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 significantly reduces metadata byte and object counts; e.g., to protect 256 byte objects against two failures (P=2), StripeFinder reduces byte amplification to 1.9x (D=4) and 1.6x (D=8). However, even StripeFinder does not provide enough savings for 100 byte objects to justify its complexity over replication.

1 Introduction

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.

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.

To make the context around this paper more specific, consider a host running a key-value server with access to multiple storage devices. (This "server" might actually be an embedded library or a kernel module, and the storage devices might be attached directly to the host or connected over a network.) A device might fail or lose some of its data due to uncorrectable bit errors. Therefore, it is desirable for the server to use some form of erasure coding so it can recover lost data.

Most storage devices today present a block namespace. It is relatively simple to employ storage-efficient erasure codes such as Reed-Solomon over block namespaces [11].

However, the use of block devices is known to add an unnecessary layer of translation and IO amplification [8]. Below we summarize this problem 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. A key-value server using a block device must translate from the key-value namespace to the block namespace. This translation requires a large map and induces heavy IO amplification, and we refer to it as a *heavyweight* translation [8].

This heavyweight translation is not so bad for spinning disks. Once the server has done the heavy lifting, disk devices don't have to do much because block addresses can be mapped to predictable locations on disk with a *lightweight* translation.

This paper is focused on flash-based solid-state drives (SSDs). Flash comprises large blocks (multiple MBs) that must be filled sequentially and erased fully. A logical block cannot be updated in place within a flash block and must be relocated, so SSDs use a large map to track block locations and perform garbage collection. Thus, block SSDs result in *two* heavyweight translations: one in the server from the key-value namespace to logical blocks, and another in the SSD from logical blocks to flash blocks.

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].) This enables an SSD to map a zone to one or more flash blocks with a lightweight translation. Heavyweight translation is left to the key-value server on the host [12].

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

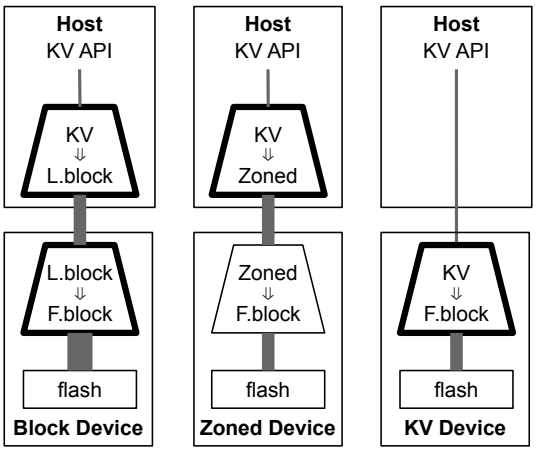

Figure 1: Translation and IO amplification with different device types. Trapezoids represent translations. Vertical lines represent IO. Heavier lines represent heavier IO/translation.

that namespace directly to flash blocks [6]. The key-value server on the host does not need to perform much 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 assume a device serves a single namespace, and we use the notions of namespace type and device type interchangeably.

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 implies that the server must perform erasure coding *over* key-value namespaces—without visibility into the data layout within each device. This paper shows that layering 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 erasure 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 our conclusions.

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.

2 Parity Coding of Key-Value Objects

Erasure coding is useful to recover from the loss of some devices or the corruption of some data due to bit errors. Corruption is a particularly big concern in flash devices, because high-density flash is prone to a high rate of bit errors.

To tolerate P full/partial failures, the server could replicate each piece of data on $P+1$ devices. Replication is the simplest form of erasure coding but incurs high storage amplification of $P+1$. Therefore, it is desirable to use *parity coding*, by which we mean an optimal code such as Reed-Solomon [11].

With 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 recover the remaining units. Recovering a corrupted unit requires the server to identify the other units in the same stripe. This is trivial with block/zoned devices, but it can be tricky with key-value devices

2.1 Parity Coding over Block/Zoned Devices

A key-value server using block/zoned devices can pack objects into blocks/zones, which can then be treated as stripe units without much difficulty. In-memory key-value stores that pack objects into fixed-size pages are similar [14].

Identifying the blocks/zones in a stripe requires little effort. A zone is a large enough unit to keep metadata on where each unit resides. A logical block is not so large, but the server can collocate successive stripes to effectively form large units and calculate the offset of a block within such a unit [4].

