Erasure Coding for Small Objects in In-Memory KV Storage

Matt M. T. Yiu, Helen H. W. Chan, and Patrick P. C. Lee
Department of Computer Science and Engineering, The Chinese University of Hong Kong
{mtyiu, hwchan, pclee}@cse.cuhk.edu.hk

ABSTRACT

We present MemEC, an erasure-coding-based in-memory key-value (KV) store that achieves high availability and fast recovery while keeping low data redundancy across storage servers. MemEC is specifically designed for workloads dominated by small objects. By encoding objects in entirety, MemEC is shown to incur 60% less storage redundancy for small objects than existing replication- and erasure-coding-based approaches. It also supports graceful transitions between decentralized requests in normal mode (i.e., no failures) and coordinated requests in degraded mode (i.e., with failures). We evaluate our MemEC prototype via testbed experiments under read-heavy and update-heavy YCSB workloads. We show that MemEC achieves high throughput and low latency in both normal and degraded modes, and supports fast transitions between the two modes.

CCS Concepts

•**Information systems** → **Key-value stores; Storage recovery strategies;** •**Software and its engineering** → *Distributed memory;*

Keywords

Erasure coding; In-memory storage

1. INTRODUCTION

Memory-centric storage systems have been proposed to keep primary data in memory to enable scalable, low-latency data access. They are often realized as *in-memory key-value (KV) stores*, which organize data in memory as KV pairs (called *objects*) to form structured storage. Enterprises have deployed in-memory KV stores for low-latency operations in social networking, web search, and analytics (e.g., [3,14,40]).

Failures are prevalent in distributed storage [28]. To maintain data availability, most existing in-memory KV stores replicate data copies across storage servers [2,4,8,10,35,40].

However, replication incurs high storage overhead and multiplies the memory cost, which is still much higher than the cost of traditional disk-based storage. RAMCloud [41] keeps replicas in secondary storage for persistence, yet accessing secondary storage for failure recovery incurs high latency, especially for random I/Os (e.g., the typical hard disk seek time is around 10ms).

This motivates us to explore *erasure coding* for in-memory KV storage. Erasure coding transforms data into fixed-size encoded chunks, such that the original data can be recovered from a subset of encoded chunks. It achieves the same fault tolerance with much lower redundancy than replication [48]. Thus, we can leverage erasure coding to keep minimum data redundancy *entirely* in memory. Failure recovery can then be done by directly accessing other working memory-based storage servers for encoded chunks, thereby maintaining low access latency. A drawback of erasure coding is its high performance overhead in terms of network bandwidth and disk I/O, especially in data updates and failure recovery. Thus, extensive studies focus on solving the performance issues of erasure-coded storage (e.g., [31, 34, 46]).

To deploy erasure coding in in-memory KV storage, we argue that two specific challenges need to be addressed in addition to mitigating overheads in data updates and failure recovery. First, Facebook's field study [15] shows that real-life KV storage workloads are dominated by *small objects*, whose keys and values are of small sizes (e.g., ranging from few bytes to tens or hundreds of bytes). In particular, one workload has keys with either 16 or 21 bytes and almost all values with 2 bytes only; another workload has up to 40% of values with only 2, 3, and 11 bytes [15]. It is infeasible to perform erasure coding directly on an extremely small object (e.g., 2 bytes), as we need to first decompose the object into chunks before encoding (§2). Second, to minimize access latency, in-memory KV stores issue decentralized requests without centralized metadata lookups. However, when failures happen, ongoing requests may need to be reverted or replayed to avoid inconsistency. In erasure coding, we also need to maintain consistency across encoded chunks that are dependent on each other.

We present MemEC, an erasure-coding-based in-memory KV store with access performance, storage efficiency, and fault tolerance in mind. MemEC works as a *high-availability distributed cache* that supports low-latency data access for read-intensive [15] or update-intensive [16, 47] workloads, and also enables fast recovery by accessing data redundancy from other working in-memory storage servers without the need of accessing secondary storage.

