FlashKey:A High-Performance Flash Friendly Key-Value Store

Madhurima Ray, Krishna Kant
Temple University, Philadelphia, PA
{madhurima.ray, kkant}@temple.edu

Peng Li, Sanjeev Trika
Intel Corporation, Aloha, OR
{peng1.li, sanjeev.trika}@intel.com

***Abstract*—Key-value stores (KVS) provide an efficient storage for increasing amounts of semi-structured or unstructured data generated by many applications. Most KVS in existence have been designed for hard-disk based storage where avoiding random accesses is crucial for good performance. Unfortunately, the resulting storage structures result in high read, write, and space amplifications when used on modern SSDs. In this paper, we introduce a KV store especially designed for SSDs, called *FlashKey*, and demonstrate that even as an initial implementation, it substantially outperforms the two most popular commercial KVS in existence, namely, Google's LevelDB and Facebook's RocksDB. In particular, we show that FlashKey achieves up to 85% improvement in average access latency, 2x improvement in tail latencies, and 12x improvement in write amplification, at comparable or better space-amplification. Furthermore, FlashKey can easily trade off space and write amplifications, thereby providing a new tuning knob that is difficult to implement in LevelDB and RocksDB.**

***Index Terms*—key-value store, read amplification, tail latency, write amplification, space amplification, YCSB, hashing**

I. Introduction

Key-value (KV) Stores are increasingly popular as they are well suited for storing and operating on streaming data generated by a variety of online applications including social media, e-commerce, scientific experiments, IoT monitoring of smart infrastructures, etc. Many KV store designs are currently in existence but are largely designed for hard disk drives (HDDs) as the backend. HDDs have very poor random read and write performance compared to their sequential performance (as shown in Table I), and the KV systems, therefore, go to great lengths to sequentialize writes, and also to front the HDDs with memory or other caches to mitigate the poor random read performance. These schemes, unfortunately, have high read and write amplification penalties. As Solid-State Disks (SSDs) become widely deployed in Datacenters, there is an opportunity to redesign the KV architecture to benefit from the SSDs' fast random performance (Table I).

In this paper, we present FlashKey - a flash friendly KV store, with data structures and algorithms that capitalize on good random IO performance by using hashing and can work with low-endurance TLC/QLC SSDs because of significantly reduced write-amplification. The solution uses a hash-table structure to provide fast "Get" operations and a "logical band" based data organization that allows partial sequentialization of "Put" data writes to the SSD since sequential writes still provide better performance and also improve the SSD write-endurance. Finally, FlashKey also includes a Log-Structured-Merge (LSM) tree structure for keys, without values, to support "Scan" operations.

This research was supported by NSF grant CNS-1422921.

TABLE I
Device IO Performance

Device (HDD:Segate SSD:Intel)	Rand R/W (MB/s)	Seq R/W (MB/s))	Endurance DWPD(GB)
HDD 10K Ent.	1.6	241	-
TLC DC P4610 1.6TB	2510/780	3200/2080	6712.329
QLC DS-P4420 7.68TB	1660/140	3200/1000	3068.493

Through a detailed experimental evaluation *on current SSDs*, we show that FlashKey, in spite of being an initial implementation, outperforms not only Google's original KV store called LevelDB, but also the state of the art and highly optimized RocksDB developed by Facebook. In particular, we demonstrate the following:

a) The unique design of FlashKey optimizes SSD accesses and thus provides a much lower write amplification than both LevelDB and RocksDB at comparable space amplification.

b) FlashKey provides a much lower average latency than both LevelDB and RocksDB. It also provides a much lower 99.999 percentile tail latency than LevelDB and RocksDB.

c) The read amplification of FlashKey is comparable and, in fact, generally lower than that for LevelDB and RocksDB.

d) FlashKey prioritizes high performance for individual record accesses over that for large key ranges. Nevertheless, it can handle range queries via additional key-only structures.

We show that FlashKey improves Get() and Put() latencies by up to 85%, and write amplification (WA) by up to 12x. The reasoning behind these improvements is as follows:

Get() performance : (1) Hash lookup eliminates the need for tree-traversal on the disk. Finding the key index needs O(1) time with one storage access on average. Typically two storage accesses are required to complete a Get() even when the hash-table is on disk. (2) A strong primary hash function reduces collision and a secondary hash is used to avoid unnecessary disk lookup(s) upon primary hash collision.

Put() performance : A typical Put(.) operation in FlashKey requires two writes to disk.

Garbage Collection (GC) performance, WA and RA: We control write and read-amplification in FlashKey by laying out the data intelligently on disk in "logical bands". The logical

bands allow for locally-sequential writes of data, and allow us to garbage collect the KV entries post updates/deletes in a smart manner using heuristics similar to those deployed inside the SSDs. Logical bands with the most invalid data-elements are garbage-collected first.

The rest of the paper is organized as follows. Section II states the key value operations and the performance metrics used to evaluate any KV store. Sections III and IV discuss the related work in the area of KV stores and the popular KV stores that are currently in use respectively. Section V discusses the architecture and algorithms of FlashKey. Section VI evaluates the performance of FlashKey in comparison to LevelDB and RocksDB on key benchmarks. Finally, section VII concludes with a summary and discussion of future work.

II. Key Value Operations and Metrics

The user-operations supported by all KV-stores are Put, Get, Delete, and (in-some-cases) Scan. These operations form the foundation for KV stores, and are sufficient for most applications. The Put(K,V) adds the Key:Value mapping, if it does not exist already, to the associative array/dictionary managed by the store. The operation updates the value for the specified key if a mapping already exists. The Get(K) operation retrieves the associated value, or returns an error if none is found. The Delete(K) operation deletes a stored K:V mapping, or returns an error if no such mapping exists. The Scan(K,range) operation allows iterate over (ordered) keys, starting with specified keys K and reading all keys upto the specified range.

The primary metrics for evaluating any KV database are:

1. **Space Amplification (SA)**. SA is the ratio between the size consumed by the database on the disk to the actual size of the user data, including keys and values. Higher SA typically reduces the amount of background operations required to sequentialize writes (resulting in lower WA and higher net performance, at the expense of higher $/GB effective cost), and vice-versa.

2. **Write amplification (WA)**. This is the ratio of total bytes of data written by the KVS to the storage device, to the bytes of data written (Put(.) request) by the user. Lower WA corresponds to fewer writes to the disk, and hence better write performance, drive-endurance, and QoS.

3. **Read amplification (RA)**. This is measured as total bytes of data read from the disk divided by the bytes of data returned to the user for their Get queries. Lower RA results in fewer reads to the disk, and hence better read performance and QoS.

4. **Average Latency**. This measures the amount of time it takes on average for each Put or Get request to finish. Lower latencies provide better responsiveness to clients and their applications.

5. **Throughput**. This measures the amount of data that can be Put and Get per unit-time, from the KV store. Throughput measures sustained performance.

