Florida International University Technical Report **TR-FIU-KFSCIS-2021-12-10**

RHIK - Re-configurable Hash-based Indexing for KVSSD

Manoj P. Saha, Bryan Kim[†], Haryadi Gunawi[‡], Janki Bhimani
Florida International University
[†]Syracuse University
[‡]University of Chicago

RHIK - Re-configurable Hash-based Indexing for KVSSD

Manoj P. Saha, Florida International University
Bryan Kim, Syracuse University
Haryadi Gunawi, University of Chicago
Janki Bhimani, Florida International University

***Abstract*—Key-Value Solid State Drive (KVSSD), a key addressable SSD technology, promises to simplify storage management for unstructured data and improve system performance with minimal host-side intervention. However, we find that the current state-of-the-art KVSSD exhibits indexing peculiarities that limit their widespread adoption. Through experiments, we observe that the performance degrades as more data are stored, and the KVSSD can only store a limited number of key-value pairs even though the amount of data stored on device is significantly lower than its capacity. To address these shortcomings, we design a novel indexing scheme for KVSSD that dynamically resizes itself to maintain high performance and support high occupancy. We implement our proposed indexing scheme on the open-source KVSSD emulator that is validated against a real KVSSD, and demonstrate its effectiveness using real workload traces and synthetic microbenchmarks.**

I. Introduction

With the advent of unstructured data, key-value databases have become prevalent in recent times. While these databases provide flexibility and scalability, it comes at a significant overhead for managing data on the host system due to layering: key-value databases are built on top of a file system, which in turn is built on top of a block device. Such an approach of using existing abstraction layers introduces inefficiencies and is the source of performance bottlenecks in systems [9]. To address this limitation, designs that bypass the file system has gained attraction [14], [6], [11], [20], [5], [16] where the underlying device provides a direct object or key-value interface to the user application. Figure 1 illustrates the block I/O and key-value (KV) I/O stack. By bypassing the file system kernel, the KV I/O stack simplifies data management and improves the storage performance by 15× [9].

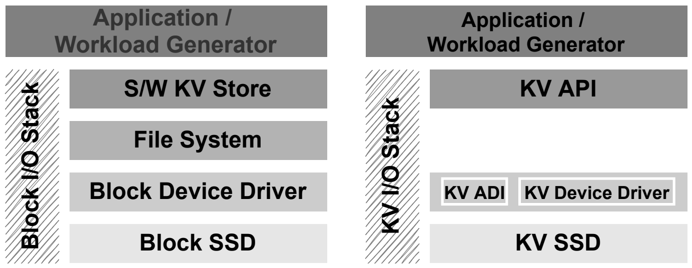


Fig. 1: Block and KV I/O Stack.

However, we are yet far away from using KVSSDs as general-purpose storage devices, the same as block SSDs. Replacing the block storage stack to KV stack gives rise to some new problems related to indexing (i.e. key-to-physical location mapping) and additional metadata management (e.g., metadata required for garbage collection and crash recovery). [7], [4], [5] identified the following challenges in designing efficient indexing and metadata management techniques for KVSSD.

First, the number of blocks that can be stored on a block SSD is pre-determined based on the SSD capacity and block size. However, pre-determining the number of KV pairs that can be stored on a KVSSD is a complex problem due to the variable size of each key-value pair. Hence, it is intractable to calculate the size of the index we need beforehand for KVSSD, in contrast to block-based SSD. In addition the size of the metadata can also become very large at times, too large to fit in the SSD DRAM. Second, the random order and variable-length of the keys means that the fixed-size sequential LBA-based index structures designed for block-based SSDs cannot be used for KVSSD. Third, efficient garbage collection and crash consistency algorithms require that some metadata (i.e. key or key signature) associated with KV pairs are also stored efficiently in each flash page, alongside the actual data. These metadata are stored in the small spare area of each flash page, usually placed after the data area in the flash page[1]. Since the number of variable-size KV pairs that can be stored in a page is also variable, the space required to store the keys or key signatures in the small spare area can often be larger than the spare area capacity, which is usually $1/32^{th}$ of the main page. Hence, it requires new metadata management techniques.

In this paper, we attempt to address the above-mentioned challenges and present the following major contributions.

- We study the limitations of different existing indexing schemes for data management in KVSSD.
- Design Re-configurable Hash-based Indexing for KVSSD (RHIK) to limit the number of flash read operations needed to fetch a mapping table entry for a key.
- We enable RHIK to support virtually unlimited number of keys until the KVSSD capacity is fully utilized.
- We implement our proposed indexing scheme on the open-source KVSSD emulator (OpenMPDK KVEMU) to mimic SSD hardware primitives and evaluate the performance of RHIK by comparing it with actual Samsung KVSSDs.

II. Design of KVSSD

We examine the various components of KVSSD and their design tradeoffs in this section.

