KVRAID: High Performance, Write Efficient, Update Friendly Erasure Coding Scheme for KV-SSDs

Mian Qin, A. L. Narasimha Reddy, Paul V. Gratz
Texas A&M University
{celery1124,reddy}@tamu.edu,pgratz@gratz1.com

Rekha Pitchumani, Yang Seok Ki
Samsung Semiconductors, Inc.
{r.pitchumani,yangseok.ki}@samsung.com

ABSTRACT

Key-value (KV) stores have been widely deployed in a variety of scale-out enterprise applications such as online retail, big data analytics, social networks, etc. Key-Value SSDs (KVSSDs) provide a key-value interface directly from the device aiming at lowering software overhead and reducing I/O amplification for such applications.

In this paper, we present KVRAID, a high performance, write efficient erasure coding management scheme on emerging key-value SSDs. The core innovation of KVRAID is to use logical to physical key conversion to efficiently pack similar size KV objects and dynamically manage the membership of erasure coding groups. Such design enables packing multiple user objects to a single physical object to reduce the object amplification compared to prior works. By applying out-of-place update technique, KVRAID can significantly reduce the I/O amplification compared to the state-of-art designs. Our experiments show that KVRAID outperforms state-of-art software KV-store with block RAID by 28x in terms of insert throughput and reduces CPU utilization, tail latency and write amplification significantly. Compared to state-of-art KV devices erasure coding management, KVRAID reduces object amplification by $\sim 2.6x$ compared to StripeFinder and reduces I/O amplification by $\sim 9.6x$ when compared to KVMD and StripeFinder for update intensive workloads.

CCS CONCEPTS

• **Hardware** → **Emerging interfaces**; **Emerging architectures**; • **Computer systems organization** → **Redundancy**; **Reliability**.

SYSTOR '21, June 14–16, 2021, Haifa, Israel
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8398-1/21/06…$15.00
https://doi.org/10.1145/3456727.3463781

Table 1: ***Object amplification comparison for insert under D data and P codes/parities configurations with 16B key and 1KB value. KVRAID-P demonstrates packing two logical objects into a physical object.***

(D,P)	REP	KVMD [28]	SF [22]	KVRAID	KVRAID-P
(4, 2)	3	4.5	1.8	1.5	0.75
(8, 3)	4	5.38	1.78	1.38	0.69

KEYWORDS

Key-value store, Key-value SSD, Redundancy, RAID

1 INTRODUCTION

Persistent Key-Value (KV) stores are an essential component in many large-scale applications [3, 7, 18]. Modern persistent key-value store engines are built on top of block devices (hard drives [5, 24] or flash-based SSDs [6, 8, 19]), resulting in high CPU utilization and I/O write amplification factors (WAF). This high CPU utilization comes from multiple software stack layers required to translate from key space to device block space including key-value data management, file system, block I/O layer, device driver, etc. [16]. Another problem of modern key-value stores are high I/O write amplification [20, 30]. In order to leverage the performance characteristics of HDDs and SSDs, state-of-art key-value stores use the Log Structure Merge (LSM) Tree [5, 8, 25] as a fundamental data structure to manage key-value objects. Although Log Structure Merge Trees significantly reduce the WAF compared to B-Trees [25], they still suffer from relatively high WAF (~10-40x) due to background compaction [20, 23, 30].

Recently, both academia [11, 14, 31] and industry [16] have proposed key-value interfaced devices to replace the conventional software-based key-value engines built on block interface devices. Key-value SSDs simplify the software stack for key-value store applications, reducing their overall CPU/memory usage [16]. Further, by consolidating redundant software indirections, KVSSDs also reduce the overall write amplification [16]. The reduction of CPU overhead and WAF can greatly benefit today's cloud environments [1, 33].

Table 2: ***Overall I/O amplification comparison for update (same configuration as Table 1). KVMD and SF require read metadata object and read-modify-write data/code object operations for updates. For KVRAID, small ϵ (near 0) relies on better garbage collection, as discussed in Section 4.2.***

(D,P)	REP	KVMD [28]	SF [22]	KVRAID	KVRAID-P
(4, 2)	3	5	5	$1.5+\epsilon$	$0.75+\epsilon$
(8, 3)	4	7	7	$1.38+\epsilon$	$0.69+\epsilon$

The introduction of key-value interfaced [16] devices naturally raises the question of how we manage redundancy [9, 27, 32, 34] and maintain performance [15, 17] on multiple KV storage devices. KVMD [28] proposed different redundancy schemes based on the value size of a KV pair. KVMD employed replication for small value sizes (less than 128B), and for large value sizes (16KB or above), employed splitting of the record across multiple devices with associated erasure codes [10, 29, 35] to improve storage efficiency. The data and code blocks of the erasure codes can share the same user key and spread across different devices. However, For value sizes in the middle (128B to 16KB), both mirroring and splitting introduce unacceptable overhead, such as storage inefficiency (mirroring), read/write performance degradation (splitting). KVMD [28] proposed stateless packing mechanism to group multiple objects into an erasure codes/parity group (or stripe). However, to maintain the membership of each user object to the stripe, KVMD needs to create a small metadata object for each user object to keep track of the keys for other user objects in the same stripe. In order to ensure fault tolerance of metadata, KVMD further replicates each metadata object which introduces not only byte level write amplification but also object level write amplification. Table1 shows KVMD requires ~30-50% more objects than replication (REP) for insert.

To address the metadata amplification issue for the KVMD packing mechanism, StripeFinder [22] proposed an efficient metadata membership tracking method to reduce the metadata size as well as the number of metadata objects required. The main idea is to use a circular chain to keep track of the stripe membership, and group multiple metadata objects into a single object through hashing on the user key to reduce the number of metadata objects as shown in 1.

However, both KVMD and StripeFinder introduce complexity for updating an object in a stripe. KVMD employs **in-place** update mechanism, which means it needs to read back the code blocks in the stripe and update multiple code blocks as well as the user object, which introduces significant read/write I/O amplification. The update throughput of KVMD is an order of magnitude slower compared to put and get. To make matters worse, if the update object's value size changes, it may cause an unbalanced stripe that leads to extra performance and storage overhead. StripeFinder doesn't discuss the update scenario. However, in theory, it encounters a similar issue as KVMD. Even if KVMD and StripeFinder employ an **out-of-place** update mechanism by creating a new stripe for the updated object along with the new incoming objects, there will be significant read/write amplification for updating the associated metadata objects to update the stripe membership information.

