Coupling Decentralized Key-Value Stores with Erasure Coding

Liangfeng Cheng
Huazhong University of Science and Technology
Wuhan, China
lenfungcheng@hust.edu.cn

Yuchong Hu
Huazhong University of Science and Technology
Wuhan, China
yuchonghu@hust.edu.cn

Patrick P. C. Lee
The Chinese University of Hong Kong
Hong Kong, China
pclee@cse.cuhk.edu.hk

ABSTRACT

Modern decentralized key-value stores often replicate and distribute data via consistent hashing for availability and scalability. Compared to replication, erasure coding is a promising redundancy approach that provides availability guarantees at much lower cost. However, when combined with consistent hashing, erasure coding incurs a lot of parity updates during scaling (i.e., adding or removing nodes) and cannot efficiently handle degraded reads caused by scaling. In this paper, we propose a novel erasure coding model called FragEC, which incurs no parity updates during scaling. We further extend consistent hashing with multiple hash rings to enable erasure coding to seamlessly address degraded reads during scaling. We realize our design as an in-memory key-value store called ECHash, and conduct testbed experiments on different scaling workloads in both local and cloud environments. We show that ECHash achieves better scaling performance (in terms of scaling throughput and degraded read latency during scaling) over the baseline erasure coding implementation, while maintaining high basic I/O and node repair performance.

CCS CONCEPTS

• **Computer systems organization** → **Redundancy**; • **Information systems** → **Key-value stores**; **Distributed storage**.

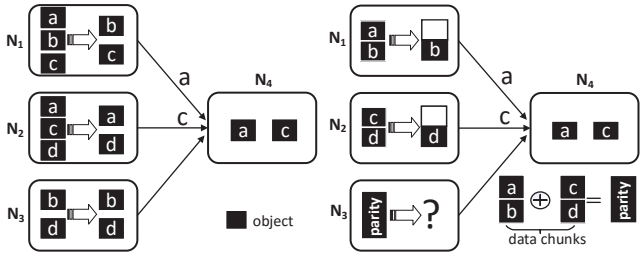
KEYWORDS

Erasure coding, Key-value stores, Scaling

ACM Reference Format:
Liangfeng Cheng, Yuchong Hu, and Patrick P. C. Lee. 2019. Coupling Decentralized Key-Value Stores with Erasure Coding. In *ACM Symposium on Cloud Computing (SoCC '19), November 20–23, 2019, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3357223.3362713

1 INTRODUCTION

Decentralized key-value (KV) stores (e.g., Swift [10], Dynamo [19], BigTable [14], Cassandra [31], and Memcached [8]) provide scalability and availability guarantees that cannot be readily achieved by traditional relational databases. To eliminate the need of centralized administration, most of the decentralized KV stores (including Swift [10], Dynamo [19], Cassandra [31], and Memcached [8]) distribute data across nodes (or servers[1]) using *consistent hashing* [29], which maps KV items (called *objects*) to nodes in a decentralized manner and allows efficient object-to-node remappings when nodes are added to or removed from KV storage.



(a) 2-way replication (b) (3, 2) cross-coding

Figure 1: Scaling (adding a new node D) for objects a, b, c, d over nodes N_1, N_2, N_3, N_4 in consistent hashing.

To maintain data availability, consistent-hashing-based KV stores (e.g., Dynamo [19] and Cassandra [31]) replicate data into multiple copies that are distributed across nodes, yet replication incurs high redundancy. Recent studies [10, 15, 30, 33, 40, 49] adopt *erasure coding* as a low-cost redundancy mechanism while maintaining high availability in KV stores. There are two erasure coding approaches. One approach is *self-coding* [30, 40], which operates on a per-object basis by dividing an object into splits that are erasure-coded. Another approach is *cross-coding* [10, 15, 33, 49], in which objects are stored across nodes and each node combines its stored objects into *data chunks*. The data chunks in different nodes are then encoded into *parity chunks*. The data and parity chunks (which collectively form a *stripe*) are distributed across different nodes, such that all original data chunks can be reconstructed from a subset of a stripe. Compared to self-coding, cross-coding provides two benefits for decentralized KV stores. First, large-scale KV store workloads are often dominated by small objects [12, 38], and cross-coding can combine small objects into chunks of fixed size to make erasure coding viable. Second, cross-coding can easily access an object without centralized metadata lookups (§2).

However, to deploy cross-coding under consistent hashing, it is challenging to handle the elasticity or resizing of the system scale, which we refer to as *scaling* in this paper. Scaling, which we define as either adding nodes (i.e., *scale-out*) or removing nodes (i.e., *scale-in*), is common and critical for real-life large-scale KV stores [3, 38] to cope with dynamic user demands (see §2.3 for details). During scaling, objects are remapped across different nodes based on consistent hashing. To show the challenges of deploying cross-coding, Figure 1 depicts the scaling process with 2-way replication

[1]The terms "nodes" and "servers" are used interchangeably in the paper.

SoCC '19, November 20–23, 2019, Santa Cruz, CA, USA

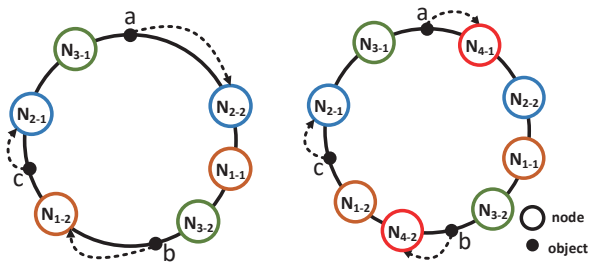
ACM ISBN 978-1-4503-6973-2/19/11…$15.00
https://doi.org/10.1145/3357223.3362713

(i.e., each object has two replicas) and (3, 2) cross-coding (see §2.2 for details), where the objects a and c are migrated to the new node N_4 based on consistent hashing. For replication (Figure 1(a)), the scaling process can simply migrate a and c (or their replicas) to N_4. On the other hand, for cross-coding (Figure 1(b)), the scaling process not only migrates a and c to N_4, but also changes the data chunks which contain a and c. This also causes the corresponding parity chunk to be updated. In the case of frequent scaling, the parity update cost will be significant (Challenge 1). Also, Figure 1(b) shows that the scaling process changes two data chunks of the stripe, and the stripe may fail to decode a and c during scaling. This impairs the performance of *degraded reads* (i.e., the reconstruction of an unavailable object being read from the available chunks of the same stripe). This is unfavorable for KV stores that aim for high availability while supporting frequent scaling (Challenge 2).

We address the above challenges of deploying cross-coding under consistent hashing based on the following intuitions. For Challenge 1, we find that consistent hashing and cross-coding are designed based on two inherently different mapping mechanisms: consistent hashing operates on the mapping *from an object to a node*, while cross-coding operates on the mapping *from a chunk to a node*. It means that even if there is only one object that changes its node mapping under consistent hashing during scaling, its corresponding data chunk, together with all parity chunks of the same stripe, must be properly updated as the data chunk cannot be mapped to two nodes. To address Challenge 1, our idea is to allow a data chunk to be mapped to *multiple* nodes, such that the scaling process no longer needs to change a data chunk and hence there is no need to update the parity chunks of the same stripe. For Challenge 2, we find that the scaling process may change multiple data chunks of a stripe under consistent hashing, even though only one node is added to (or removed from) the system. To address Challenge 2, our idea is to allow consistent hashing to be aware of the number of chunks of a stripe by partitioning all the nodes into the same number of *isolated* zones. We then distribute the chunks of a stripe to different zones, such that the scaling process (e.g., adding/removing a node) only affects the zone that runs scaling and at most one chunk of the stripe is changed.

Based on the above intuitions, in this paper, we propose a new erasure coding model called *fragmented erasure coding*, or FragEC, that supports *fragmented* chunks, meaning that one data chunk can be fragmented into sub-chunks that are mapped to multiple nodes. Also, we design a consistent hashing scheme based on *multiple hash rings*, which partition nodes into multiple isolated zones (i.e., one hash ring per zone). To this end, we build a decentralized in-memory KV store prototype called ECHash, which realizes FragEC in multiple hash rings. In particular, we improve the conventional node repair scheme by supporting the repair of sub-chunks instead of chunks in ECHash. We implement ECHash based on the Memcached protocol [8], which has been the building block in production large-scale in-memory KV services [3, 38]. Compared to state-of-the-art approaches, we show via experiments in both local and cloud settings that the scaling throughput increases by up to 8.3× (local) and 5.2× (cloud), and the degraded read latency reduces by up to 81.1% (local) and 89.0% (cloud).

The source code of our ECHash prototype is available for download at: **https://github.com/yuchonghu/echash**.



(a) Before scaling (b) After scaling

Figure 2: Consistent hashing with virtual nodes.

2 BACKGROUND AND MOTIVATION

We provide the background details of consistent hashing and erasure coding for decentralized KV stores. We then show via motivating examples the challenges of combining erasure coding with consistent hashing and our main ideas.

2.1 Consistent Hashing

Consider a decentralized KV store with M nodes. To balance the storage load of the M nodes, a naïve way is to map each object with key o into the node with the index $hash(o)$ *mod* M. However, during scaling (e.g., a new node is added), we need to re-hash a lot of objects to a new set of nodes; for example, the object o is mapped to the node with the index $hash(o)$ *mod* $(M + 1)$. To mitigate the re-hashing overhead, *consistent hashing* [29] is proposed with the following three steps: (i) calculating the hash values of all nodes and mapping them to a *hash ring*; (ii) calculating the hash values of all the objects' keys and mapping them to the same hash ring; (iii) storing each object in its nearest node in the hash ring along the clockwise direction. Note that the hash ring is divided into M *regions* by the M nodes, each of which is responsible for the region between itself and its predecessor node in the hash ring. In this way, the addition/removal of a node only affects its adjacent nodes in the hash ring, while other non-adjacent nodes remain unaffected.