Parity coding can be implemented as the bottom layer within the key-value server, or it can be encapsulated within a filesystem layered below the key-value server 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, 13].

2.2 Parity Coding over Key-Value Devices

When using key-value devices, there are *two* key-value interfaces: at the frontend, between the application and the server, and at the backend, between the server and the devices.

In the absence of erasure coding, the server can store application-provided objects directly on the devices without much translation. Given a key, it only needs to know the device that might hold the corresponding object. It can achieve this using a small *home map*, H, based on either partitioning the key range or hashing the keys. Even though H maps all possible keys, its memory footprint is small.

With parity coding, the server might need to transform application objects (e.g., by splitting or packing) before storing them. We refer to application objects as *frontend objects* and objects stored on the devices as *backend objects*. The next two sections describe how objects might be parity encoded.

2.3 Splitting Large Objects

For a large-ish object ($\geq$16 KB, give or take a factor of two), parity coding is simple. The server splits the object into D data units of (almost) equal size, computes P parity units, and stores the $D+P$ units as backend 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 is larger than 2–4 KBs so that the aggregate read throughput from flash devices is not greatly reduced.

The server can map a frontend key to the backend objects generated by splitting without storing a large map. Here we describe the method employed in KVMD [10]. The server generates the backend key for the ith unit in the stripe by extending the frontend key k as k:i. Furthermore, it stores this unit on device $H(k{:}i) = (H(k)+i) \bmod N$, where N is the number of devices in the system ($N \geq D+P$). 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 create a large number of even smaller backend objects. Below we describe two alternatives for small objects that we refer to as *multi-packing* and *uni-packing*.

2.4.1 Multi-Packing

It is tempting to follow the approach used on block/zoned devices: pack many small objects into a large *intermediate* object, and encode this object using splitting as described earlier. (An advantage of this approach is that it can pack objects of widely different sizes into a stripe.) The server assigns a key k_m for the intermediate object, which is extended during splitting to generate backend keys of form $k_m{:}i$.

This method is relative simple, but it belies a massive inefficiency: the server must keep a large map to translate a frontend key k to the backend key $k_m{:}i$. This re-introduces double translation (one in the server and one in the devices) and thereby negates 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 server composes a stripe where each data unit is a *single* frontend object. To that end, it buffers recently-received frontend objects. When it has accumulated D 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 server places the D data objects on their home devices using the frontend key of each object (or a simple extension thereof) as the backend key. (Because there is no significant difference between frontend and backend keys of data objects, we will refer to them as data keys.) The server can place the P parity objects on any P of the remaining $N-D$ devices. For each parity object, it assigns a fresh backend key k_p.

A big advantage of uni-packing over multi-packing is that the server can service read requests without needing a large map to translate frontend keys to backend keys.

However, uni-packing faces its own challenges. First, the server might need to buffer recently-received objects longer than usual, as it encodes only objects of similar sizes that are destined to distinct devices. It can loosen the size constraints, but that might increase the parity overhead. Or, it can generate a shorter stripe after a wait threshold, but that also increases the parity overhead. Similarly, when a data object is deleted, the server might need to effectively shrink the stripe.

Second, more importantly, the server needs to store a lot of stripe metadata. To recover a corrupted object, the server needs to fetch the other data and parity objects in the same stripe. The server cannot intuit the keys of these objects because the key space is huge and sparsely filled, so it stores 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 data objects in the stripe. Finally, the metadata itself must be stored redundantly to tolerate failures.

While both challenges are of immense practical concern, the second challenge is more fundamental and less amenable to workarounds, and it is the focus of this paper. In the next section, we analyze the metadata overhead 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 its metadata overhead formally.

For each stripe, KVMD creates a metadata object that holds the sequence of $D+P$ keys in the stripe. We refer to this metadata object as a *stripe object*. KVMD stores multiple instances of the stripe object:

- For each data key k_d in the stripe, KVMD stores a separate version of the stripe object and assigns it a key that is an extension of k_d, say, $k_d{:}1$.
- To tolerate P failures, KVMD stores P clones of each of the above versions. For data key k_d, these clones are assigned the keys $k_d{:}1$, $k_d{:}2$, …, $k_d{:}P$ and stored on distinct devices using the home map, as described in Section 2.3.

(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.