[1]Each NAND flash page is divided into a data area to store actual data, and a spare area to store metadata such as error correction code, logical-to-physical location mapping information, bad block marker

A. Device Interface and Command Processing

The interface between user applications and KVSSD has been ratified by SNIA KV API[2] [13]. Applications can use API calls to directly execute atomic or group operations on KV pairs in synchronous or asynchronous mode. The KV API library in the host communicates with the KVSSD through a kernel or user-space device driver over the NVMe interface [1]. Five vendor-specific NVMe commands are used in Smasung KVSSD to enable KV operations over NVMe - `put`, `get`, `delete`, `exist`, and `iterate`. The `put`, `get`, and `delete` commands are used to store, retrieve, and remove KV pairs. The `exist` command is used to confirm the existence of one or more KV pairs. Lastly, the `iterate` command is used to enumerate keys or KV pairs with the particular search prefix [17]. However, [8] has shown that Samsung's NVMe command interface for KVSSD can be inefficient at times, and proposed coalescing of multiple KV API requests into a single NVMe compound command.

User applications send get, put or delete requests to KVSSD using the KV API library calls. These requests are transformed and passed to the storage device through vendor-specific NVMe commands by the kernel or SPDK driver. Inside KVSSD, the keys are processed first with a combination of local index and global index, and then the data is accessed. Samsung KVSSD also supports iterate operations with the use of a log-structured iterator manager [1], [7], [17].

B. In-Storage Data and Metadata Management

Indexing: The index structure of KVSSDs, can not use the mathematically derived traditional mapping of fixed size Logical Block Address (LBA) to fixed-size Physical Block Address (PBA) or Physical Page Address (PPA). Traditionally, for a block SSD, the number of LBA's are fixed for a give device capacity and the LBA range can be sequentially labelled as 0, 1, ..., n-1, n.

Contrary, the keyspace of KVSSD is larger than the number of blocks on a similar block SSD and is often sparsely populated. Both keys and values in KVSSD are variable in size. Ideally, unlimited number of variable-length keys assigned arbitrarily by the user applications, should be allowed by indexing scheme until the device capacity is full. For example, for a 3.84TB block-based SSD with 512B logical blocks, 7.5 billion LBAs can be stored until the device is full. However, we cannot calculate the maximum number of keys in advance for which the KV indexing scheme need to be designed because each key value pair can be of different sizes. In addition, unlike LBAs, which are numbered sequentially from 0 to the highest possible LBA (limited by the storage device capacity), the keys can be chosen arbitrarily by user applications.

The above properties have important ramifications for indexing. FTLs in block SSD optimizes the mapping table by reducing the number of entries needed to maintain the LBA to physical location relationships. For example, a hybrid FTL can coalesce entries for multiple LBA's into just one entry for a single large block. However, such optimizations are not possible in KVSSD, instead a separate entry is needed for each key in the FTL. To handle the arbitrary nature of the keys, Samsung KVSSD uses a multi-level hash table as the primary index [7]. [4] demonstrated a simple hash table index with a data layout that slices variable-size data into different partitions similar to a buddy memory allocation system. Some other researchers have followed suit [3]. However, his approach has a serious bottleneck - index size. For example, if the hash table index has 7.5 billion keys, where the average key size is 20B, and physical addresses are 5B in length, then the minimum amount of space needed to store the translation table would be 188GB. Given the scarcity of integrated RAM, it would be impossible to fit the entire translation table into KVSSD DRAM and be flushed to persistent storage. Once the index outgrows the SSD DRAM capacity, I/O performance suffers [18], [5].

To handle the issue of index scalability, [16] presented a LSM-tree based FTL for KVSSD. [5] optimized the LSM-tree-based index for KVSSD FTL by removing Bloom filters, separation of keys and values and pinning multiple levels in the SSD DRAM for faster data access. However, an LSM-tree-based index still requires a higher amount binary search operations during metadata lookups, since we don't know for sure which SSTable file contains the corresponding record. This paper presents a hash-based index that reduces the number of search operations by reducing the number of flash reads for record lookup.

Data Layout: KVSSD is made by extending the block-based SSD firmware, i.e., the underlying physical hardware of SSDs is still the same. On the host-side, the user applications need to handle the memory management functions of variable length keys and values. These variable length KV pairs, along with associated internal metadata, are directly stored as variable-length blobs in a log-like manner on the NAND flash [12]. The physical locations of these blobs are stored in the index.

Garbage Collection: KVSSD garbage collection (GC) mechanism is adapted from block-SSD and incorporates support for variable-length KV pairs. After selecting victim flash blocks, cleaning operation gets triggered, using the same policy as in block-SSD. To identify invalid blocks, GC scans the keys in flash pages (in block-SSD the key identifiers or LSNs are stored in the spare area of each flash page) in the and then validates the keys in the global hash table. Invalid (already deleted or updated) KV pairs are then erased from the device [7], [12].