In this paper, we focus on records of small to medium size(128B to 4KB) which can greatly benefit from grouping multiple objects into one erasure code group. We explore an alternative way to keep track of the membership information for each stripe by translating the user/logical keys to physical/device keys. The physical keys are designed as 64-bit monotonically increasing numbers which enable easy identification of a stripe. Given a stripe, it is easy to compute the physical keys on rebuild. We leverage well-established in-storage data structure LSM-Tree [5, 25] to map the logical keys to physical keys. The LSM-Tree introduces acceptable write overhead and amplification since it allows converting a logical key to physical key (as a KV pair) to a large I/O (large value size in KV devices), which also significantly reduces the number of objects required for metadata compared to KVMD and StripeFinder. With compression which is well established in existing LSM-Tree implementation such as LevelDB [5], RocksDB [8], the metadata write overhead can be further reduced.

The introduction of logical keys to physical keys mapping enables packing multiple objects into a single physical object within an erasure code group. Recent studies on KV-SSD show noticeable read/write performance scale-down (~15-20%) as the number of objects managed in the KV-SSD grows. The reduction of number of objects on the devices through our design can help maintain performance as the number of user objects scales. Table 1 shows that KVRAID-P (packing two logical objects to a physical object) can reduce object amplification by more than 2x for insert compared to StripeFinder (SF) and by more than 4x for update compared to KVMD and StripeFinder.

This paper makes the following contributions:

- Presents a design employing erasure coding on a group of small-medium variable size records for fault tolerant storage over key-value storage devices.
- Proposes several novel ideas for organizing such a system, among them: slab allocation within the KV store, logical to physical key mapping and maintaining multiple states of data within the system.
- Evaluates the proposed system on real KVSSD devices to show that the proposed solution can improve

performance while reducing CPU load overheads for data protection, and improving lifetime of the system through reduction of write operations.

2 BACKGROUND AND MOTIVATION

KVRAID's focus is building an erasure code management mechanism for KVSSDs. In this section, we first provide a brief overview of the KVSSD devices. Then we introduce the related work on managing erasure codes across multiple small objects.

2.1 Key-value SSDs

The idea of key-value interface devices has been proposed by both academia [14, 31] and industry [16]. In Figure 1 we show the results of experiments we conducted on Samsung KV-PM983 KVSSDs to evaluate performance characteristics for different value sizes from 128B to 2MB. For all the experiments, we use 64 threads to issue the requests to the device. The experimental setup details are described in Section 5.

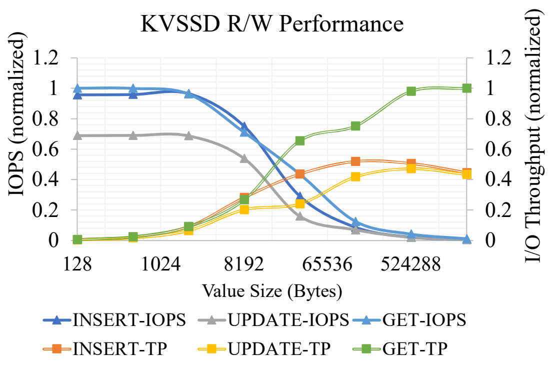

Figure 1: *KVSSD performance characteristics under different value size.*

There are two interesting performance characteristics of KVSSDs as shown in the figure.

(1) The IOPS performance for both insert and get remain nearly the same for value sizes between 128B to 4KB. (Slightly lower from 2KB to 4KB)
(2) The update (overwrite the existing keys) IOPS performance is noticeably lower (~30%) than insert operations.

The steady IOPS performance for insert and get for smaller records implies that by packing multiple KV objects in a single I/O, we can increase the overall device bandwidth while not sacrificing performance. Further, we can leverage the performance benefit of insert operations (in comparison to updates) by our logical to physical translation technique (by always inserting new physical keys for update operations). Section 3 will elaborate more details of how we leverage these KVSSD performance characteristics in our design.

Another important characteristic of the KVSSDs is the read/write performance scales down as the number of objects in the devices grows. We observed that read/write performance diminishes by upto 30% when the number objects in the device exceeds 500 million. Similar observations were made in recent work [11]. This characteristic motivates us to reduce the number of objects inside the device in our design.

2.2 Managing multiple objects for erasure coding over KV devices

In this section, we briefly introduce how the state-of-art works KVMD [28] and StripeFinder [22] manage multiple objects in an erasure code group (also referred as a stripe). KVMD and StripeFinder both group multiple user objects into a single stripe and write user data objects and code objects (calculated from the user data objects through erasure coding) across different devices. Figure 2 illustrates how KVMD and StripeFinder use metadata objects to keep track of the membership information within a stripe.

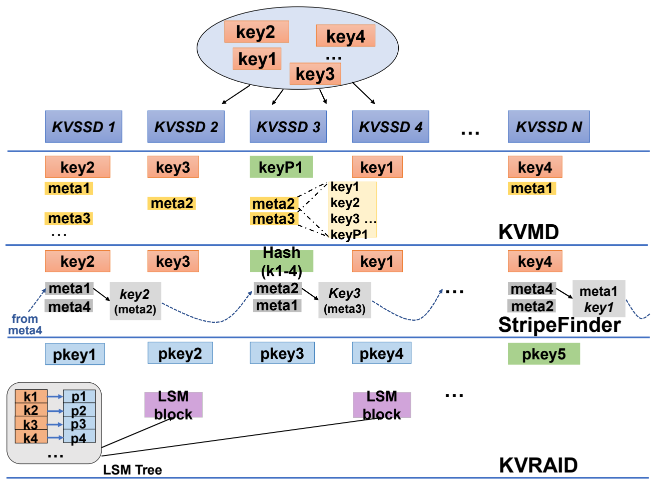

Figure 2: *Comparison of KVMD StripeFinder and KVRAID on how to manage multiple objects in a single erasure coding stripe.*

In KVMD, every user object is associated with a metadata object (replicated on multiple devices for fault tolerance). The metadata object key is assigned based on the user key. (For example if the user key is key1, the associated metadata object key is key1:1). The value of the metadata object stores all the user keys and code keys within the same stripe. When rebuild happens, every surviving user key can retrieve all the data/code objects in the stripe through the information in the metadata object.

StripeFinder takes a step further to reduce the metadata overhead of KVMD. Instead of storing all the data/code objects keys within a stripe in the metadata object, StripeFinder creates a finder object (metadata object) that only stores the adjacent data object keys within the stripe as shown in Figure 2 and forms a ring chaining the data object keys within a stripe. During update and rebuild, each user object can walk the ring to retrieve all the data objects keys within a stripe. The code object keys are generated through all data object keys through a hash function. Code key can be easily retrieved after retrieving all the data object keys. To further optimize, StripeFinder groups multiple finder objects together through hashing (User keys hashed to the same bucket use the same finder object) to reduce the object amplification [22].

