Hybrid Data Reliability for Emerging Key-Value Storage Devices

Rekha Pitchumani and Yang-suk Kee,
Memory Solutions Lab, Samsung Semiconductor Inc.

https://www.usenix.org/conference/fast20/presentation/pitchumani

This paper is included in the Proceedings of the
18th USENIX Conference on File and
Storage Technologies (FAST '20)
February 25–27, 2020 • Santa Clara, CA, USA
978-1-939133-12-0

Hybrid Data Reliability for Emerging Key-Value Storage Devices

Rekha Pitchumani
Memory Solutions Lab
Samsung Semiconductor Inc.

Yang-suk Kee
Memory Solutions Lab
Samsung Semiconductor Inc.

Abstract

Rapid growth in data storage technologies created the modern data-driven world. Modern workloads and application have influenced the evolution of storage devices from simple block devices to more intelligent object devices. Emerging, next-generation Key-Value (KV) storage devices allow storage and retrieval of variable-length user data directly onto the devices and can be addressed by user-desired variable-length keys. Traditional reliability schemes for multiple block storage devices, such as Redundant Array of Independent Disks (RAID), have been around for a long time and used by most systems with multiple devices.

Now, the question arises as to what an equivalent for such emerging object devices would look like, and how it would compare against the traditional mechanism. In this paper, we present Key-Value Multi-Device (KVMD), a hybrid data reliability manager that employs a variety of reliability techniques with different trade-offs, for key-value devices. We present three reliability techniques suitable for variable length values, and evaluate the hybrid data reliability mechanism employing these techniques using KV SSDs from Samsung. Our evaluation shows that, compared to Linux mdadm-based RAID throughput degradation for block devices, data reliability for KV devices can be achieved at a comparable or lower throughput degradation. In addition, the KV API enables much quicker rebuild and recovery of failed devices, and also allows for both hybrid reliability configuration set automatically based on, say, value sizes, and custom per-object reliability configuration for user data.

1 Introduction

Modern applications require a simpler, fast and flexible storage model than what the traditional relational databases and file systems offer, and key-value stores have emerged as the popular alternative and the backbone of many scalable storage systems [1–3]. To meet the needs of such applications and to simplify the process of storing such user data even further (without added software bloat), modern storage devices have undergone a new key-value face-lift [4–8].

The Samsung Key-Value (KV) SSDs [4, 5] have incorporated the key-value store logic with the NAND flash SSD firmware, and has adopted a key-value user interface, instead of the traditional block interface to store and retrieve user data. The commercial success and widespread adoption of devices such as these will be the first step towards more intelligent and smart storage devices. A practical issue in the adoption of these devices is the identification and evaluation of suitable data reliability techniques for data stored in these devices.

Traditional systems with multiple block storage devices employ fixed-length, block-based data reliability techniques to overcome data loss due to data corruptions and device failures, and Redundant Array of Independent Disks (RAID) [9] has been the de-facto standard for these devices. KV devices, on the other hand, allows for the storage and retrieval of variable-length objects associated with variable-length user keys. Their storage semantics and as such, the data reliability techniques/recovery mechanisms are different from traditional block devices.

In this work, we address this need for a tailored data reliability solution for KV devices and present KVMD, a hybrid data reliability manager for such devices. KVMD is to KV devices as RAID is to block devices. We present four different configurable reliability techniques, all suitable for variable-length data addressed by variable-length keys, to be used in KVMD: *Hashing, Replication, Splitting* and *Packing*. These techniques serve as counterparts to the traditional RAID0, RAID1, and RAID6 architectures. We also present the different throughput, storage and reliability trade-offs of these mechanisms, enabling the users to make an informed decision.

In addition, we present three different modes of KVMD operation: a *standalone* mode, where the workload size and characteristics may remain more or less the same and is known beforehand to the user, and the user can choose a single reliability technique for all data, a *hybrid* mode, where the user can configure different reliability techniques for KVs with value sizes in different pre-configured ranges, and a *custom* mode, where the user can specify a reliability technique per KV pair and can be used in combination with either the standalone mode or the hybrid mode.

We also evaluate the above individual techniques for different value sizes, in both the *standalone* and the *hybrid* mode (since *custom* mode is just a functional extension and the performance characteristics does not require a separate evaluation), using Samsung's NVMe Key-Value SSDs (KV SSDs). We show that, when compared to the Linux mdadm-based RAID throughput degradation for block devices, data reliability for KV devices can be achieved at a comparable or lower throughput degradation. KVMD, enabled by the flexible KV interface, also provides much quicker rebuild and recovery compared to Linux mdadm-based RAID. Finally, we conclude that, thanks to the flexible, modern device interface, KVMD for KV devices not only provides custom configuration convenience for the users, but is also either equivalent or superior to schemes for block devices in many ways.

2 Key-Value SSDs

Storage device technologies have undergone tremendous changes since the first disk drive was introduced several decades ago. Yet, the traditional random access block interface is still being used to access most modern storage devices, even the NAND flash SSDs, until recently. Here, we describe the enterprise grade NVMe Key-Value Solid State Drives from Samsung [4].

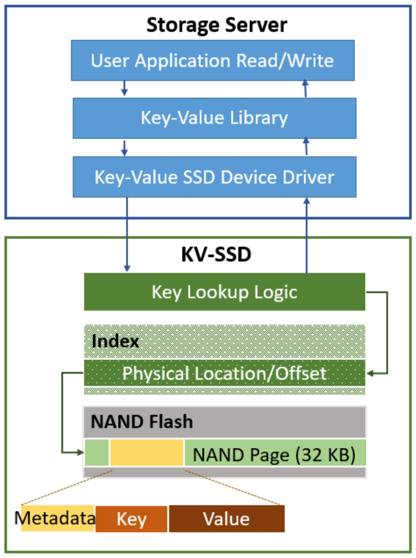


Figure 1: **Key-Value SSD IO path.** Key Lookup logic is added to the device.

NAND flash density has grown tremendously over the years; internal parallelism and read/write bandwidth of the devices have improved drastically. In addition, a NAND flash memory cell can be read and programmed only in units of pages of size 8-32 KB, and a page can be programmed only after an erase, done in larger units of size 4-8 MB. To handle such device characteristics and manage the placement and retrieval of the host-addressable 4 KB logical blocks to the storage media, traditional NAND flash solid state drives already come equipped with very capable hardware and enhanced firmware.

API Kind	APIs
Device API	kvs_[open/close]_device, kvs_get_device_[info/capacity/utilization], kvs_get_[min/max]_[key/value]_length, kvs_get_optimal_value_length
Container API	kvs_[create/delete/open/close]_container, kvs_list_containers, kvs_get_container_info
Key-Value API	kvs_[store/retrieve/delete]_tuple[_async], kvs_get_tuple_info, kvs_exist_tuples[_async]
Iterator API	kvs_[open/close]_iterator, kvs_iterator_next[_async]

Table 1: **Samsung Key-Value SSD API**

The KV SSDs used for evaluation in this paper use the same hardware resources as those of their block SSD counterparts used for evaluation. The KV firmware is based on the block firmware and has modifications to support the storage, retrieval and cleanup of variable-length values and key. Whether the KV IO throughput matches the block IO throughput, or what the effects of increased hardware resources on KV IO throughput would be, are the topics for another paper altogether, and will not be discussed here for the sake of brevity.