III. MOTIVATION

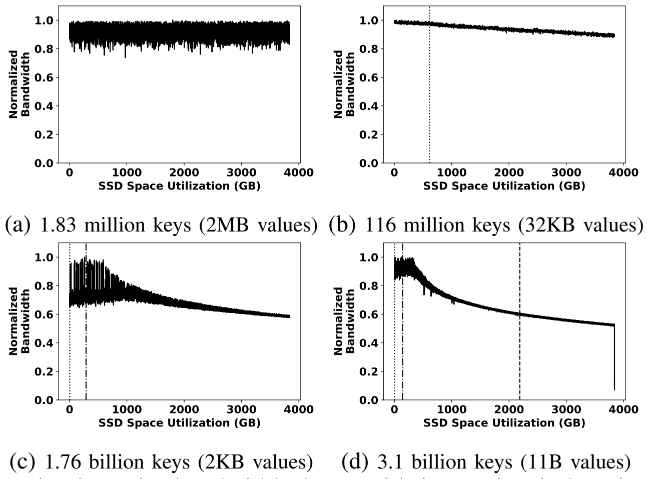
KVSSDs use existing SSD hardware and implements a KV firmware by modifying block-based firmware to enable direct KV operations support. While this design choice allows easy conversion of block SSD to KVSSD, but it has many limitations. We elaborate the drawbacks of the current design in the context of KV-indexing from our experiments using real KVSSDs in this section.

[2]Although NVMe 2.0 standard ratifies the device interface as well, it was not available at the time when this work was implemented. This work uses Samsung KVSSD command interface instead.

Performance drops as index size increases: The hash-based index in current KVSSD increases the tail latency and hampers performance as the size of the index grows [5] because the index becomes too large to fit in the SSD DRAM. This behavior is visible in Fig. 2. Fig. 2a depicts that KVSSD can maintain performance over time when the index size id small (only 1.83 million keys in index) even though the device capacity is fully utilized. Fig. 2b-2d show how write performance drops as the index grows larger and larger, from 1.83 million to 3.1 billion keys in the index. The patterned vertical lines in the Fig. 2b-2d shows the point where the current index outgrows the previous index.

The page-based, or block-based or hybrid flash translation tables that are used within block SSD to index the physical data, can no longer be directly used to support the KVSSDs, as the number of blocks need to be stored in the index can be pre-determined due to the fixed block size, but it is not straight-forward to know the number of keys that needs to be stored in the index due to the variable key-value sizes. Hence, a multi-level hash table that can be scaled as needed seems a much better choice, compared to a one-level hash table with fixed size. In addition, since a separate record needs to be maintained in the index for each KV pair, irrespective of their sequentiality, the index optimization mechanisms used in block-based SSD FTLs can no longer be employed in KVSSDs. For example, for a block SSD, a single entry in the index can be maintained for all the sequentially arranged LBAs in a large physical NAND flash block. However, for KVSSDs, each key needs a separate entry in the index. This can lead to extremely large indexes for KVSSD. In certain situations index size can be hundreds of GBs. Since such a large index can't be fitted in the SSD DRAM, maintaining such a multi-level index can incur large latency due to repeated flash operations needed to maintain the index. This paper presents a Re-configurable Hash-Index for KVSSD (RHIK) that guarantees a maximum of one flash access for each index record access.

Index supports only a limited number of keys: We observe that KVSSD also supports only a limited number of keys compared to its capacity. Even though Samsung KVSSD supports an unfathomably large keyspace (consisting of $2^4 + 2^5 + ... + 2^{254} + 2^{255} keys$) [17], our experiments reveal that a 3.84TB PM983 KVSSD can store a maximum of approximately 3.1 billion KV pairs (with value lengths of 1KB or lower). Thus, we next analyze the number of keys required by most of the existing KV stores.



(a) 1.83 million keys (2MB values) (b) 116 million keys (32KB values)

(c) 1.76 billion keys (2KB values) (d) 3.1 billion keys (11B values)

Fig. 2: Write bandwidth drops with increasing index size.

TABLE I: Diversity in the request sizes for different workloads.

Baidu Atlas - Write		FB Memcached - ETC	
Request Size	Requests	Request Size	Requests
0-4KB	1.2%	0-11B	40%
4-16KB	1.0%	12-100B	10%
16-32KB	0.8%	101B-1KB	45%
32-64KB	1.2%	1KB-1MB	5%
64-128KB	1.7%		
128-256KB	94.1%		
34 million-2.7 billion keys (4TB KVSSD)		24-744 billion keys (4TB KVSSD)	

Table I shows distribution of requests sizes in Baidu's Atlas KV store deployment [10] and Facebook's Memcached deployment [2]. Typical write workloads in Baidu's deployment consists of 94.1% writes with request sizes between 128KB and 256KB, while request sizes falling between 0B and 128KB takes up the rest. If we fully utilize a 4TB SSD capacity to store such a workload, theoretically, it would result in KV pair counts ranging between 34 million and 2.7 billion. Even though the KV pair count is highly volatile, the range is still within the maximum number of KV pairs supported by a Samsung PM983 3.84TB KVSSD. However, when we consider a typical workload in Facebook's Memcached KV store deployment, the amount of KV pairs that a 4TB drive should support falls between 24 billion and 744 billion KV pairs. Also, another recent work characterizes workload distributions of three RocksDB deployments - UDB, ZippyDB, and UP2X - at Facebook [19]. The average KV pair length in these KV stores falls between 57B and 153B. To support such workloads, a 4TB KVSSD should be able to handle between 26 billion and 700 billion keys.

