for PPA 9 in DRAM. This *per-page invalid count* is used during GC to check whether or not we can skip reading a page; if the count of invalid grains inside a page is the same as the maximum grains per page, the read can be skipped. This structure is memory-resident, and never goes to flash during normal operation.

(6) The *store* finishes by updating the cached hash table section with the new *H50 G806* mapping.

What we Get The introduction of *invalid mapping pages* means that we don't need to rely on the grain bitmap for validity checking. Setting a smaller grain size now affects how much invalid mapping data we write and read. Hence, we trade DRAM space for extra flash reads (at the beginning of GC) and writes (to flush invalid mapping pages). The extra amount of flash reads and writes is small. If we assume a 4TB SSD with 8 channels, 8 LUNs per channel, 16KB pages, and 256 pages per block [12], we arrive at 16K 256MB superblocks on disk. Consider that in a 50% full 256MB superblock, we need to perform roughly 32K reads and 32K writes to copy valid data. If we assume a 64B grain size in *Plus*, that's 4M grains per superblock, or 2M invalid grains on a 50% full superblock. Each invalid hash index to grain mapping is 8 bytes, and thus we had to write 4K extra pages to record invalid mappings, and read 4K extra pages to get invalid mapping data at the beginning of GC; an overhead of roughly 12%. GC is done in

the background, and invalid mapping page flushes are also sent to the background, which means foreground requests don't need to wait for either to complete.

Memory Accounting In *Plus*, the memory overhead consists of a per-superblock *invalid mapping buffer*, a per-page *invalid count*, and a per-superblock list of flushed invalid mapping pages. If we consider a 4TB KVSSD with 256MB superblocks as before, we have 16K superblocks total. With 4K pages, that results in 64MB of invalid mapping buffers that need to reside in DRAM. The per-page invalid counter is 1 byte per-page, which adds an additional 1GB of DRAM overhead. In a 256MB superblock, we have a maximum of 8K invalid mapping pages. If we take the worst case scenario of half of the superblocks having the maximum number of invalid mapping pages on flash, we need to store 4K * 8K * 4B worth of data to index the invalid mapping pages[2], which is around 256MB.

Minus cached hash table sections, the total memory usage of *Plus* with a 64B grain size is roughly 1320MB. In *Original*, a 64B grain size results in an 8GB grain bitmap.

B. Plus Part Two - Reduced Mapping Table Size

In both *Original* and *Plus*, the maximum hash table size is equal to the size of the disk divided by the grain size. The hash table size is fixed in *Original*; every hash table page on flash stores every hash index to grain mapping, including those of yet-to-be assigned hash indexes (empty mappings).

The ability to use a smaller grain size in *Plus* thanks to the invalid mapping page scheme thus comes at a cost; the size of the hash table increases by a factor of 8 (512B grain vs 64B grain) even if most of the grains are unused. Consider a scenario where we fill the KVSSD with 1024B KV pairs; 2 grains in *Original* and 16 grains in *Plus*. *Plus's* hash table will be significantly larger, even though we are indexing the same amount of pairs.

The second part of *Plus* is a straightforward but important change to the caching scheme. In *Plus*, only live hash index to grain mappings are stored in DRAM and on flash. Logically, sections of the hash table in *Plus* represent the same sections as in *Original*; only the physical layout changes. During eviction and hash table GC, *Plus* groups live hash indexes from different hash table sections until it gets a flash page worth of indexes, then writes out the page.

Plus will store hash index to grain mappings as they arrive, and thus can't directly find an entry using a simple offset calculation as in *Original* because mappings are out of order. To speed up the hash index to grain mapping search, hash table sections in *Plus* are organized as a two level table. The first level is a sorted range of hash indexes (separator keys) that point to parts of the second level of the table. In each part of the second level, an unsorted list of hash indexes to grain mappings less than or equal to the separator key is contained. The first level does not contain pointers to the lower levels, saving space; parts of the lower level are at fixed locations in the hash table section (page) and don't change. Additionally, only the first level is sorted to avoid the CPU overhead of sorting each part of the second level on every insert.

The first level of the table reserves four grains when using a 64B grain, which introduces a 6% space overhead on the hash table only. Compared to a linear search, the two level table reduces *Plus's* average number of comparisons needed to find a hash index to grain mapping in a full hash table section from 256 to 10.