Thus, KVMD stores $D\times P$ instances of the stripe object. We name this version of uni-packing "unipack-md." We believe a more robust version would store $P+1$ clones of the stripe object for each data key. (With only P clones, P failures can wipe out all clones find-able using a data key k_d. Now, if object k_d is deleted, the server will not be able to find its stripe metadata, which it needs to re-encode the stripe.) 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 backend objects (including data, parity, and metadata) to the number of frontend (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 does not require 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 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.

	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].

We define *byte amplification* (β) as the ratio of the total size of all backend objects to the total size of all frontend objects. A backend data or parity object is roughly the same size as a frontend object, W. Each stripe object stores $D+P$ keys and is of size $K+(D+P)K$.

(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.

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

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 is equivalent to objects of 64 B to 2 KB. The table manifests two surprises. First, for small object sizes, β_{md2} is even higher than β_{repl}. We require uni-packing to provide at least, say, 20% savings over replication to be worth its complexity and be practically useful. 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: $\partial\beta_{\text{md2}}/\partial D = 0 \implies D_{\text{opt}} = \text{sqrt}(XP/(P+1))$.

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 server, 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, which adds optimizations to uni-packing to reduce byte and object amplification.

4.1 Reducing byte amplification

In KVMD, each stripe object contains all $D+P$ keys in the stripe, and this information is repeated in multiple versions of the stripe object. StripeFinder replaces each stripe object with a *finder* object, which contains a single key such that the

finder objects for a stripe form a ring. Thus, given any data key, the server can recover all other data keys in the stripe.

Specifically, suppose the data objects in a stripe have keys k_1, k_2, ..., and k_D. A finder object associated with data key k_d (whose key is an extension of k_d, such as k_d:1) contains the single key k_{d+1} (if $d<D$) or k_1 (if $d=D$). A flag is set in the finder object associated with k_1 to identify the start of the sequence, because parity computation is order sensitive.

Like stripe objects, each finder object is cloned $P+1$ times to tolerate P failures. For data key k_d, these clones are assigned the keys k_d:1, k_d:2, ..., k_d:$(P+1)$ and stored on distinct devices using the home map, as described in Section 2.3.

The server generates the keys for parity objects using a strong hash of the sequence of data keys. Given this strong hash value, σ, the P parity objects are assigned the keys σ:1, σ:2, ..., σ:P. Thus, given any data key, the server can recover all other data keys (through finder objects), and subsequently recover all parity keys (by computing the strong hash over the data keys). This mechanism is sufficient in a server that accesses parity objects only to recover data objects.

A server that might access parity objects independently of data objects, such as when scrubbing the devices, needs an additional mechanism to repair a corrupted parity object. Namely, affix the key of the first data object, k_1, to each parity object. This enables the server to find all data and parity objects in the stripe and repair corrupted parity objects. It assumes, however, that at least one copy of the affixed data key (in the P parity objects) has survived corruption.

The server also needs to be able to recover the identities of the devices holding the parity objects. If there are only $D+P$ devices in the system, the server can infer these devices by excluding the devices holding the data objects (which, in turn, are determined from the home map). If the system includes more devices ($N > D+P$), information identifying these devices can be embedded in the finder objects associated with the first data object in the sequence.

We name this optimized version "unipack-sf." For each stripe, it stores D data, P parity, and $D(P+1)$ finder objects. The size of each finder object is $K+K$. In addition, unipack-sf affixes a data key to each parity object to enable scrubbing-based repair. Thus, the byte amplification of unipack-sf is

$$\beta_{sf} = \frac{(D+P)W + D(P+1)(K+K) + PK}{DW} = (1+P/D) + 2(P+1)/X + P/(DX) \quad (2)$$

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

Table 4 shows that unipack-sf reduces byte amplification relative to unipack-md2 for a wide range of configurations: dramatically for small objects and significantly even for 1 KB objects with 16 B keys (X=64). In particular, unipack-sf provides useful savings over replication for protecting objects as small as 128 B (X=8) against two failures. However, even unipack-sf is not usable for tiny objects such as those studied at Facebook, where the object-key ratio is below 6.