Thus, supporting a large index and sustaining performance with increasing index size is impossible in current KVSSD. RHIK proposes a novel hash-based indexing design that can ensure predictable metadata access overheads by limiting the number of levels in multi-level hash index to just two, with the top level being integrated RAM resident, and the bottom level being flash resident. RHIK also supports virtually unlimited keys in the index by conservatively re-configuring the index.

IV. RHIK Architecture and Implementation

In this section, we illustrate RHIK's architecture and discuss how our design choices address the issues pointed out in Section III. Our proposed indexing model strives to fetch the K2P mapping table entry for a key with at most one flash read, and can be re-configured to handle virtually unlimited number of keys. Even while storing large number of KV pairs in the device, RHIK is designed to minimize the flash access overhead of the internal metadata fetching operations.

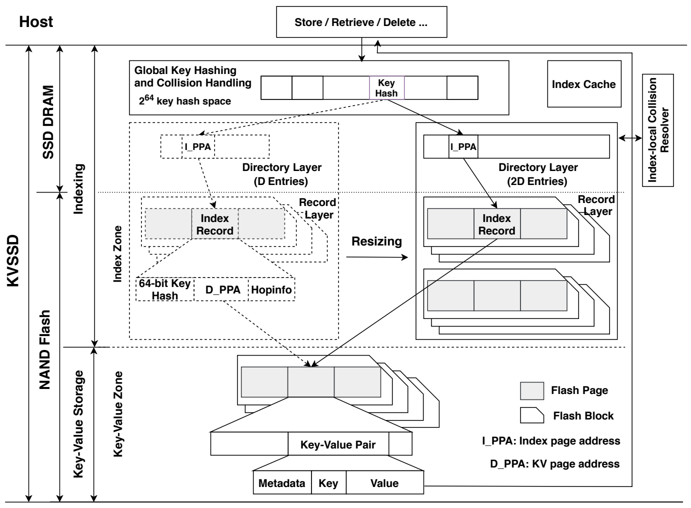


Fig. 3: Re-configurable Indexing

A. Re-configurable Hash Index for KVSSD

RHIK is designed as a two-level hash table with high occupancy. The first layer in RHIK works like a directory, containing D entries (Eq. 2), and accessed from SSD DRAM. At the same time, a periodically updated persistent copy of these D entries resides on flash. The second layer, or record layer, holds fixed-size independent hash tables that store the metadata of corresponding KV pairs and is served from flash unless available in the integrated RAM. Figure 3 shows the overall architecture of our RHIK in relation to the internal component of the KVSSD such as NAND flash and integrated RAM.

RHIK transforms variable-sized keys coming from the application into fixed-size (64-bit) key signatures using a simple hash function such as MurmurHash2. This fixed-size key signature is used as an identifier of the key within the scope of the index, and the size of the key signature can be configured at initialization to contain higher or lower number of bits, although we are using 64-bit key signature by default. The 64-bit key signature is used to select the appropriate directory and record corresponding to the key using a variable hash function in the directory layer and a fixed hash function in the record layer. The variable hash function uses D least significant bits of the 64-bit key signature to compute the target bucket in the directory layer. The value of D can be set arbitrarily or using Eq. 2, based on anticipated workload. As the index is resized, the value of D is updated. Every entry in the directory layer points to a flash page containing a unique hash table index in the record layer. In record layer, each record in the hash table stores the 64-bit key signature, the physical address of the KV pair on flash, and information related to index occupancy for each bucket (also known as hopinfo). The size of the record layer hash tables are fixed and can contain R index records (which translates into a fixed hash function for all hash tables in the record layer). R chosen such that the total storage space required to store each hash index equals the size of a flash page. See Eq. 1 where p is the flash page size, kh is the key signature size, ppa is the physical page address size, and hi is the hopinfo size. When RHIK is deployed in a generic environment and information about the expected workload is unknown, it can be initialized conservatively to handle a limited number of keys. Later, in this section we explain RHIK's process of resizing index.

$$R = \lfloor \frac{p}{kh + ppa + hi} \rfloor \tag{1}$$

$$D = \frac{anticipated\ number\ of\ keys}{R} \tag{2}$$

To further explain the operations of RHIK , we discuss how the store, retrieve, and delete operations are handled in it. When KVSSD receives a store request for a KV pair, it first computes a 64-bit key signature using a simple hash function. Then a key exist operation is executed in RHIK to identify whether the key already exists or not. Suppose no record corresponding to the key is found. In that case, the KV pair is written on flash and the corresponding record is stored in the index. A retrieve operation takes the key as input, computes the hash of the key to lookup corresponding record in the index. If the key exit operation returns that the key might exist. The KV pair is retrieved. The key is matched before the value is returned, to ensure that no key signature collision has occurred. The key signature is computed to handle a delete operation, and the corresponding record is fetched from the index. The record is then fetched from flash to match the request key, similar to retrieve operation. If a match is found, the record is deleted from the index, and the corresponding KV pair is marked stale for garbage collection.