However, the basic consistent hashing design leads to load imbalance, as the regions may have different sizes and be mapped by different numbers of objects. To achieve load balancing, one approach is to assign v *virtual nodes* to each *physical node*, such that the hash ring is divided into $M \times v$ regions by $M \times v$ virtual nodes [19]. Thus, an object that is covered by a virtual node's region will be distributed to the physical node that corresponds to the virtual node. Figure 2(a) depicts this idea, in which there are three objects a, b, and c, and $M = 3$ physical nodes N_1, N_2, and N_3; each physical node is assigned $v = 2$ virtual nodes (e.g., node N_1 has two virtual nodes N_{1-1} and N_{1-2}). Thus, the hash ring is divided into 6 regions. Here, the objects a and b are covered by the regions $(N_{3-1}, N_{2-2}]$ and $(N_{3-2}, N_{1-2}]$, and are distributed to N_2 and N_1 before scaling, respectively. Figure 2(b) depicts how we re-distribute the objects when a new node N_4 is added. Here, a and b are covered by the regions $(N_{3-1}, N_{4-1}]$ and $(N_{3-2}, N_{4-2}]$ after scaling, respectively, and will be relocated to the new node N_4. Modern decentralized KV stores [8, 10, 19, 31] adopt consistent hashing for deterministic lookups of objects and efficient scaling.

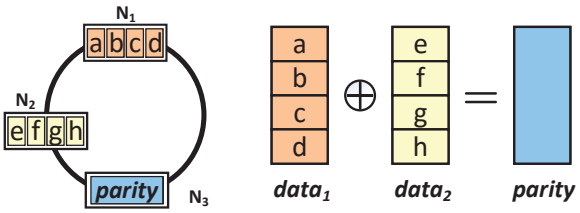


Figure 3: An illustration of $(3, 2)$ cross-coding.

2.2 Erasure Coding

An erasure coding scheme, denoted by an (n, k) code, can be constructed by two configurable parameters n and k. An (n, k) code organizes KV objects as fixed-size *data units*. For every set of k data units, the KV store encodes them into additional $n - k$ equal-size *parity units*, such that any k out of the n data/parity units (collectively called a *stripe*) suffice to rebuild the original k data units. An (n, k)-coded KV store with M nodes contains multiple stripes that are independently encoded, so our discussion only focuses on a single stripe. Suppose that $n < M$. Then we distribute all the n units of a stripe across n of the M nodes to tolerate the failures of any $n - k$ out of the n nodes.

Erasure coding protects data storage against failures with a significantly low degree of redundancy, while tolerating the same number of *any* combination of failures as traditional replication. For example, in Figure 1 (§1), both the 2-way replication and $(3, 2)$ coding can tolerate any single node failure, but $(3, 2)$ coding incurs less storage redundancy (1.5×) than 2-way replication (2×). Previous studies (e.g., [26, 43, 47]) also show via quantitative reliability analysis that erasure coding achieves much higher durability (in terms of mean-time-to-failures) than replication in general.

While erasure coding is more storage-efficient than replication, it incurs higher performance overhead during the reconstruction of unavailable objects, including (i) the degraded reads to an unavailable object and (ii) repairing all lost objects in a failed node. The reason is that reconstructing any unavailable object needs to retrieve multiple available objects of the same stripe for decoding. This is in contrast to replication, which can directly retrieves another replica of the unavailable object for reconstruction.

There are two ways of applying (n, k) coding for KV stores: *self-coding* [30, 40] and *cross-coding* [10, 15, 33, 49]. For self-coding, we partition each object into k data units and encode them into $n - k$ parity units. We then store all the n units in n nodes. Self-coding is suitable for storing large objects (e.g., over 1 MB [40]) in big data analytics. On the other hand, for cross-coding, we distribute objects across different nodes, and in each node, we combine multiple objects into a fixed-size data chunk (which corresponds to a data unit). We encode the k data chunks into $n - k$ parity chunks of the same size, and store all the n chunks in n nodes.

To elaborate how cross-coding works, Figure 3 illustrates $(3, 2)$ cross-coding with consistent hashing for eight objects a to h. The eight objects are first distributed across two nodes N_1 and N_2 via consistent hashing. There are two data chunks $data_1$ (composed of a, b, c, and d) and $data_2$ (composed of e, f, g, and h). A parity chunk $parity$ is generated by XOR-ing $data_1$ and $data_2$, and is placed in the third node N_3. Figure 3 shows two merits of cross-coding over self-coding in decentralized KV stores.

- Cross-coding can support small objects simply by combining them into chunks of fixed size. In fact, real-life KV store workloads (e.g., Facebook [12]) are often dominated by small objects whose sizes range from few bytes to tens or hundreds of bytes. In particular, one of Facebook's five workloads [12] has values with 2 bytes only, while another workload has up to 40% of values with only 2, 3, and 11 bytes.
- Cross-coding can directly calculate (rather than look up) the location of an object via consistent hashing. For example, we can calculate that a is stored at N_1. Thus, cross-coding can easily issue object access requests without centralized metadata lookups, thereby enabling decentralized object management in KV stores.

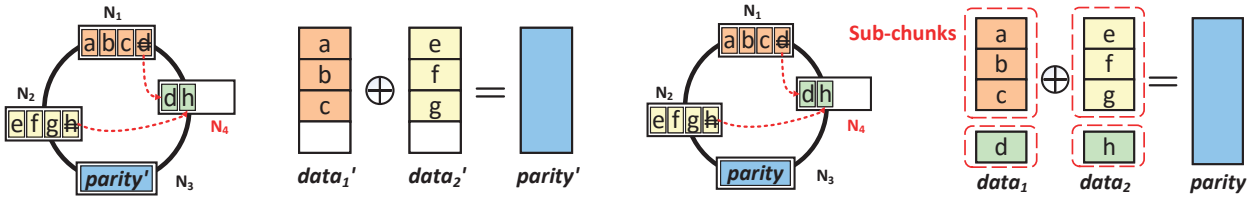
In contrast, it is inefficient for self-coding to be applied to decentralized KV stores. First, it is inappropriate to divide extremely small objects (e.g., 2 bytes) into data units. Also, the fact that each object is divided into small data units makes object access requests require centralized lookups to determine and manage the locations of all small data units before reconstructing the object. Thus, we believe that cross-coding is a sound erasure coding technique for decentralized KV stores. In this paper, we focus on cross-coding.

2.3 Scaling

To handle the dynamic storage demands, a KV store may perform *vertical scaling* (i.e., adding/removing resources of an existing node) or *horizontal scaling* (i.e., adding/removing nodes in a KV store). In this paper, we focus on horizontal scaling, and refer to it as "scaling" in short. Our work considers two scaling processes: (i) *scale-out*, in which we add s nodes to a KV store and denote this process as (n, k, s)-*scaling*, and (ii) *scale-in*, in which we remove s nodes from a KV store and denote this process as $(n, k, -s)$-*scaling*. Note that we still maintain (n, k) coding before and after scaling. A scaling process performs two steps: (i) *object migration*, which redistributes objects due to node additions or removals, and (ii) *parity updates*, which re-compute new parity chunks based on relocated objects. We argue that improving the scaling performance is critical in practice, due to the following two reasons.

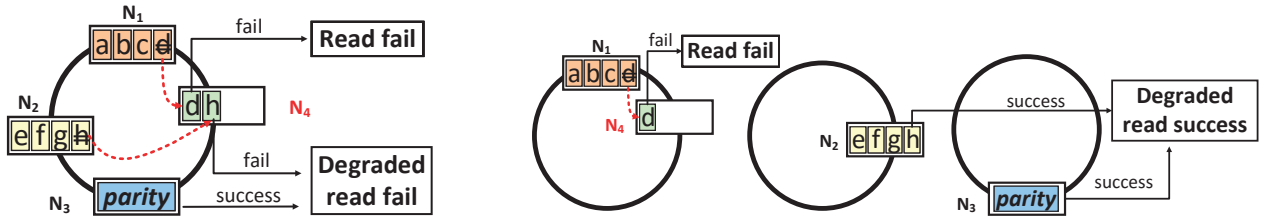
Scaling always happens: Many decentralized KV stores [19, 31] serve hundreds of millions of users at peak times. Due to the huge number of users, real-life KV stores have to *frequently* adapt to users' current performance and reliability service-level agreements, and allow users to dynamically scale-out or scale-in their resources based on their desired request loads. For example, Amazon Web Services (AWS) provides a KV caching service called ElastiCache [3] that can effectively scale in-memory KV storage instances (e.g., Memcached [8]) to meet the current user demands. It also supports auto-scaling [4], which automatically scales the storage capacity to balance performance and monetary costs.

Scaling traffic is significant: Scaling for erasure-coded distributed storage systems inevitably triggers substantial *scaling traffic* (i.e., the amount of transferred data during scaling) [27, 48, 50], as the scaling process needs to migrate data chunks to different nodes and update parity chunks based on the new data chunk layout. Similarly, decentralized KV stores using consistent hashing and cross-coding also incur significant scaling traffic. First, scaling triggers traffic for object migration, since the regions associated with some of the nodes are changed after node additions/removals, and the objects



(a) No fragmented data chunks (b) With fragmented data chunks

Figure 4: (3, 2, 1)-scaling with fragmented data chunks. Without fragmented data chunks (figure (a)), when a new node N_4 is added, we need to update *parity* due to the change of $data_1$ and $data_2$. By allowing fragmented data chunks (figure (b)), we do not need to update *parity*; each part in a dotted frame refers to a sub-chunk.