(D, P)	parity	repl	unipack-md2 / unipack-sf					
	$1+P/D$	$P+1$	$(1+P/D)+(P+1)(1+D+P)/X$ / $(1+P/D)+2(P+1)/X+P/(DX)$					
			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	2.3	1.8	1.5	1.4	1.3	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	3.1	2.3	1.9	1.7	1.6	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	2.8	2.0	1.6	1.4	1.3	1.3

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

4.2 Reducing Object Amplification

The second optimization in StripeFinder reduces the number of objects. The basic idea is to combine some number of finder objects residing on a device (possibly belonging to different stripes) into a *combined* finder object. The combined object contains a sequence of finder *entries*. Each entry is a pair of keys (k_d, k_{d+1}), where k_d is the data key it is associated with and k_{d+1} is the key of the next data object in the stripe.

The combined finder object must be accessible using any of the data keys for which it holds an associated finder entry. The server achieves this by using a hash function that partitions the data keys into B buckets, keeping a single (combined) finder object for each bucket on every device, and using the bucket number as the key of the combined 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 may need to be split or merged to keep the average bucket size close to C (within a factor of 2 or so). The server achieves this 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 range of values of C, which can be characterized as follows:

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

The table shows that a relatively modest value of C such as 10 is sufficient to get useful savings over replication.

Combining multiple entries into a finder object implies that the server must search for an entry within the object, but this is not a concern because there are only about 10 entries. A bigger problem is that adding or removing a finder entry

(D, P)	parity	repl	unipack-md2	unipack-sf2		
	$1+P/D$	$P+1$	$P+2+P/D$	$1+P/D+(P+1)/C$		
				C=5	C=10	C=20
(4, 1)	1.25	2.0	3.25	1.65	1.45	1.35
(4, 2)	1.50	3.0	4.50	2.10	1.80	1.65
(8, 2)	1.25	3.0	4.25	1.85	1.55	1.40

Table 5: Object amplification for tolerating P failures.

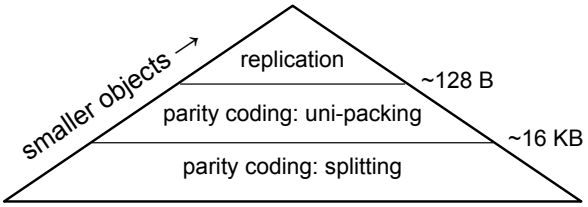

Figure 2: Erasure coding choices for different object sizes.

requires a read-modify-write of the combined object. Thus, even though unipack-sf2 preserves the optimization provided by unipack-sf in bytes stored, it increases 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 about 16 KB can be parity encoded by splitting and striping across multiple key-value devices. Splitting does not require metadata or induce wastage from padding, so it provides the optimal byte amplification of $1+P/D$ to tolerate P failures. The only constraint is that the splits be large enough to avoid creating too many objects and diminishing the aggregate read throughput from SSDs.

Parity coding of smaller objects has proven tricky. One option is to use multi-packing, where multiple application 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), which defeats the purpose 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 application object is stored as a separate object on a device, so application keys can be used without translation. However, this method requires a lot of metadata. We have shown that it creates even more backend 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 (e.g., for 360 B objects with 16 B keys), it fails to provide practically useful savings (of at least 20%) over replication.

We have proposed an optimized version of uni-packing called StripeFinder. It reduces byte count by storing stripe information in a ring of metadata objects to avoid unnecessary repetition. As a result, it provides useful savings over replication even when the object-key ratio is as small as 8 (e.g., for 128 B objects with 16 B keys). However, it too fails to provide 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 combining small pieces of metadata into larger 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 our conclusions for erasure coding over key-value devices. Large-ish objects ($\geq$16 KB) can be parity encoded using splitting, which provides the optimal byte amplification of $1+P/D$. Tiny objects (with object-key ratio below 8) are best replicated, which is the simplest form of erasure coding but incurs high amplification of $1+P$. Small objects (of size between the above two ranges) can be parity encoded using uni-packing, which is much more 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, differences in the sizes of objects in a stripe, 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 workable solution might be to add extensions to the key-value device interface to help maintain the stripe metadata efficiently. For example, small metadata objects could be tagged as "infrequently-accessed" so the key-value device might store them more economically.

Finally, there might *not* be a strong need for storage-efficient erasure coding over key-value devices. This would happen if the amount of data stored on key-value devices is a small fraction of that stored on block/zoned devices, which in turn could happen because of key-value separation within the server or in the application. In that case, for key-value devices, simplicity and robustness might be more important than storage efficiency, and replication might be good enough for all but large objects that can be parity encoded using splitting.

Acknowledgments

Thanks to anonymous reviewers and Tomasz Barszczak 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] 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.

[13] 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.

[14] 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.