StripeFinder: Erasure Coding of Small Objects over Key-Value Storage Devices (An Uphill Battle)

Umesh Maheshwari
Chiku Research

Abstract

Emerging key-value storage devices are promising because they rid the storage stack of the intervening block namespace and reduce IO amplification. However, pushing the key-value interface down to the device level creates a challenge: erasure coding must be performed *over* key-value namespaces.

We expose a fundamental problem in employing parity-based erasure coding over key-value namespaces. Namely, the system must store a lot of per-stripe metadata that includes the keys of all objects in the stripe. Furthermore, this metadata must be find-able using the key of each object in the stripe.

A state-of-the-art design, KVMD, does not quantify this metadata overhead [10]. We clarify that, when storing D data and P parity objects, KVMD stores $D \times P$ metadata objects, each of which stores $D+P$ object keys. This nullifies the benefit of parity coding over replication in object count. For small objects, it might also nullify the benefit in byte count; e.g., to protect 256 byte objects with 16 byte keys against two failures (P=2), KVMD would cause byte amplification of 2.8x (D=4) and 3.3x (D=8) vs. 3x with plain replication.

We present an optimized version, StripeFinder, that reduces the metadata byte count by a factor of D and the metadata object count by a configurable factor; e.g., to protect 256 byte objects against two failures (P=2), StripeFinder reduces byte amplification to 2.2x (D=4) and 1.9x (D=8). However, even StripeFinder does not provide enough savings for 128 byte objects to justify its complexity over replication. Overall, parity coding of small objects over key-value devices seems to be an uphill battle, and tiny objects are best replicated.

1 Motivation

A key-value store maps application-selected keys to variable-size values. The storage engine of many database systems, including MongoDB [3] and MySQL/MyRocks [9], can be viewed as employing key-value storage. Most database systems today use their own custom key-value stores, but it is desirable to standardize interfaces and create shareable implementations on which optimization efforts can be focused.

A distinctive feature of key-value storage is that the key space is huge and sparsely filled. Most key-value stores support a maximum key size of 16 B to multiple KBs. This huge space enables applications to select meaningful names as keys instead of using storage addresses that happen to be available.

A key-value store translates from the key namespace to the underlying storage namespace. This translation amplifies reads and writes to underlying storage, such as when merging sorted runs, reading over multiple runs, and garbage collecting obsolete versions. This IO amplification is often 5–50x [8].

Most storage devices today present a block namespace. Below we recap how a key-value store layered over a block namespace compounds IO amplification and motivate the emergence of two new types of storage namespaces. We contrast the fundamental characteristics of the three namespace types, as this is deeply relevant to the discussion in this paper.

1.1 The Problem with Block Namespace Type

A block namespace is a sequence of fixed-size blocks (typically 0.5 or 4 KB) also known as "logical" blocks.

This simple namespace type has been a good choice for hard disk drives (HDDs). Block addresses can be mapped to disk locations systematically, and blocks can be updated in place, so this mapping is mostly static and requires very little memory. We refer to translations that need only a small map and cause little IO amplification as being *lightweight*.

This paper is focused on flash-based solid-state drives (SSDs). Internally, flash comprises large blocks (multiple MBs) that must be filled sequentially and erased fully. To avoid confusion with logical blocks, we refer to flash blocks as *segments*—as in LFS [12]. A logical block cannot be updated in place within a segment and must be relocated, so SSDs need a large map to track the location of each block.

Thus, a key-value store built over block SSDs incurs two heavyweight translations: (1) from key namespace to block namespace, and (2) from block namespace to flash segment namespace. Each heavyweight translation consumes memory and compute and imposes IO amplification.

1.2 Emerging Namespace Types

The SSD industry is working on standardizing two new types of storage namespaces to remove double translation.

Zoned Namespace: This namespace is a sequence of large fixed-size *zones* (typically $\geq$ 10 MB), each of which must be filled sequentially and erased fully. (The actual spec is more nuanced [1].) An SSD might map a zone to one or more flash segments, which is a lightweight translation. Heavyweight translation is left to the key-value store on the host [13].