Figure 1 illustrates the major components in the IO path of a KV SSD. The Samsung KV-SSDs use the Non-Volatile Memory express (NVMe) interface protocol, developed for low-latency, high-performance non-volatile memory devices connected via PCIe. As seen in the figure, the variable-length KV pair is stored along with any internal metadata in the NAND flash page in a log-like manner, and the index stores the physical location/offset of this variable-length blob, instead of storing a fixed 4 KB data in a log-like manner and indexing the 4 KB block location. The firmware now also has hash-based key lookup logic instead of the traditional logical block number based lookup. In addition, the garbage collection logic is also equipped to deal with variable-length KV pair cleanup. Kang et al. [5] describe the design and benefits of these devices in more detail.

User applications in the storage server can use the KV library API, and the KV library in turn talks to the KV SSD device driver to talk to the KV SSDs. The open-source KV-SSD host software package provides the KV API library and access to both a user-space and kernel device driver for the KV SSDs [10]. Samsung Key-Value SSD API is listed in Table 1 and the detailed description of the API can be found in the KV API spec provided with the host software package.

As can be seen in the table, the API provides management calls to open/close a device and get information such as device capacity/utilization, min/max key and value lengths supported and optimal value length. The API also includes the concept of containers to group KV pairs. The KV API includes both asynchronous and synchronous calls to store, retrieve and delete KV pairs. Further, a user can get information about KV pair or check the existence of keys in the device. Finally, a user can open an iterator set on a predicate and can iterate over either key only or key and value in lexicographic key ordering.

3 KVMD Design

KVMD is a virtual device manager for multiple KV devices. As shown in Figure 2, KVMD handles the KV operations sent to the virtual device and stores the user data chunks in underlying KV devices it manages. KVMD's reliability manager relies on multiple pluggable reliability mechanism (RM) implementations, and can handle huge value sizes unsupported by the underlying KV devices. It can also have an optional data/metadata caching layer to improve performance.

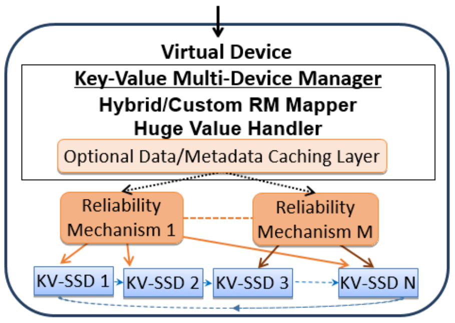


Figure 2: **KVMD Reliable Device.** Reliability device encapsulated the underlying KV devices and employs a hybrid reliability manager.

The virtual device layer works in a stateless manner, i.e., it does not have to maintain any KV to device mapping to work. KVMD can operate in three modes:

- A *standalone* mode, where the workload size and characteristics may remain more or less the same and/or is known beforehand to the user, and the user chooses a single reliability mechanism for all KV data stored in the group of devices,
- A *hybrid* mode, where the user pre-configures different reliability mechanisms for KVs in different value size ranges, and
- A *custom* mode, where the user can by default set either the *standalone* mode or the *hybrid* mode, and in addition specify a reliability technique per KV pair, upon which the specified technique will be used for the KV pair regardless of the default setting.

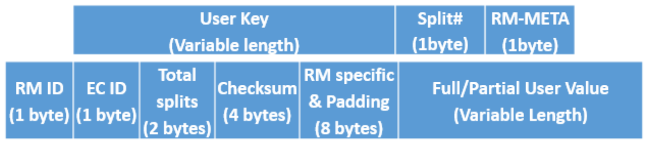
The size-thresholds and the corresponding reliability mechanisms of the *hybrid* mode are specified using a configuration file. The *custom* mode is activated if the individual store call specifies a RM different than the default configuration. The configuration file is also used to specify any RM specific parameters and the erasure code implementation to use for the RM.

The KVMD manager is responsible for the creation and deletion of the underlying device abstraction layer, which handles queue-depth maintenance and calls to the underlying storage devices. The individual RMs share the underlying device abstraction objects owned by the hybrid manager. The underlying device order specified during the virtual device creation is retained by the KVMD manager. This ordering is used to determine the adjacent devices (preceding and following devices) in a circular manner. The virtual device's API is designed to be very similar to that of the KV SSD API as seen in Table 1, with an additional rebuild device call, to recover from entire device failure and rebuild the device contents, and the ability to optionally specify custom RM for stores. KVMD supports both the synchronous and asynchronous versions of the store, retrieve and delete calls, in addition to the synchronous rebuild device call.

3.1 Hybrid-Mode Operations

We will describe the operations of KVMD in the *hybrid* mode, since *custom* mode is similar and the *standalone* mode is the simpler straightforward version.

RM Determination. Since KVMD is stateless and can operate without the optional caching layer, when the user issues a KV call, KVMD does not know if the key already exists. The underlying RMs can handle inserts and updates differently. Hence, all Store/Retrieve/Delete operations has to first determine which RM was used to write a KV pair previously, if the KV pair already exist. This information, along with other metadata is stored in the beginning of all values, as shown in Figure 3, the structure of internal values.



User Key (Variable length)
Split# (1byte)
RM-META (1byte)
RM ID (1 byte)
EC ID (1 byte)
Total splits (2 bytes)
Checksum (4 bytes)
RM specific & Padding (8 bytes)
Full/Partial User Value (Variable Length)

Figure 3: **Internal Key and Value Structures.** KVMD Metadata is stored along with user key and values.

'RM ID' identifies the RM used to store the KV pair, 'EC ID' identifies the erasure code used by the RM, 'Total splits' stores the number of splits a huge object was split into (discussed next under 'Huge Object Handling'), and 'Checksum' field stores checksum and ensures that the data read back hasn't been corrupted and is used to detect failure. Individual RMs determines how the checksum is calculated and stored. Other RM specific metadata is also stored with the value, followed by padding.

KVMD reads part/entire KV pair for every operation, to determine the RM used to write the KV pair and then proceed with the operation, by forwarding the request to the corresponding RM. To aid in this determination by the hybrid manager, all RMs adhere to the below rules:

1. Place the first copy/chunk of the KV pair on the primary device, determined using the same hash function on the key, modulo the number of devices,
2. Store at-least the first copy/chunk/info using the same key as the user key,
3. Store metadata such as RM identifier, EC identifier, at the beginning of the value.

Huge Object Handling. Underlying KV SSD devices may have limits on the max value sizes supported, owing to their internal limitations. For example, the Samsung KV SSD used in this work has an upper size limit of 2 MB. Individual RMs may also have a maximum value size it can support for a KV-pair, based on the underlying device's maximum size minus the metadata size and its own configuration parameters, such as the number of devices a value is split and stored into. If the value size exceeds the maximum size supported by a RM, then KVMD splits the KV pair into multiple KV pairs, where each split's size is determined by maximum size supported by the RM and stores them all using the same RM no matter the residual size of the splits.

As Figure 3 shows, internal keys have additional metadata bytes in addition to the user key field, such as 'split number' and any 'RM specific metadata'. Split number is zero for both the first split of a huge value and a KV that does not have any splits. Thus, the min and max key sizes supported by KVMD are 2 bytes lesser than the underlying KV SSD supported key sizes. During a read, if the metadata in the value indicates that it is part of a huge object, KVMD issues additional IO requests to deal with the huge objects as needed.