6. **Quality of Service (QoS)**. A problem with several storage solutions is that, while they provide good average performance, they suffer from a highly variable performance. Real world deployments, however, prefer a deterministic latency, especially for the reads. Thus an important QoS metric for read operations is Read Tail Latency (RTL), which is measured as 99 percentile or even higher percentile of read latency.

III. Related Work

The existing works on KVS belong to four primary classes: in-memory KVS [1] (e.g., Memcached, Redis); Persistent-Memory based KVS [2]–[5](e.g pmemkv *https://github.com/pmem/pmemkv*); HDD or NAND flash-based KVS (e.g., LSM based KV); and all the other types of KVS (e.g., designed for SMR drives or Open-FTL) [6]–[16]. Here, our discussion is limited to the Flash based KV stores for relevance.

Flash-based KV Store: In [17], the author used KV isolation similar to FlashKey, where the value is stored separately in a circular buffer -"vLog", indexed by the LSM tree. However, the vLog by design is inefficient for GC, as it uses FIFO for key selection and relocation during GC. Also, during GC, each key is validated by an LSM tree lookup which slows down the GC as a key lookup in [17] needs $O(\log(n))$ time, as compared to O(1) time in FlashKey.

In HashKV [18] the author separates the keys from its values based on the value-sizes. Small values are stored with keys in the LSM, and large values are stored in separate datastore indexed by the LSM tree. Such an approach would work well if items with small values are more popular than others, but otherwise, the additional search in the LSM may be detrimental to performance. In our case, all our lookups take on an average O(1) time independent of KV sizes or key access profile. HashKV also partitions the datastore into groups and uses key hashes to map the value across these fixed groups, whereas FlashKey allows more flexibility as it places KVs within arbitrary LBands.

In the paper [19] the author caches the KV pairs in flash, indexed by an in-memory Cuckoo-hashed table for performance as the KVS is on HDD. An extended work [20], optimizes the performance of hashing during a collision by linear chaining of the records mapped to the same index in flash. μDepot [21] uses a DRAM-based two-level hopscotch hashtable indexing that can be dynamically resized as the database grows. To restrict lookup(s) to a single IO, the index uses more bytes which helps to cover a large address space but increases DRAM footprint. In our work, the hashtable does not store the key which reduces the space requirement and update latency.

SILT [22] uses three different stores where a key is initially inserted into a write-optimized LogStore, indexed by in-memory hashtable, then eventually merged to one of the many on-flash HashStore, and finally ends up in an immutable SortedStore with even lower in-memory index. The key concern of SILT is to reduce the DRAM usage for indexing, perhaps at the cost of higher WA/SA, which is different from our objectives.

SlimDB [23] picks up some of the design features of SILT and tries to optimize the index and filtering, compaction algorithms and reduces the memory footprint further. PebblesDB [24] uses Fragmented LSM (FLSM) to increase the write throughput (↓ WA) at the cost of higher read latency and SA [25] than RocksDB/LevelDB. The WA in FlashKey is lower than both the RocksDB and LevelDB with similar SA, and also the performance improvement comes at the cost of architecture change without sacrificing the read performance.

IV. Current Key-Value Stores

A large number of KV databases are currently available where they are typically implemented on top of the underlying storage layer, although storage level implementations also exist [26], [27]. Two of the most popular and widely deployed KVSes are RocksDB by Facebook and LevelDB by Google. They are both based on *Log-structured merge (LSM) tree* [28] and we will compare FlashKey performance against these.

A. Architecture of Log Structured Merge-Tree

LSM tree is a data structure that converts small random writes to large sequential writes by batching them in memory. It stores the sorted files in multiple tree levels to provide O($\log N$) lookup time. The sequentialization makes LSM Tree better than B-Trees. LSM tree is a good structure for HDDs because it minimizes random accesses. It provides indexed access to files on HDDs with high insert, update, delete and scan operations [29].

A LSM tree consists of two or more components $C_{-1}, C_0, .. , C_i$ of exponentially increasing sizes. Component C_{-1} stays in-memory and is updated in-place whereas the others are on-disk and updated out-of-place. For better Put(.) latencies, an insert to a LSM-tree gets added to a memory buffer at C_{-1} (and also appended to a Write-Ahead Log for persistence). Once C_{-1} exceeds a certain size, a contiguous and sorted segment of entries is removed from C_{-1} and flushed onto the disk. This flush may cause level C_0 to exceed its size limit which is then merged with C_1 using an algorithm similar to mergesort. The process, called **"compaction"**, merges overlapping parts of C_0 and C_1, and the new merged file replaces the old files of C_1.

Compaction is a background process that removes overwritten and deleted data. Compaction amplifies reads and writes, since two files with overlapping data records must be both read and the most up to date data written to a third file. Depending on the rate and the extent of compaction, the system exhibits different trade-offs. Infrequent compaction increases the space amplification (SA) whereas frequent compaction increases both RA and WA significantly and may cause both reads and writes to stall as it uses more of both CPU and IO.

The data lookup in LSM-tree largely relies on in-memory caches for performance on cache-hits. Upon a cache miss, LSM tries to locate the data in C_{-1} first with a linear search starting from the most recent data to get the latest update if any. Upon missing out of C_{-1}, the read cascades through the following levels. Thus, in the best case, a lookup can be resolved at C_{-1}, however, in the worst case, it can travel through all $C_0, .., C_i$. Hence a single read operation can be amplified by reading a number of different components in the LSM tree.

B. LevelDB

LevelDB implements the LSM-tree as discussed in section IV-A. For the in memory component (C_{-1}) it uses sorted skiplist-based "**M**emtable". The on-disk part ($C_i, \forall i \geq 0$) is divided into levels L_0 through L_6, where each level L_i is *N* times (*N: level size multiplier*) bigger than the adjacent level L_{i-1}. Usually, N is set as 10. Each level is constituted of several "**S**orted **S**tring **T**able (SST)" files. In LevelDB, an insert of a KV pair first goes to the "mutable" memtable. As multiple inserts exhaust the capacity of the memtable, it switches to a new memtable. The old memtable is marked as "immutable" and in the background flushed as a SST file to the disk at level L_0. Therefore overlap in key ranges is possible in the level L_0, but is not allowed for other levels.

The cache-miss on lookup requires a search in the memtables and the SST files from L_0 to L_6 in order which increases the RA. For level L_0 (with overlap), a lookup might require exploring all the SST files (worst case) and for the other levels lookup only touches one SST file per level (no overlap), and does a binary-search within each SST. LevelDB also requires metadata lookup which increases the RA further. It is needed to identify which SST files at each level to look into.