Key-Value Namespace: Here, the SSD presents a key-value namespace and does the heavyweight translation from

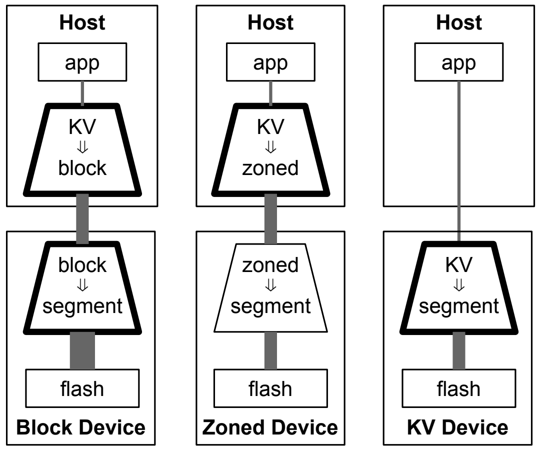

Figure 1: Translation and IO amplification with different device types. Trapezoids represent translations. Vertical lines represent IO. Heavier lines represent heavier IO/translation.

that namespace directly to flash segment namespace [6]. The host OS and applications do not need to perform translation.

The intent is that a future SSD might serve multiple namespaces, each configurable to a supported namespace type: block, zoned, or key-value. However, for simplicity, we equate namespace types and device types and refer to them interchangeably, as if a device serves a single namespace.

Both zoned and key-value device types rid the storage stack of the intervening block namespace, enable a stack with only one heavyweight translation, and reduce IO amplification on flash (each to about the same level). An advantage of key-value device type is that it keeps the residual IO amplification close to flash media, where bandwidth is most plentiful, and frees up bandwidth on higher-level resources such as PCIe and main memory. This is shown in Figure 1.

However, there is an understated challenge with using key-value devices. Pushing the key-value namespace down to the device level creates need for erasure coding *over* key-value namespaces—without visibility into the data layout within each device. This paper shows that layering parity-based erasure coding over key-value namespaces adds complexity and high overhead in the number of objects and bytes stored.

1.3 Outline

Section 2 exposes the challenge of parity coding over key-value devices. Section 3 analyzes the overhead in a state-of-the-art design, KVMD [10]. Section 4 presents an optimized version, StripeFinder. Section 5 states the conclusions and suggests some directions for future work.

This paper presents a conceptual analysis of design alternatives. It does not cover many details needed to build a real system, e.g., buffering objects before parity coding, atomic update of data and parity, detailed layouts, rebalancing when scaling the number of devices, and performance analysis.

The paper is written in the context of a host with direct-attached devices, but most of the arguments are applicable to a disaggregated system with network-attached devices.

2 Parity Coding of Key-Value Objects

Systems with multiple storage devices employ erasure coding because a device might fail, become inaccessible, or lose some data due to bit errors. We use the term "erasure coding" regardless of whether a device is a drive within a host or a node within a distributed system. Also, note that erasure codes include simple but storage-inefficient codes such as replication as well as optimal/MDS codes such as Reed-Solomon codes [11]; we refer to the latter loosely as *parity codes*.

When parity coding, data is encoded into stripes, each a sequence of D pieces of original data and P pieces of parity. We refer to each piece of data or parity as a *stripe unit*. Any D units in a stripe are sufficient to reconstruct the remaining units. Reconstruction requires that, given the identity of one stripe unit, the system be able to identify the other units in the same stripe. We refer to this information as *stripe metadata*.

2.1 Parity Coding over Block/Zoned Devices

When a key-value store is built over block/zoned devices, objects are packed into blocks/zones, which can be parity encoded without much difficulty. In-memory key-value stores that pack objects into fixed-size pages are similar [15].

In this case, parity coding is simple because identifying the blocks/zones within a stripe requires little metadata. A zone is a large enough stripe unit to keep a small map. A logical block is not so large, but the system can collocate successive stripes to effectively form large stripe units [4].