1) Collision Management: Since RHIK used fixed-size hash tables in the record layer, handling collisions within this scope is imperative. To handle index-local collisions and achieve high index occupancy in the record layer hash tables, By default, RHIK employs Hopscotch hashing with hopinfo size 32, and can be configured at initialization. Suppose an empty record slot can not be found within these confines. In that case, an uncorrectable error is returned, and the operation is aborted. Such aborts, however, are not frequent and more details about such collisions can be found in Section V .

2) Resizing Index: RHIK can be initialized conservatively to support a limited number of keys in the beginning to reduce space wastage related to provisioning a large under-utilized hash index structure. Once the total occupancy of RHIK reaches a pre-defined threshold (e.g., 80%), its resizing function is triggered. Every time while resizing, a new index is initialized with double the capacity of the current active index. The directory layer of the new index is doubled, compared to the directory layer in the current index. The number of independent hash tables in the record layer also doubles, while the size of each independent hash table in the record layer remains the same. Entries from the old index are then inserted in the new index. Maintaining a single global index is important to serve queries quickly later. Our key to achieving faster migration from the old index to the new index lies in the fact that we store the 64-bit key signatures corresponding to each key stored in the KVSSD inside the hash indexes in the secondary layer. We reuse these key signatures to rearrange the records in the new index quickly. The KV pairs stored in the device are not accessed to recompute the key signatures. During the migration, all new commands are kept into the submission queue until the end of the migration process to avoid any inconsistency of the persistent data from newly incoming requests.

After migrating to the new index, the flash pages containing the old index records are marked stale for the garbage collector, and all operations are pointed to the new index. We recognize the trade-off that all the hash-based data management schemes has some additional garbage collection due to their index management. However, they provide excellent performance for all the queries while runtime due to their efficiency in searching the data. From our study, we see that most key value stores have high read intensity, so we assume the above short-coming to be a valid design choice.

3) Membership Checking: Since the key-exist operation is an integral part of KV operations, as noted earlier in this section, achieving faster membership checking is crucial to ensure faster performance. We propose a probabilistic membership[3] checking by reusing the key signatures, instead of traditional Bloom filter-based approaches. Since the key signatures serve the purpose of a unique key identifier within the confines of the index and are stored inside every record, it can be used to check membership, instead of Bloom filters. We believe that this approach is better compared to the Bloom filter, considering the variable size of the key range and the bottleneck of scaling Bloom filters efficiently. However, as index occupancy reaches millions of entries, the probability of getting a collision in the 64-bit global key signature space increases. That means, the probability of correctly identifying that a key-exist just from the key signature decreases, and additional flash reads to retrieve the actual key from the flash is needed. We can omit such expensive operations for all KV operations other than explicit key-exist operation requests. For example, since retrieve operations were intended to read the KV pair from flash, we can recheck whether the request key matches with the retrieved key before returning the value to the requesting application. Write and delete operations implement the same technique to simplify membership checking. An alternative is to use a higher resolution hashing for key signature generation. For example, using 128-bit key signatures, instead of 64-bit signatures, will reduce the chances of key signature collisions [15]. By carefully selecting the hashing function to generate key signatures, we can omit the extra steps of rechecking the actual keys from flash.

4) Fast Metadata Processing: Limiting the index to two levels and resizing the index to accommodate new metadata ensures that RHIK maintains a predictable execution time for handling metadata. Since the directory layer of the index remains small in size (0.005 bytes/key for 32KB flash page), it can be maintained in the embedded DRAM without incurring much space. Thus, RHIK 's design ensures that retrieving index records for any keys stored in the device will take a maximum of 1 flash read (for retrieving the record layer hash table). We compare RHIK with existing solutions using overall performance during the store and retrieve commands and rate of change in key exist operation execution as the index grows.

5) Improving Maximum Value Size Limit: RHIK removes the index-induced limit on maximum value size, as observed in NVMKV, by implementing a key-value aware indexing scheme. We decouple the data packing problem from the index. We assume that the flash storage is logically partitioned into small storage units[4] and implement an extent-based data packing technique to store KV pairs sequentially, similar to EMT-FTL [3], as depicted in Fig. 4. The index only stores the starting address of the KV pair on flash. The metadata stored with each KV pair is then used to retrieve the KV pair. This approach enables the storage of large values.

B. Garbage Collection

Existing KVSSD garbage collector (GC) can be modified to work with the proposed data layout. To identify stale data, GC needs to scan the key signatures in each flash page of a block, and check if the data is valid or stale (updated or deleted KV pairs) by querying the index. Stale data can then be discarded. Victim block selection and merging operations can proceed according to existing GC algorithms.