Compaction is triggered whenever any level L_{i-1} exceeds its size limit; during which LevelDB removes all the invalid key value pairs (deleted or updated). In the worst case, while compacting level L_{i-1} to level L_i, LevelDB can read at most *N* files from level L_i, merge-sort and write back to level L_i. This can increase the WA up to *N* times. With level L_0 to L_6, new data propagates to higher-numbered levels through a series of compactions which can increase the WA to even higher. A garbage collection method is called at the end of every compaction to remove the obsolete files that are currently not in use. Hence, in LevelDB, the SA always stays bounded, at the cost of high WA.

C. RocksDB

RocksDB, which extends LevelDB, is an embedded persistent key-value storage system. It has many additional features [30] beyond LevelDB, including multithreaded compaction, multithreaded memtable insertion, reduced DB mutex holding, optimized compaction, etc. In addition to the LevelDB style compaction, RocksDB includes Universal compaction where instead of aggressively merging a smaller SST with a larger one, the system waits for several sorted runs of similar size before merging them into a big run. This new compaction method reduces WA at the cost of doubling SA at times. Multi-threaded compaction allows RocksDB to run parallel compactions across several non-overlapping regions of the KV store simultaneously as long as resources are available.

To reduce the SA, instead of statically setting the level target size as 10x of the prior level, RocksDB introduces a feature called dynamic level size adjustment where the size target of levels is changed dynamically based on the size of the last level in a bottom-up manner. As the database size changes, it dynamically adjusts the size of each level to be $\frac{1}{10}$th the size of the data stored on the next higher-numbered level.

V. FlashKey

In this section, we define the architecture of FlashKey. Given the excellent random IO performance of Flash and other emerging NVRAM technologies, we believe that hashing is a viable approach for locating KV records. Our current FlashKey architecture, illustrated in figures 1 and 2, uses simple hashing, although more complex mechanisms such as cuckoo hashing for better space utilization or locality sensitive hashing (LSH) to have better range query performance - may also be useful. FlashKey uses (1) a hash table (HT) for storing per key metadata, (2) a flat file structure on disk to store KV pairs, in a set of *Logical bands* (LBands), (3) an optional LSM tree to store the unique set of keys, and (4) other system-level metadata structure. All of the above data structures are stored on disk, but the HT is generally cached in the host-DRAM. These are further described in the following subsections.

A. Data Structures and Layout

1) Hash Table (HT): Each key is hashed using a *primary hash function* that maps a given key to an index into the hash table (HT). For each hash index, the HT stores the metadata (described further below) of up to two keys that map to this index. If more keys map to this hash index, the second HT entry becomes a pointer to a common overflow area where the metadata of all the keys mapping to this index is maintained in the form of a linked list. We henceforth call each HT entry as a "node" (as shown in Figure 1), in which the first two nodes are in the HT itself, and the rest are in the overflow area.

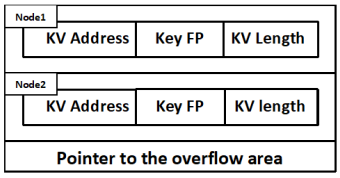

Fig. 1. A Hash Table entry

The HT stores metadata per KV pair that contains the address of the KV pair on the disk ("KV-addres") constitutes of LBand_seq_number and offset within that LBand, the length of the KV pair ("KV-length"), and a secondary hash value or Key Fingerprint ("Key-FP") to reduce disk lookup on collision as shown in Figure 1. We do not store the actual key in the HT entry. So when an input key is mapped to the HT entry, in the best case, a single lookup suffices to determine the potential value location on disk. In the worst-case, all the nodes in the corresponding linked list need to be read from the storage media, to locate the (potentially) matching record. However, by choosing a suitable (non-cryptographic) hash function and HT size, the overhead of hashing can be kept very low [31]–[33]. In our experiments, we observe that with 36B HT entry size and a HT size equal to twice the maximum database size, the collision probabilities are under 4%.

As the database grows, the hash space is filled in with the incoming KV pairs. If the HT load factor (*No-of-keys-stored / capacity*) exceeds 90%, we allocate a larger HT and migrate all the entries from the old to the new HT. With suitable sizing of the database, such a move should be rare and can be done during periods of very low activity, although a live-migration type of approach can be used to further reduce the impact. The space for the old HT could then be reclaimed.

The HT is kept on the SSD, and cached in host DRAM for performance. FlashKey permits write-back caching of the HT in memory which allows the user to control the cache size. Zero DRAM consumption for HT places the entire table on disk - minimizing DRAM cost, but requiring additional IO for HT accesses. At the other extreme, the entire HT may be kept in DRAM, improving performance at the expense of DRAM cost and startup/recovery times. With suitable logging mechanisms (discussed later), the unflushed dirty portions of HT can be recovered from its disk version and LBands in case of power loss. In our experiments, we use both in-memory and on-disk HT and compare the impact of the two bookend implementations of HT in sections V-B6 and VI-B7. Depending on the requirement, the user can select an approach that balances these tradeoffs.

2) Logical Band (LBand): The storage media inside most NAND flash-based SSDs is organized into multiple physical sections, which is the basic unit for defragmentation. We extend this idea in FlashKey to store key-value data, rather than disk-sector data. Accordingly, we divide the entire logical storage space into multiple logical bands (LBands) [34].

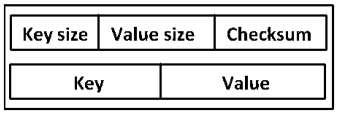

Fig. 2. An LBand entry

Each LBand is designed to hold roughly a thousand KV pairs by default, and this can be reconfigured as needed. Each LBand stores LBand-metadata, multiple LBand entries. Each LBand is in one of three states: empty, open, or closed. A linked list for empty bands called "EmptyLBandList" is used whenever FlashKey needs to open a new LBand to write new (or defragged) KV pairs. The selected band is moved from the empty state to open state. When an open band is filled up, it is moved to a closed state. The "ClosedLBandList" is used to identify bands that are filled and closed. At any given time there is only one LBand which can be in the "Open" state, into which the system writes the new/compacted KV pairs.

a) LBand Metadata: Each open or closed LBand starts with band-metadata, which includes the sequence number of the LBand. The sequence number is used during system recovery. Each closed band includes metadata at the end of the band, which specifies the band's state (closed).

b) LBand Entry: Each entry in the LBand as shown in the Figure 2, stores the actual data for the KV pair. Each LBand entry also includes a 16 bytes header that has additional

information about this KV pair, that specifies the length of the key and the value, and checksum for error detection purposes. This metadata associated with each entry allows FlashKey to support variable size key-value pairs.

3) LSM Tree: Hashing intentionally destroys the locality and hence the key ordering information by randomly spreading the keys across the entire hash space. This makes Range-Scan(.) operations problematic. We allow the user to configure FlashKey to support Range-Scan(.). To support scan, FlashKey stores the set of unique keys in a separate LSM tree structure. Each time a new insert is requested, we insert only the key into the LSM tree, without any value [35]. No subsequent (value) update for that key goes to the LSM tree. Hence the space required by the LSM tree is very low (2.7% for both 50GB and 100GB database) compared to the database size. In case of deletion, we delete the key from the LSM tree as well. During the range queries, we retrieve an iterator with the set of sorted keys within the given range from the LSM tree and iterate over it to get the values using the HT and LBand.