Parity coding can be implemented as the bottom layer of a key-value store, or it can be encapsulated within a filesystem below the key-value store and above the block/zoned devices. In fact, hyperscale filesystems with multi-MB blocks/extents, such as HDFS and Windows Azure Storage, can be seen as providing zoned-like namespaces and have been extended to support parity coding [5, 14].

2.2 Parity Coding over Key-Value Devices

First, consider a system with multiple key-value devices in the absence of any redundancy. A set of objects can be partitioned across the devices based on partitioning the key range or hashing the key. Given a key k, the system can use a function or a small map H to identify the *home* device $H(k)$ that might hold the object. Even though H maps all possible keys, its memory footprint is small.

We refer to the overarching system that interfaces with applications as the *external* key-value store and the objects it receives from applications as *external* objects. The external store might transform external objects (e.g., through splitting or packing) and generate additional objects (e.g., for parity and metadata) and store the resultant objects, which we refer to as *internal objects*, on the underlying key-value devices. The external store can access the underlying devices only through their key-value namespaces.

The next two sections describe how large and small objects might be parity encoded in such a system.

2.3 Splitting Large Objects

Parity coding a large-ish object (say, $\geq$ 16 KB) is relatively simple. The external store splits the object into D data units of (almost) equal size, computes P parity units, and stores the $D+P$ units as internal objects on distinct devices.

The size of a unit need not be a multiple of some sector size, because key-value devices support variable-size objects. The only desirable constraint is that the unit size be larger than a couple of KBs so that the aggregate read throughput from flash-based SSDs is not greatly reduced. This constraint also limits the number of internal objects per GB of SSD capacity.

There are several options to map an external key to the corresponding internal keys efficiently—without storing a large map. Here we describe a generalization of the method employed in KVMD [10]. For each unit, the external store appends the unit number i to the external key k to generate an internal key k:i. It stores this unit on device $H_s(k,i)$, where H_s is a *home-sequence* function, e.g., $H_s(k,i) = (H(k)+i) \bmod N$, where N is the number of devices. This preserves the locality and load-balancing provided by the home map H while ensuring that the units are placed on distinct devices.

This paper is focused on small objects, so we do not discuss splitting much further in this paper.

2.4 Packing Small Objects

Splitting small objects is undesirable because it would diminish read throughput and also create a large number of internal objects. Below we describe two options for encoding small objects that we refer to as *multi-packing* and *uni-packing*.

2.4.1 Multi-Packing

Here, the external store packs multiple small objects into a large *intermediate* object, and stores this large object using splitting as described in Section 2.3. (An advantage of multi-packing is that it can pack objects of widely different sizes into a stripe.) The external store assigns a key k_m for the intermediate object, which is then extended during splitting with unit numbers to generate internal keys of form k_m:i.

This method is simple and perhaps the most intuitive option, but it belies a massive inefficiency: the external store must keep a large map to translate an external key k to the internal key k_m:i. This would re-introduce double translation (one in the external store and one within the key-value devices) and thereby negate the benefit of using key-value devices.

One solution is to create such large stripe units that the key-value devices do not need a heavyweight translation, but this same result is better achieved using zoned devices instead. Therefore, we do not discuss multi-packing further.

2.4.2 Uni-Packing

Here we generalize the method employed in KVMD [10], where it is called "packing." The external store composes a stripe where each data unit is a *single* external object. To that end, it buffers recently-written objects. When it has accumulated D external objects of similar sizes that are destined to D distinct devices based on the home map H, it packs them into a stripe. To generate P parity objects, it temporarily pads all data objects to the size of the largest such object.

The external store places the D data objects on their home devices (which are distinct by construction) using their external keys as internal keys. It can place the P parity objects on any P of the remaining $N-D$ devices. For each parity object, it assigns a fresh internal key k_p such that $H(k_p)$ matches the chosen home device.