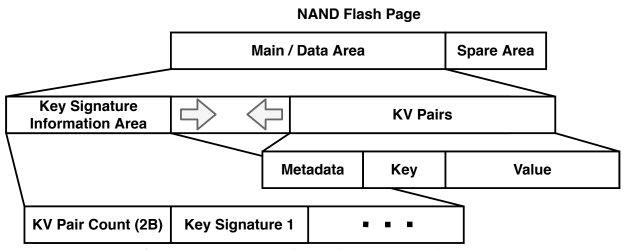


Fig. 4: RHIK data layout scheme.

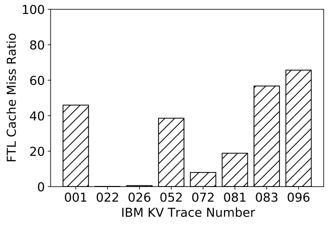
C. RHIK *on KV Emulator*

We develop an advanced version of the KV Emulator by extending OpenMDK KV Emulator. The new KV Emulator imitates the fundamental hardware primitives of an SSD, such as flash blocks, pages, etc. We implement RHIK in the new KV Emulator. We implement separate indexing and data layout schemes.

V. Evaluation

In this section, we evaluate the performance of RHIK . The goal of our evaluation is to determine whether the use of RHIK is able to dynamically scale the index as per the workloads requirement, without drastic increase in scaling time with larger index size. Thus, first we analyze the impact of resizing in RHIK . Then, we evaluate RHIK using read and write KV operations performed in sync and async way. Finally,

[3]The probability of identifying a key being stored in flash when it is not, is decided by the probability of getting the same key signature for two different keys, and depends on the hash function and key distribution.


(a) Cache miss ratio of for different clusters in IBM KV traces.

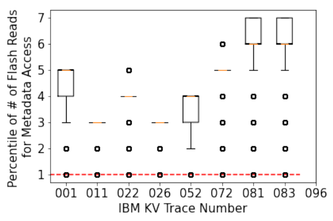
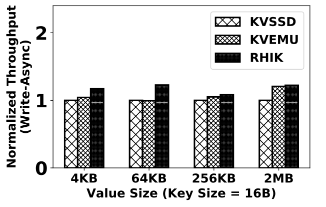
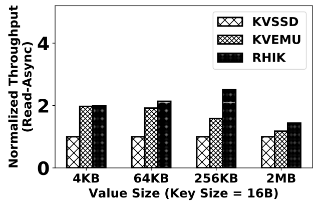
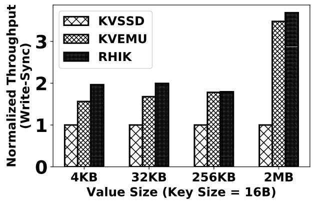

(b) Percentile of flash accesses required for metadata access for different clusters in IBM KV traces.

Fig. 5: Performance comparison of RHIK and 8-level Multi-Level Hash Index.


(a) Normalized throughput for async writes with different value sizes.


(b) Normalized throughput for async reads with different value sizes.


(c) Normalized throughput for sync writes with different value sizes.

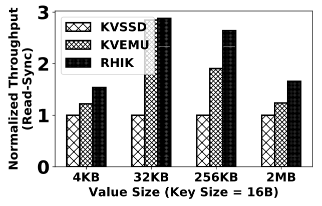

(d) Normalized throughput for sync reads with different value sizes.

Fig. 6: I/O Performance Analysis

we discuss in-depth sensitivity analysis to better understand the impact of resizing with respect to percentage of collisions and index occupancy in our RHIK . Overall, our evaluation addresses three broad questions:

- How does RHIK perform compared to the existing solution?
- How fast RHIK scales as the index grows?
- How RHIK behaves for different configurations?

A. Platform Setup

We test RHIK on a system comprising of 2x Intel Xeon Silver 4208 CPU @ 2.10GHz processors, 192GB DDR4 DRAM, and Samsung PM983 NVMe KVSSD (firmware version ETA51KCA). Since the emulator resides in system memory and is limited by the available memory in the host system, we set our emulated KVSSD capacity at 150GB. We configure our emulator with erase blocks consisting of 256 flash pages of size 32KB each, as default. We communicate with the KVSSD (both emulated and real) through SNIA KV API and run our experiments using KVBench and custom scripts.

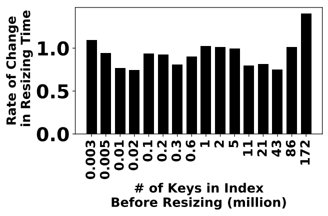


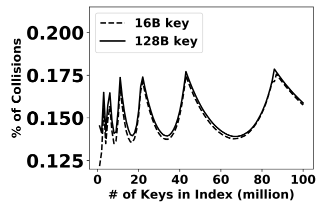
Fig. 7: Rate of change of the resizing time to double index capacity.