As shown in Tables 1 and 2, while KVMD and StripeFinder improve storage efficiencies compared to mirroring, their performance, especially update performance needs further improvement. Unlike KVMD and StripeFinder, our KVRAID design chooses a different route by translating user/logical keys to physical keys and uses physical keys to keep track of erasure code group information. In the following sections, we will elaborate how the key translation idea helps reduce the object amplification and improve the update performance compared to the state-of-art works.

3 DESIGN OVERVIEW

The difference between managing erasure coding/RAID on block devices exposing block address (BA) interface and key-value interface devices has been discussed in prior works [22, 28]. Our KVRAID design focuses on small to medium size objects (128B to 4KB). We propose to translate logical keys to physical keys and efficiently manage erasure code group membership information through physical keys. Such a design enables the possibility of packing multiple logical objects into a single physical object to maintain high write throughput and reduce the number of objects managed by the device.

3.1 Parity group formation with packing

In this section, we describe how KVRAID leverages the logical keys to physical keys translation to apply erasure coding on multiple logical objects and keep track of membership information.

Our design aims to work across a wide range of object sizes. Our approach is inspired by a typical O/S memory allocation policy, the "slab allocator", together with an enhanced packing approach to achieve storage efficiency while reducing object amplification and I/O complexity. Figure 3 demonstrates how our design works. To handle the variable size objects in key-value applications, we pre-define multiple fixed size *slabs* according to the object size distribution of the applications. We assume we know the maximum size of the object as the max slab size. Then, we group the objects in two dimensions. First, based on the observation that KVSSD has

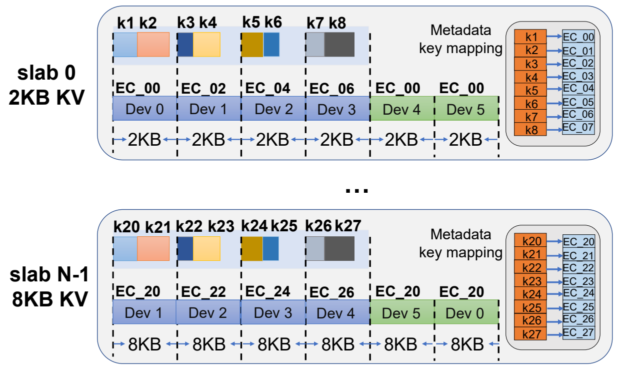

Figure 3: ***Packing KV objects with variable size* slabs *for parity group formation. The physical keys from key mappings can be used to locate the physical value and the offset for the packed user value.***

similar IOPS performance across value size from 128B to 4KB as shown in Section 2.1, we accumulate and pack multiple similar size objects to form a data chunk for erasure codes. Second, we group multiple of those data chunks to form a parity group and apply erasure coding on these chunks.

When applying erasure parity on multiple objects, which can be of variable size, a single object may be cut off into multiple code chunks causing fragmentation issues. Besides, packing may incur rewriting overhead when an object is updated with a different size, requiring a read back and repacking of the whole parity group. In order to reduce the fragmentation of objects across devices, we align each data chunk to an erasure code chunk size while tightly packing objects within a data chunk as shown in Figure 3 (since we need to read/write them in one shot anyway). In summary, our approach has the following advantages:

- **Device Bandwidth Utilization Efficiency** Since we pack multiple objects in to a single data chunk, this has the effect of increasing overall I/O size to the device, which can increase the device bandwidth utilization due to our observation in Figure 1.
- **Storage Efficiency** Due to the multiple slab sizes design, we can reduce the storage overhead for the parity chunks/objects. **Note: we don't need to pad zeros to the data objects since the KV device can handle variable size I/O.**
  The slab sizes determine the total storage of the parities objects and the computation overhead for performing encoding and decoding. (Consider there is only one slab, the slab size/parity object size needs to be the largest record size in the dataset).
- **Less Object and I/O Amplification** By packing multiple logical objects into one physical object (data objects in the parity group), we effectively reduce the

number of objects managed in each KV device compared to KVMD and StripeFinder. We will further optimize *insert* and *update* operations to reduce the I/O amplification in the following sections.

3.2 Batch writes

Now we consider how new parity groups are formed for a small write case. For example, for the EC[4,2] configuration, when there is a new *insert* request, we first assign a unique key to this new request. This physical key is a combination of the slab id and unique sequence number. The sequence number will determine the parity group information (device key and device id for the erasure code chunk) for this request. Then we calculate the 2 erasure code chunks in memory (pack zeros for the first data chunk to align code size, i.e. slab size and zero filling data chunks for other 3 data chunks in the group). Finally, we write the first data chunk (we don't need to write the padding zeros to device since the device can accept arbitrary size values) and the two erasure code chunks to three separate devices. When another request arrives in the same slab, we need to take the following steps, read the old erasure code chunks, pack the new object with the previous object, re-calculate the new erasure codes, write the new data chunk and update the two erasure code chunks in the devices. This continues with more write requests until the parity group is full.

In our design, we use batch writes or *big writes* to address the I/O overhead. We accumulate the data in memory until sufficient number of objects arrive to form a full parity group before we issue the I/Os for data/code chunks and commit all the requests. In this case, forming each parity group costs 6 writes in total. This *batch writes* technique is widely used in storage systems [3, 18]. In order to limit the impact on I/O latency, we bound the time for forming a full parity group, at which time available objects are written as a partial parity group to the devices. Application writes are not returned as completed until all the writes have been sent to the devices, thus providing reliability for completed writes.

3.3 Lazy deletion

Consider an update operation. This record is already part of an existing parity group. In order to update this record, we have to update both the data chunk it belongs to and the two code chunks (parities) as well in case of EC[4, 2]. This again converts one write into multiple read and write I/Os (data chunk and corresponding code chunks).

In order to avoid these problems, we always update objects *out of place*. We treat an update as a deletion of the old object and an insert of the new object (into a new parity group). To facilitate this approach, an object can have three states on the device: *Valid-alive*, *Invalid-alive* and *Empty*. A

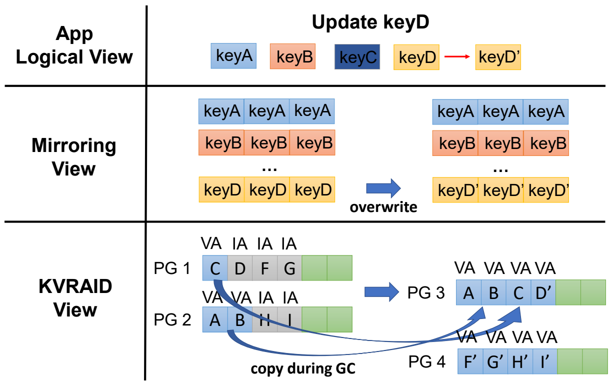

Figure 4: *Lazy deletion demonstration. VA stands for Valid-alive records and IA stands for Invalid-alive records.*

valid-alive object is a valid record. An Invalid-alive object is only maintained on the device to maintain the parity group consistent and for recovery reasons. An empty record (within an empty parity group) will be deleted from device periodically. With such out-of-place updates, we can combine insert and update operations together into a batch write operation with appropriate metadata updates. The invalid-alive objects are garbage collected later in order to create empty groups. This lazy deletion approach allows parity groups to accumulate more deletions over time to make garbage collection more efficient. Figure 4 illustrates an example of how our lazy deletion approach works. The accompanying write reduction benefit will be demonstrated in Section 4.2 quantitatively.