A big advantage of uni-packing over multi-packing is that an external object can be read from an underlying device using only its external key, without needing a large map.

However, uni-packing creates its own problems. First, the external store might need to buffer recently-written objects longer than usual, as it encodes only objects of similar sizes that are destined to distinct devices. It can push out a shorter stripe after a wait threshold, but that increases the parity overhead. Similarly, when an external object is deleted, the external store must re-encode the other objects in the stripe before deleting the object on the underlying device.

Second, more importantly, the external store needs to keep a lot of metadata to enable reconstruction. To reconstruct a lost object, the external store needs to fetch the other data and parity objects in the same stripe. Because it cannot intuit the keys of these objects, it must store per-stripe metadata that includes the key of every object in the stripe. Furthermore, this stripe metadata must be find-able using the key of any of the external objects in the stripe. Finally, the metadata itself must be stored redundantly to tolerate failures.

In the next section, we analyze the metadata overhead of the representation employed in KVMD.

3 Metadata Overhead in KVMD

The KVMD paper [10] does not quantify the metadata overhead, perhaps because it is focused on objects that are 1 KB or larger and the overhead is assumed to be small. Here we quantify and analyze the metadata overhead formally.

In KVMD, stripe metadata is stored as an object, which we refer to as a *stripe object*. Thus, each stripe includes three

(D, P)	parity	repl	unipack-md	unipack-md2
	$\frac{D+P}{D}$	$\frac{D(P+1)}{D}$	$\frac{D+P+DP}{D}$	$\frac{D+P+D(P+1)}{D}$
	$1+P/D$	$P+1$	$P+1+P/D$	$P+2+P/D$
(4, 1)	1.25	2.0	2.25	3.25
(4, 2)	1.50	3.0	3.50	4.50
(8, 2)	1.25	3.0	3.25	4.25

Table 1: Object amplification for tolerating P failures.

types of objects: D external data objects, P parity objects, and *multiple* instances of the stripe object driven by two factors:

- For each external key k in the stripe, KVMD stores a separate instance of the stripe object with internal key $k\alpha$ (k extended with constant α to mark this as a metadata object).
- Each of the above instances must survive P failures, so KVMD stores P clones of each. The ith clone is assigned the internal key $k\alpha{:}i$ and stored on a distinct device using the home-sequence function, H_s.

Thus, KVMD keeps $D\times P$ instances of the stripe object. We name this version of uni-packing "unipack-md." We believe a more robust version would need $P+1$ clones for each external key. (With only P clones, P failures can wipe out all clones find-able using an external key k. Now, if object k is deleted, the external store will not be able to find its stripe metadata, which it needs to re-encode other objects before it can delete k on the underlying device.) This robust version keeps $D(P+1)$ instances of the stripe object, and we name it "unipack-md2." In the rest of this paper, we focus on the robust version.

We define *object amplification* of a code as the ratio of the number of objects stored internally (including data, parity, and metadata) to the number of external data objects. Table 1 shows object amplification for protecting D data objects against P failures. Here, "parity" refers to a hypothetical and optimal code that does not require stripe metadata, and "repl" refers to replication, which never requires stripe metadata.

The object amplification of unipack-md and unipack-md2 is even higher than that of replication, which is high to begin with. This is problematic because the performance of key-value stores is often limited by the number of objects stored rather than the number of bytes stored, especially after optimizations such as key-value separation [7].

Now we analyze the number of bytes stored. Suppose the average key size is K and the average value size is V. We calculate the average object size, $W=K+V$, and the *object-key ratio*, $X=W/K$. The object-key ratio is relevant because we show later that the metadata overhead of uni-packing is

	AVG-K	SD-K	AVG-V	SD-V	AVG-W	$X=\frac{W}{K}$
UDB	27.1 B	2.6 B	126.7 B	22.1 B	153.8 B	5.7
ZippyDB	47.9 B	3.7 B	42.9 B	26.1 B	90.8 B	1.9
UP2X	10.5 B	1.4 B	46.8 B	11.6 B	57.3 B	5.5

Table 2: Stats from key-value data at Facebook [2].