Hash table sections in plus expand by a grain at a time as needed. When a hash table section is the size of a page (the maximum size), it is possible to sort hash index to grain mappings during the final expansion, and replace the two level table scheme with the offset calculation scheme as in *Original*.

TABLE I
YCSB WORKLOADS.

Workload	Composition
A	50% read 50% update
B	95% read 5% update
C	100% read
D	95% read 5% update (latest distribution)
F	50% read-modify-write 50% read

V. EVALUATION

A. Machine and NVMeVirt Setup

All of the evaluations are carried out on a machine with 256GB DRAM and two 20-core Intel Xeon Gold 5218R CPUs. NVMeVirt is loaded with one request dispatcher thread, one IO worker thread (which copies data to and from memory, representing flash), one background GC thread, and one background eviction thread. This four-core setup is intended to mimic the amount of cores in a real KVSSD tested in previous work [13]. For accurate performance, we configure the CPUs that NVMeVirt uses on our test machine to run at a lower clock frequency than the default, described later in section V-C.

We configure NVMeVirt to represent a small, fast NVMe disk; a 128GB disk with 8 channels, 8 LUNs per channel, 4KB pages, and 32 pages per block. This results in a scaled-down version of our 4TB running example where we have 16K superblocks total on disk. Additionally, we use the same flash, PCIe, and firmware speeds as in the original conventional FTL implementation in NVMeVirt. These parameters were arrived at by testing against a modern NVMe SSD.

B. Workloads

We test *Original* with a 512B grain size against *Plus* with a 64B grain size. For these tests, we use the YCSB [14] workload suite. The YCSB workloads are shown in Table I. We skip workload E, as hash-table FTLs are not well suited for range queries. Before the YCSB workload tests, the disk is preconditioned. We populate the disk with 100M KV pairs (roughly 80% of the disk when mapping data is taken into account), and then perform 100M random overwrites. This ensures constant GC throughout the runs with writes, and

[2]A physical page address is 4B.

represents a realistic scenario wherein a disk has already been in use. The YCSB workloads are run for 10M operations each, with 8B keys and 1000B values.

C. CPU Adjustments

The CPUs on our test machine are significantly more powerful than those in typical SSDs, which can lead to to unrealistically fast benchmark results. To obtain accurate performance numbers, we run the *Original* source code with a POSIX memory backend (unrelated to NVMeVirt) on two different configurations:

Baseline: An ARM device with weaker cores similar to those in modern computational SSDs [15]. We set up an 8GB in-memory disk with 4M KV pairs and run YCSB A, B, and C for 10 million operations, 50 times each. Average runtimes are collected as the baseline. The tests are single-threaded and pinned to a single core.

Test Machine: We repeat the tests using the same code and in-memory backend on our test machine, again pinned to a single core. The CPU frequency of the core is adjusted until the YCSB test runtime matches the ARM device's runtime within a second. NVMeVirt's cores are set to this CPU frequency for the evaluation.

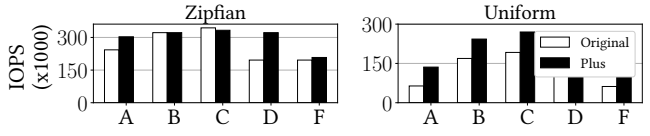

Fig. 2. YCSB workload performance.

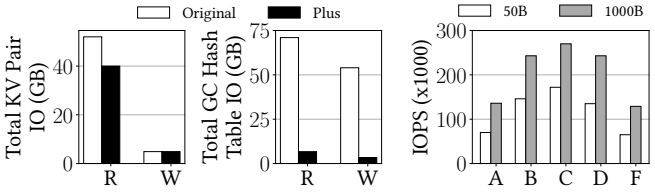

Fig. 3. Total amount of KV data read and written (left) due to user requests, and total amount of hash table data read and written during GC (middle). Both for YCSB A Uniform. Finally, performance of *Plus* with a smaller value size and significantly more total KV pairs (right).

D. YCSB

The YCSB results are shown in Figure 2.

***Plus* generally outperforms *Original*.** In write-heavy tests, *Plus* outperforms *Original* by up to 1.25× with high locality (zipfian), and up to 2.1× without locality (uniform). In the read-heavy tests, *Plus* outperforms by up to 1.4× without locality. *Plus* performs better because it reads in and writes out less data at every stage of execution. When evicting from the cache, *Plus* packs live mapping table entries from several different mapping pages into a page, because it only writes out live hash table indexes. In comparison, *Original* writes out the entire hash table section, even if most mapping aren't used. During GC, *Plus* again is able to pack more mapping table entries into a single page than *Original*, resulting in less write amplification. *Plus* also reads much less data during GC because hash table sections are smaller. Finally, due to *Plus* having a smaller grain size, it benefits from a larger logical area over which to generate hash indexes. This reduces hash collisions when compared to *Original*, meaning *Plus* needs to read less KV pair data overall.