4 IMPLEMENTATION

KVRAID was implemented from scratch in C/C++ as a user space library with simple key-value interface. It supports different redundancy levels and different erasure coding schemes. In this section, we focus on the how to reduce the metadata (logical keys to physical keys mapping) overhead and garbage collection related details and deliver a more complete picture of our KVRAID design.

4.1 Metadata management

The key idea of KVRAID requires dynamic management of a translation table of user-side logical keys to device-side physical keys. By classifying the objects by value length, KVRAID groups similar size objects into a fixed-size slab and forms a parity group according to write request order, then applies erasure code on it. Figure 5 demonstrates the core structures for metadata management. When an application write request (*insert* or *update*) comes, KVRAID determines which slab the object will fall in by object size. That slab will assign a unique physical key for accumulated request objects and apply packing and erasure coding on them. The logical (user) key and physical (device) key mapping will be filled into the mapping table. Finally, the packed data chunks and code (parity) chunks for each parity group will be written to separate devices as KV objects.

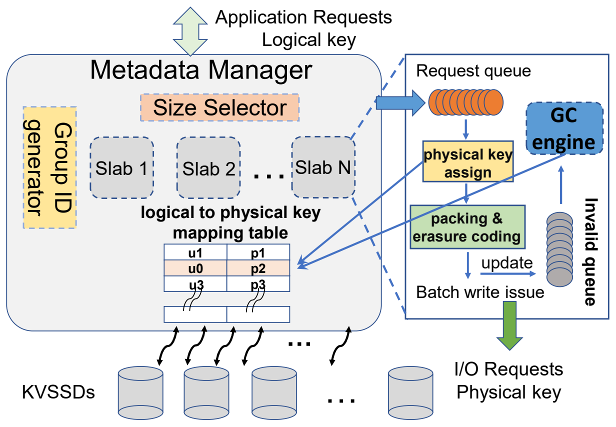

Figure 5: *KVRAID metadata management data structure layout.*

Slab. The slab contains a *request queue* which accumulates the incoming write requests and dequeues requests in a batch to form a parity group. We use a global monotonically increasing unique group id number for each parity group after packing and erasure coding of the packed objects. The code and parity objects (physical objects being written to device) will use the unique group id number and slab id to construct the physical key. Figure 6 shows an example of how physical key and value are constructed before writing to the devices. The 56bit sequence number is calculated as $group_{id} \times k + offset \times p$, where k is the number of data objects in an erasure code group, p is the number of records packed into a single erasure code object and offset is the relative position of the erasure code objects.

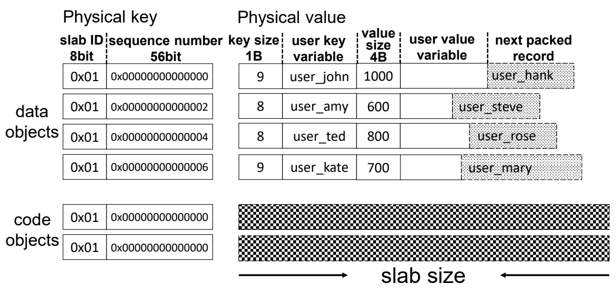

Figure 6: *Physical key/value format.*

The monotonically increasing physical keys can also help determine the device membership for each physical object (physical objects are assigned to devices with round-robin algorithm like RAID6 [27].) Besides, monotonically increasing keys also provide a versioning effect which will help us for crash recovery (see Section 4.3). For the physical value, we don't need to actually write the padding zeros used in erasure coding to device (referred as virtual zeros padding in KVMD [28]) since we are aware of the slab size (erasure code object size) and the user value size (embedded in the physical value). We also embed the user key into the physical value for future garbage collection and crash recovery (see Section 4.2, 4.3).

For slab size choice, ideally we can use prior knowledge on the dataset value size distribution to find the optimal slab sizes to minimize the erasure coding computation and storage overhead as mentioned in 3.1. Our initial evaluations have shown that the results are not highly sensitive to the slab sizes. (We use four slabs in our evaluation)

Mapping table. The mapping table maps each user logical key to a physical key to get the erasure code data object and the internal packed position (offset) of the record. As shown in Figure 6, the physical key of user_john and user_hank will be 0x0100000000000000 and 0x0100000000000001 respectively. By accessing the mapping table, we can extract the slab id, physical key of the erasure code data object and the packed position within the data object to retrieve the user record.

In our design, we examine two possible approaches for persisting metadata. First, we propose to employ persistent memory (like Intel Optane [12]) to store the metadata. In this case, we can leverage in-memory data structure like hashtable and balanced trees to store the metadata to achieve minimal metadata overhead for insert, update and read.

Second, when NVM is unavailable, we propose to use the in-storage data structure LSM-Tree [26] to store the metadata externally on KVSSDs with redundancy. The LSM-Tree structure converts small writes (logical key to physical key pair) to larger writes (typically 16KB block size for LSM-Tree implementation). Such a design significantly reduces the metadata write overhead and number of metadata objects required in the device compared to KVMD and StripFinder which require metadata objects proportion to the total number of logical objects managed in the system. For example, for an EC[4,2] configuration with 100 million 16B-key objects, KVMD requires 300 million metadata objects and StripeFinder(C=10) requires 30 million metadata objects, while KVRAID only requires 0.22 million metadata objects(16KB block size for LSM-Tree without compression).

In our implementation we port LevelDB [5] to KVSSD storage. The levelDB on-disk structures (such as manifest, sstables blocks, etc.) are stored as key value pairs in KVSSD. To provide redundancy to the mapping table, we split the large SSTable blocks (64KB) to multiple objects with erasure coding and store each object to a separate KVSSD. Other levelDB related files (stored as KV pairs) are replicated. We also store a magical KV record (with replicas) to indicate the existence of in-storage mapping table.