Store. After huge objects are split into multiple objects, the *RM* to use to store the key is determined using the configured size-threshold. The KV pair is then read from the primary device. If the KV already exists, RM_{prev} that was previously used to store the KV pair is extracted from the metadata stored with the value, along with the number of splits stored. Then, RM_{prev}'s *update* (for matching split numbers) and *delete* (for excess splits stored previously) methods are called to let RM_{prev} handle these in a *RM* specific way. Finally, the new *RM*'s *store* method is called to store all the KV pair splits.

Retrieve. The KV pair is read from the primary device. If it exists, RM_{prev} is determined, along with the user value size and the number of splits for the KV pair. The *retrieve* request could require additional calls to read from multiple splits, or just call RM_{prev}'s *complete_retrieve* method to complete the initial read as the *RM* sees fit. Finally, the user requested data is assembled to the user value buffer.

Delete. The KV pair is first read from the primary device. If it exists, RM_{prev} and the number of splits are determined from the metadata. Then, RM_{prev}'s *delete* method is called for all the splits.

Rebuild Device. On device failure, KVMD can rebuild all the KVs that would have been present in the failed device to a new device by iterating over all the keys present in the devices adjacent to the failed device, and performing per-KV repairs. Some RMs may require iterating over both the device in front of a failed device and that which is after, while some may require iterating over just one of device. The hybrid manager first obtains the list of drives to iterate from, from all of the underlying RMs, before starting the rebuild process.

3.2 Reliability Mechanisms

This section describes the 4 different reliability mechanisms we implemented and can be plugged into our framework. Table 2 shows the metadata information stored with the different RMs, and will be discussed further below.

3.2.1 Hashing

Hashing does not add any redundancy/data protection. Similar to RAID 0, its purpose is load balancing and request distribution to all underlying devices. It simply hashes the key and stores a single copy in the primary device, and directs all *retrieve* and *delete* calls to the primary device. When a device fails, any recovery attempt fails and user data stored in the device will be lost.

3.2.2 Single Object Replication

Replication is a simple, popular redundancy mechanism in many storage systems that is applied per object (KV pair). The primary device, determined by the key hash, stores the primary copy of the object. As shown in Figure 4, copies of the object are written to $r-1$ consecutive devices when any write happens, in addition to the primary copy, where r is the user-configurable number of replicas, and consecutive devices are determined in a circular fashion. Since 3-way replication is a popular configuration in many systems, including distributed systems, r is set to be 3, by default.

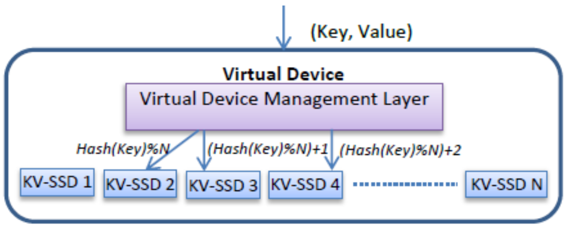


Figure 4: *Replication* stores r copies (3 here) of the data in r consecutive devices.

All copies are identical and stored under the same key in the different devices. r, the number of replicas, is also stored along with other metadata, as shown in Table 2. All RM's also have a *num_user_key* method to return the number of devices from the primary device that would store the user key as is, without any RM specific key-metadata. Replication's *num_user_key* returns r, for example. The *update* method of all RMs obtain uk_{new}, the return value from the new RM's *num_user_key* method. If uk_{new} is less than r, Replication deletes the final $r - uk_{new}$ KVs. Finally, the *store* is passed on to the new RM, and its *store* method is called.

The *retrieve* method reads the entire value from the primary device, verifies the checksum, strips the metadata from the value and copies the user requested data onto the user buffer and returns it, if the checksum verifies. If checksum error occurs, the value is retrieved from one of the replicas, rewritten to the device that failed, and the correct data is returned to the user. The *delete* method issues delete calls on all r consecutive devices starting from the primary device.

Replication has high storage costs and write overhead, but low read and recovery costs. Since the mechanism works per-object and does not have any dependency on any other KV pair, there is no added update overhead. Replication is a good choice for very small values, where the high storage overhead is not a big strain on the system, and keeping the object intact in one piece and independent of other objects is better for performance.

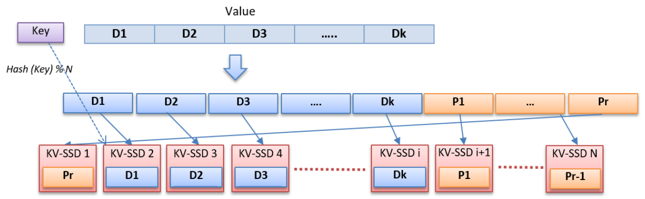


Figure 5: *Splitting* splits the value into k equal-sized objects and add r parity objects.

3.2.3 Single Object Erasure Coding - Splitting

Splitting is a single object erasure coding mechanism, that splits the user object into k equal-sized objects, adds r parity objects using a systemic MDS code and writes the $k+r$ objects to $k+r$ consecutive devices using the same user key. The code is $(4,2)$ Reed Solomon code, by default, similar to RAID 6, though the code and parameters are configurable by the user.

As shown in Figure 5, the first data shard is placed in the primary device determined by the hash, and the other data and parity shards are placed in consecutive devices in a circular fashion. The size of the shard has to be supported by the erasure code implementation and the underlying devices, and the final shard is zero padded for parity calculation purposes, if shards cannot be evenly divided into k shards of supported size. The *ec* in Table 2 indicates the erasure code implementation to use with splitting. Due to space considerations, we will only describe one ec implementation, our best performing equivalent to RAID 6 that is used for all evaluations in the Evaluation section. The original user value size before splitting is also stored as part of the metadata, to be of use when the last shard is lost or needs recovery, in order to recover the right value content without the zero padding.

Similar to Replication, the *update* method obtains uk_{new} from the new RM and if uk_{new} is less than $k+r$, deletes the final $k+r-uk_{new}$ devices. *Delete* method issues deletes for all $k+r$ KVs. The *retrieve* method reads all splits from the k data devices asynchronously. If all their checksums verify, then metadata is stripped from the splits and they are reassembled and sent to the user. If $f <= r$ checksums fail, the required number of parity shards are read and the failed shards are recovered, rewritten to the failed devices and the user requested value is returned back to the user. If the number of failures, $f > r$, then recovery will fail and an error will be returned back to the user.

Splitting reduces the storage overhead. The read and write overhead and throughput reduction is determined by the erasure coding mechanism, code parameters and the size of the values. Similar to replication, splitting does not have any dependency to any other object; hence, no added update overhead. Splitting is recommended for big value sizes where the multiple request processing overhead does not have a huge impact on overall throughput.

3.2.4 Multi-Object Erasure Coding - Packing

Packing is a multi-object erasure coding mechanism that packs up-to k independent objects from k different devices into a single reliability set. The packing is a logical packing, purely for the sake of parity calculation. The user objects are stored in their own primary devices as determined by their hash values, independent of each other, and thus, do not intro-

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	1	$\cdots$
RM	**Value Metadata**																**Key Metadata**	
Hashing	1	0	Splits		Checksum				Padding								None	
Replication	2	r	Splits		Checksum				Padding								None	
Splitting	3	ec	Splits		Checksum				Value Size				k	r	Padding		None	
Packing	4	ec	Splits		Checksum				k	r	Padding						U/M/P	
	Metadata Value																	
Packing	ck	r	Key Size		Var-length Key				Value Size				Repeat $\cdots$ $(k+r-1$ more KVs)					

Table 2: **KVMD Metadata stored per RM.**

duce any privacy concerns. It adds r parity objects to every reliability set and stores them in r devices different from the original k devices. Erasure code can be any implemented using any systemic MDS code. The default, as before, is $(4,2)$ Reed-Solomon code, similar to RAID6 erasure coding.