(D, P)	parity	repl	unipack-md2					
	$1+P/D$	$P+1$	$(1+P/D)+(P+1)(1+D+P)/X$					
			X=4	X=8	X=16	X=32	X=64	X=128
(4, 1)	1.25	2.0	4.3	2.8	2.0	1.6	1.4	1.3
(4, 2)	1.50	3.0	6.8	4.1	2.8	2.2	1.8	1.7
(8, 2)	1.25	3.0	9.5	5.4	3.3	2.3	1.8	1.5
(D_{opt}, 2)	1.00	3.0	5.8	3.9	2.8	2.2	1.8	1.5
	D=∞	D=1	D=2	D=2	D=3	D=5	D=7	D=9

Table 3: Byte amplification (β) for tolerating P failures.

inversely proportional to this ratio. Table 2 is copied from a recent study of key-value data at Facebook [2]; we added two columns to the right for W and X. The table shows that object-key ratios for these data sets are small—less than 6.

We define *byte amplification* (β) as the ratio of total bytes stored internally to the size of external data. An internal data or parity object is the same size as an external object, W. Each stripe object stores $D+P$ keys and is of size $K+(D+P)K$. (Keys for parity objects are assigned internally and might be more compact than external keys, but we ignore that nuance.)

$$\begin{aligned}\beta_{\text{md2}} &= \frac{(D+P)W+D(P+1)(K+(D+P)K)}{DW} \\ &= (1+P/D)+(P+1)(1+D+P)/X \qquad (1)\end{aligned}$$

Table 3 shows byte amplification from encoding D objects to tolerate P failures. For unipack-md2, it shows results for a range of object-key ratios (X) from 4 to 128. If object keys are 16 B, this range translates to objects of 64 B to 2 KB. The table manifests two surprises. First, for small object sizes, β_{md2} is higher than β_{repl}. We consider a parity code that does not provide at least, say, 20% savings over replication as not worth its complexity and therefore *impractical*. The table shows impractical combinations in red.

The second surprise is that β_{md2} for (D=8,P=2) is *higher* than that for (D=4,P=2), even though increasing D is supposed to *lower* amplification. The anomaly is explained by Eq 1: one term in β_{md2} is inversely proportional to D (parity overhead) and another is proportional to D (metadata overhead). This makes β_{md2} hit a lower bound for some optimal value of D.

$$\frac{\partial \beta_{\text{md2}}}{\partial D} = 0 \implies D_{\text{opt}} = \sqrt{XP/(P+1)} \qquad (2)$$

The last two rows of Table 3 show results for tolerating two failures (P=2) when D is set optimally for each value of X. Thus, β_{md2} cannot be reduced below the values shown here. E.g., to protect 256 B objects with 16 B keys (X=16) against two failures, β_{md2} cannot be reduced below 2.8.

Note that byte amplification from stripe metadata results in higher space usage as well as write amplification. There are other contributors to write amplification within the external store, such as re-encoding a stripe when an object is deleted or updated, which we do not analyze in this paper.

4 StripeFinder: Optimizing Uni-Packing

This section presents StripeFinder, an optimized version of uni-packing to reduce byte and object amplification.

4.1 Reducing byte amplification

The basic idea is to share stripe metadata as much as possible. Specifically, StripeFinder keeps two kinds of metadata:

- *Stripe record*, which contains $D+P$ keys and is cloned $P+1$ times. P of these clones are affixed to the P parity objects, and one is stored by itself in a stripe object. There is synergy in storing stripe records within parity objects because both need to be updated when the stripe composition changes.
- *Finder object*, which maps a single external key k to a single internal key k_p of one of the parity/stripe objects (which, in turn, contain a stripe record). The finder object is assigned an internal key $k\alpha$ and holds the key k_p as value. It is cloned $P+1$ times, each clone ($k\alpha$:i) holding the key of a *different* parity/stripe object (k_p).