B. Results and Analysis

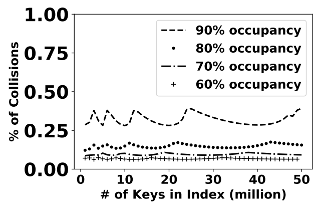
First, we compare RHIK's performance with a multi-level hash index to demonstrate the advantages of our approach. To evaluate RHIK's performance in scenarios where the index size is significantly smaller, approximately the same size, or significantly larger than the available SSD DRAM cache, we limit the SSD DRAM cache budget to 10MB (i.e. for a 10GB SSD) only. We replay the real-world IBM Cloud Object Store KV traces on this SSD setup to demonstrate how RHIK outperforms a traditional multi-level hash index. Out of the eight clusters in Fig. 5, four clusters (i.e. 022, 026, 052, and 072) need very small index compared to SSD cache budget. Two clusters (i.e. 083 and 096) need significantly large index. Fig. 5a shows the cache miss ratio for each of these workloads when emulated on a KVSSD with 10MB cache budget for FTL. A large index with a traditional multi-level hash index incurs high cache miss ratio, along with higher number of flash reads for fetching each record from flash. Whereas, RHIK requires only one flash read to fetch metadata from flash (Fig. 5b).

Next, we evaluate RHIK 's performance using multiple sequential workloads with variable key-value size consolidating to 1GB. Since KVSSD and RHIK both maintain unordered index with KV pairs written in a log-like manner on flash, and KV operations are largely dominated by key handling operations [8], the performance of sequential workloads are not significantly different from Uniform or Zipfian workloads in KVSSD for most cases. Figure 6a compares RHIK write performance (asynchronous) with Samsung KVSSD and OpenMPDK KV Emulator. For almost all value sizes, RHIK achieves higher throughput. The throughput comparison for read performance is shown in Figure 6b. We observe that RHIK is able to perform better with large value sizes during read. We observe that the performance trends of the normalized throughput of the OpenMPDK KV Emulator differs from KVSSD. We believe that this difference in the performance trends may be due to the IOPS model used by the OpenMPDK KV Emulator. Since, our extended KV Emulator also uses the same IOPS model, we intend to measure RHIK 's performance improvements relative to OpenMPDK KV Emulator performance.

We further evaluate RHIK's resizing performance. Instead of resizing the index whenever a collision occurs, RHIK takes a conservative approach in resizing the index, to reduce space wastage and unnecessary flash reads. The index resize function is triggered when the index contains a preset number of



(a) Effect of key size on the percentage of collision in index of different sizes.



(b) Effect of occupancy on the percentage of collision in index of different sizes.

Fig. 8: Sensitivity Analysis

records. By default, we set RHIK 's index occupancy threshold at 80%. Upon reaching the threshold, RHIK is resized to double it's current size. We analyze the rate of increase of RHIK 's resizing time with an increase in the index size to evaluate our approach . From Figure 7, we observe that RHIK mostly maintains the rate lower than equal to 1, which shows that the rate of increase of the resizing time remains constant even while doubling the capacity with increased index size. The latency of operations during the resizing is expected to increase, depending upon the size of the reconfigured index. For example, to resize the index with capacity of 11 million keys, the scaling time was 5 milliseconds in our experiments, while resizing to a capacity of 345 millions keys took 172 milliseconds. To the best of our knowledge none of the existing indexing schemes [21] discuss this property (index capacity) while this is very important for KVSSDs.

C. Sensitivity Analysis

Since RHIK uses fixed-length key hashes from wide global address space, we analyze if collision trends are similar across different key sizes (Figure 8a). Our analysis shows, different size keys show similar collisions trends. We also analyzed the effect of index occupancy on collision to determine optimal occupancy threshold for better collision management (Figure 8b). Collision handling degrades heavily above 80% index occupancy.

VI. Discussions

In this work, we investigate the indexing schemes for data management in KVSSDs. We propose an orthogonal idea of designing a resizable indexing technique, which is critically important for KVSSDs to avoid over-allocation of storage space or constrained number of key-value pairs. Based on our investigation, we plan to advance in the following major directions;

- **Real-time index scaling:** Current implementation of RHIK keeps I/O requests in submission queue on halt while re-configuring the index. This increases the tail latency of I/O requests during that period. A real-time re-configuration approach would mitigate the issue to a large extent.
- **Collision Management:** To optimize metadata storage and management, RHIK collision handling algorithm has to reject some keys. The application needs to generate a new key and issue a new I/O request in such instances. A robust collision management algorithm with hyper-local scaling of the index may be used to solve this issue. By re-configuring only the buckets
- **Integrated Iterator Support:** Although RHIK does not offer iterator support in its current form, including iterator support would be easy due to RHIK's structure. By generating key signatures with 4B prefix and 4B suffix of the original key, and by careful partitioning of the keys inside the multi-level index, we can easily add support for iterator operations. Such a design would also be able to support key-value iterator operations, currently absent in Samsung KVSSD.
- **Integrate advantages of hash-based and log-structured merge based indexing:** Enhance our resizable indexing scheme, which takes advantage of fast querying operations of a hash-based indexing scheme along with the techniques to simultaneously reduce the internal write amplification of KVSSDs to improve their lifetime. We wonder, what is the feasibility of the co-existence of these two in a single indexing scheme? Is there any limitation regarding feasibility?
- **Long-term relevance of RHIK :** We anticipate and assume that KVSSDs will be cost and performance efficient devices. They will be beneficial to most of our NoSQL workloads in evolving times. Thus, designing an optimal indexing scheme for these KVSSDs is critically important. However, as the technology is not yet easily available to buy at the current time, nothing can be said about its adaptation in our existing storage stack. What are the worst-case scenarios wherein a system such as RHIK is rendered unnecessary? How likely is it that those scenarios play out in reality? Can such indexing techniques be re-purposed to be used with other system layers such as within key-value stores, or to take advantage of persistent memory?