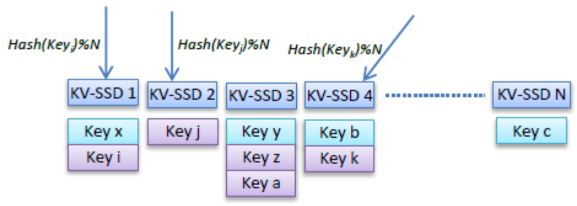


Figure 6: *Packing* packs k different objects into a single reliability set.

Figure 6 shows different keys placed in different KV SSDs based on their hashes, say key i in KV SSD 1, key j in KV-SSD 2 and key k in KV-SSD 4, etc.,. Packing queues recent write requests for each device, and chooses up-to k objects, each from a different device's queue, to be erasure coded in a set (for example, keys x, y, b, and c, the ones marked in blue in the figure, can form a reliability set). Erasure coding of the set of selected objects results in r parity objects, which are written to r different devices.

Erasure coding requires the data sizes to be the same, but there is a difference in size of the objects in a set. This challenge is overcome by virtual zero padding, i.e., the objects are padded with zero's for erasure coding, but the zero padding is not actually written to the device, as shown in Figure 7. The object value buffers and the parity objects are of the same size as the largest object in the set, rounded to a size supported by the erasure code implementation and the underlying devices.

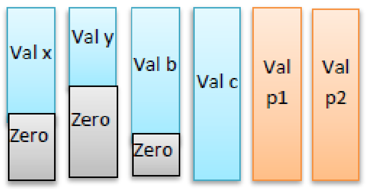


Figure 7: *Packing* pads the different values virtually with zeroes, for the sake of erasure coding.

Retrieve is straightforward; the object is read from the primary device and the checksum is verified to ensure the value isn't corrupted. Object recovery, in case of device failure or checksum failure, and recalculation of parity, in case of updates to an object in a set, requires knowledge of the erasure code set. The RM needs to know which keys were grouped together to calculate parity, and hence are in a set.

Set information together with the actual size of each object (to recreate the objects with actual size without any zero padding) is stored as metadata objects in each of the devices. The metadata objects store the number of user objects, ck and parity objects, r in the set, along with all keys in the set, and their value sizes, as shown in Table 2. Here, ck stands for current k. We want k objects to be packed every time, but we don't want to wait too long in the queue. Hence, after a wait time threshold, available number of objects, $ck <= k$ is chosen to be packed. The RM specific identifier byte in the key is used to store indicators identifying the key as a user object or parity object or metadata object. The Metadata object for each user key is replicated and the number of replicas is set be r.

The *update* method first regroups the erasure code set the key is part of, without the key, before the write can be passed on to the new RM that will be used to store the key. First, the metadata object is read from the device, followed by the rest of the KV objects in the set and they are rewritten into a new reliability set. Once complete, the metadata object for the key is deleted, following which the store if forwarded to the new RM. Delete happens in a similar manner, except the user object is also deleted along with the deletion of the metadata object. While KVMD supports both synchronous and asynchronous calls, the underlying grouping operation in case of updates and deletes are synchronous in our current implementation, affecting performance. Hence, it is recommended to use packing for objects not expected to be updated much; reads are fast for such objects and storage overhead is also small. Similar to splitting, write throughput degradation is determined by the erasure code implementation, parameters and the size of the objects.

4 Evaluation

In this section, we evaluate both software RAID for block devices and KVMD reliability mechanisms for KV SSDs and present the results. The evaluation is done on a Linux server running CentOS 7.4. The machine has Intel(R) Xeon(R) Gold 6152 CPU @ 2.10GHz with two NUMA nodes and 22 cores per CPU, and 64 GB memory. The machine has 6 NVMe SSDs, and the same SSDs are used in both the RAID and KVMD evaluations. For the RAID evaluation with block devices, enterprise-grade block firmware is used, while a KV firmware is used for the KVMD evaluation with KV-SSDs. The KVMD virtual device is formed with all 6 devices for all tests in all evaluations. Replication is configured with total 3 replicas, while packing and splitting is configured to use 4 data devices and 2 parity devices.

KVMD is implemented as a user-space reliability library that works on top of Samsung's open source KVAPI library to access the KV-SSD devices. Unlike RAID, KVMD has hash calculation and 32-bit checksum calculation and verification overhead for every operation. After a test of couple of different implementations, we settled on the crc32 IEEE checksum calculation function using Intel's ISA-L library [11], since we found it to have the least performance degradation. For erasure coding, we use a Reed Solomon coding implementation for any k and r using the Intel ISA-L library [11]. Additional KVMD overhead includes memory allocations/frees and memory copies during external to internal key/value conversions and vice-versa.

The goal of the experiments in this section is to evaluate the performance degradation incurred by the different RMs under different settings and to compare it against the performance degradation incurred by Linux software RAID. But, a RAID vs KVMD comparison is not an apples-to-apples comparison. The device capabilities and internal operations are different. Hence, their absolute throughput numbers are also different. ***In more realistic KVS settings, KV SSDs outperform block SSDs with host KV software stack.*** To learn more, we refer the readers to the work by Kang et al. [5].

The results in this section are presented with 2 y-axis. The left-hand y-axis and the bar plots show the absolute throughput numbers, and the secondary right-hand y-axis represents the percentage throughput achieved and is the axis for the lollipop plot (the red sticks with the small spheres at the end, on top of the bars). The lollipops on top of each bar shows the percentage throughput achieved by the RM or RAID scheme represented by the bar, with respect to the first bar in the category, and provides a sense of the performance overhead. Since this work is not about the device implementation, and the numbers are from prototype firmware, and absolute performance numbers of final products are likely to be different from those presented here, **we encourage the readers to focus on the lollipop plots rather than the bar plots**, as we do in the rest of the section.

Our evaluation uses Fio's [12] asynchronous engine for RAID device and kvbench's [13] asynchronous benchmark supplied with the KV SSD host software package for KVMD device. Hence, we use fixed block and value sizes for our experiments. Since, this is a new emerging device, we also do not make any assumptions regarding the popularity of KV sizes based on previous studies, and have chosen 1 KB, 4 KB, 16 KB and 64 KB value and block sizes.