(a) Single hash ring (b) Multiple hash rings

Figure 5: (3, 2, 1)-scaling with multiple hash rings. With a single hash ring (figure (a)), the normal read or the degraded read to an object may fail during object migration. With multiple hash rings (figure (b)), the degraded read to an object will work when the object is migrated to another node.

need to be migrated to the other nodes that are currently in charge of their storage (§2.1). More importantly, scaling triggers a lot of additional traffic for parity updates after object migration, since many migrated objects will change the data chunks with which they are associated (before or after scaling), and the KV store must transfer data to re-compute the parity chunks. Thus, mitigating the scaling traffic due to parity updates is necessary to accelerate the scaling process and avoid disturbing normal KV storage applications.

2.4 Motivating Examples

In this work, our main goal is to minimize the scaling traffic due to parity updates in decentralized KV stores that build on consistent hashing and cross-coding. We use two toy examples to illustrate the challenges and our main ideas posed in §1.

Example 1: We address Challenge 1 by allowing a data chunk to be fragmented into sub-chunks that are stored in different nodes. We motivate this idea via an example of (3, 2, 1)-scaling in Figure 4; here, we only show the physical nodes and omit the virtual nodes for simplicity. Suppose that a new node N_4 is added and two objects d and h are migrated from the old nodes N_1 and N_2, respectively, to N_4. Note that the migrations of d and h change their corresponding data chunks, so we need to update the parity chunk of the same stripe. Also, we need to create a new parity chunk for d and h after they are moved to N_4, so as to provide them with fault tolerance (the creation of a new parity chunk is not shown in the figure).

To elaborate, in Figure 4(a), the data chunks $data_1$ and $data_2$ are changed to $data_1'$ and $data_2'$ (which no longer have d and h), respectively. In the traditional erasure coding design, the whole parity chunk *parity* has to be updated to *parity*$'$. On the other hand, in Figure 4(b), suppose that we allow the data chunks $data_1$ and $data_2$ to be fragmented across N_1, N_2, and N_4 as sub-chunks. Then $data_1$ and $data_2$ remain unchanged (logically), even though their sub-chunks (i.e., d and h) are (physically) migrated to the new node N_4. In this case, we do not need to update the parity chunk *parity*.

Example 2: In Example 1, the failure of N_4 causes both d and h to be unavailable and leads to Challenge 2. We address this challenge by partitioning all the nodes into n isolated zones that are managed by multiple hash rings and distributing the chunks of each stripe across distinct hash rings (one chunk per hash ring). We motivate this via an example of (3, 2, 1)-scaling in Figure 5, which illustrates the difference between a single hash ring and the multiple hash rings. In Figure 5(a), both d and h may have not been migrated to the new node N_4 in the early phase of scaling, so issuing a read to d through N_4 will fail. Instead, we may issue a degraded read to d by decoding d from $data_2$ and *parity* in N_2 and N_3, respectively. However, the degraded read to d will not work if h has now been migrated away from N_2 in which $data_2$ resides, as $data_2$ no longer contains h for the decoding. In contrast, in Figure 5(b), we distribute all the chunks $data_1$, $data_2$, and *parity* across $n = 3$ distinct hash rings, such that the addition/removal of a node only occurs at one hash ring at a time. In this way, we can issue a degraded read to d by decoding it from other chunks in other hash rings.

Motivation: Example 1 motivates us to design a new erasure coding model, called FragEC (§3), that enables fragmented data chunks; that is, each data chunk can be fragmented into sub-chunks (called *fragments*) across different nodes instead of being stored in a single node. In this way, we keep the data chunks unchanged during scaling and hence do not need to update the parity chunks, thereby eliminating the scaling traffic due to parity updates. Example 2 further motivates us to design a FragEC-based KV store, called

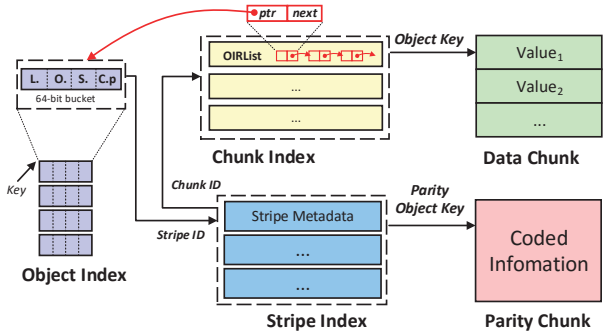


Figure 6: FragEC **model.**

ECHash (§4), that manages all the nodes via multiple hash rings and distributes the chunks of a stripe across distinct hash rings, so as to ensure that all objects remain available during scaling.

3 FragEC MODEL

In this section, we introduce the FragEC model, which allows each data chunk to be fragmented and stored in multiple nodes, so as to efficiently integrate cross-coding with consistent hashing.

Object organization: For object indexing, we introduce the *Object Index* (Figure 6), which is a hash table that maps an object's key to a 64-bit bucket. Each bucket comprises four values (Table 1): the object's length, the object's offset in the chunk in which the object resides, the ID of the stripe in which the object resides, and the chunk index within the stripe (indexed from 0 to $n-1$).

Coding details: We store objects with hybrid-encoding [15], which performs erasure coding on the values of objects while storing the keys and metadata in replication. We encode the objects via (n, k) cross-coding described in §2.2: (i) we select multiple objects from the Object Index; (ii) we combine the selected objects' values into k fixed-size data chunks (4 KiB as the default size [15, 49]) until the chunk size is reached; (iii) we encode the k data chunks into $n-k$ parity chunks of the same size using Reed-Solomon coding [41], a popular erasure code construction adopted in production (e.g., [21, 39]); (iv) we distribute all objects of the k data chunks via consistent hashing across k nodes, such that each object can be accessed directly via consistent hashing, while distributing parity chunks across $n-k$ different nodes in a round-robin fashion [15]. Note that when nodes are later added to or removed from the KV store, FragEC may partition the k data chunks into fragments that are distributed across more than k nodes.

For each data chunk, we prepend a unique 32-bit Chunk ID for chunk identification; for each stripe, we also prepend a unique 32-bit Stripe ID for stripe identification. For the i-th ($1 \le i \le n-k$) parity chunk of a stripe, we treat it as an object and generate its key based on its Stripe ID and i.

Chunk organization: Note that FragEC allows an object to be directly read and written via consistent hashing. Also, FragEC supports update and degraded read operations by locating the data and parity chunks of the same stripe. To this end, FragEC leverages *Chunk Index* and *Stripe Index* for chunk organization (Figure 6). The Chunk Index is a hash table that maps the Chunk ID of a data chunk to a list showing how the data chunk organizes the objects. We call this list *Object Index Reference List* (or *OIRList*), which specifies the references of all the object indices in the Object Index, as well as the organization order of all the objects in the data chunk. The Stripe Index is a hash table that maps the Stripe ID of a stripe to the stripe metadata, which includes the Chunk IDs of the k data chunks and the $n-k$ parity chunks.

Bits	Values	Description
0-11	Length	Length of the object value
12-23	Offset	Offset of the object within the chunk
24-55	Stripe ID	Identifier of the stripe
55-63	Chunk index	Index of the chunk within the stripe (0 to $n-1$)

Table 1: Object Index's bucket format.

Enabling fragmented data chunks: Existing erasure-coded KV stores mainly bind each data chunk to a specific node. On the other hand, FragEC stores objects in different nodes via consistent hashing and records objects' information in the Object Index. It also generates data chunks from objects based on the Object Index and records how the objects form the data chunks in the OIRList. In other words, the Object Index records objects that can reside in different nodes based on consistent hashing, and each data chunk generated based on the Object Index can be formed by objects that are spread more than one node. Thus, FragEC realizes fragmented data chunks that *decouples* the relation between data chunks and nodes, such that each data chunk does not change during scaling if its OIRList stays unchanged, thereby incurring no parity updates.

4 ECHash DESIGN

We present ECHash, a decentralized KV store with the following design goals:

- **No parity updates during scaling:** ECHash realizes FragEC, such that there is no need to update parity chunks during scaling (§4.3).
- **Efficient degraded reads:** ECHash proposes a *multi-hash-ring* design, such that degraded reads can be efficiently issued during scaling (§4.4).
- **Efficient node repair:** ECHash proposes a *fragment-repair* design, which issues the repair of a failed node at the fragment level rather than at the chunk level (§4.5).

4.1 Architecture

We first introduce the architecture of ECHash and state our design assumptions.

ECHash targets consistent-hashing-based *in-memory* KV stores that are commonly deployed in decentralized environments [3, 11, 38], while its design idea is applicable for KV stores with persistent storage as well. Currently, ECHash is implemented atop Memcached [8] (§5.1). Integrating ECHash into other decentralized KV stores must deal with the compatibility issues; for example, Swift [10] (a persistent KV store) uses a different consistent hashing scheme from Memcached. We pose the integration to other decentralized KV stores as future work.