Invalid queue. To facilitate the lazy deletion, each slab also maintains an *invalid queue* from the *update* and *delete* requests for further garbage collection. To enable efficient garbage collection, the invalid queue is implemented as a hash-table with the parity group id as the key and the value is the group offsets for all invalid-alive records in that group. Thus, the garbage collection thread can linearly scan this structure and identify the near empty group to perform further reclamation. In our KVRAID implementation, we did not persist this *invalid queue* structure. On KVRAID closing, our GC engine will enforce cleaning all the partially full parity groups and moving the valid objects in them to new parity group to compact space utilization (with the assumption that the KVRAID is not frequently opened and closed). Also, we can gracefully recover from crash during this final "compaction" by leveraging the monotonically increasing sequence number of the physical keys. (see Section 4.3).

4.2 Garbage collection

Garbage collection (GC) is a critical component in our KVRAID design to trade-off I/O amplification and storage efficiency. In our current implementation, the garbage collection is triggered by a storage utilization threshold. A separate thread does garbage collection for each slab in the background periodically. Figure 7 demonstrates how garbage collection works in each epoch. As mentioned in Section 4.1, we maintain the *invalid-alive* objects in a hash-table structure.

The GC thread first scans the invalid queue and finds the candidate reclaim groups which are above the threshold of minimum *invalid-alive* entries. Then it will read the *valid-alive* records in the reclaim groups from devices, extract the user keys and insert them into the slab request queue.

After the *valid-alive* objects in the reclaimed group is committed in a new parity group, the GC thread will update the mapping table with the new metadata. Finally, we can delete the data and parity objects in the reclaimed groups from devices to release device space.

Take an example of the EC[4, 1] configuration, consider 12 objects are updated in four parity groups as shown in Figure 7. In the ideal case, update without GC kick-in would cost 1.25 writes per updated-object under the *batch writes* and *lazy deletion* criteria. Reclaiming the 4 parity groups will cost 1.25 writes per *valid-alive* object (4 in total). The average write amplification for those 4 parity groups that

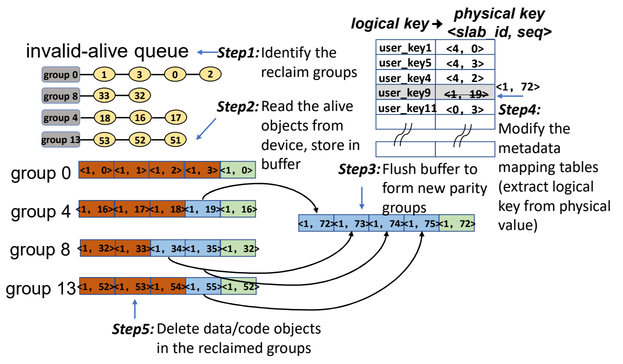

Figure 7: *Garbage collection implementation.*

were updated becomes $\frac{12\times1.25+4\times1.25}{12} = 1.67$[1] which is still smaller than the replication cost (2 with the same redundancy level). To be more general, the average write amplification for KVRAID under GC is as follows.

$$WAF = \alpha \times \frac{k+r}{k} + \frac{1-\alpha}{R_{invalid}} \times \frac{k+r}{k} \tag{1}$$

In equation 1, k is the number of data chunks and r is the number of code chunks in erasure code, α is the fraction of groups that are not updated on the device. $R_{invalid}$ is the average ratio of the *invalid-alive* objects per parity group ($\frac{12}{16}$ in the previous example) before GC. In our implementation, we can tune the $R_{invalid}$ parameter for garbage collection aggressiveness to trade off between storage efficiency and write amplification.

4.3 Recovery

Device failure. The erasure code determines the number of device failures that can be tolerated. For example EC[4, 2] can tolerate at most two simultaneous device failures. By the design of the construction of the physical keys, we can extract all physical keys (data and parity chunks) of each parity group from any logical key from the group by indexing the logical to physical key mapping table. On a device failure, we can then issue concurrent read I/Os to retrieve the surviving data/parity objects in the parity group and decoding over the erasure group to recover the objects on the failed devices. For the in-storage metadata implementation, the metadata itself is also protected by erasure code, which can also be rebuilt.

Host failure. Due to maintaining the logical to physical key mapping table, host side failure (either hardware or software) may corrupt the mapping table. Another source of corruption from host failure is the "compaction" process for the *invalid-alive* queue when closing KVRAID. We may

[1]For in-storage metadata, extra I/O is required for key mapping lookup. However, a small cache on the relatively small key mappings can significantly reduce the amortized I/O for metadata access.

terminate before the "compaction" process is completed or lose the *invalid-alive* queue data since it is volatile. For the mapping table corruption, we can rebuild the mapping table purely from device since we embed the logical (user) key in the physical value. The recovery process will retrieve all the physical key value objects on all KVSSDs, identify the data objects (from the physical key) and extract the logical keys inside and rebuild the mapping table. For the *invalid-alive* queue corruption, the monotonically increasing sequence number implicitly conveys the version information. By comparing all the physical keys' sequence numbers with the same logical key, we can distinguish the *invalid-alive* objects and the *valid-alive* one (with the largest sequence number).

5 EVALUATION

In this section, we evaluate KVRAID system performance on real KVSSD hardware and compare with state-of-art software KV stacks (RocksDB) on block SSDs with RAID and state-of-art erasure coding management for KV devices.

5.1 Experimental setup

Hardware and configurations: In our evaluation, we use a real system (Intel Xeon Gold 6152 platform with 256GB DRAM) with six Samsung KVSSDs devices (PM983). We evaluate our KVRAID with an EC[4,2] erasure coding configuration (4 data objects and 2 code objects).

For RocksDB (on block SSDs) configuration, we disabled compression to make a fair comparison with KVRIAD (current KVSSD firmware doesn't support compression). We set up 2GB block cache and use direct I/O for flush and compaction which is a common industry setup [2]. For KVRAID, we employ 4 uniformly distributed slabs based on value size range (100B to 4000B). We implemented a host side LRU cache for a fair comparison with RocksDB's block cache.

Comparison schemes: In our evaluation, we compared seven different schemes to thoroughly investigate our KVRAID design. RocksDB-raid10 and RocksDB-raid6 are running on six block SSDs with software RAID. Other 5 schemes are running on KVSSDs. The block SSDs and KVSSDs share the same SSD hardware but with different firmware. All schemes have the same, device-failure tolerance level (tolerate 2 simultaneous device failures).

I **Rocksdb-raid10:** RocksDB on block devices with Linux nested RAID10 configuration (two raid1 devices each with 3 SSDs, i.e. mirroring, and then apply raid0 on the two raid1 devices, i.e. striping).