We name this version "unipack-sf." For each stripe, it stores D data objects, P parity objects, 1 stripe object, and $D(P+1)$ finder objects. The total number of objects per stripe is one *more* than that in unipack-md2. But unipack-sf stores fewer bytes because finder objects for different external keys within a stripe share the same set of stripe records, so it stores $P+1$ instances of the stripe record instead of $D(P+1)$ in unipack-md2. It does store $D(P+1)$ finder objects, but a finder object contains only one key, so the object size is only $K+K$.

To tolerate P failures, the stripe object is placed on a device different from where the P parity objects are placed. The $P+1$ clones of a finder object $k\alpha$ are stored on $P+1$ distinct devices using the home-sequence function with keys of form $k\alpha$:i.

To look up the stripe record using an external key k, the external store iterates over the finder object clones ($k\alpha$:i) until it finds one that is not lost to failure and that points to a parity object that is also not lost. This adds a non-obvious constraint: If a device holds a finder object $k\alpha$:i as well as a parity/stripe object k_p for the same stripe, that finder object must point to k_p. Otherwise, a failure of this device will cause *two* finder object clones for the same external key to become ineffective: (1) $k\alpha$:i and (2) some other $k\alpha$:j that points to k_p.

Now we analyze the byte amplification of unipack-sf.

$$\begin{aligned}\beta_{\text{sf}} &= \frac{(D+P)W+(P+1)(D+P)K+K+D(P+1)2K}{DW} \\ &= (1+P/D)+(P+1)(1+P/D)/X+1/(DX)+2(P+1)/X \\ &= (1+P/D)+(P+1)(3+P/D)/X+1/(DX) \qquad (3)\end{aligned}$$

Note that β_{sf} does not suffer the anomaly of β_{md2}, because it decreases monotonically with D to an asymptotic value. (In practice, D cannot be increased arbitrarily for a fixed value of P, because that increases the risk of more than P failures, and also because uni-packing requires the system to accumulate D objects of similar sizes before they can be encoded.)

(D, P)	parity	repl	unipack-md2					
			unipack-sf					
	$1+P/D$	$P+1$	$(1+P/D)+(P+1)(1+D+P)/X$					
			$(1+P/D)+(P+1)(3+P/D)/X+1/(DX)$					
			X=4	X=8	X=16	X=32	X=64	X=128
(4, 1)	1.25	2.0	4.3	2.8	2.0	1.6	1.4	1.3
(4, 1)	1.25	2.0	2.9	2.1	1.7	1.5	1.4	1.3
(4, 2)	1.50	3.0	6.8	4.1	2.8	2.2	1.8	1.7
(4, 2)	1.50	3.0	4.2	2.8	2.2	1.8	1.7	1.6
(8, 2)	1.25	3.0	9.5	5.4	3.3	2.3	1.8	1.5
(8, 2)	1.25	3.0	3.7	2.5	1.9	1.6	1.4	1.3
(D_{opt}, 2)	1.00	3.0	5.8	3.9	2.8	2.2	1.8	1.5
(∞, 2)	1.00	3.0	3.3	2.1	1.6	1.3	1.1	1.1

Table 4: Byte amplification for tolerating P failures. White rows are for unipack-md2; gray rows are for unipack-sf.

Table 4 shows that unipack-sf provides significant reduction in byte amplification for a broad range of configurations. (Lower byte amplification also reduces write amplification.) In particular, when protecting 256 B objects with 16 B keys (X=16) against two failures, unipack-sf provides practically useful savings over replication. However, it too struggles with encoding 128 B objects (X=8), in which case D must be increased to 12 to eke out 20% savings over replication. Thus, even unipack-sf is not practical for tiny objects such as those studied at Facebook, where the object-key ratio is below 6.

4.2 Reducing Object Amplification

Unipack-sf employs one more object per stripe than unipack-md2, which is high to begin with; see Table 1. This is because unipack-sf keeps a large number of small finder objects. Now we add an optimization to reduce the number of finder objects.