Figure 7 shows ECHash's architecture for in-memory KV storage. ECHash mainly comprises multiple servers for storing objects,

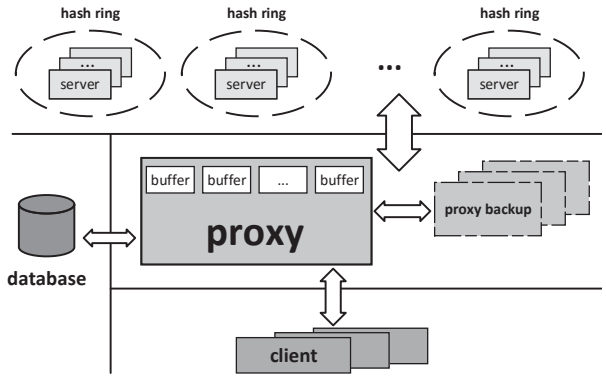


Figure 7: ECHash **architecture.**

multiple clients for interfacing with user applications, as well as a proxy for relaying clients' requests and distributing objects across the servers. Each server allocates a memory region for storing objects that are distributed by the proxy. The servers are organized via multiple hash rings based on three principles: (i) all the M nodes are dispersed across n isolated hash rings, so that all M nodes can be managed by the n hash rings; (ii) one hash ring contains at least one node for storing objects, so that (n, k) coding can be performed across the n hash rings; and (iii) scaling occurs at no more than $n - k$ hash rings at a time, so that any object remains available as (n, k) coding can tolerate up to $n - k$ failures. The proxy serves as a front-end interface for clients to access objects in servers. It implements FragEC and performs erasure coding, scaling, degraded reads, and node repair operations. It has n buffers corresponding to n hash rings for gathering data objects to form data chunks. It also stores the Object Index, the Chunk Index, and the Stripe Index. ECHash currently assumes a single proxy. To avoid the single-point-of-failure of the proxy, ECHash can be attached with multiple backup proxies for synchronizing the proxy-side metadata and the server connection status. It can also be attached with a database for persistent storage.

The proxy-based design is also used in cloud storage studies (e.g., [16, 46]) and production systems (e.g., OpenStack [9]). Our ECHash implementation builds on a Memcached client, which acts as the proxy. We emphasize that the proxy is not the bottleneck in the scaling process, as the scaling performance mainly depends on the object migration and parity update overheads. Our FragEC model for consistent hashing is orthogonal to the proxy-based design. Also, our evaluation fairly compares all schemes under the same proxy-based setting (§5).

In this work, we assume that the size of each object is fixed and will not change after the object is stored, but the sizes of different objects may be different. We also do not consider space reclamation of deleted objects, which can be done offline in the background.

4.2 Basic Requests

ECHash supports four basic requests in normal mode (i.e., without failures): write, read, update, and delete.

Write inserts a new object into ECHash. The proxy stores the object in one of the servers, by first selecting a hash ring via $hash(key)\ mod\ n$ (where key is the object's key) and further determining a server in the selected hash ring to store the object via consistent hashing. To realize FragEC, the proxy appends the value of each object to the buffer of the corresponding hash ring. As soon as k of the n buffers of the hash rings reach the chunk size, these k buffers will be flushed and the objects in each of the flushed buffers are combined into each of the k data chunks. The proxy associates each data chunk with a unique Chunk ID, and updates the corresponding ORILists in the Chunk Index. Also, the proxy encodes the k data chunks into $n - k$ parity chunks, and associates the parity chunks of the same stripe with a unique Stripe ID. The Object Index records the information of the flushed objects and the parity objects as described in Table 1. For parity chunks, the proxy chooses the $n - k$ hash rings that are different from the hash rings of the k data chunks. It places each parity chunk in each of the $n-k$ selected hash rings via consistent hashing, and records the stripe metadata in the Stripe Index.

Read retrieves an object from ECHash. The proxy first selects the hash ring based on the object's key. It then retrieves the object in the hash ring via consistent hashing.

Update changes an existing object's old value into a new value. The proxy first selects the hash ring based on the object's key. It then finds the object's Stripe ID, offset, and length from the Object Index. It locates the whole stripe by the Stripe Index (which contains all Chunk IDs and parity keys). It retrieves all the parity chunks and the old values of the objects. It computes all the parity chunks' new values, and writes the new object and the updated parity chunks to the servers.

Delete removes an object from ECHash. The proxy treats delete requests equivalently as update requests by updating the object's value to zero-bytes. The free space of the deleted objects can later be reclaimed in the background (§4.1).

4.3 Scaling

ECHash performs scaling when a new node is added or removed, in which it performs object migration and parity updates (§2.3). Here, we describe the scale-out process with $s = 1$ (i.e., adding a new node); the case of $s > 1$ can be generalized in a similar way. We also discuss the scale-in case, which can be viewed as the reverse of the scale-out process.

Object migration: Recall from §2.1 that the scaling process under consistent hashing will change the regions of nodes and migrate some objects among nodes. In ECHash, it migrates the affected objects and updates the corresponding metadata. Specifically, suppose that the addition of a new node is associated with v virtual nodes (§2.1). ECHash first identifies the v new regions covered by the new node's v virtual nodes that are mapped to the hash ring. It then traverses the Object Index to identify the objects being covered by the v new regions (i.e., the objects that need to be migrated) in the hash ring. It also identifies the v old virtual nodes in which the migrated objects reside before scaling (i.e., the nodes that are adjacent to the v new virtual nodes in the hash ring along the clockwise direction). Finally, it retrieves the objects from the physical nodes of the corresponding v old virtual nodes and stores them in the new node. Figure 8 depicts the object migration process in ECHash (with the same setting as in Figure 2). In the figure, ECHash first

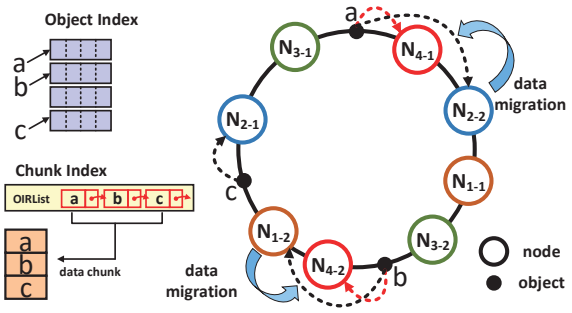


Figure 8: Illustration of ECHash's scaling with $s = 1$.

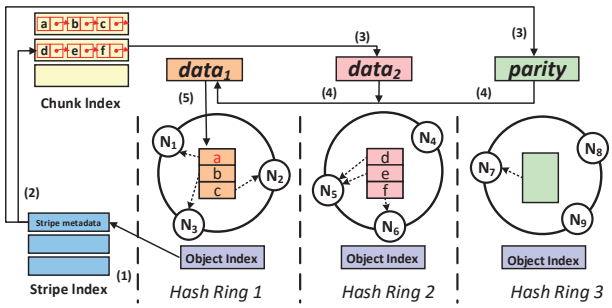


Figure 9: Workflow of the degraded read to object b: (1) selecting Hash Ring 1 and looking up the Object Index to obtain the Stripe ID, offset, and length of b; (2) from the Stripe ID, obtaining the stripe metadata and the chunk IDs of the chunks $data_2$ and $parity$; (3) from $data_2$'s Chunk ID, constructing the data chunk $data_2$ based on its OIRList in Hash Ring 2, as well as the parity chunk in Hash Ring 3; (4) decoding $data_1$ from $data_2$ and $parity$; (5) retrieving object b by its offset and length from the decoded $data_1$.

identifies two new regions $(N_{3-1}, N_{4-1}]$ and $(N_{3-2}, N_{4-2}]$ due to the addition of the new node N_4. It then identifies the objects a and b covered by the two regions from the Object Index. It finds the two old virtual nodes N_{2-2} and N_{1-2}, reads and deletes a and b from N_2 and N_1, respectively, and writes them to the new node N_4.

Parity updates: Recall from §3 that under FragEC, ECHash does not change the OIRList of each data chunk during scaling. Although some objects have been migrated due to consistent hashing, the object organization within each data chunk determined by the OIRList keeps unchanged. Thus, ECHash does not need to change any data chunk after scaling, thereby eliminating parity updates and significantly reducing the scaling traffic. For example, in Figure 8, a data chunk is constructed by three objects a, b, and c, while the objects a and b are migrated to the new node N_4 during scaling. Note that ECHash can ensure via the data chunk's OIRList that the data chunk is still formed by a, b, and c after scaling.

Discussion: Scale-in can be in general viewed as the reverse of scale-out, yet they incur different savings of parity chunk updates in ECHash. Specifically, for scale-out, objects are migrated from existing nodes to newly added nodes, so ECHash can save the parity updates of the stripes that involve existing nodes. In contrast, for scale-in, objects are migrated from the removed nodes to other existing nodes, so ECHash can save the parity updates of the stripes that involve the removed nodes.

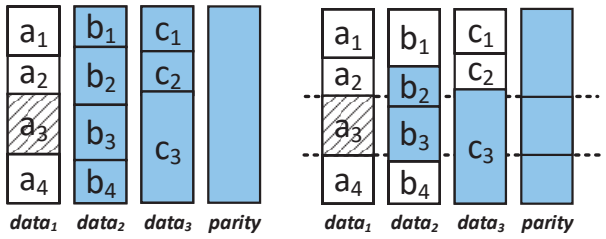
4.4 Degraded Reads

We first describe the basic approach of performing degraded reads to objects. Suppose that there are k data chunks and $n-k$ parity chunks of a stripe, and one of the k data chunks now becomes unavailable. If we want to read an object that belongs to the unavailable data chunk, then we decode the unavailable data chunk by retrieving any k out of the remaining $n-1$ available chunks via the FragEC model, and we access the object within the decoded data chunk.