4) Other System Level Metadata Structures: FlashKey also maintains other system-level metadata structures including LBand invalidity table, a linked list for empty and closed bands, and statistical information. This information is kept in the host main memory during run time, and flushed to storage device periodically and when the FlashKey is closed. It is loaded/rebuilt during startup.

The LBand invalidity table records how much data is invalid in each LBand. We use this table to select which LBand should be selected for Garbage Collection/Compaction. The number of entries in this table equals the total number of LBand(s) in the system. Each entry contains the current amount of invalid bytes in the corresponding LBand. The table is updated after each PUT/DELETE operation. More specifically, if an existing key is being overwritten or deleted, the LBand index and the length of the stale KV pair can be found from the HT. Then, the corresponding invalidity table entry is selected, and the length of the stale KV pair is added into that entry. Storing the KV length in both HT entry along with the KV header at LBand, helps to reduce the number of disk reads as the KV is deleted or updated at the cost of a little higher storage space.

The statistical information includes the currently available free space, the amount of data that has been: (a) written to and read from the device, (b) written and read by the host, etc.

B. Algorithms

In this section, we describe the basic methods of FlashKey.

1) Data Retrieval using Get(.): During the Get operation, FlashKey computes the HT entry index of the input key. Then it loads the HT entry from the storage device to the host main memory if required. It then computes a secondary hash based on the key and tries to match it to the key fingerprint stored in the node. If the secondary hash matches as well, then it uses the LBand index and offset stored in that node to read the KV pair from the LBand. Next FlashKey compares the key read from the LBand with the key from the input and returns the value in case the two keys match and the value is not a delete token which signifies that the key has already been deleted. Otherwise, it searches through all the nodes at the HT index until a match is found or the nodeList ends, in which case it returns not found to the host.

2) Data Insertion using Put(.): FlashKey tracks the location to be written in the currently open LBand, namely LBand_Write_Position. During Put(.) operation, FlashKey checks if space is available to put the incoming KV pair in the currently open LBand. If space is insufficient, then it closes the current LBand and opens a new clean band. It triggers garbage collection to reclaim storage space by throwing away the invalid KV pairs in case the available clean bands are low in number ($<$ predefined Threshold). Once space is available, FlashKey writes the KV pair to LBand_Write_Position.

After the KV pair is written, FlashKey updates the corresponding hash-table entry. The logic it follows is similar to finding a key match in the Get operation. If no match is found, then the algorithm allocates a new node at the corresponding HT index entry and treat this as a new insert. The new node might be first or second, or any node in the overflow linked list. For any new insert, FlashKey also creates an entry with just the key in the LSM tree. On the other hand, in case a match is found, FlashKey updates the HT entry of the existing KV pair with the recent location of KV in the LBand.

3) Data deletion using Delete(.): The delete operation is similar to the Put operation, except that instead of writing any value with the key, FlashKey creates a new entry where it drops a "DELETE token" for the deleted key at the current open LBand Write Position. For the entry created in the LBand, it also updates the corresponding HT entry. The deleted (marked) key is eventually discarded during Garbage Collection. However, FlashKey removes the deleted (invalid) key from the LSM tree during the delete operation, so that the LSM tree only contains the valid keys.

4) Range Queries using Scan(.): Given the starting key and the range for the scan operation, FlashKey gets the set of valid keys within that range from the LSM tree, iterates over those keys, and calls the Get(.) method to retrieve their values.

5) Garbage Collection : Garbage collection (GC) is triggered when the number of empty LBands falls below a certain threshold. During GC, FlashKey selects the source LBand with the most invalid KV pairs using the invalidity table. For the source LBand, it reads each KV and determines whether it is a valid key by matching its location with that pointed to by the HT. It then relocates the valid KV pairs to the current open LBand and updates the address of the KV pair at the corresponding HT node. For any valid KV pair, if the value at the LBand contains a delete token, FlashKey removes the entry for the KV pair from the LBand and the HT. When the relocation completes, FlashKey marks the source LBand as "empty" and pushes it to the empty band list. For the sake of simplicity we use single GC thread, however, the performance can be enhanced further by having multiple of them, in which case all the GC threads and the foreground threads write to

the HT and the open logical band. This requires locking the LBand entry at the corresponding HT index and the LBand-offset increment. Due to the small granularity of the lock, the lock has no major impact on the performance.

6) Clean Shutdown, Startup and Power Loss Recovery: During a clean (proper/not power-loss) database shutdown, all the dirty data in memory is written back to the storage device which includes (1) HT dirty entries, (2) LBand metadata for the open LBand, in part to identify where we left-off in the band, (3) LBand status table including the LBand invalidity information, (4) statistics, etc. A flush(.) command ensures that the data is non-volatile before the database shuts down.

During restart from a clean shutdown, FlashKey first loads the LBand status table to build the empty/closed LBand list. Then it reopens the last opened LBand and restores the write pointer to the appropriate offset. Next, it loads the open LBand metadata and journal. Finally, FlashKey loads the invalidity table and the statistics and gets ready for new host commands.

Restart/recovery from a dirty/improper shutdown depends on the system configuration. As stated earlier, the HT can be optionally cached in DRAM. A user can choose the right configuration to balance the DRAM consumption, the required number of disk reads and writes, and the recovery time. Configuration selection also impacts the startup/recovery time after a system crash or power failure. To assist recovery, FlashKey also periodically writes its state-information to the storage device, including (1) dirty HT writes, (2) LBand invalidity table, (3) LBand Status table and (4) statistical information. Our current implementation does not cover all corner cases that can cause inconsistencies under the power loss situation. However, based on where the HT and all metadata is placed, we can identify the following three cases relevant for power loss recovery.

On disk: In this configuration, both shutdowns and restarts happen very fast. Even under sudden power loss, as everything is backed up on disk, minimal recovery is required for proper cleanup, potentially of the last KV and a few metadata structures.

Fully in memory: This requires saving and loading the data structures during shutdown and restart respectively. Since we always persist the data on disk, thus under sudden power loss condition, we must playback all non-empty LBand(s) starting from the lowest sequence number in ascending order to recover the key data-structures. For each LBand entry in a non-empty LBand, we compute the hash, populate the HT entry, and update the associated metadata structures.

On disk and write-back cached in memory: We generally flush the dirty portions of the HT and other structures as we close an LBand as it gets filled. So to recover from the power loss, we restore the last saved KVS state, and then replay only the current open LBand. However, to reduce the flush frequency, FlashKey can be configured to flush on every N^{th} LBand that is closed; in that case, we playback the last N LBand(s). The above configuration provides control over the time to recover from power loss conditions by adjusting the flushing frequency. In the extreme case we may never flush and incur the maximum recovery time as described above.