VII. Conclusion

As NoSQL applications become increasingly data and memory intensive, the synergistic use of KVSSDs becomes a compelling possibility. How to best utilize these disparate technologies to build a unified solution continues to remain an open question. In this paper, we explore indexing schemes to build a resizable indexing scheme, RHIK , with fast, horizontal scaling to support running variable workloads on KVSSDs. We designed a key-value aware indexing scheme that requires a maximum of one flash reads to fetch metadata, and can be reconfigured to accommodate more keys as the index grows, while maintaining the maximum flash access limit to just one. A preliminary prototype of RHIK is implemented by extending the OpenMPDK KV emulator. RHIK 's approach is promising, and it particularly explores new directions to resolve challenges for efficient indexing scheme to use KVSSD.

References

[1] *KV SSD host software including APIs and drivers.* https://github.com/OpenMPDK/KVSSD.

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[3] Tai Chang, Jen-Wei Hsieh, Tai-Chieh Chang, and Liang-Wei Lai. Emt: Elegantly measured tanner for key-value store on ssd. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2021.

[4] Y. Chen, M. Yang, Y. Chang, T. Chen, H. Wei, and W. Shih. Kvftl: Optimization of storage space utilization for key-value-specific flash storage devices. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 584–590, 2017.

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

[6] Y. Jin, H. 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)*, pages 373–384, 2017.

[7] Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang-suk Kee, Francisco Londono, Sangyoon Oh, Jongyeol Lee, and Daniel D. G. Lee. Towards building a high-performance, scale-in key-value storage system. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, page 144–154, New York, NY, USA, 2019. Association for Computing Machinery.

[8] Sang-Hoon Kim, Jinhong Kim, Kisik Jeong, and Jin-Soo Kim. Transaction support using compound commands in key-value ssds. In *Proceedings of the 11th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'19, page 1, USA, 2019. USENIX Association.

[9] Samsung Memory Solutions Lab. *Samsung Key Value SSD enables High Performance Scaling*, 2017. Retrieved from https://www.samsung.com/semiconductor/global.semi.static/Samsung_Key_Value_SSD_enables_High_Performance_Scaling-0.pdf.

[10] C. Lai, S. Jiang, L. Yang, S. Lin, G. Sun, Z. Hou, C. Cui, and J. Cong. Atlas: Baidu's key-value storage system for cloud data. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14, 2015.

[11] Stanley Miao. *Key/Value SSD Design Overview and Use Cases*, 2019. Retrieved from https://www.flashmemorysummit.com/Proceedings2019/08-07-Wednesday/20190807_SOFT-202-1_Miao.pdf.

[12] Rekha Pitchumani and Yang-Suk Kee. Hybrid data reliability for emerging key-value storage devices. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 309–322, Santa Clara, CA, February 2020. USENIX Association.

[13] SNIA Technical Position. *Key Value Storage API Specification Version 1.0*, 2019. Retrieved from https://www.snia.org/tech_activities/standards/curr_standards/kvsapi.

[14] LLC SDxCentral. *Kinetic Open Storage Project*. Retrieved August 15, 2020 from https://www.sdxcentral.com/projects/kinetic-open-storage-project/.

[15] Wikiedia. *Birthday problem*. Retrieved Aug 20, 2020 from https://en.wikipedia.org/wiki/Birthday_problem.

[16] S. Wu, K. Lin, and L. Chang. Kvssd: Close integration of lsm trees and flash translation layer for write-efficient kv store. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 563–568, March 2018.

[17] Sang yoon Oh. *KV-SSD Seminar*, 2018. Retrieved August 15, 2020 from https://github.com/OpenMPDK/KVSSD/wiki/presentation/kvssd_seminar_2018/kvssd_seminar_2018_fw_introduction.pdf.

[18] Osmar Zaiane. *Indexing Hashing*, 1998. Retrieved from https://www2.cs.sfu.ca/CourseCentral/354/zaiane/material/notes/Chapter11/node1.html.

[19] zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, Santa Clara, CA, February 2020. USENIX Association.

[20] Itai Ben Zion. Key-value ftl over open channel ssd. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, page 192, New York, NY, USA, 2019. Association for Computing Machinery.

[21] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 461–476, Carlsbad, CA, October 2018. USENIX Association.