There are two major cases that trigger degraded reads to objects. The first case refers to a transient or permanent server failure that leads to unavailable objects. In this case, it can be handled via the above basic approach of degraded reads. The second case refers to a scaling operation (e.g., adding a new node), which is frequent in decentralized KV stores (§2.3). In this case, the scaling operation may migrate multiple objects of a stripe to other new nodes based on consistent hashing. As shown in Example 2 in §2.4, the degraded reads cannot be applied directly under traditional consistent hashing if we organize all the servers in a single hash ring. Handling degraded reads becomes more challenging in a single hash ring if the scaling operation involves multiple nodes, since many objects will be migrated. Currently, our ECHash implementation keeps a copy of every object in a persistent MySQL database (§5.1). If an object cannot be decoded (e.g., ccMemcached in our evaluation (§5.1)) , it will be retrieved from the MySQL database. This increases the degraded read latency.

To efficiently deal with degraded reads under scaling, ECHash organizes nodes in *multiple* hash rings. Specifically, it writes an object by first selecting a hash ring, followed by distributing the objects within the selected hash ring (§4.2). To issue a degraded read to an unavailable object, ECHash first selects the hash ring in which the object resides. It locates the object's stripe from the Object Index using the object's key. It looks up the stripe metadata by Stripe ID to obtain the Chunk IDs of available data chunks and parity chunks for decoding, so as to decode the data chunk in which the requested object resides. Here, ECHash can retrieve the objects of the retrieved data chunks from their OIRLists as well as the parity chunks to decode the data chunk in which the requested object resides. Finally, it recovers the requested object from the decoded data chunk from the Object Index using the offset and length information. A key advantage of using multiple hash rings, as opposed to using a single hash ring, is that we can stage the additions/removals of nodes to ensure that objects remain available at any time (assuming that the number of failures is within the tolerable limit), while still preserving the existing consistent hashing design in each hash ring (i.e., during scaling, we can re-distribute the objects in the same hash ring based on the original consistent hashing scheme).

Figure 9 depicts the above degraded read workflow for an unavailable object b stored in the data chunk $data_1$ (composed of a, b, and c), by retrieving the data chunk $data_2$ (composed of d, e, and f) and the parity chunk $parity$ for decoding. Note that the chunks $data_1$, $data_2$, and $parity$ reside in three different hash rings.



(a) Chunk-repair (b) Fragment-repair

Figure 10: Illustration of chunk-repair and fragment-repair.

4.5 Node Repair

We now study how ECHash repairs a node failure, in which it reconstructs all the objects of the failed node through chunk decoding. Based on FragEC, there are two types of data to repair: the original objects and the parity chunks, so we need to repair them separately. However, we find that it is inefficient to repair the original objects in ECHash. The reason is that FragEC enables fragmented data chunks and operates on fragments instead of chunks, while the traditional repair in erasure coding operates on chunks (or called *chunk-repair*). In other words, even if there is only one object as a fragment in the failed node, the chunk-repair still needs to decode the whole data chunk by retrieving other available data chunks and parity chunks of the same stripe. For example, Figure 10(a) shows how to repair the object a_3 in a failed node based on chunk-repair. There are three data chunks (i.e., $data_1$, $data_2$, and $data_3$) and one parity chunk (i.e., $parity$) in the stripe, and the object a_3 resides in $data_1$. To repair a_3, the chunk-repair approach needs to decode $data_1$ (composed of a_1, a_2, a_3, and a_4) by reading all the other available chunks (i.e., $data_2$, $data_3$, and $parity$). This means that all the objects of $data_2$ (composed of b_1, b_2, b_3, and b_4) and $data_3$ (composed of c_1, c_2, and c_3), as well as the parity chunk, all have to be retrieved in order to repair the only one object a_3.

In view of this, ECHash enhances its repair performance to handle the above inefficiency by proposing a *fragment-repair* approach for node repair. Our idea is inspired by repair pipelining [32], in which the repair of an erasure-coded chunk can be done by decomposing a chunk into smaller sub-chunks and performing the repair at the sub-chunk level. The fragment-repair approach works as follows.

Object repair: The main idea of fragment-repair in ECHash is to only retrieve the necessary objects to repair the fragments of a failed node. To this end, ECHash can use the OIRLists to retrieve all the necessary objects, since any object in each data chunk can be exactly located via the OIRLists and we can identify all the necessary objects to repair the required fragments by searching through the OIRLists.

Specifically, to repair the fragments of a failed node (called *failed fragments*), ECHash performs four steps. First, for each failed fragment, ECHash identifies all the positions of objects of the failed fragment in its data chunk and stores the information in a list called *PosLenList*, which includes the offsets and lengths of the objects of the failed fragment. Second, ECHash locates the fragment's stripe by the Stripe ID from the Object Index via any key of the objects in the failed fragment. Third, ECHash retrieves the necessary objects from the chunks of the fragment's stripe via their offsets and lengths in the PosLenList. Finally, ECHash decodes the failed fragment from the necessary objects as well as the parity chunks determined by the stripe metadata. For example, Figure 10(b) shows how to repair the object a_3 in a failed node based on fragment-repair. To repair a_3, the fragment-repair approach only retrieves b_2 and b_3 in $data_2$, c_3 in $data_3$, and $parity$, so as to decode a_3.

Parity repair: To repair a parity chunk, ECHash first obtains the Stripe ID, and locates the available data chunks of the same stripe via the Chunk IDs and the stripe metadata. It then retrieves all data objects for each data chunk in the stripe, and combines them into the data chunks via their OIRLists, offsets and lengths. Finally, it encodes the data chunks into the parity chunk being repaired.

Discussion: The fragment-repair approach is only applied to repairing the objects but not the parity chunks. For example, in Figure 10(b), to repair object a_3, ECHash has to retrieve the whole parity chunk $parity$, since ECHash treats each parity chunk as an object (§3), which cannot be divided into fragments. We can further improve the repair performance by reading the fragments of the parity chunk $parity$ instead of the whole $parity$, and we pose this issue as future work.

4.6 Discussion

Finally, we discuss three open issues in our ECHash design: consistency, degraded writes, and metadata management. We describe how they can be addressed in ECHash, while their implementation and evaluation are posed as future work.

Consistency: It is critical for ECHash to provide consistency support in both proxy-based design and erasure-coded updates. For the proxy-based design, we need to ensure that the proxy and its backup proxies maintain consistent states. This can be realized via the standard coordination systems, such as Zookeeper [28] and etcd [5].

For erasure-coded updates, we need to ensure that if a data chunk is updated, all $n-k$ parity chunks of the same stripe in (n,k) cross-coding are consistently updated via atomic broadcast to reflect the update of the data chunk [15]. One way to handle consistent parity updates is based on the two-phase commit protocol. Specifically, ECHash can allocate a temporary buffer in each server for holding parity chunks. First, the servers store the parity chunks in the buffer and acknowledge the proxy whether the required parity chunks have been correctly stored. After the proxy receives the acknowledgements from all servers that store the parity chunks, it can notify all the servers to commit all parity chunks from the temporary buffers; otherwise, it notifies all the servers to rollback and discard the parity chunks. We may also leverage a piggybacking approach to reduce two rounds of messages in the two-phase commit process into one round as in [15].

Degraded writes: ECHash addresses the fault tolerance in accessing an object through degraded reads (§4.4) and node repair (§4.5). However, it may fail to write objects to a server via consistent hashing if the server is failed. In this case, ECHash can re-hash the object to another non-failed redirected server to temporarily store the object, and maintain the metadata information for the redirection in the proxy. When the failed server is restored, the

proxy can migrate any re-hashed object from the redirected servers to the restored server.

Metadata management: ECHash has low metadata overhead as shown in our evaluation (§5.5). Currently, we store the metadata in the proxy (§4.1). For fault tolerance, we may store and replicate the metadata in the servers.

5 EVALUATION

In this section, we present evaluation results on ECHash in both local and cloud settings. We show the performance gain of ECHash in scaling compared to state-of-the-arts.

5.1 Implementation

We implement ECHash atop the Memcached protocol [8], which is widely used in decentralized in-memory KV stores (e.g., Amazon ElastiCache [3]). Specifically, we extend `libMemcached` [7] with about 3,600 SLoC in C++ on Linux as the proxy that performs FragEC (§3), supports basic requests (§4.2), scaling (§4.3), degraded reads (§4.4), and node repair (§4.5). ECHash performs (n, k) cross-coding based on Reed-Solomon codes [41], where (n, k) is varied from $(5, 3)$ [15] to $(8, 6)$. Each data chunk is set to 4 KiB [15, 49] by default in coding. ECHash leverages two APIs of Intel's ISA-L [6] to accelerate the erasure coding computations: `ec_init_tables`, which specifies coding coefficients, and `ec_encode_data`, which specify encoding/decoding operations.

To show ECHash's performance improvements, we also extend Memcached and implement the cross-coding scheme used in existing erasure-coded Memcached systems (e.g., Cocytus [15] and BCStore [33]). We call this Memcached implementation with the existing cross-coding scheme ccMemcached. ccMemcached manages all nodes in a single hash ring, and records the metadata as in ECHash on the proxy side. Based on cross-coding, ccMemcached combines the objects that are distributed to the same node via consistent hashing into the fixed-size data chunks. It then encodes k data chunks that are stored in different nodes into $n - k$ parity chunks and writes them to other different nodes.

5.2 Summary of Findings

We evaluate the performance of Vanilla (i.e., the vanilla Memcached without erasure coding), ccMemcached, and ECHash in various operations, including basic I/O , scale-out and scale-in, degraded reads during scaling, and node repair. We conduct these experiments in both a local setting (i.e., a local testbed cluster) and a cloud setting (i.e., Amazon EC2 and Elasticache Memcached). In our experiments, we focus on evaluating the scaling operations based on erasure coding without considering replication, as replication incurs significantly more redundancy than erasure coding at the same reliability level (§2.2).