MemEC supports small objects via a new *all-encoding* data model, which encodes objects in entirety, including keys, values, and metadata, so as to significantly reduce storage redundancy for fault tolerance. We carefully design the index structures for the all-encoding data model to reduce memory footprints, such that all objects and index structures are kept entirely in memory. Our analysis shows that our all-encoding data model saves up to 60% of redundancy over the replication- and erasure-coding-based approaches used by existing in-memory KV stores. See §3.

MemEC allows decentralized requests for reads, writes, updates, and deletes in normal mode (i.e., no failures), which is the common case in practice. It ensures graceful transitions between normal mode and degraded mode (i.e., with failures), such that each request in degraded mode will be centrally coordinated and redirected from a failed server to another working server. It also maintains both availability and consistency during transitions. See §4 and §5.

We implement a MemEC prototype that can be fully deployed on commodity hardware in a cloud environment. We conduct testbed experiments under read-heavy and update-heavy workloads generated by YCSB [23]. In normal mode, MemEC achieves comparable performance to both Memcached [7] and Redis [8] (e.g., latencies are in the range of few milliseconds). It also efficiently operates in degraded mode, and completes transitions between normal and degraded modes within milliseconds. See §6 and §7.

The source code of our MemEC prototype is available at **https://github.com/mtyiu/memec**.

2. ERASURE CODING

Erasure coding is typically constructed by two configurable parameters n and k (where $k < n$). We treat data as a collection of fixed-size units called *chunks*. Every k original chunks (called *data chunks*) are encoded into $n-k$ additional equal-size coded chunks (called *parity chunks*), such that the set of the n data and parity chunks is called a *stripe*. We consider erasure codes that are *maximum distance separable (MDS)*, meaning that the encoding operations ensure that any k of the n data and parity chunks of the same stripe can sufficiently decode the original k data chunks, while incurring the minimum storage redundancy (i.e., $\frac{n}{k}$ times the original data size). Reed-Solomon (RS) codes [44] are one well-known example of MDS erasure codes.

A storage system contains multiple stripes of data that are independently encoded, and hence our analysis focuses on a single stripe. The n chunks of each stripe are stored in n distinct servers to tolerate server failures. We call the server that stores a data (parity) chunk a *data (parity) server*. Since each stripe is stored in a different set of servers, a server may act as either a data server or a parity server for different stripes; in other words, the naming of a data or parity server is logical.

Each update to a data chunk will trigger an update to every parity chunk of the same stripe. To mitigate the network overhead of parity updates, we leverage the *linearity* property of erasure coding without transferring any existing parity chunk for the update. For example, RS codes encode data chunks by linear combinations based on Galois Field arithmetic. Specifically, a parity chunk (denoted by P) is encoded by k data chunks (denoted by $D_1, D_2, \cdots, D_k$) via a linear combination as $P = \sum_{i=1}^{k} \gamma_i D_i$ for some encoding coefficients γ_i's ($1 \leq i \leq k$). Suppose that a data chunk

D_i is now updated to D_i'. We can compute the new parity chunk as $P' = P + \gamma_i(D_i' - D_i)$, where $(D_i' - D_i)$ is called the *data delta*. Thus, when a data server applies an update to a chunk, it sends the data delta directly to each parity server, which can then use the data delta to compute the new parity chunk. Previous studies have exploited the linearity property for parity updates in RAID [18] and distributed storage [12, 16, 50].

Erasure coding generally requires the chunk size be large enough. For example, RS codes require $n \leq 2^w - 1$, where w is the number of bits of a chunk [43]. To deploy RS codes, it is typical to set w as a multiple of 8 to align with machine word boundaries; in other words, a chunk needs to be at least one byte long. For an extremely small object (e.g., 2 bytes long [15]), dividing it into k (e.g., $k > 2$) data chunks for encoding can be infeasible.

3. DATA MODEL

In this section, we propose a data model for small objects in in-memory KV stores. Each object in a KV store comprises three fields: a key, a value, and metadata, such that both the key and value have variable sizes, while we assume that the metadata has a fixed size for simplicity. The key is a unique identifier for an object; the value holds the actual content of the object; the metadata holds the object attributes such as the key size, the value size, creation/modification timestamps, etc. Also, the KV store has an *index structure* that holds the references to all objects for object retrievals.