II **Mirroring:** Two replicas in different KVSSD devices which is equivalent to **Rocksdb-raid10** in case of storage efficiency.

III **Rocksdb-raid6:** RocksDB on block devices with Linux RAID6 configuration.

IV **Small write IS (KVMD):** Update data and corresponding code objects in a parity group in sequence with metadata stored on storage devices (This resembles *KVMD packing* approach for **update** operations which requires read and update code objects in a parity group. For **insert**, KVMD packing is similar to Batch writes IS. We mainly use this scheme to model update operations of KVMD and StripeFinder.).

V **Batch writes IS:** KVRAID which employs batch writes and lazy deletion techniques with metadata stored on storage devices.

VI **Batch writes IM:** Same as IV except for metadata stored in memory,

VII **Batch writes pack IM:** KVRAID which employs packing 2 objects into an erasure code chunk along with batch writes with in-memory metadata.

Workload: We use the Yahoo! Cloud Serving Benchmark (YCSB) [4] to generate our workloads with 64 client threads(for RocksDB experiments, we use 16 client threads to load the data to mitigate contention, for update/get queries we use 64 client threads. We evaluate with a dataset with a uniform distribution of values over the range of 100B to 4000B. The distribution of request operations is zipfian. To cover a full spectrum of real workloads, we evaluate our systems with YCSB default workloads and different write/read (update/get) ratios, including (90%:10%), (70%:30%), (50%:50%), (30%:70%) and (10%:90%).

5.2 Experimental results

We ran the YCSB benchmark on real KVSSD and SSD devices, for each of our experiments, we first load 200 million records with variable value sizes (around 400GB of total data). The key size is ~25 bytes. Then we perform another 200 million operations with different update/get ratios on the dataset (all keys exist). For KVRAID, we used a soft capacity of 320GB for each KVSSD device to make sure GC kicks in during the run phase. For RocksDB experiments, we perform an extra warm-up phase to finish compactions in the load phase. The raw data written in each experiment varies from ~0.66TB to ~2.3TB. Before each experiment, we format the devices to reset the internal device state.

5.2.1 Throughput performance. Figure 8 (a), (b) show the overall throughput performance and CPU utilization (through Linux time utility) for different redundancy schemes (for rocksdb, we collected overall performance due to its irregular compactions, for KVRAID schemes, we collect the steady performance by omitting small period of start and end of each run). In data load phase, RocksDB with software RAID6 performs significantly worse (~28x in pure insert

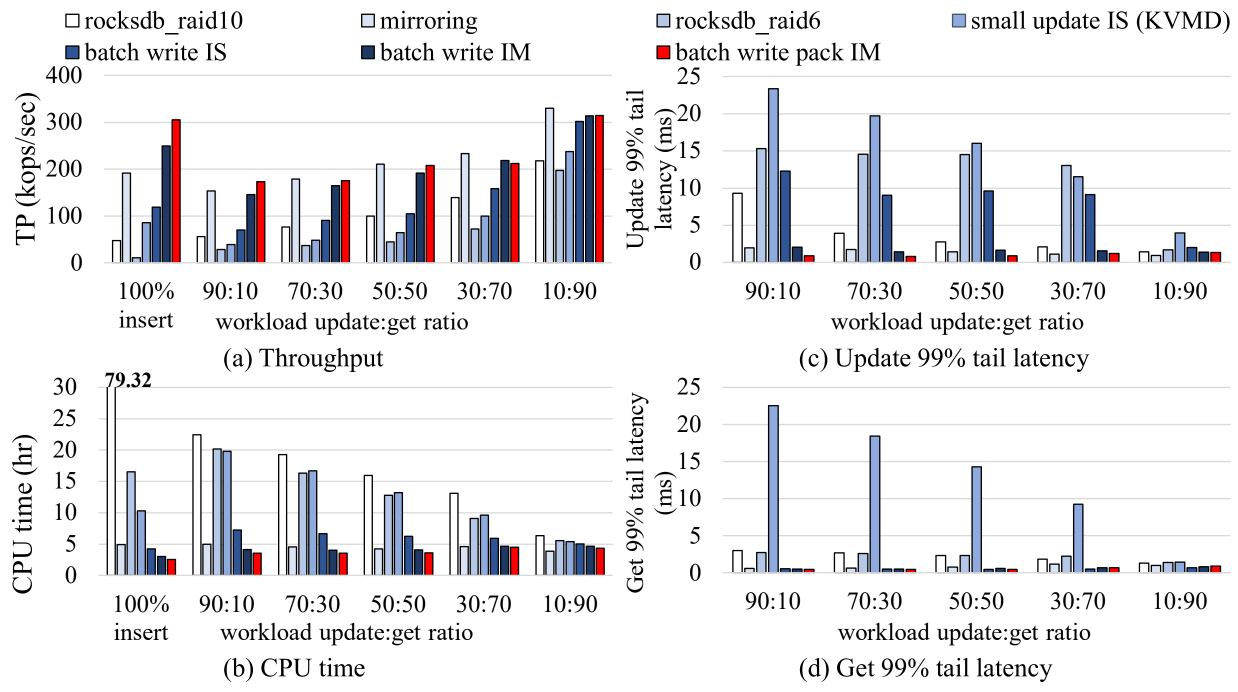

Figure 8: *Performance and CPU utilization for different redundancy schemes under YCSB workloads.*

workloads) with much higher CPU utilization (~31x) compared to our best KVRAID implementation (VII) on KVSSD. In mixed update/get workloads, KVRAID (VII) still outperforms RocksDB with software RAID by ~4x and reduces CPU utilization on an average by 4.1x. This demonstrates the performance advantage of our KVRAID design compared to the traditional block RAID on software KV stores. The saved CPU cycles can benefit other jobs, which is critical in cloud environments [1]. Besides, KVRAID (VII) outperforms KVMD packing (IV) by 3.7x and reduces CPU utilization by 4.6x for update intensive workloads because of batch writes and lazy deletion techniques. (KVMD needs to read-modify-write the code objects for updates).

Within the KVRAID implementations, *batch writes* with in-memory mapping table implementation (VI) achieves similar performance compared to mirroring. Packing 2 objects into an erasure code chunk (VII) can outperform mirroring by 59% for 100% insert case (loading data) and sustain performance for heavy update workloads. For read heavy workloads, batch writes with packing performs slightly worse compared to non-packing due to the latency cost of waiting for more records to group into a parity group.