We highlight our evaluation results as follows:

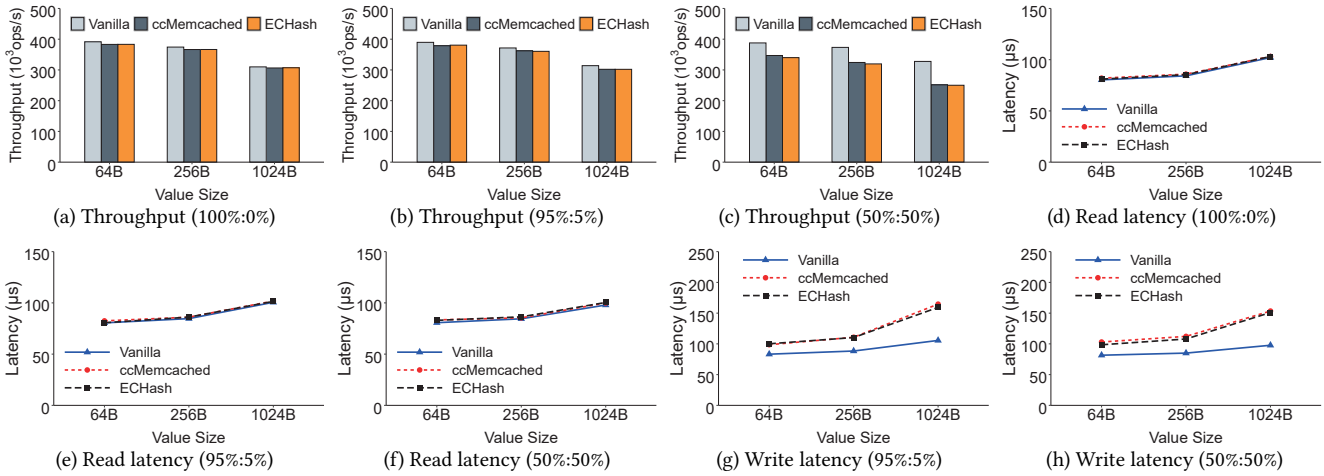
- ECHash incurs low overhead. It has similar throughput to Vanilla and ccMemcached, and incurs only a small increase in write latency compared to Vanilla.
- ECHash achieves high scale-out and scale-in performance. It achieves up to 8.3× (local) and 5.2× (cloud) of scale-out throughput compared to ccMemcached.
- ECHash accelerates degraded read operations during scaling. It reduces the degraded read latency during scaling by up to 81.1% (local) and 89.0% (cloud) compared to ccMemcached.
- ECHash has close node repair performance to ccMemcached. It smartly repairs the data stored in failed nodes based on fragments instead of chunks to maintain the similar node repair performance to ccMemcached.

5.3 Performance in the Local Setting

Setup: We conduct testbed experiments in a local environment with commodity configurations. To set up the architecture in §4.1, we deploy a cluster of seven physical machines, among which one machine acts as the proxy, one machine acts as the backup proxy, and the remaining five machines run Memcached instances that act as servers. Each physical machine runs Red Hat Enterprise Linux Server release 6.5 with the Linux kernel version 2.6.32. All physical machines are equipped with an 8-core Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40 GHz and 100 GiB of RAM, and are interconnected via a 10 Gb/s network. We also deploy an additional MySQL-5.1.71 database to store all the objects persistently in case both the read and degraded read operations fail. We configure 45 Memcached instances among the five servers, with the same configuration applied to Vanilla, ccMemcached, and ECHash.

Workloads: We use workloads generated by Yahoo! Cloud Serving Benchmark (YCSB) [18] to evaluate Vanilla, ccMemcached, and ECHash. Specifically, we set the size of the key for each object using the default setting in YCSB (which is a variable size of around 20 bytes). We set the size of the value of an object as 64 bytes, 256 bytes, and 1 KiB according to Facebook's Memcached workload; that is, 99% of Facebook Memcached request sizes are no more than 800 bytes, and the median size is 135 bytes [38]. We limit the total data size to 4 GiB and the numbers of 64-byte, 256-byte, and 1-KiB objects are 64 million, 16 million, and 4 million, respectively. We evaluate the systems with different read/write ratios (read:write), including read-only (100%:0%), read-mostly (95%:5%), and write-intensive (50%:50%) [15, 49].

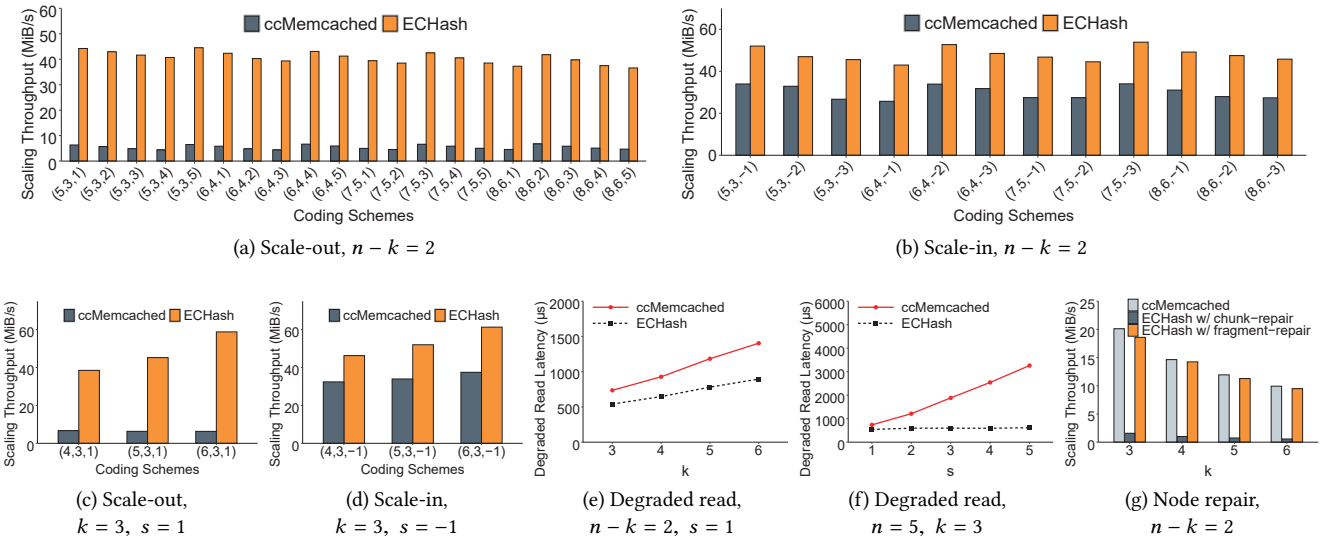
Experiment 1 (Basic I/O performance): We compare Vanilla, ccMemcached, and ECHash in terms of throughput (in number of operations per second), read and write latency under different read/write ratio workloads. Our goal is to show that ECHash maintains high basic I/O performance. Figures 11(a) and 11(b) show that ECHash has similar throughput performance for read-only and read-mostly scenarios compared to Vanilla and ccMemcached. Figure 11(c) shows that ccMemcached and ECHash incur small performance overhead under write-intensive scenarios compared to Vanilla, since Vanilla does not perform additional coding nor store parity chunks for data availability as opposed to ccMemcached and ECHash. Figures 11(d)-11(f) show that ECHash has similar read latency for read-only, read-mostly, and write-intensive scenarios compared to Vanilla and ccMemcached. Figures 11(g)-11(h) show that ccMemcached and ECHash have similar write latency for read-mostly and write-intensive scenarios, but both of them have higher write latency compared to Vanilla due to chunk encoding and hence a longer I/O path in the write operations.



(a) Throughput (100%:0%) (b) Throughput (95%:5%) (c) Throughput (50%:50%) (d) Read latency (100%:0%)

(e) Read latency (95%:5%) (f) Read latency (50%:50%) (g) Write latency (95%:5%) (h) Write latency (50%:50%)

Figure 11: Experiment 1: Comparison of throughput, read and write latency of three read/write ratio workloads in (5, 3) coding in the local setting.



(a) Scale-out, $n-k=2$ (b) Scale-in, $n-k=2$

(c) Scale-out, $k=3,\ s=1$ (d) Scale-in, $k=3,\ s=-1$ (e) Degraded read, $n-k=2,\ s=1$ (f) Degraded read, $n=5,\ k=3$ (g) Node repair, $n-k=2$

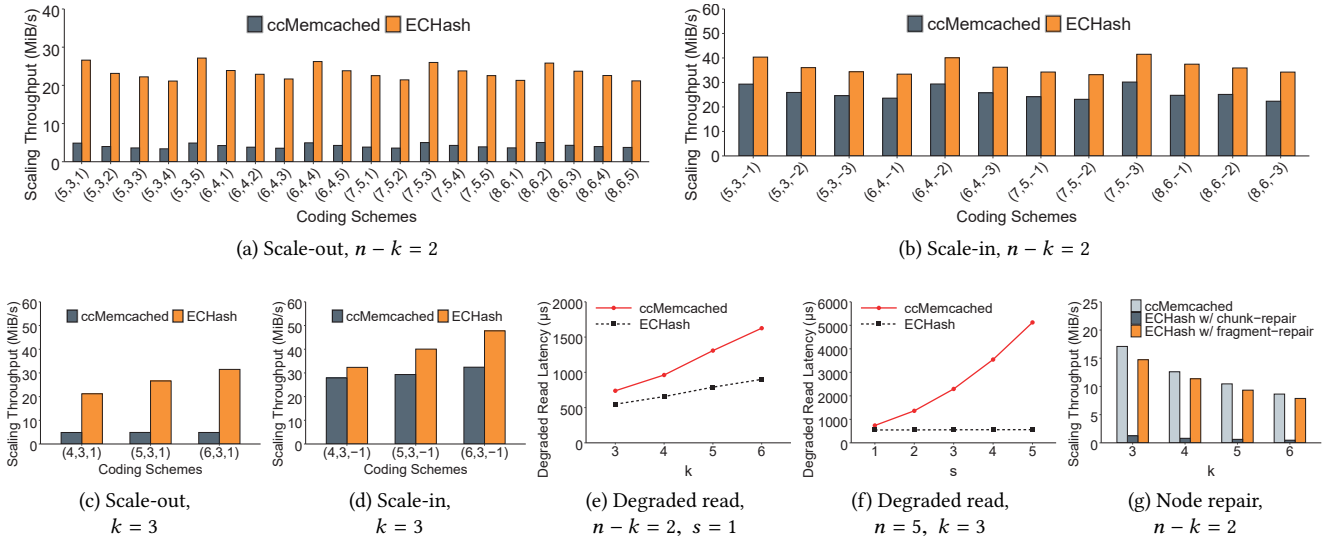
Figure 12: Experiments 2, 3, and 4: Comparisons of scaling throughput, degraded read latency, and node repair throughput in the local setting.