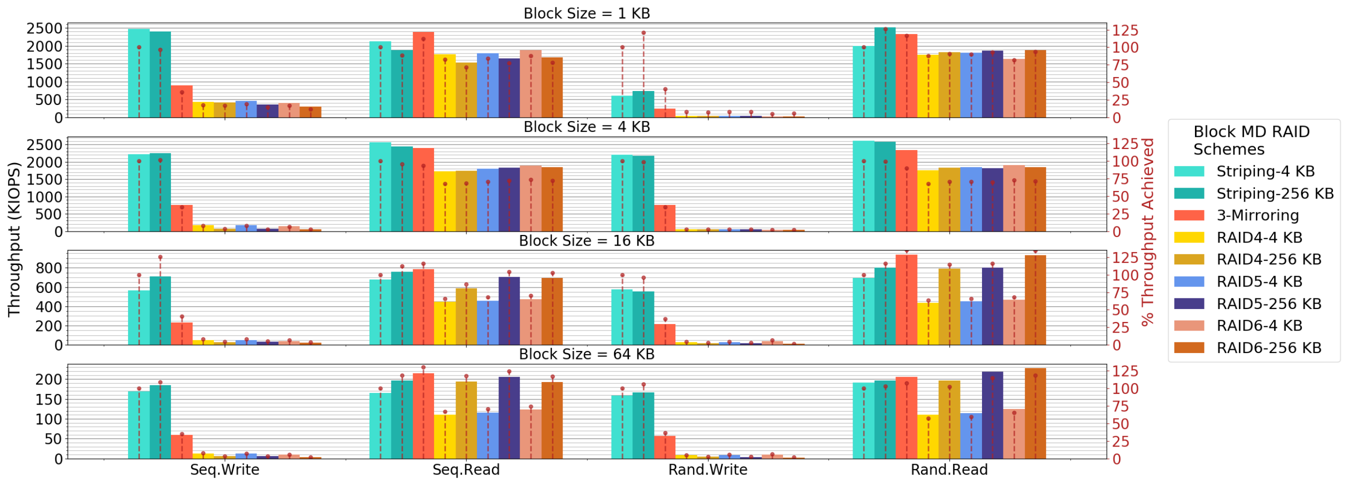


Figure 8: RAID Throughput for block sizes from 1 KB to 64 KB.

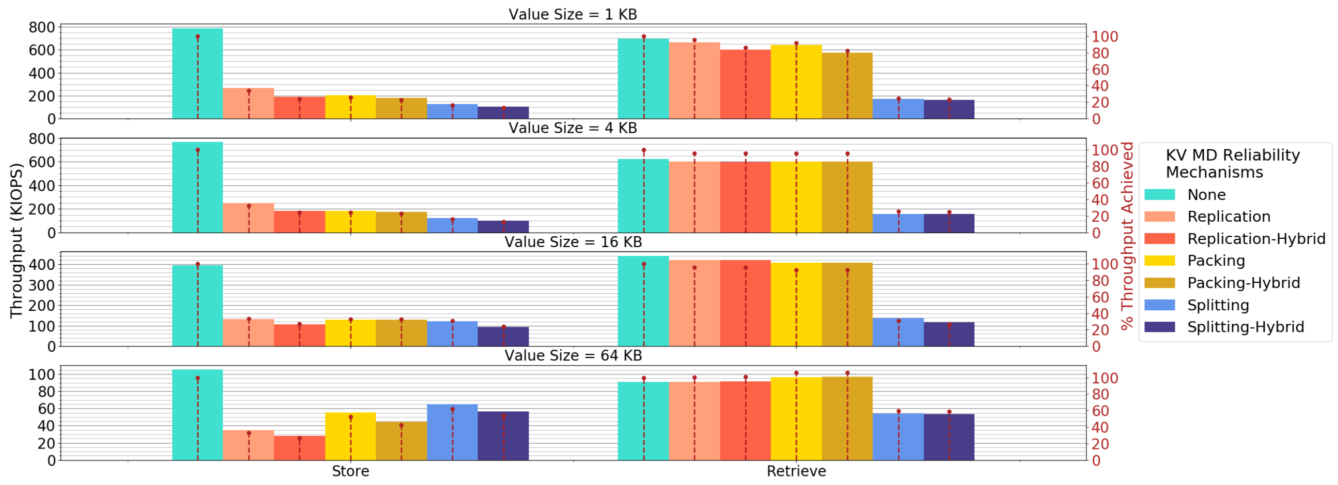


Figure 9: KVMD RM throughput for various value sizes. The Hybrid mode runs of each RM shows the performance impact due to the extra reads in the mode.

4.1 Block Device RAID Performance

Mdadm software RAID is used to create RAID devices on top of NVMe block devices, in striping, mirroring, raid4, raid5 and raid6 configurations, and were tested with both 4 KB and 256 KB chunk size. Mirroring is configured as 2 virtual devices each with 3 physical devices, for 3-way mirroring similar to 3-way replication, with default settings. We measured performance with a total of 6, 12, 24 and 36 threads and found 24 threads to be lowest number of threads to perform the best. The results shown in the Figure 8 are all with 24 fio threads. Our workload sequence was sequential write, followed by sequential read, followed by random write and finally random reads.

Striping achieves the aggregated throughput of all 6 devices. As can be seen from the figure, most RAID writes incur heavy throughput degradation and perform at a much lower rate, compared to striping. The throughput degradation of mirrored writes is as expected, roughly 1/3rd of the aggregated throughput. But all other writes are unexpectedly worse, due to ready-modify-write operations.

Reads perform much better, though we can see significant performance degradation for small block sizes and for small stripe sizes, even though no additional functionality such as read verification or decoding is performed. Read performance is degraded even for larger block sizes, if the stripe size is smaller than the block size.

We observe mirroring to be the best option for performance, with a constant, understandable performance degradation for writes of all block sizes and best read performance in most cases, but has high storage costs. While erasure coding has better storage costs, very high write throughput degradation and uncalled for read throughput degradation in some configurations makes mdadm RAID erasure coding very undesirable for high performance NVMe SSDs.

4.2 KV SSD KVMD Performance

In the presented results, 'None' signifies pure KV SSD read/write throughput, obtained with 6 threads (one for each device). The same number of threads (6) is used to issue IO to the KVMD device, in all cases.

4.2.1 Fixed Value Sizes

KVMD is evaluated only in the standalone (configured only with the RM tested) and the hybrid (configured with all RMs, but only the RMs being tested are exercised) mode, since custom mode is only a functional extension. Even though only one RM is being exercised in the hybrid mode, the possibility of multiple RMs triggers an additional read request for every operation (to check if it already exists) before continuing with the operation. Hence, we observe a slight throughput degradation in the hybrid mode, compared to the standalone mode.

Store and Retrieve. Figure 9 shows the store and retrieve performance for the RMs. Replication achieves roughly 1/3rd the aggregated 6 drive throughput, since it writes 3 objects for every object the user writes, similar to RAID 3-way mirroring. As seen in the figure, the write performance degradation is as expected, in spite of the additional hashing, checksum calculation and memory copy operations. Replication-Hybrid issues 4 requests for every write operation and hence incurs a higher, but expected performance degradation. The read throughput is very close to that of the drives without any reliability, and the slight performance degradation observed is due to checksum verification and memory copy operations for every read operation.

Packing issues the additional read request in both the standalone and the hybrid mode. Hence, the write throughput is similar in both modes. In the tested configuration, it groups every 4 user write into 14 total writes to the device - 4 user writes + 2 parity writes + 8 metadata object writes. The metadata writes are of smaller size than user writes; hence, for small value sizes where metadata write throughput is similar to object write throughput, the write throughput is close to 4/14 of the aggregated device throughput, but for larger value sizes where metadata writes are not as significant as object writes, the write throughput is close to 1/2 of the aggregated device throughput. The read throughput is similar to replication read throughput, since both read the user object in a similar fashion without any other dependency and additional IO requests. Hence, performance characteristics become similar to replication in many cases, even though the space amplification is way less.

Splitting splits the objects into 4 equal parts of size 1/4th the user object size and writes 2 more parity objects of same size as the splits. Hence, splitting issues 6 writes 1/4th the size of the user object, and its write throughput can only be 1/6th of the KV-SSD throughput for the smaller value size. As can be seen in the figure, its write throughput is the lowest among all the RMs for small value sizes, but catches up as value size increases and becomes better than others for larger value sizes. As in the case of replication, additional read request in case of hybrid mode results in slightly more throughput degradation. Reads have the same pattern as writes, but the smaller object performance is better than writes, because every user read only issues 4 read requests to the devices (to read all 4 splits), while every user write issues 6 write requests to the device (to write the 4 splits + 2 parity objects).