VI. Evaluation and Analysis

This section presents the experimental and analytic comparison of FlashKey to both LevelDB and RocksDB. In particular, section VI-A details the system and workload configuration, and section VI-B presents the experimental results. Section VI-C provides an analytical model and an associated comparison of key performance metrics.

A. Experimental Setup

Benchmarks: In our evaluation, we use the popular benchmark called db_bench and compare it with our implementation in FlashKey. Even though both the benchmarking tools that come with LevelDB and RocksDB are named "db_bench", they use different key generation streams. Therefore, it is required to test against them separately, henceforth called *(db_bench_L)* and *(db_bench_R)* respectively. db_bench_L only works with LevelDB and FlashKey. db_bench_R only works with RocksDB and FlashKey. These evaluations illustrate the primary performance characteristic differences between FlashKey and the baseline KVSes.

TABLE II
DB_BENCH_R & DB_BENCH_L WORKLOADS

Workload 1	fillsequential	loads 50M KV
Workload 2	readrandom	50M random reads
Workload 3	overwrite	100M random writes
Workload 4	readwhilewriting(1:1)	1-R, 1-W, 50M reads/thread
Workload 5	readwhilewriting(3:1)	3-R, 1-W, 50M reads/thread

In section VI-B6, we use the Yahoo! Cloud Server Benchmark (YCSB) [36] for a system level evaluation. We use mapkeeper bindings to couple YCSB with FlashKey and LevelDB systems, and rocksjava binding with RocksDB. YCSB workload configurations are shown in Table III.

TABLE III
YCSB WORKLOAD CONFIGURATION

W	Ops Ratio	Operations
A	50:50	Read/Update
B	95:5	Read/Update
C	100	Read
D	95:5	Read/Insert
E	95:5	RangeScan/Insert
F	50:50	Read/Rd-Mod-Wr

KVS and System Configuration: To reduce variables that introduce significant variance, we disabled system and KVS features that can be equivalently added to all databases and systems. In particular, we turned off application based data caching, while leaving operating system level caching (page cache) in place. Also, except for a few cases, the tests are run using a single thread (multi-threading is generally turned off). The resulting configuration parameter information for RocksDB is shown in Table IV, and LevelDB configuration follows suite. The experiments were performed on an Intel7 pro NVMe SSD installed on a server running CentOS-7 with 8 Intel Core$^{(TM)}$ i7-7700 CPU executing at a rate of 3.6 GHz, and a 32 GB memory.

FlashKey Configuration: In FlashKey we use a simple modular hash function to generate index to HT, and a second 64-bit Jenkins [33] to generate the key fingerprints (the second

hash that is included in the node). It is possible to use any low-overhead hash function that provides good collision performance. Hash-table size is set to 100M entries, which is 2x the number of KV pairs in the database. More sophisticated mechanisms, such as Cuckoo hashing, may allow for a smaller hash space, but this is not explored here. Our experimental results use hash-table in memory unless otherwise stated.

TABLE IV
ROCKSDB CONFIGURATION

Parameters	Value
All params except those below	default
block_restart_interval	1
write_buffer_size	128M
compression_type	None
level0_file_num_compaction_trigger	2
cache_size	0

Other tools and parameters: A significant part of our evaluation focuses on WA and RA metrics. To calculate these metrics, we use the nvme-cli *https://github.com/linux-nvme/nvme-cli* tool to read the SSD counters to measure the IO issued to the disk. Combined with workload information, the WA and RA metrics are then calculated as defined in section II.

B. Experimental Results and Discussion

For our experiments with db_bench_L and db_bench_R, we use workload 1 of Table II to fill the database with KV pairs and run different benchmarks as stated in Table II above to measure the performance. We chose key size of 16B and value size of 1024B. Note that although KV's allow for rather large key-sizes, small key sizes are more efficient and popularly used in real applications [17].

1) Space Amplification Comparison: Space amplification (SA) significantly impacts KVS performance, WA, RA, and cost. An ideal comparison of KVS systems would evaluate these metrics with SA kept constant. Unfortunately, this is not practical with the baseline KVS solutions in the study as the SA for them varies widely within a run. We can only capture the final SA as the workload stops and there is no information provided for the intermediate SA (maximum SA) at any time within a run. In our comparison, we align SA of FlashKey to the final reported SA of the baseline KVS. We explain below (1) how the SA of the baseline KV system changes, (2) what is the impact of the change on WA, and (3) how FlashKey is different from them.

During database-load, all KVS systems use little to no overprovisioning, with space amplification between 1.02 and 1.12, as the KV pairs are loaded sequentially in the database.

Next, under intensive update traffic, SA increases due to out-of-place updates. As SA increases, LevelDB and RocksDB try to bound the space utilization through repetitive compactions. The frequent compaction hurts WA(↑), RA(↑), latency(↑), and throughput(↓). Both KVSes acquire more space to defer compaction as per their policies to limit the impact on performance and WA; this increases their SA during run-time to some unspecified (and not-reported) amount. LevelDB and RocksDB only report the database size (space-utilization) on disk for the KVS at the end of the test-runs, which is a snapshot post compaction. This results in significant under-reporting of their actual space utilization and hence calculation of the SA. In contrast, FlashKey allows the database to grow to only a fixed amount of additional LBand(s), and triggers GC as needed to keep the space-utilization bounded. In our experiments, we configure FlashKey space-utilization (and hence SA) to match the final-calculated SA of the baseline KVS per experiment. This is an extremely conservative comparison, therefore, against both LevelDB and RocksDB. We also note that since FlashKey tightly controls the space amplification at all times, it can operate with a limited amount of storage space and low cost, whereas RocksDB exits with an error if they are unable to secure the necessary amount of space during the runs.

2) Write Amplification Comparison: The graphs in Figure 3 compare the improvement in write amplification (WA) with FlashKey over RocksDB and LevelDB when running under the db_bench_R and db_bench_L benchmarks respectively with their SA aligned. While comparing WA, we only consider workloads with writes, i.e., workloads 3, 4 and 5.

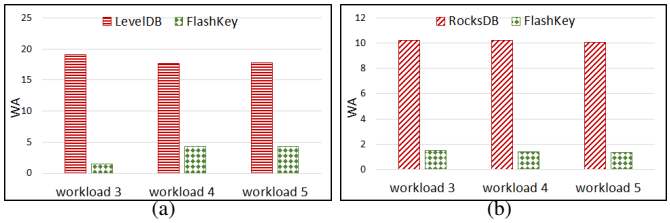

Fig. 3. Comparison of WA of FlashKey against LevelDB and RocksDB using db_bench_L & db_bench_R respectively.