Experiment 2 (Scale-out/Scale-in performance): We compare ccMemcached and ECHash in terms of scaling throughput, which is equal to the amount of scaling traffic divided by the total running time. We consider different coding schemes (i.e., (n, k)) and different numbers of added/removed nodes (i.e., $\pm s$).

Figures 12(a) and 12(b) show the results of scale-out and scale-in for different coding schemes with $n-k=2$ (i.e., the maximum number of tolerable failed nodes is two). In Figure 12(a), the scale-out throughput of ECHash is significantly higher than that of ccMemcached, because the former does not consume any bandwidth and computation overhead to update parity chunks. For example, in (8, 6, 1)-scaling, the scaling throughput of ECHash is 8.3× compared to that of ccMemcached. In Figure 12(b), the scale-in throughput of ECHash is also higher than that of ccMemcached, but with the same values of n, k, and s, the improvement of $(n, k, -s)$-scaling is not as significant as that of (n, k, s)-scaling. For example, in $(8, 6, -1)$-scaling, the scaling throughput of ECHash is 67.1% higher than that of ccMemcached.

Figures 12(c) and 12(d) show the results of scale-out (i.e., $s=1$) and scale-in (i.e., $s=-1$) with different numbers of fault tolerance, in which $n-k$ ranges from 1 to 3. We find that compared to ccMemcached, the scaling throughput gain of ECHash increases with $n-k$. The reason is that a larger $n-k$ implies that more parity chunks are stored in a stripe, so ECHash, which eliminates the parity updates during scaling (Goal 1 in §4), can achieve a higher performance gain compared to ccMemcached.

Experiment 3 (Degraded read performance during scaling): We compare ccMemcached and ECHash in terms of the degraded



(a) Scale-out, $n-k=2$ (b) Scale-in, $n-k=2$

(c) Scale-out, $k=3$ (d) Scale-in, $k=3$ (e) Degraded read, $n-k=2,\ s=1$ (f) Degraded read, $n=5,\ k=3$ (g) Node repair, $n-k=2$

Figure 13: Experiment 5: Comparison of scaling throughput, degraded read latency, and node repair throughput in the cloud setting.

read latency during scaling, which is equal to the average latency of all the degraded read operations during scaling. Note that if the degraded read operation fails, then we continue to accomplish the degraded read operation by accessing the persistent MySQL database.

Figure 12(e) shows the degraded read latency results under different coding schemes (i.e., (n,k)) with $s=1$ (i.e., adding a new node). We see that ECHash incurs lower degraded ready latency than ccMemcached by 26.5-36.2%, since it can perform all the degraded read operations during scaling (Goal 2 in §4), while ccMemcached has to access the persistent MySQL database for failed degraded read operations. Note that the degraded read latencies of both ECHash and ccMemcached increase with k, mainly because a larger k means that we need more data chunks to decode the unavailable object in a stripe (§4.4).

Figure 12(f) shows the degraded read latency results under different numbers of added nodes (i.e., s) with $(n,k)=(5,3)$. We find that ECHash maintains the degraded read latency, because k keeps constant and a degraded read to an unavailable object can be done successfully by decoding it from k available chunks. In contrast, ccMemcached increases the degraded read latency when the number of added nodes increases since more added nodes may cause more object re-distribution. As a result, more degraded read operations may fail (§4.4). For example, under (5, 3, 5)-scaling. ECHash reduces the degraded read latency by 81.1% compared to ccMemcached.

Experiment 4 (Node repair performance): We compare ccMemcached and ECHash with chunk-repair and fragment-repair schemes in terms of the node repair throughput, which is equal to the storage capacity of a node over the repair time. We consider different values of k with $n-k=2$. Figure 12(g) shows that fragment-repair, which is proposed for the efficient repair of data objects (§4.5), significantly improves the node repair performance of chunk-repair. For example, fragment-repair increases the node repair throughput by up to 16.5× (when $k=6$). Note that compared to ccMemcached, ECHash with fragment-repair still has a small degradation of repair performance, ranging from 7.6% (when $k=3$) to 4.5% (when $k=6$). The reason is that the current fragment-repair scheme has not been applied to parity chunks yet (see our discussion in §4.5).

5.4 Performance in the Cloud Setting

Setup: We conduct experiments on Amazon Cloud (EC2 [2] and Elasticache [3]) to evaluate ECHash and ccMemcached in a cloud environment. To set up the architecture in §4.1, we deploy 45 `cache.r4.large` Memcached instances of Elasticache in US-West (North California) that act as the servers, two `m5d.2xlarge` instances of Amazon EC2 in US-West (North California) that act as the proxy and the backup proxy, and one `m5d.2xlarge` instance of Amazon EC2 that hosts a persistent MySQL database.

Experiment 5 (Performance in the cloud setting): Figures 13(a)-13(g) repeat the experiments of Figures 12(a)-12(g) in the cloud setting. Figures 13(a)-13(d) show that compared to ccMemcached, ECHash improves the scale-out throughput and the scale-in throughput by up to 5.2× (when $k=8$) and 53.2% (when $k=8$), respectively. We find that the improvements in the cloud setting are not as significant as in the local setting due to the following reason. Table 2 compares the average throughput and read/write latency in both the local and cloud settings, and we see that the performance in the cloud setting is generally worse than that in the local setting. Thus, the cloud setting slows down both object migration and parity updates in the node scaling process (§4.3). The object migration (which appear in both ccMemcached and ECHash) is totally composed of read and write operations, while the parity updates (which appear in ccMemcached only) includes parity chunk computation in addition to the read and write operations. Thus, in ccMemcached, the object migration slows down more significantly than the parity updates in the cloud setting, implying that ECHash, which eliminates the

Settings	Throughput (10^3 ops/s)	Latency (μs)	
		Read	Write
Local	366.5	81.0	110.3
Cloud	269.3	118.1	155.5

Table 2: Average throughput and read/write latency of ECHash **in both local and cloud settings.**

Value size	64 bytes	128 bytes	256 bytes	512 bytes	1 KiB
Ratio	31.7%	17.2%	9.1%	4.7%	2.5%

Table 3: Ratio of the metadata size to the object data size under $(5, 3)$ **coding.**

parity updates, has less improvement in the scaling performance in the cloud setting.

Figures 13(e) and 13(f) show that similar to the local setting, ECHash still outperforms ccMemcached in the cloud setting in the degraded read latency during scaling. For example, ECHash reduces the degraded read latency by 89.0% compared to ccMemcached in (5, 3, 5)-scaling. The improvement of ECHash in the cloud setting here is higher than in the local setting, since ccMemcached's MySQL queries become slower in the cloud setting, which has less available network bandwidth than in the local setting.

Figure 13(g) shows that similar to the local setting, the node repair performance of ECHash is close to that of ccMemcached.

5.5 Metadata Analysis

Recall that the metadata in ECHash includes the Object Index, the Chunk Index and the Stripe Index, all of which incur storage overhead on the proxy side. Thus, we evaluate the ratio of the metadata size to the object data size, in which we vary the value size. Table 3 shows that the ratio of the metadata size to the object data size decreases with the value size. For example, when the value size equals 1 KiB, the metadata only takes up 2.5% of the data size. Note that the metadata is mainly needed for object reconstruction from failures. Thus, the metadata can be stored in a secondary storage device, so as to further save memory resources.

6 RELATED WORK

Erasure coding in storage systems: Erasure coding has been extensively employed in distributed file systems [13, 25–27, 32, 37, 43, 44, 48, 50] and KV stores [15, 17, 33, 36, 40, 45, 49]. Prior studies on erasure coding focus on improving storage efficiency (e.g., [37]), repair performance (e.g., [26, 32, 43, 44]), update performance [13, 33], consistency [17], memory management [15, 25, 36, 40, 49], and scaling performance [27, 45, 48, 50]. In particular, to improve scaling performance in erasure-coded storage, previous studies focus on minimizing the scaling traffic in distributed storage systems [27, 48, 50] or simplifying key-to-node mappings in decentralized KV stores [45], yet they target the scaling operations that change the redundancy parameters (i.e., (n, k)). In contrast, ECHash keeps the redundancy parameters unchanged. Also, it targets decentralized KV stores based on consistent hashing.