Updates and Deletes. For updates and deletes, we show the 4 KB value size results only in Figure 10, since other value sizes follow a similar pattern. The normalized throughput degradation of replication and splitting is similar to the read and write pattern observed earlier. This is because there are no other special update and delete handling procedure for both and they are both limited by the underlying device throughput for the workload and the number of IO requests. But packing performs poorly in both cases, as expected, because our current implementation operates synchronously and has to rewrite objects in a group to new groups before the update/delete could proceed. We believe packing's update and delete performance can be improved further with more engineering effort, but will still be inherently limited.

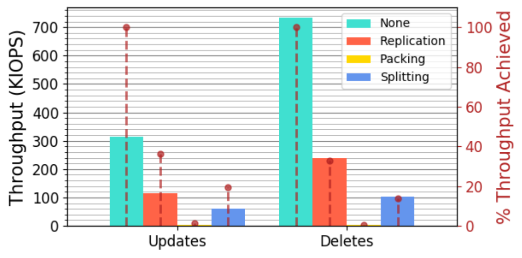


Figure 10: Update and delete throughput for 4 KB values.

4.2.2 Mixed Value Sizes

In this section, we measured the store and retrieve throughput for mixed value sizes, 4 KB, 16 KB and 64 KB in the ratio 30:40:30, and present the results in Figure 11. KVMD was configured in the hybrid mode with value size thresholds configured in such a way that 4 KB objects are handled by replication, 16 KB objects by packing and 64 KB objects by splitting. As can be seen in the figure, the read and write throughput degradation is as expected, retaining the performance characteristics of the underlying RMs exercised by the workload.

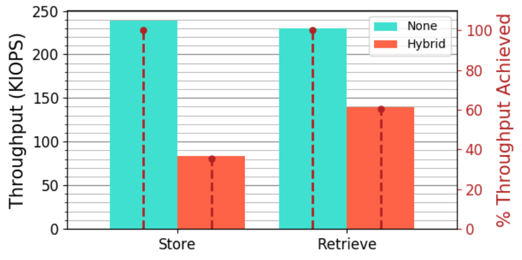


Figure 11: Mixed value size throughput measured in *hybrid* mode configured with all 3 RMs.

4.3 Rebuild Performance

In this section, we present the time taken to rebuild a failed device with very little user data. For this test, we write 1 million 4 KB user objects using the individual RMs/blocks for RAID that is roughly about 4 GB of user data, and then format/fail one of the underlying devices. We present the run time of RAID device repair and KVMD rebuild device functionality in Figure 12. As shown, KVMD reduces repair time drastically compared to RAID, since it is able to and is designed to rebuild only the user data that was written to the failed device as opposed to RAID which traditionally rebuilds the entire failed device. Reduced repair time further increases the reliability of the data stored, as shown in next section.

Time taken, in case of KVMD, is proportional to the RM read/write throughput, decode speed and the number of user objects in the device. Replication has higher write throughput, no decode cost and fewer user objects in the devices and is the quickest. Packing has fewer user objects, but slightly lower write throughput than replication and decode cost, and hence, is slightly slower than replication. Splitting has the lowest write throughput for the workload size, decode size and number of objects, and hence, takes the most time among the RMs. While these KVMD measurements are done using a synchronous, one key at-a-time recovery implementation, it can be improved further with a multi-threaded and/or asynchronous implementations.

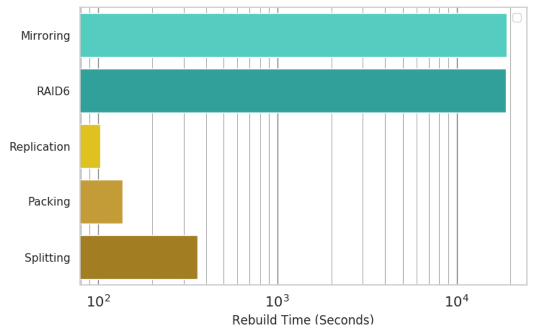


Figure 12: Single device failure rebuild times for RAID and the various KVMD RMs.

5 Analysis

In this section, we provide reliability analysis for KVMD and provide a comparison between the RAID levels and KVMD reliability mechanisms.

5.1 Reliability Analysis

We provide reliability analysis for KVMD, using standard Markov model, and follow the methodology commonly followed by other researchers [14, 15]. As is common in literature, for the sake of simplicity, we are going to assume that data failures are independent and are exponentially distributed, and do not consider correlated failures, even though we are aware that correlated failures are common, and their presence changes the model.

We use the metric mean time to data loss (MTTDL), to compare the reliability of the different mechanisms against each other. The MTTDL of the system is determined by the MTTDL of a reliability set, MTTDL$_{\text{set}}$, normalized by the total number of reliability sets in the system, N_{RS}.

$$MTTDL = \frac{MTTDL_{set}}{N_{RS}} \tag{1}$$

Let the average size of a user object be O, and the capacity of the underlying devices be C. Then, under *replication*, $N_{RS} = C/nO$, where n is the number of replicas. Under *splitting*, $N_{RS} = C/n(O/k)$, where n is determined by the code parameters, k and r, and is equal to $k+r$. Under *packing*,

$N_{RS} = C/n(O+M)$, where n is equal to $k+r$ as well and M is the average size of a metadata object.

MTTDL$_{\text{set}}$ is a function of mean time to failure (MTTF), mean time to repair (MTTR), the total number of objects in the set (N), and the number of parity/redundant objects in the set (G). MTTF is the average interval of time that an object will be available before failing, and MTTR is the average amount of time needed to repair an object after a failure.

Since MTTF is out of our control and is dependent on the underlying device failure rates, MTTDL$_{\text{set}}$ is affected by two factors: $a)$ the number of object failures that can be tolerated before losing user data, and $b)$ the speed at which objects can be repaired. The reliability of the system is also dependent on the number of valid sets stored in the system, unlike RAID which is dependent on the capacity of the system, and not just valid data.

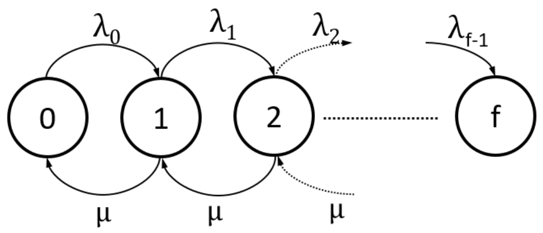


Figure 13: The Markov model used to calculate MTTDL$_{\text{set}}$.

We compute MTTDL$_{\text{set}}$ using a standard Markov model depicted in Figure 13. The numbers on the states represent the number of objects lost in the set, and f denotes the number of object losses that result in a failure and unrecoverable data loss for data in the set. The number of states for a given system depends on the configuration parameters and characteristics of the reliability mechanism. For *replication*, $f = n$, where n is the number of replicas, and for *splitting* and *packing*, $f = r+1$, where r is the number of parities in the erasure code configuration.