Figure 3(a) shows the WA improvements of FlashKey over LevelDB, and Figure 3(b) shows the same in comparison to RocksDB. In comparison to LevelDB at similar SA, FlashKey achieves WA improvement in the range of 4-12x on these workloads, with a maximum of 12x gain for workload 3. In comparison to RocksDB, FlashKey exhibits WA improvement of ∼7x, which remains consistent across all the workloads. The WA improvement that FlashKey achieves over RocksDB is less than the gain achieved in comparison to LevelDB, as RocksDB incurs comparatively lower WA than LevelDB at the cost of higher SA than that of the LevelDB.

As stated earlier, both LevelDB and RocksDB allow for greater SA during the run than the ending value that we equalized for a conservative comparison. RocksDB is more lax in controlling SA than LevelDB, which allows it to do a bit better in WA. Even so, FlashKey achieves far better WA as compared to RocksDB.

3) Read Amplification Comparison: The graphs in Figure 4 show the change in RA achieved by using FlashKey against the other KV Stores. We compare the RA for workloads with random read requests, i.e., workloads 2, 4, and 5.

When tested against RocksDB, for all the workloads, FlashKey performs 1.5-2X better in terms of RA. Compared to LevelDB, on workload 2 and 5, FlashKey improves the read-amplification by 1.2X and 1.4X respectively. However, for the workload 4, (i.e. 1:1 read-to-write ratio) the RA with FlashKey

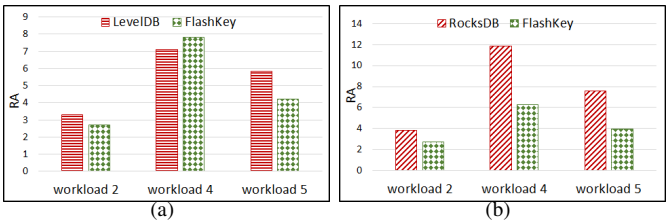

Fig. 4. Comparison of RA of FlashKey against LevelDB and RocksDB using db_bench_L & db_bench_R respectively.

is somewhat greater (10%) than that of the LevelDB. Note that these two workloads (4 and 5) represent "read while writing" situation, and thus the read performance entirely depends on how eagerly the writes are done. LevelDB did only 1/2 as much writes as FlashKey during the entire read period thereby eking out better read performance than FlashKey. This is because of the strict compaction (with very high WA 18X) to keep the SA (1.24X) low, which helps random reads but at the cost of fewer writes, which itself saves on compaction overhead. The impact of fewer writes can be seen on the Average delay graphs of FlashKey over LevelDB discussed in the following section. Note that, we still observe some improvement in RA with workload 5 as multithreading hides the impact of successive access to the same key.

4) Average Latency Comparison: Figure 5 compares the average latency for the workloads (2-5) at an IO queue-depth of 1. For all the workloads, FlashKey obtains improvements in average latency over both RocksDB and LevelDB. For LevelDB, the improvement is lower ($\sim 7\%$) for the read-write mix workloads (4 and 5) for reasons mentioned above. When tested with db_bench_L, FlashKey achieves a maximum of 84% improvement with workload 3. The average latency for LevelDB is very high due to frequent compaction which also broadens the scope for having a higher latency gain when tested with the write-intensive workload. The latency improvement of FlashKey is due to its efficient O(1) architecture and algorithms that address the shortcomings of LevelDB.

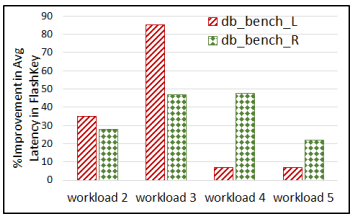

Fig. 5. Comparison of Average Latency of FlashKey against LevelDB and RocksDB using db_bench_L & db_bench_R respectively

Overall, when compared to RocksDB, FlashKey achieves improvement in the range 22-50% in latency, with the highest improvement under workload 3. The gain over RockDB is lower as RocksDB improves latencies with reduced compaction at the cost of higher SA (SA in RocksDB was 1.7X compared to 1.24X of LevelDB). For workloads 4 and 5, FlashKey attains an average latency improvement over the RocksDB in the range (22-48)%.

5) Read QoS Comparison: Since FlashKey is designed to support fast Get requests on SSDs (via randomized hash-table lookups rather than LSM tree lookups), it shines on read tail latency (RTL) introduced earlier. Figure 6 illustrates RTL of FlashKey compared to both (a) LevelDB and (b) RocksDB. Lower numbers are better on the graph. Especially on "5 9s", aka 99.999P RTL, FlashKey is 2x better than both the baselines. Table V summarizes the RTL improvements.

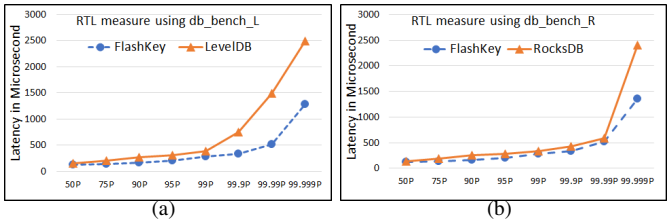

Fig. 6. Comparison of Random Read QoS of FlashKey against (a) LevelDB and (b) RocksDB with workload 2 of db_bench_L & db_bench_R respectively.

6) Throughput Comparison: In this section, we compare the throughput of FlashKey, LevelDB, and RocksDB under the YCSB system benchmark, with configuration as detailed in section VI-A. For throughput analysis, we used the YCSB benchmark as its traffic generation is based on real cloud workloads. In contrast, db_bench traffic is rather artificial. For our experiments, we used a 50 GB database in which the keys are 23 bytes and values are 1000 bytes long. To keep the database size consistent, as recommended by the developers [36], we execute the workloads of Table III in the following order: (1) Load 50M records in the database with workload A; (2) Run workload A, B, C, F, and D in order for 50M operations each; (3) Drop the database - i.e. completely delete it; (4) Load 50M records with workload E; (5) Run workload E for 1M operations (each scan cost 100 reads on average). Except for the record and the operation counts, all the other YCSB workload parameters are kept unchanged from their defaults.

TABLE V
PERCENTAGE IMPROVEMENT IN RANDOM READ QOS

Metric	LDB	RDB
RTL-50P	18.5%	10%
RTL-99.999P	48.43%	43.7%

YCSB runs atop a selected KVS. In our runs, we kept the KVS configurations unchanged from those detailed earlier for db_bench_L and db_bench_R. We previously discussed the difficulty of alignment of the SA of FlashKey with the SA of the baseline KV systems, and it is even more difficult to achieve this alignment with YCSB as some of the workloads dynamically insert new records, which increases the size of the database. We use a constant SA of 1.23x for FlashKey.