The total KV pair IO due to user requests and total GC mapping IO for YCSB A uniform are shown in the left and middle of Figure 3. *Plus* reads roughly 25% less KV data (less collisions), and writes and reads out 90% less hash table data.

***Plus* performs well as value size decreases.** We test *Plus* against itself to see how performance holds at the same device occupancy with small values, and thus more total KV pairs. We repeat the YCSB tests with 50B values and 32GB disk, at the same device occupancy as the previous YCSB tests. This results in the disk hosting roughly 330M KV pairs. The results are shown on the right hand side of Figure 3. Performance falls both because GC takes longer with more grains to check, and the mapping cache is less useful when there are more KV pairs on the disk. However, while *Plus* is managing 16X more KV pairs in the 50B value size case, performance drops by only 50% in the worst case.

Invalid mapping page reads and writes are a negligible fraction of overall IO. Invalid mapping pages are used in *Plus* to determine which pairs we need to copy during GC. In the YCSB A (write-heavy) uniform run, roughly 40MB of invalid mapping page data was both written and read. In the same run, roughly 100GB total of user and mapping data was written to the disk (due to user requests, mapping data, and GC).

***Plus* Reduces TCO for a System.** Given that *Original* uses a 512 byte grain size, space amplification compared to *Plus* is a simple calculation. At a 40 byte value size, *Original* will only be able to fill the 128GB disk with roughly 260M KV pairs (accounting for space used by the map). In comparison, *Plus* will be able to hold roughly 2.1B, or 8× more.

The combination of invalid mapping page scheme and improved cache in *Plus* results in a lower total cost of ownership (TCO) for a system; it can store more KV pairs (reducing the amount of disks needed), it performs better (decreasing the amount of disks needed to hit throughput targets), and it writes less data (increasing device lifetime), all on the same hardware.

***Plus* marginally underperforms in YCSB C Zipf.** In the YCSB C read-only workload with high locality, *Plus* is 3% slower than *Original*. In both schemes, the cache hit ratio is similar; the working set is small enough that both *Plus* and *Original* can cache all of its mappings. *Plus* pays a slight overhead for finding hash index to grain mappings in its two-level table scheme that *Original* does not, and this is the source of the overhead. Considering the large range of benefits *Plus's* caching scheme brings outside of this specific test, we deem the trade-off worthwhile.

TABLE II
CURRENT KVSSD SIMULATOR AND EMULATOR OFFERINGS. *Sim* REFERS TO SIMULATOR, *Emu* REFERS TO EMULATOR.

Name	Type	SW	Accurate	Extensible
PinK [13]	Sim	OSS [8]	No	Yes
KVEMU [16]	Emu	Private	No	No
Samsung KVSSD	Emu	OSS [17]	Partial	No
NVMeVirt [5]	Emu	OSS [18]	Partial	No
This Work	Emu	OSS [19]	Yes	Yes

VI. RELATED WORK

KVSSD FTL Improvements PinK [13] is an LSM-tree based KVSSD FTL that improves on the basic LSM-tree design. Different to the hash-table based KVSSD FTLs in this work, LSM-tree based KVSSD FTLs provide native range queries, which are commonly used in KV store deployments. RHIK [20] is a dynamically resizable hash-table based KVSSD indexing scheme. Mapping table entries in *Plus* are resizable by nature; they get bigger as they hold more entries. RHIK is thus unnecessary when following the *Plus* design.

KVSSD Simulators and Emulators In table II we survey the availability of KVSSD simulators and emulators. We note that there exist several real KVSSDs, but these are either expensive or impossible to acquire [13], [21], [22]. Despite Samsung's KVSSD Emulator and NVMeVirt's existing KVSSD FTL being open source and modifiable, we do not classify them as extensible because they both work by using timings from in-house testing of Samsung's KVSSD, and don't present actual FTL logic. For the same reason, we class both emulators as partially accurate with regards to performance; they can emulate one type of KVSSD only.