5.2.2 Tail latency. Figure 8 (c), (d) show the 99% tail latency for update/get queries. Compared to KVRAID, RocksDB incurs higher update tail latency for write-heavy workloads due to compaction. Even with in-storage metadata implementation (V), KVRAID reduces the update tail latency by 33% compared to RocksDB in RAID6 (III). With packing and in-memory metadata (VII), KVRAID significantly reduces the update tail latency (more than 10x) compared to RocksDB in RAID6 (III). Batching the writes and packing incurs higher update tail latency for workloads with less write traffic. However, thanks to our adjustable backoff timer, the increase of update tail latency for 10% update ratio compared to 90% ratio is less than 47%. Compared to KVMD packing (IV), KVRAID(VII) reduces update tail latency by 22.5x for update-intensive workloads since we avoid in-place-update for code objects. For get operations, RocksDB requires multiple read I/Os on different LSMT levels, which leads to worse get tail latency. KVRAID(VII) consistently requires only a single I/O for each get query (for in-storage metadata, extra I/O is required for key mapping lookup. However, due to the key mapping size is relatively small, the amortized I/O for metadata access is considerably small). For batch writes with in-storage metadata (V), KVRAID can still reduce get tail latency by ~4.3x compared to RocksDB in RAID6 (III).

5.2.3 I/O Amplification. KVRAID's **out-of-place** update design through batch writes and lazy deletion enabled by the logical keys to physical keys translation greatly reduces the read/write amplification compared to state-of-art KVMD [28] and StripeFinder [22]. Figure 9 demonstrates the comparison for different redundancy schemes for KVSSDs. Here we only show KVMD results since StripeFinder uses a similar mechanism. Compared to mirroring, KVMD requires more I/Os for an update since it needs to read the old data and code objects to calculate the new erasure code objects. However, KVRAID(V) always forms a new erasure code group for the updated objects which yields much less overall I/O amplification. For 90% update ratio workload, KVRAID(V) yields 1.7x less I/O amplification compared to Mirroring(I) and 5x less compared to KVMD(IV)). For KVMD(IV) the update I/O cost is more than what is shown in Table 2 since the variable length object may change value size during update (changes slab), while Table 2 only considers fixed length objects. By applying packing technique (two logical objects into a physical object), KVRAID(VII) reduces overall I/O amplification by 9.6x compared to KVMD(IV). Packing more logical objects will yield better I/O amplification reduction. In our implementation, we didn't use per object metadata design of KVMD(IV). KVMD/StripeFinder will result in more read I/Os for reading the metadata objects.

For normal get operations, KVMD and StripeFinder can use the user keys to retrieve value from device directly. While KVRAID needs to go through logical key to physical keys translation which may require additional I/Os for in storage metadata implementation. However, the indirection overhead for get can be alleviated by caching of the LSM-Tree mapping table. For in-memory metadata implementation, this indirection overhead is negligible.

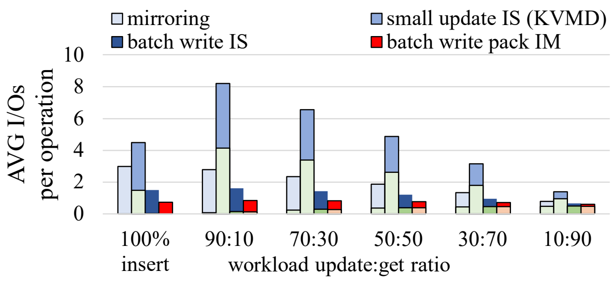

Figure 9: ***Average write/Read I/Os amplification comparison. (Top and bottom half of the stacked bar show the average write and read I/Os respectively.)***

5.2.4 Write Amplification Factor. As the flash technology evolves (from SLC to TLC, or even QLC), device lifetime is becoming a bigger concern [13, 21]. A major advantage of our KVRAID design is reducing write amplification factors (WAF), as a result of erasure coding and our batch writes design. As shown in Figure 10, RocksDB on software RAID6 (III) introduces ~18x WAF in total compared to our best KVRAID implementation (RAID10 is even higher). This is mainly due to the fact that the multiple software layers (LSM tree, block RAID, etc.) are independent and unaware of each other. Our KVRAID design, however, can directly manage erasure codes at KV object level to comprehensively optimize the WAF. Compared to KVMD packing (IV), KVRAID (VII) reduces ~3.8x WAF for update-intensive workloads. KVMD needs to rewrite all code objects in the parity group for every data object update, while KVRAID's batch writes keep the WAF similar to pure insert case. Our batch writes design can also take advantage of the data reduction from erasure coding by delaying the code/parity updates (by retaining data as invalid-alive) which reduces the WAF.

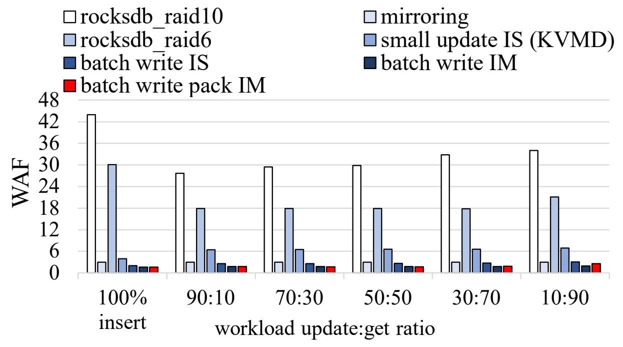

Figure 10: ***Write amplification factors for different redundancy schemes (WAF is measured as total data write to devices from redundancy schemes over the application write data).***

In KVRAID, batch writes with packing (VII) significantly reduce the amount of WAF compared to mirroring (II) (63% reduction across all workloads). This advantage can translate into increased device lifetime.

6 CONCLUSION

This paper proposed, implemented and evaluated a novel design, KVRAID, an erasure coding-based redundancy scheme for KV SSDs. KVRAID extends the idea of "slab allocator" to maintain erasure codes in multiple sizes to handle variable key-value object lengths. KVRAID employs a level of indirection from logical keys to physical keys that allows multiple objects to be packed into a single object on the device. By leveraging *batch writes* and *lazy deletion* with garbage collection, KVRAID can achieve high performance, low write amplification while maintaining the storage efficiency realized from erasure coding. Our measurements show that KVRAID outperforms existing software KV stack and replication schemes on performance, I/O amplification and write amplification, which can provide better energy efficiency and lifetime to KVSSD devices.

7 ACKNOWLEDGEMENT

We thank the anonymous reviewers for their valuable comments. We appreciate Bradley Settlemyer from LANL for his insightful feedback on earlier version of this paper. This research is supported in part by Samsung Corporation and the National Science Foundation, under grant NSF-1823403 and I/UCRC-1439722.