Figure 7 shows the performance in terms of *Throughput* of FlashKey compared to RocksDB and LevelDB when tested with YCSB workloads. For all the workloads except workload E, FlashKey's performance exceeds that of both RocksDB and LevelDB. Since our design is optimized for point insertions and lookups, and not for range queries, workload E (range-scan based workload) is

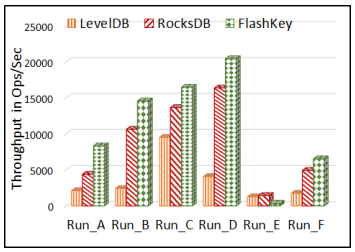

Fig. 7. Throughput Comparison using YCSB

as-expected slower when run on FlashKey. Range scan in FlashKey essentially does random reads and LSM traversal, plus HT lookups. Hence, for the workload E which constitutes of majorly range scan, FlashKey is not as performant as either RocksDB or LevelDB due to not keeping the data in order. However, for the rest of the cases, we achieve better performance even without keeping the data in order.

Among the other workloads, while testing with workload A which is a 50:50 intermix of reads and updates, FlashKey achieves the maximum gain up to 90% against RocksDB over the others. In case of workload C which is purely read traffic, RocksDB comes closest to FlashKey, but is still 17% below the throughput of FlashKey.

7) Effect of moving the entire HT from memory to disk: As explained in section V-A, the HT in FlashKey can be (1) entirely contained in the Host-DRAM, (2) on disk and write-back cached into the DRAM, or (3) on disk. Depending upon which location we choose for the HT, there are tradeoffs in DRAM cost, recovery time, access latency, WA, RA, etc. Here we compare the bookend configurations (1) and (3). Table VI lists the average latency in microseconds (denoted as AL), WA and RA measurements for random read, random write, and read-write mix workloads (workloads 2, 3 and 5 respectively of Table II). We conclude from the measurements that FlashKey is significantly more performant than RocksDB even when the hash-table is entirely on disk, with significantly better WA, latencies, and RA, especially on typical mixed read-write workloads. While a comparison against LevelDB is not shown in Table VI, note that LevelDB is slower than RocksDB, and our gains against LevelDB are even better.

TABLE VI
FLASHKEY - IN MEMORY VS ON DISK HT COMPARISON

Workload	Metric	RDB	Disk	Mem
2	AL	125	133	90
	RA	3.5	4.6	2.7
3	AL	346	250	183
	WA	10.2	3.8	1.5
5	AL	671	597	537
	WA	10.1	4.9	1.3
	RA	7.6	3.7	4

C. Algorithmic Analysis of FlashKey

We define, the key size:K, value size:V, the maximum number of KV pairs in the system:N and the SSD capacity:C. FlashKey consumes a small amount of SSD capacity for HT and other system level information. The system level information is less than 1 MiB and is ignored in the following discussion. The HT size is $36N$ (bytes) since each HT entry contains 36 bytes. As we use a hash function and hash-table size that largely avoids collisions, in most of the cases, we find our desired data with a high probability (95%+ in all our experiments) within the first two nodes.

The remaining SSD capacity is divided into multiple LBands where each LBand contains multiple LBand entries. Each entry contains a 16B LBand entry header and the actual KV pair. There is also a header associated with each LBand called LBand header (32B/LBand) which is different from the LBand Entry header. For simplicity, we skip this header for now. At any given time, FlashKey does not use all the LBands' storage capacity for current KV pairs. Instead, a fraction of this capacity, henceforth denoted as OP, is reserved as over-provisioning for efficient compaction.

Now, among K, V, N, C, and OP, we have the following relationship:

$$C = (1 + OP)((16 + K + V)N + 36N).$$

If the HT size is much smaller than SSD, we have $N = \frac{C}{(1+OP)(16+K+V)}$. The SA can be calculated as follows:

$$SA = \frac{(1+OP)((16+K+V)N+36N))}{N(K+V)}.$$

With low collision probability, for the Get(.) operation (both sequential and random), FlashKey reads the HT entry first, and then loads the KV pair from the corresponding LBand entry; hence the RA (not accounting for GC yet) is: $RA = \frac{16+K+V+36}{K+V}$=$1 + \frac{52}{K+V}$. Note that the RA (and also WA) are defined here in terms of bytes read (or written) rather than the IOs performed.

For the Put(.) operation, FlashKey writes the KV pair to the current open LBand first and then updates the HT entry. For the sequential put operations (with no valid data requiring relocation during GC), the WA is; $WA = \frac{16+K+V+36}{K+V}$=$1 + \frac{52}{K+V}$. Whereas for random Put(.) operations, there will be additional writes during the data relocation in GC and the corresponding WA is;

$$WA = \Big(1 + \tfrac{52}{K+V}\Big) X \Big(\tfrac{1+OP}{1+OP+LW(z)}\Big),$$

where LW() is the Lambert W function [37], and $z = -(1 + OP)X(e^{-(1+OP)})$.

For example, let us assume $C = 4$ TB, average key size $K = 20$ B, and average value size $V = 800$ B, and the minimum OP is 18%. Under this configuration, the maximum number of keys will be $N = \frac{4TB}{1.18(16+20+800)}$= 4.05 Billion. Since we need additional space for HT and other metadata, we set the maximum number of keys to 4 Billion. The HT size is set to $4X10^9X36X2 = 288$ GB, and the SA will be $SA = \frac{4TB}{4X10^9X(20+800)B}$=1.22.

RA under sequential and random Get(.) will become $1 + \frac{52}{820} = 1.1$, and WA under sequential Put(.) is also 1.1. The WA under the random Put(.) will be $1.1X3.46 = 3.81$, where 3.46 is the WA under 18% OP. These WA, SA, and RA measurements assume to have all the elements of FlashKey stored on the disk (worst case). A mixed random read and write workload introduces GC due to the random writes/updates. The GC requires extra reads and writes to the disk, and results in higher RA and WA. Please note that the above analysis provides only an estimate based on mathematical modeling; it does not incorporate the impact of sector-aligned I/O. This is different from the disk reads and writes which are issued in practice. As such, the mathematical analysis provides a lower bound, and the experimental results included above provide a better understanding of true behavior on real workloads.

VII. Conclusions and Future Work

In this paper, we presented a new KVS system for SSDs called *FlashKey* that is based on hashing of keys, logical-banding of KV pairs, combined with an empty-value LSM-tree. We showed both experimental and analytical evaluation of FlashKey vs. popular LevelDB and RocksDB KV systems and demonstrated O(1) lookup of Get(.) requests while maintaining low RA and reducing WA by up to 12x. Average access and tail latencies are improved by up to 85% and 2x respectively. These improvements persist as database size increases, e.g., with 100M KV pairs and SA aligned at 1.57X, compared to RocksDB, FlashKey achieves 20-60% lower latency, $\sim 9X$ lower WA, and 1.5-2X lower RA for db_bench_R.