From the survey we conclude that there doesn't yet exist an accurate, extensible KVSSD emulator. This work is thus the first generally useful KVSSD emulator available.

VII. CONCLUSION

Hash-table based KVSSD FTLs are unable to efficiently handle smaller KV pairs in their current form. This work introduces a new hash-table KVSSD FTL that enables small value storage with higher performance and longer device lifetime.

ACKNOWLEDGMENTS

This work was supported by an Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (No. RS-2021-II211363), an Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (No. RS-2024-00349594), an Institute of Information & communications Technology Planning & Evaluation (IITP) under the Artificial Intelligence Convergence Innovation Human Resources Development (IITP-2024-RS-2023-00255968) grant funded by the Korea government(MSIT), and Samsung Electronics.

REFERENCES

[1] Snia key value storage api specification. [Online]. Available: https://www.snia.org/keyvalue

[2] C. Duffy, J. Shim, S.-H. Kim, and J.-S. Kim, "Dotori: A key-value ssd based kv store," *Proceedings of the VLDB Endowment*, vol. 16, no. 6, pp. 1560–1572, 2023.

[3] M. Qin, Q. Zheng, J. Lee, B. Settlemyer, F. Wen, N. Reddy, and P. Gratz, "Kvrangedb: Range queries for a hash-based key–value device," *ACM Transactions on Storage*, vol. 19, no. 3, pp. 1–21, 2023.

[4] J. Koo, J. Im, J. Song, J. Park, E. Lee, B. S. Kim, and S. Lee, "Modernizing file system through in-storage indexing," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021, pp. 75–92.

[5] S.-H. Kim, J. Shim, E. Lee, S. Jeong, I. Kang, and J.-S. Kim, "Nvmevirt: A versatile software-defined virtual nvme device," in *21st USENIX Conference on File and Storage Technologies (FAST 23)*, 2023, pp. 379–394.

[6] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul, "Spdk: A development kit to build high performance storage applications," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2017, pp. 154–161.

[7] G. Haas and V. Leis, "What modern nvme storage can do, and how to exploit it: High-performance i/o for high-performance storage engines," *Proceedings of the VLDB Endowment*, vol. 16, no. 9, pp. 2090–2102, 2023.

[8] Pink source code. [Online]. Available: https://github.com/dgist-datalab/PinK

[9] Y. Kang, R. Pitchumani, P. Mishra, Y.-s. Kee, F. Londono, S. Oh, J. Lee, and D. D. Lee, "Towards building a high-performance, scale-in key-value storage system," in *Proceedings of the 12th ACM International Conference on Systems and Storage*, 2019, pp. 144–154.

[10] A. Gupta, Y. Kim, and B. Urgaonkar, "Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings," *Acm Sigplan Notices*, vol. 44, no. 3, pp. 229–240, 2009.

[11] Z. Cao, S. Dong, S. Vemuri, and D. H. Du, "Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020, pp. 209–223.

[12] X. Zhang, S. Pei, J. Choi, and B. S. Kim, "Excessive ssd-internal parallelism considered harmful," in *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*, 2023, pp. 65–72.

[13] J. Im, J. Bae, C. Chung, S. Lee *et al.*, "Pink: High-speed in-storage key-value store with bounded tails," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 173–187.

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[15] Nvidia bluefield networking platform. [Online]. Available: https://www.nvidia.com/en-us/networking/products/data-processing-unit/

[16] S.-H. Kim, J. Kim, K. Jeong, and J.-S. Kim, "Transaction support using compound commands in key-value ssds," in *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[17] Openmpdk kvssd emulator. [Online]. Available: https://github.com/OpenMPDK/KVSSD/tree/master/PDK/core/src/device_abstract_layer/emulator

[18] Nvmevirt source code. [Online]. Available: https://github.com/snu-csl/nvmevirt

[19] Source code for this work. [Online]. Available: https://github.com/snu-csl/kvvirt

[20] M. P. Saha, B. S. Kim, H. S. Gunawi, and J. Bhimani, "Rhik: Reconfigurable hash-based indexing for kvssd," in *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, 2023, pp. 319–320.

[21] Y. Jin, H.-W. Tseng, Y. Papakonstantinou, and S. Swanson, "Kaml: A flexible, high-performance key-value ssd," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 373–384.

[22] D. Min and Y. Kim, "Isolating namespace and performance in key-value ssds for multi-tenant environments," in *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*, 2021, pp. 8–13.