REFERENCES

[1] Dan Ardelean, Amer Diwan, and Chandra Erdman. 2018. Performance Analysis of Cloud Applications. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 405–417. https://www.usenix.org/conference/nsdi18/presentation/ardelean

[2] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 209–223. https://www.usenix.org/conference/fast20/presentation/cao-zhichao

[3] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*. USENIX Association, Seattle, WA. https://www.usenix.org/conference/osdi-06/bigtable-distributed-storage-system-structured-data

[4] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[5] J. Dean and S. Ghemawat. 2017. LevelDB: Google's fast key value store library. *Github release 1.2* (2017).

[6] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2010. FlashStore: High Throughput Persistent Key-value Store. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 1414–1425. https://doi.org/10.14778/1920841.1921015

[7] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. *SIGOPS Oper. Syst. Rev.* 41, 6 (Oct. 2007), 205–220. https://doi.org/10.1145/1323293.1294281

[8] Facebook. 2015. Rocksdb. https://rocksdb.org/.

[9] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. ACM, New York, NY, USA, 29–43. https://doi.org/10.1145/945445.945450

[10] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. 2012. Erasure Coding in Windows Azure Storage. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX, Boston, MA, 15–26. https://www.usenix.org/conference/atc12/technical-sessions/presentation/huang

[11] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. 2020. PinK: High-speed In-storage Key-value Store with Bounded Tails. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 173–187. https://www.usenix.org/conference/atc20/presentation/im

[12] Intel. 2019. Intel Optane DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[13] Jaeyong Jeong, Sangwook Shane Hahn, Sungjin Lee, and Jihong Kim. 2014. Lifetime Improvement of NAND Flash-based Storage Systems Using Dynamic Program and Erase Scaling. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*. USENIX, Santa Clara, CA, 61–74. https://www.usenix.org/conference/fast14/technical-sessions/presentation/jeong

[14] Y. Jin, H. Tseng, Y. Papakonstantinou, and S. Swanson. 2017. KAML: A Flexible, High-Performance Key-Value SSD. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 373–384. https://doi.org/10.1109/HPCA.2017.15

[15] Asim Kadav, Mahesh Balakrishnan, Vijayan Prabhakaran, and Dahlia Malkhi. 2009. Differential RAID: Rethinking RAID for SSD Reliability. In *HotStorage 2009: 1st Workshop on Hot Topics in Storage and File Systems* (hotstorage 2009: 1st workshop on hot topics in storage and file systems ed.). Association for Computing Machinery, Inc. https://www.microsoft.com/en-us/research/publication/differential-raid-rethinking-raid-for-ssd-reliability/ (best paper award.).

[16] Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang-suk Kee, Francisco Londono, Sangyoon Oh, Jongyeol Lee, and Daniel D. G. Lee. 2019. Towards Building a High-performance, Scale-in Key-value Storage System. In *Proceedings of the 12th ACM International Conference on Systems and Storage* (Haifa, Israel) *(SYSTOR '19)*. ACM, New York, NY, USA, 144–154. https://doi.org/10.1145/3319647.3325831

[17] J. Kim, J. Lee, J. Choi, D. Lee, and S. H. Noh. 2013. Improving SSD reliability with RAID via Elastic Striping and Anywhere Parity. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 1–12. https://doi.org/10.1109/DSN.2013.6575359

[18] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 35–40. https://doi.org/10.1145/1773912.1773922

[19] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. 2011. SILT: A Memory-efficient, High-performance Key-value Store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. ACM, New York, NY, USA, 1–13. https://doi.org/10.1145/2043556.2043558

[20] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 133–148. https://www.usenix.org/conference/fast16/technical-sessions/presentation/lu

[21] Youyou Lu, Jiwu Shu, and Weimin Zheng. 2013. Extending the Lifetime of Flash-based Storage through Reducing Write Amplification from File Systems. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*. USENIX, San Jose, CA, 257–270. https://www.usenix.org/conference/fast13/technical-sessions/presentation/lu_youyou

[22] Umesh Maheshwari. 2020. StripeFinder: Erasure Coding of Small Objects Over Key-Value Storage Devices (An Uphill Battle). In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*. USENIX Association. https://www.usenix.org/conference/hotstorage20/presentation/maheshwari

[23] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. 2015. NVMKV: A Scalable, Lightweight, FTL-aware Key-Value Store. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 207–219. https://www.usenix.org/conference/atc15/technical-session/presentation/marmol

[24] Michael A. Olson, Keith Bostic, and Margo Seltzer. 1999. Berkeley DB. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Monterey, California) *(ATEC '99)*. USENIX Association, Berkeley, CA, USA, 43–43. http://dl.acm.org/citation.cfm?id=1268708.1268751

[25] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-structured Merge-tree (LSM-tree). *Acta Inf.* 33, 4 (June 1996), 351–385. https://doi.org/10.1007/s002360050048

[26] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-structured Merge-tree (LSM-tree). *Acta Inf.* 33, 4 (June 1996), 351–385. https://doi.org/10.1007/s002360050048

[27] David A. Patterson, Garth Gibson, and Randy H. Katz. 1988. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '88)*. ACM, New York, NY, USA, 109–116. https://doi.org/10.1145/50202.50214

[28] Rekha Pitchumani and Yang-Suk Kee. 2020. Hybrid Data Reliability for Emerging Key-Value Storage Devices. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 309–322. https://www.usenix.org/conference/fast20/presentation/pitchumani

[29] James S. Plank. 2013. Erasure Codes for Storage Systems. A Brief Primer. 38 (11 2013), 300.

[30] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. ACM, New York, NY, USA, 497–514. https://doi.org/10.1145/3132747.3132765

[31] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An Efficient Design and Implementation of LSM-tree Based Key-value Store on Open-channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands) *(EuroSys '14)*. ACM, New York, NY, USA, Article 16, 14 pages. https://doi.org/10.1145/2592798.2592804

[32] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington) *(OSDI '06)*. USENIX Association, Berkeley, CA, USA, 307–320. http://dl.acm.org/citation.cfm?id=1298455.1298485

[33] Erci Xu, Mai Zheng, Feng Qin, Yikang Xu, and Jiesheng Wu. 2019. Lessons and Actions: What We Learned from 10K SSD-Related Storage System Failures. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 961–976. https://www.usenix.org/conference/atc19/presentation/xu

[34] Heng Zhang, Mingkai Dong, and Haibo Chen. 2016. Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 167–180. https://www.usenix.org/conference/fast16/technical-sessions/presentation/zhang-heng

[35] Zhe Zhang, Andrew Wang, Kai Zheng, Uma Maheswara G, and Vinayakumar. 2018. Introduction to HDFS Erasure Coding in Apache Hadoop. https://blog.cloudera.com/blog/2015/09/introduction-to-hdfs-erasure-coding-in-apache-hadoop.