While FlashKey performs well compared to both RocksDB and LevelDB, there is scope for significant improvements. To accelerate the garbage collection process and power loss recovery, we plan to include a data structure called LBand Journal per LBand. The band journal can be used to record the hashes and locations of the table entries, i.e., hash index and HT node information of all the key-value pairs stored in that respective LBand. It may be extended to include key-prefixes for the keys stored in the LBand. Also, as mentioned in section V-B6, we plan to explore the power loss situations in detail including the potential for inconsistencies and data loss, and how those could be minimized. We expect that a well designed LBand journal can significantly enhance the robustness of FlashKey against power loss episodes.

Additionally, the memory needs of the hash table can be reduced significantly by hashing on a key range (by ignoring least significant few bits of the key) rather than on individual keys. The number of bits to ignore can be chosen to ensure that the space taken by the HT is no more than some specified fraction of the database size. Such a mechanism also helps with the range searches and will be the most useful for databases with small key-value sizes. The HT can also be cached partly in the memory based on a prediction of the most active regions of the key space. Finally, more sophisticated hashing mechanisms can be employed, e.g., Cuckoo hashing to increase HT space utilization, and locality sensitive hashing (LSH) to enhance sequential access performance. In the future, persistent memory can also be exploited to keep the HT.

References

[1] X. Hu *et al.*, "Lama: Optimized locality-aware memory allocation for key-value cache," in *Proc. of USENIX ATC*, 2015.

[2] K. Wu, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Towards an unwritten contract of intel optane ssd," in *USENIX HotStorage*, 2019.

[3] A. Eisenman *et al.*, "Reducing dram footprint with nvm in facebook," in *Proc. of ACM EuroSys*, 2018.

[4] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Redesigning lsms for nonvolatile memory with novelsm," in *Proc. of USENIX ATC*, 2018.

[5] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y.-r. Choi, "Slm-db: Single-level key-value store with persistent memory," in *Proc. of USENIX FAST*, 2019.

[6] T. Yao, J. Wan, P. Huang, X. He, F. Wu, and C. Xie, "Building efficient key-value stores via a lightweight compaction tree," *ACM TOS*, 2017.

[7] T. Yao *et al.*, "Geardb: A gc-free key-value store on hm-smr drives with gear compaction," in *Proc. of USENIX FAST*, 2019.

[8] R. Pitchumani, J. Hughes, and E. L. Miller, "Smrdb: key-value data store for shingled magnetic recording disks," in *ACM SYSTOR*, 2015.

[9] P. Wang *et al.*, "An efficient design and implementation of lsm-tree based key-value store on open-channel ssd," in *Proc. of ACM Eurosys*, 2014.

[10] J. Zhang, Y. Lu, J. Shu, and X. Qin, "Flashkv: Accelerating kv performance with open-channel ssds," *ACM TECS*, 2017.

[11] Z. Shen, F. Chen, Y. Jia, and Z. Shao, "Didacache: A deep integration of device and application for flash based key-value caching," in *USENIX FAST*, 2017.

[12] M. Bjørling, J. González, and P. Bonnet, "Lightnvm: The linux open-channel SSD subsystem," in *USENIX FAST*, 2017.

[13] Y. Jin, H. Tseng, Y. Papakonstantinou, and S. Swanson, "Kaml: A flexible, high-performance key-value ssd," in *IEEE HPCA*, 2017.

[14] J. Li, Z. Chen, Z. Chen, N. Xiao, F. Liu, and W. Chen, "Kv-ftl: A novel key-value-based ftl scheme for large scale ssds," in *Proc. of IEEE (HPCC/SmartCity/DSS)*, 2017.

[15] L. Marmol, S. Sundararaman, N. Talagala, and R. Rangaswami, "NVMKV: a scalable, lightweight, ftl-aware key-value store," in *Proc. of USENIX ATC*, 2015.

[16] S. Lee, M. Liu, S. Jun, S. Xu, J. Kim *et al.*, "Application-managed flash," in *Proc. of USENIX FAST*, 2016.

[17] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: Separating keys from values in ssd-conscious storage," in *Proc of USENIX FAST)*, 2016.

[18] H. H. W. Chan, Y. Li, P. P. C. Lee, and Y. Xu, "Hashkv: Enabling efficient updates in KV storage via hashing," in *USENIX ATC*, 2018.

[19] B. Debnath, S. Sengupta, and J. Li, "Flashstore: high throughput persistent key-value store," *Proc. of the VLDB*, 2010.

[20] ——, "Skimpystash: Ram space skimpy key-value store on flash-based storage," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, 2011, pp. 25–36.

[21] K. Kourtis, N. Ioannou, and I. Koltsidas, "Reaping the performance of fast NVM storage with udepot," in *Proc. of USENIX FAST*, 2019.

[22] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "Silt: A memory-efficient, high-performance key-value store," in *ACM SOSP*, 2011.

[23] K. Ren, Q. Zheng, J. Arulraj, and G. Gibson, "Slimdb: A space-efficient key-value storage engine for semi-sorted data," *Proc. of VLDB*, 2017.

[24] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees," in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, October 2017.

[25] Y. Li, C. Tian, F. Guo, C. Li, and Y. Xu, "Elasticbf: elastic bloom filter with hotness awareness for boosting read performance in large key-value stores," in *2019 USENIX Annual Technical Conference*, 2019, pp. 739–752.

[26] A. Devulapalli *et al.*, "Data structure consistency using atomic operations in storage devices," in *IEEE SNAPI*, 2008.

[27] M. Minglani *et al.*, "Kinetic action: Performance analysis of integrated key-value storage devices vs. leveldb servers," in *IEEE ICPADS*, 2017.

[28] R. Sears and R. Ramakrishnan, "blsm: A general purpose log structured merge tree," in *Proc of ACM SIGMOD*, 2012.

[29] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, 1996.

[30] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum, "Optimizing space amplification in rocksdb," in *CIDR*, 2017.

[31] C. Estébanez, Y. Saez, G. Recio, and P. Isasi, "Performance of the most common non-cryptographic hash functions," *Software: Practice and Experience*, 2014.

[32] C. Henke, C. Schmoll, and T. Zseby, "Empirical evaluation of hash functions for multipoint measurements," *ACM SIGCOMM Comput. Commun. Rev.*, 2008.

[33] R. J. Jenkins, "Hash functions for hash table lookup," ONLINE, 1995-1997. [Online]. Available: http://burtleburtle.net/bob/hash/evahash.html

[34] P. Li and S. N. Trika, "Techniques to manage key-value storage at a memory or storage device," Mar. 29 2018, uS Patent App. 15/279,279.

[35] S. N. Trika, D. Park, P. Li, F. R. Corrado, and R. A. Dickinson, "Data management system employing a hash-based and tree-based key-value data structure," Jan. 31 2019, uS Patent App. 15/856,686.

[36] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proc. ACM SoCC*, 2010.

[37] P. Desnoyers, "Analytic modeling of ssd write performance," in *Proceedings of the ACM SYSTOR*, 2012.