The forward state transitions happen on failures and backward transitions happen on recovery. Failures are assumed to be independent, at the rate $\lambda = 1/MTTF$. Since the objects in a set are distributed to N different devices, when the state is i, there are $N-i$ objects intact in a set, and the rate at which an object is lost, λ_i is equal to $(N-i)\lambda$. For recoveries, we assume a fixed recovery rate, μ for recovering a single object and moving from state i to $i-1$. While it is possible for some RMs to recover chunks in parallel and/or to move from state i to 0 directly with slightly different recovery rates, for the sake of simplicity, we model only serial recovery.

$$MTTDL_{set} \simeq \frac{\mu^{f-1}}{(N)_{(f-1)}\lambda^{f}} \tag{2}$$

In equation 2, $(a)_{(b)}$ stands for $(a)(a-1)\cdots(a-b)$.

Table 3 lists the factors affecting the reliability of the system and how. While increasing the MTTF of the underlying devices will increase the reliability of the system, and vendors try their best to do the same, for a given hardware type they do not change much and out of user control. But the rest can be controlled by the user. For same number of parity objects, increasing the number of data objects decreases MTTDL, but not so much. But, adding an additional parity/replica to a set increases MTTDL by orders of magnitude.

Control Factors	Impact on MTTDL
$\uparrow MTTF / \downarrow \lambda$	$\uparrow$
$\uparrow N$	$\downarrow$
$\uparrow f$	$\Uparrow$
$\downarrow MTTR / \uparrow \mu$	$\Uparrow$
$\downarrow \mu \times N_{RS}$	$\Uparrow$
$\downarrow$ Write Amplification (WA)	$\Uparrow$

Table 3: Factors affecting the reliability of the system.

Reducing the time taken to recover an object has a high positive impact on the reliability system. Because $\mu >> \lambda$, reducing MTTR by half has a much higher impact on MTTDL than doubling MTTF. Similarly, reducing the total time taken to repair and rebuild a device by working only on the reliability sets instead of the entire device as done by RAID, improves reliability tremendously.

Finally, for devices such as SSDs, write amplification has a negative impact on the lifetime of the device and reduces the MTTDL. Even though increased space utilization reduces MTTDL, it has been shown that data protection provided by parity improves data lifetime if the configurations are right [16]. *Replication* has a high space utilization negatively affecting the MTTDL. *Splitting* can be configured to have lower space utilization for the same MTTDL. The space utilization of *packing* can vary based on how many writes are available in the device queues and can be higher than configured. Further, updates on packed objects can increase write amplification even further, as the parity needs to be updated again.

5.2 RAID vs KVMD Comparison

Table 4 provides a comparison between the characteristics of RAID for block SSDs and KVMD for KV SSDs. For the read/write characteristics of RAID, we refer the readers to the original RAID publication [9]. KVMD calculations are given for the *standalone* mode, derived by calculating the number of IO requests issued for a given number of user requests.

Since Replication writes everything r times, its write overhead and space utilization is $1/r$, but reads are straightforward, with no additional overhead. Splitting has N writes for every user write and k to 1 reads for every read based on the whether the read is a partial read or not. Packing can end up packing 1 to k user object in a group, and in case of updates can rewrite the whole group for a single update similar to RAID6. Similar to RAID6 calculation, we do not show additional reads required. While metadata writes are

	Block SSD		KV SSD		
	RAID 1	RAID 6	Replication	Packing	Splitting
Writes	$1/r$	$[1/N, (N-2)/N]$	$1/r$	$[1/(N+m), k/(N+m)]$ where m (metadata) $= [r, rk)]$	$1/N$
Reads	1	1	1	1	$[1/k, 1]$
Rebuild Time	$\Uparrow\Uparrow$ ($\propto$ Device capacity)	$\Uparrow\Uparrow$ ($\propto$ Device capacity)	$\downarrow$ ($\propto$ Number of user objects)	$\uparrow$ ($\propto$ Number of user objects)	$\Uparrow$ ($\propto$ Number of user objects)
Space Utilization	$1/r$	$(N-2)/N$	$1/r$	$[1/(r+1), k/N]$ metadata is additional, but assumed small	k/N
Write Amplification	$\Uparrow$	[$\uparrow$ for stripe aligned and sized writes, $\Uparrow\Uparrow$ for most writes]	$\Uparrow$	$\uparrow$ for inserts $\Uparrow\Uparrow$ for updates	$\uparrow$
Pros & Cons	Similar writes for all sizes. Best reads. Low MTTDL due to WA.	Very poor writes and good reads. Poor, workload-dependent MTTDL due to WA.	Similar to RAID 1. Best for small, hot objects.	Best reads. Best inserts. Very poor updates. Good, workload-dependent MTTDL.	Writes/reads $\propto$ value & request sizes. Best MTTDL. Best for large values.

Table 4: Comparison between RAID levels and KVMD RMs. Here, N is the total number of devices in a group, r is the number of replicas or parity devices, and $k = N - r$.

additional, it is the cost paid for high read performance while keeping space overhead lower than Replication. But as seen from the results, bigger the objects, lesser the metadata impact. Variable sizes complicate the space overhead calculation, but we keep it rounded and simple and ignore metadata space since it is assumed small (but is dependent on the key sizes).

The RM specific factors affecting MTTDL are also shown, for easier comparison and informed selection. Finally, the pros and cons of each and how they compare against each other is given. The comparison shows KVMD can provide for KV-SSDS all that RAID provides for block SSDs and more. While the table provides the characteristics of individual RMs under KVMD, the overall read/write performance and MTTDL in the hybrid mode in the presence of mixed value sizes will be determined by the RM configuration for value size ranges, ratio of the user requests and the average size of the objects served by the different RMs configured.

6 Related Work

Plenty of Maximum Distance Separable (MDS) block erasure codes exist to add data redundancy and failure tolerance, such as Reed-Solomon codes [17], Cauchy Reed-Solomon [18], Blaum-Roth [19], etc.,. Our work presents ways to use them all for variable-length key-value data as well. Qin et al., [20] investigated reliability issues in object-based storage devices, but considers them only as network-attached devices and study mechanisms for very large systems with thousands of nodes. While they provide reliability analysis for replication and object grouping, they do not discuss practical considerations such as variable length handling while grouping, or the impact the various schemes have on read/write performance.

Even though many modern distributed, cloud scale systems are built on top of an object-based model, they still use block storage devices underneath and either rely on the redundancy mechanism the underlying block devices employ, such as RAID [21], or provide redundancy at a higher level such as file-level redundancy rather than at a variable-length object level [15, 22, 23], where, the writes are buffered until a fixed-length block (mostly append-only large blocks) is full and replication/erasure coding is applied to these blocks and the resultant blocks are spread across different storage nodes.

In recent years, researchers have proposed a number of resilient, in-memory, distributed key-value caching solutions. Though they need to maintain key to physical location mappings, which is not required for KV devices, and do not have the same performance characteristics and workloads as that of our target system, they do share commonalities such as variable-length values and addressing scheme. Cocytus [24] uses replication for metadata and keys, and erasure coding for values by splitting the value into k parts, adding m parity parts and storing the resulting $k+m$ parts. EC-Cache [25] erasure codes the variable-length objects by splitting and storing the $k+m$ resulting parts in $k+m$ servers. KVMD also explores both replication and splitting as one of the options.

7 Limitations & Future Directions

While we cover a variety of reliability techniques and a hybrid reliability manager to use the different techniques simultaneously, for different user needs and value sizes, by no means is the work complete. In this section, we will discuss some of the limitations of the current design and implementation, and directions for future enhancements of our work.