The basic idea is to group a configurable number (C) of *finder entries* placed on a device into a *grouped finder object*. A finder entry maps a single external key k to the key of a parity/stripe object k_p. A grouped finder object contains roughly C finder entries as an array of pairs (k, k_p).

The finder object must be accessible using any of the external keys that it holds. This is achieved by hashing external keys into B buckets, keeping a single finder object for each bucket on every device, and using the bucket number as the internal key of that finder object. The number of buckets B is configured to result in roughly C entries in each bucket. (As the total number of entries stored on a device changes, buckets can be split or merged to keep the bucket size close to C. This can be achieved by using more or fewer bits from the hash value to assign bucket numbers.)

We name this optimized version "unipack-sf2." Table 5 shows how unipack-sf2 reduces object amplification for a

(D, P)	parity	repl	unipack-md2	unipack-sf2		
	$1+P/D$	$P+1$	$P+2+P/D$	$1+(P+1)/D+(P+1)/C$		
				C=8	C=16	C=32
(4, 1)	1.25	2.0	3.25	1.8	1.6	1.6
(4, 2)	1.50	3.0	4.50	2.1	1.9	1.8
(8, 2)	1.25	3.0	4.25	1.8	1.6	1.5

Table 5: Object amplification for tolerating P failures.

range of values of C, which can be characterized as follows:

$$\omega_{\text{sf2}} = \frac{D+P+1+D(P+1)/C}{D} = 1+(P+1)/D+(P+1)/C$$

The table shows that a relatively modest value of C such as 16 is sufficient to get useful savings over replication.

Grouping multiple entries into a finder object implies that the external store must search for an entry within the object, but this is not a concern because there are only about 16 entries. A bigger problem is that adding or removing an entry requires a read-modify-write of the finder object. Thus, even though unipack-sf2 preserves the optimization provided by unipack-sf in bytes stored, it does increase write amplification. This write amplification could be reduced if the key-value device interface were to support an "append" operation.

5 Conclusion

We explored multiple design options for erasure coding of objects over emerging key-value devices such as KV SSDs [6].

Objects larger than, say, 16 KB can be parity encoded by splitting and striping across multiple key-value devices. Splitting does not require much metadata, so it preserves the low byte amplification of parity coding: $1+P/D$ to tolerate P failures. The only constraint is that the splits must be large enough to avoid creating too many internal objects and diminishing the aggregate read bandwidth from SSDs.

Parity coding of smaller objects has proven tricky. One option is to encode small objects using multi-packing, where multiple external objects are first packed into a large object and then split. However, this requires maintaining a large map to translate keys and causes double translation (one above and one below the device namespace), defeating the benefit of using key-value devices. We conclude that multi-packing is best implemented over zoned devices instead, which avoids translation below the device namespace.

A recent design, KVMD [10], employs uni-packing, where each external object is stored as a separate stripe unit, so external keys can be used internally without translation. However, this method requires a lot of metadata. We showed that it creates even more internal objects than replication, which is problematic because the performance of key-value stores is often limited by object count. Furthermore, for small objects, it also adds a high overhead in byte count. In fact, when the ratio of object size to key size is below 24, it fails to provide practically useful savings (of at least 20%) over replication.

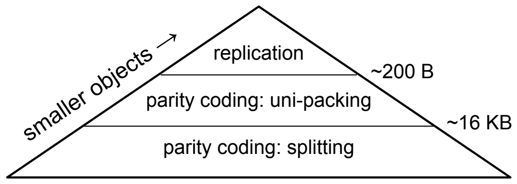

Figure 2: Erasure coding choices for different object sizes.

We proposed an optimized version of uni-packing called StripeFinder. It reduces byte count by sharing metadata, thus providing useful savings over replication even when the object-key ratio is as small as 12 (e.g., for 192 B objects with 16 B keys). However, it too fails to provide practically useful savings for tiny objects such as those studied at Facebook, where the object-key ratio is below 6 [2].