The erasure coding models in the above studies assume that each data chunk is tightly coupled to a specific node. Our FragEC model decouples the relation between one data chunk and one node, and it is applicable for general erasure-coded storage systems for high scalability. To avoid rebuilding the whole data chunk during repair, we propose the fragment-repair algorithm by repairing the partial fragments in a data chunk.

Decentralized KV stores: There have been many studies on building decentralized KV stores. In particular, Dynamo [19] and Cassandra [31] are two well-known decentralized KV stores, both of which provide high availability and scalability via replication and consistent hashing (note that our discussion in this paper is based on the original Dynamo paper [19], while Amazon DynamoDB [1], the commercial database service at Amazon that builds on the principles of Dynamo, may have a different implementation). Memcached [8] is one well-known in-memory KV store that serves as a data caching solution in Facebook [38] and Twitter [11]. ECHash builds atop the Memcached protocol and uses consistent hashing for high scalability, but applies erasure coding (rather than replication) for high availability with low redundancy. Nevertheless, ECHash's design is applicable to state-of-the-art decentralized KV stores in general.

Recent studies have focused on optimizing the designs of decentralized in-memory KV stores in different aspects, such as memory efficiency [40, 49], parallelism [35], concurrency [20, 34], availability [15, 33], memory management [20, 24, 42], and load balancing [23, 40]. Some closely related studies to ours include: Facebook's Memcached [38], which scales memcached clusters with replication; ElMem [22], which proactively migrates hot data through scaling for power and operator cost savings; Ring [45], which proposes stretched Reed-Solomon coding to simplify key-to-node mappings when the redundancy parameters (n, k) change (see our discussion above). ECHash complements prior in-memory KV storage optimizations by improving scaling performance under consistent hashing in erasure-coded decentralized in-memory KV storage.

7 CONCLUSIONS

We study how to effectively apply erasure coding to decentralized KV stores based on consistent hashing, so as to provide low-cost fault-tolerant storage via erasure coding, while preserving the scaling feature (i.e., adding or removing nodes) via consistent hashing. We propose FragEC, a novel fragmented erasure coding model that supports efficiently scaling without incurring parity updates. We also design a new consistent hashing scheme based on multiple hash rings to support efficient degraded reads. To this end, we realize a FragEC-based KV store prototype system, ECHash, atop the Memcached protocol and further design a fragment-repair scheme to improve the node repair performance. Testbed experiments in both local and cloud settings demonstrate the efficiency of ECHash in basic I/O, scaling, degraded reads and node repair.

Acknowledgments: We thank our shepherd, Doug Terry, and the anonymous reviewers for their comments. This work was supported by the National Natural Science Foundation of China (61872414 and 61872411), Fundamental Research Funds for the Central Universities (2017KFYXJJ065), Alibaba Innovation Research, Key Laboratory of Information Storage System Ministry of Education of China, and Research Grants Council of Hong Kong (GRF 14216316 and CRF C7036-15G). The corresponding author is Yuchong Hu.

REFERENCES

[1] Amazon DynamoDB. https://aws.amazon.com/dynamodb.
[2] Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2.
[3] Amazon Elasticache. https://docs.aws.amazon.com/elasticache.
[4] AWS Autoscaling. https://aws.amazon.com/autoscaling.
[5] etcd. https://etcd.io.
[6] Intel ISA-L. https://github.com/01org/isal.
[7] LibMemcached. https://libmemcached.org.
[8] Memcached. https://memcached.org.
[9] Openstack. https://openstack.org.
[10] Openstack Swift. https://swift.org.
[11] Twemcache is the Twitter Memcached. https://twitter.com/twemcache.
[12] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proc. of ACM SIGMETRICS*, pages 53–64, 2012.
[13] J. C. Chan, Q. Ding, P. P. Lee, and H. H. Chan. Parity logging with reserved space: Towards efficient updates and recovery in erasure-coded clustered storage. In *Proc. of USENIX FAST*, pages 163–176, 2014.
[14] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. on Computer Systems*, 26(2):1–26, 2008.
[15] H. Chen, H. Zhang, M. Dong, Z. Wang, Y. Xia, H. Guan, and B. Zang. Efficient and available in-memory KV-store with hybrid erasure coding and replication. *ACM Trans. on Storage*, 13(3):25, 2017.
[16] M. Chen and E. Zadok. Kurma: Secure geo-distributed multi-cloud storage gateways. In *Proc. of ACM SYSTOR*, pages 109–120, 2019.
[17] Y. L. Chen, S. Mu, J. Li, C. Huang, J. Li, A. Ogus, and D. Phillips. Giza: Erasure coding objects across global data centers. In *Proc. of USENIX ATC*, pages 539–551, 2017.
[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. of ACM SoCC*, pages 143–154, 2010.
[19] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. of ACM SOSP*, pages 205–220, 2007.
[20] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. In *Proc. of USENIX NSDI*, pages 371–384, 2013.
[21] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *Proc. of USENIX OSDI*, pages 61–74, 2010.
[22] U. U. Hafeez, M. Wajahat, and A. Gandhi. ElMem: Towards an elastic Memcached system. In *Proc. of IEEE ICDCS*, pages 278–289, 2018.
[23] Y.-J. Hong and M. Thottethodi. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *Proc. of ACM SoCC*, page 13, 2013.
[24] X. Hu, X. Wang, Y. Li, L. Zhou, Y. Luo, C. Ding, S. Jiang, and Z. Wang. LAMA: Optimized locality-aware memory allocation for key-value cache. In *Proc. of USENIX ATC*, pages 57–69, 2015.
[25] Y. Hu, Y. Wang, B. Liu, D. Niu, and C. Huang. Latency reduction and load balancing in coded storage systems. In *Proc. of ACM SoCC*, pages 365–377, 2017.
[26] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in Windows Azure Storage. In *Proc. of USENIX ATC*, pages 15–26, 2012.
[27] J. Huang, X. Liang, X. Qin, P. Xie, and C. Xie. Scale-RS: An efficient scaling scheme for RS-coded storage clusters. *IEEE Trans. on Parallel and Distributed Systems*, 26(6):1704–1717, 2015.
[28] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proc. of USENIX ATC*, pages 1–14, 2010.
[29] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. of ACM STOC*, pages 654–663, 1997.
[30] C. Lai, S. Jiang, L. Yang, S. Lin, G. Sun, Z. Hou, C. Cui, and J. Cong. Atlas: Baidu's key-value storage system for cloud data. In *Proc. of IEEE MSST*, pages 1–14, 2015.
[31] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.
[32] R. Li, X. Li, P. P. Lee, and Q. Huang. Repair pipelining for erasure-coded storage. In *Proc. of USENIX ATC*, pages 567–579, 2017.
[33] S. Li, Q. Zhang, Z. Yang, and Y. Dai. BCStore: Bandwidth-efficient in-memory KV-store with batch coding. In *Proc. of IEEE MSST*, 2017.
[34] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proc. of ACM EuroSys*, page 27, 2014.
[35] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *Proc. of USENIX NSDI*, pages 429–444, 2014.
[36] W. Litwin, R. Moussa, and T. Schwarz. LH* RS: A highly-available scalable distributed data structure. *ACM Trans. on Database Systems*, 30(3):769–811, 2005.
[37] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, et al. f4: Facebook's warm blob storage system. In *Proc. of USENIX OSDI*, pages 383–398, 2014.
[38] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling Memcache at Facebook. In *Proc. of USENIX NSDI*, pages 385–398, 2013.
[39] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The Quantcast File System. *Proc. of VLDB Endowment*, 6(11):1092–1101, 2013.
[40] K. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran. EC-Cache: Load-balanced, low-latency cluster caching with online erasure coding. In *Proc. of USENIX OSDI*, pages 401–417, 2016.
[41] I. Reed and G. Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial & Applied Mathematics*, 8(2):300–304, 1960.
[42] T. Saemundsson, H. Bjornsson, G. Chockler, and Y. Vigfusson. Dynamic performance profiling of cloud caches. In *Proc. of ACM SoCC*, pages 1–14, 2014.
[43] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing elephants: Novel erasure codes for big data. In *Proc. of VLDB Endowment*, volume 6, pages 325–336, 2013.
[44] M. Silberstein, L. Ganesh, Y. Wang, L. Alvisi, and M. Dahlin. Lazy means smart: Reducing repair bandwidth costs in erasure-coded distributed storage. In *Proc. of ACM SYSTOR*, pages 1–7, 2014.
[45] K. Taranov, G. Alonso, and T. Hoefler. Fast and strongly-consistent per-item resilience in key-value stores. In *Proc. of ACM EuroSys*, page 39, 2018.
[46] M. Vrable, S. Savage, and G. M. Voelker. Bluesky: A cloud-backed file system for the enterprise. In *Proc. of USENIX FAST*, pages 19–19, 2012.
[47] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proc. of Springer International Workshop on Peer-to-Peer Systems*, pages 328–337, 2002.
[48] S. Wu, Y. Xu, Y. Li, and Z. Yang. I/O-efficient scaling schemes for distributed storage systems with CRS codes. *IEEE Trans. on Parallel and Distributed Systems*, 27(9):2639–2652, Sep 2016.
[49] M. M. Yiu, H. H. Chan, and P. P. Lee. Erasure coding for small objects in in-memory KV storage. In *Proc. of ACM SYSTOR*, page 14, 2017.
[50] X. Zhang, Y. Hu, P. P. Lee, and P. Zhou. Toward optimal storage scaling via network coding: From theory to practice. In *Proc. of IEEE INFOCOM*, pages 1808–1816, 2018.