Concurrency Control. Currently, KVMD does not implement concurrency control, and assume that the applications will implement concurrency control at their desired level. While the device guarantees consistency in case of concurrent asynchronous operations on the same key, it does not guarantee any ordering. If the application does not implement concurrency control, KVMD can be in an inconsistent state. Though Packing synchronizes all updates and deletes to protect against the concurrent update of two members in a group, it does not protect against concurrent inserts of the same key. Replication might result in different versions of the data in different devices. Splitting might have shards from two different versions in a mingled state, resulting in an inconsistent state. This can be avoided by a lock-based implementation, or through a multi-version implementation.

Crash Consistency. KVMD returns a success only after all replicas/shards (including the parity shards)/entire reliability sets (including the parity objects) are written to the device. It is once again assumed that the user/application can replay the write if it receives a failure. In cases where it cannot do so, such as during a crash, a consistency check module and KV restoration mechanism is required. While implementing a consistency check module similar to device rebuild is simple, an efficient mechanism requires design changes. Once detected, inconsistency in case of Replication can be resolved using a consensus algorithm. In case of Packing, inconsistent groups can be regrouped as long as the KV pair (actual or recovered) checksum can be verified. Partial writes in case of Splitting that has $<= r$ shards in a different version can also be recovered, others can't be. A multi-version based update mechanism can provide crash consistency with some additional impact on performance.

Optional Data/Metadata Caching Layer. The optional data/metadata caching layer shown in Figure 2 has also not yet been implemented. The benefits of a read cache is known. KVMD's value metadata is small in size and caching the metadata can avoid the initial read in case of hybrid mode and reduce the performance gap between the hybrid mode and the standalone mode. While the read of non-existent keys is quick in Samsung KV SSDs, it might not be the case with other devices, and the metadata cache could be very useful in those cases. Packing's metadata object can also be cached in the metadata caching tier, and can help improve the update/delete performance by eliminating the metadata object reads. With the metadata objects in memory and with additional in-memory only metadata per object and group, better regrouping of objects across multiple sets can be performed.

Performance. Current Packing implementation has high update and delete performance penalty, due to inefficient synchronous regrouping. A multi-version based design can enable delaying the regrouping and make room for more efficient regroup operations. Combined with the above metadata caching and in-memory metadata, current Packing inefficiencies can be greatly reduced making it a viable and competing choice. Since the current update performance of the device is roughly half the insert/delete performance, a new version insert and old version delete should have similar performance as the current update, and can solve many of the current limitations.

Picking the Right RM. Picking the right RM can be challenging for users, since the throughput is a function of the device capabilities, the RM parameters, and workload characteristics. Users usually have some intuitive knowledge about the average size of the objects in their system, and their update and delete characteristics. With some performance measurements of the underlying devices, workload information and our evaluation, the right size thresholds can be picked by an informed user. The application/user can also use the custom mode for outliers in a size threshold. Nevertheless, in reality, manual picking is hard due to the changing nature of the workload and/or limited user knowledge. Automatic size threshold determination, size threshold outlier detection and outlier custom mode utilization, to minimize space overhead while maintaining a performance level, are promising future directions for our work.

Capacity Utilization. We have also not considered the value sizes and capacity utilization of the underlying devices. Object distribution that avoids uneven capacity utilization, while maintaining the stateless design is an important future work as well.

8 Conclusion

KVMD, our hybrid reliability manager for multiple key-value storage devices, is configurable per the user needs and workload needs. KVMD can be used in the standalone mode by tiered storage systems that have fixed object size/other workload characteristics, while the hybrid mode enables object-size based configuration for a more general setting. The custom mode can be used to switch RMs for objects with certain characteristics, say hot objects, and is applied per object, giving maximum control to the user. We presented four RMs for KVMD: hashing, replication, packing and splitting, all suitable for variable-length KV objects, with different storage, throughput and reliability trade-offs. We also presented a theoretical analysis and practical evaluations of the RMs using Samsung KV SSD prototypes. Finally, we conclude that KVMD is superior to schemes for block devices in many ways.

References

[1] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 2008.

[2] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. *SIGOPS Oper. Syst. Rev.*, 2007.

[3] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 2010.

[4] Samsung Key Value SSD enables High Performance Scaling. http://www.samsung.com/semiconductor/global/file/insight/2017/08/Samsung_Key_Value_SSD_enables_High_Performance_Scaling-0.pdf, August 2017.

[5] Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang-suk Kee, Francisco Londono, Sangyoon Oh, Jongyeol Lee, and Daniel D. G. Lee. Towards Building a High-performance, Scale-in Key-value Storage System. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19.

[6] Y. Jin, H. W. Tseng, Y. Papakonstantinou, and S. Swanson. KAML: A Flexible, High-Performance Key-Value SSD. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 373–384, Feb 2017.

[7] The Seagate Kinetic Open Storage Vision. https://www.seagate.com/tech-insights/kinetic-vision-how-seagate-new-developer-tools-meets-the-needs-of-cloud-storage-platforms-master-ti/, 2016.

[8] Rekha Pitchumani, James Hughes, and Ethan L. Miller. SMRDB: Key-value Data Store for Shingled Magnetic Recording Disks. In *Proceedings of the 8th ACM International Systems and Storage Conference*, SYSTOR '15, pages 18:1–18:11, 2015.

[9] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-performance, Reliable Secondary Storage. *ACM Computing Surveys*, June 1994.

[10] OpenMPDK. KV SSD host software package. https://github.com/OpenMPDK/KVSSD.

[11] Intel(R) Intelligent Storage Acceleration Library. https://github.com/01org/isa-l, 2018.

[12] Flexible I/O Tester. https://github.com/axboe/fio.

[13] KV Benchmark. https://github.com/OpenMPDK/KVSSD/tree/master/application/kvbench.

[14] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G. Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. XORing Elephants: Novel Erasure Codes for Big Data. *Proceedings of the VLDB Endowment*, March 2013.

[15] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, 2010.

[16] Sangwhan Moon and A. L. Narasimha Reddy. Does RAID Improve Lifetime of SSD Arrays? *ACM Transactions on Storage (TOS)*, June 2016.

[17] I. S. Reed and G. Solomon. Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 1960.

[18] Johannes Blömer, Malik Kalfane, Richard Karp, Marek Karpinski, Michael Luby, and David Zuckerman. An XOR-Based Erasure-Resilient Coding Scheme. Technical Report TR-95-048, International Computer Science Institute, 1995.

[19] M. Blaum and R. M. Roth. On Lowest Density MDS Codes. *IEEE Transactions on Information Theory*, January 1999.

[20] Qin Xin, Ethan L. Miller, Thomas Schwarz, Darrell D. E. Long, Scott A. Brandt, and Witold Litwin. Reliability mechanisms for very large storage systems. In *Proceedings of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'03)*, MSST '03, 2003.

[21] Introduction to Lustre Architecture. http://wiki.lustre.org/images/6/64/LustreArchitecture-v4.pdf, 2017.

[22] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows

Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, 2011.

[23] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, November 2006.

[24] Haibo Chen, Heng Zhang, Mingkai Dong, Zhaoguo Wang, Yubin Xia, Haibing Guan, and Binyu Zang. Efficient and Available In-Memory KV-Store with Hybrid Erasure Coding and Replication. *ACM Transactions on Storage*, 2017.

[25] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. EC-cache: Load-balanced, Low-latency Cluster Caching with Online Erasure Coding. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, 2016.