StripeFinder also reduces metadata object count by grouping small pieces of metadata into large objects. This gets the object count close to that provided by optimal parity coding. However, the improvement incurs write amplification from having to read-modify-write larger metadata objects.

Figure 2 summarizes the conclusions for erasure coding over flash-based key-value devices. Large-ish objects ($\geq$16 KB) can be parity encoded using splitting, which provides the optimal storage efficiency of $1+P/D$. Tiny objects (with object-key ratio below 12) are best replicated, which is the simplest erasure code but has the worst storage efficiency of $1+P$. Small objects (of size between the above two ranges) can be parity encoded using uni-packing, which is relatively complex but provides useful savings over replication.

The view shown in Figure 2 is optimistic because it is based on a conceptual analysis and does not fully account for the complexities and inefficiencies that might be associated with uni-packing in practice. These inefficiencies might arise from having to wait longer to accumulate objects that can be uni-packed together, handling object deletion and size-changing updates, using elaborate metadata structures, etc.

Overall, parity coding of small objects over key-value devices seems to be an uphill battle. The main challenge is not so much the variable size of objects, but the fact that the key space is huge and sparsely filled, making it difficult to use a simple function or small map to determine the set of objects in a stripe. This seems to be an intractable problem, but it would be useful to examine it through the lens of coding theory.

On the other hand, a practical solution might be to add extensions to the key-value device interface to help maintain the stripe metadata efficiently. For instance, small metadata objects could be tagged as "infrequently-accessed" so the key-value device might store them more economically.

Without a robust solution, erasure coding of small objects might become the Achilles' heel of key-value storage devices.

Acknowledgments

Thanks to anonymous reviewers for their comments. This research was conducted while the author was affiliated with Hewlett Packard Enterprise (HPE). The views expressed here are those of the author and do not necessarily reflect the official position of HPE. The author can be contacted at `umesh@alum.mit.edu`.

References

[1] Matias Bjørling. From Open-Channel SSDs to Zoned Namespaces. In *Linux Storage and Filesystems Conference (Vault 19). USENIX Association, Boston, MA*, 2019.

[2] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, 2020.

[3] Kristina Chodorow. *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. O'Reilly Media, Inc., 2013.

[4] Mark Holland and Garth A Gibson. Parity Declustering for Continuous Operation in Redundant Disk Arrays. *ACM SIGPLAN Notices*, 27(9):23–35, 1992.

[5] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure Coding in Windows Azure Storage. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 15–26, 2012.

[6] Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang-suk Kee, Francisco Londono, Sangyoon Oh, Jongyeol Lee, and Daniel DG Lee. Towards Building a High-Performance, Scale-In Key-Value Storage System. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 144–154, 2019.

[7] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-Conscious Storage. *ACM Transactions on Storage (TOS)*, 13(1):1–28, 2017.

[8] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, Raju Rangaswami, Sushma Devendrappa, Bharath Ramsundar, and Sriram Ganesan. NVMKV: A Scalable and Lightweight Flash Aware Key-Value Store. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, 2014.

[9] Yoshinori Matsunobu. InnoDB to MyRocks Migration in Main MySQL Database at Facebook. In *SREcon17 Asia/Australia*. USENIX Association, 2017.

[10] Rekha Pitchumani and Yang-suk Kee. Hybrid Data Reliability for Emerging Key-Value Storage Devices. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 309–322, 2020.

[11] Irving S Reed and Gustave Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[12] Mendel Rosenblum and John K Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[13] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 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*, pages 1–14, 2014.

[14] Mingyuan Xia, Mohit Saxena, Mario Blaum, and David A Pease. A Tale of Two Erasure Codes in HDFS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 213–226, 2015.

[15] Matt MT Yiu, Helen HW Chan, and Patrick PC Lee. Erasure Coding for Small Objects in In-Memory KV Storage. In *Proceedings of the 10th ACM International Systems and Storage Conference*, pages 1–12, 2017.