3.1 Existing Data Models

We revisit two existing data models that achieve fault tolerance via redundancy, namely all-replication and hybrid-encoding. Both data models have been realized in existing in-memory KV stores.

All-replication stores multiple replicas for each object, including the key, value, metadata, and reference to the object. It has been used in Facebook's Memcache [40], DXRAM [35], RAMCloud [41], and various open-source in-memory KV stores (e.g., [2, 4, 8, 10]). If there is a failed object, the KV store retrieves its replica from another working server. Most KV stores directly store replicas in memory. To reduce memory footprints, RAMCloud stores replicas in secondary storage, but this incurs expensive I/Os when there are intensive random reads to failed objects.

Hybrid-encoding combines erasure coding and replication. It applies erasure coding to the value only, while applying replication for the key, metadata, and reference to the object. Its rationale is that if the value size is significantly large, applying erasure coding to values only suffices to reduce storage redundancy. Specifically, hybrid-encoding performs (n, k) erasure coding across object values, such that the values of multiple objects are combined to form a data chunk, and the k data chunks in the same stripe are encoded to $n - k$ parity chunks. For each object, hybrid-encoding replicates the key, metadata, and any index information across $n - k + 1$ servers, including the data server that stores the object (which is embedded in the data chunk) and the $n-k$ parity servers. Thus, every object can tolerate any $n-k$ server failures. Hybrid-encoding has been adopted by existing KV stores, such as LH*RS [38], Atlas [36], and the recently proposed in-memory KV store Cocytus [50]. In particular, Cocytus focuses on the value size of at least 1KB
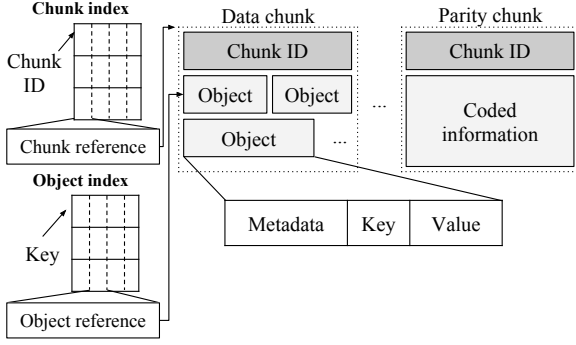
Figure 1: All-encoding data model. Each server keeps a local copy of both object and chunk indexes, while the chunks are distributed across servers.

in evaluation [50]. The drawback of hybrid-encoding is that if the value size is very small, it still incurs significant storage overhead due to the replication of keys and metadata.

3.2 Our Data Model

We now propose a new data model called **all-encoding**, whose idea is to apply erasure coding to objects *in entirety* (including keys, values, and metadata) without replication. In addition, we carefully design our index structure to limit its storage overhead.

Data organization: Figure 1 shows the data organization of the all-encoding model. We organize data as fixed-size chunks, which form the units of erasure coding. A larger chunk size provides better storage efficiency by mitigating the storage overhead due to indexing (see details below), yet it also incurs higher computational overhead in encoding/decoding. Our current implementation chooses 4KB as the default chunk size to balance the trade-off. For each chunk, we prepend a unique fixed-size *chunk ID* (8 bytes in our case) for chunk identification in a server.

All-encoding performs erasure coding across objects in entirety. Suppose that every object is small, such that its total size is smaller than the chunk size (i.e., the object can fit into a single chunk). Thus, each data chunk can contain multiple objects, each of which starts with metadata, followed by a key and a value. We append new objects to a data chunk until the chunk size limit is reached, and *seal* the data chunk so that no new objects can be further added. Sealed data chunks will later be encoded to form parity chunks belonging to the same stripe.

For indexing, we introduce two hash tables: the *chunk index*, which maps a chunk ID to a chunk reference, and the *object index*, which maps a key to an object reference. We emphasize that we do not need to keep redundancy for both indexes for fault tolerance; instead, each server only keeps a local copy of both indexes. The reason is that when a server fails, we can reconstruct the failed chunks from other available servers by erasure coding, and rebuild both indexes by reinserting the references of the reconstructed objects and chunks into the object and chunk indexes, respectively. In other words, the redundancy of both indexes is implicitly included in chunk-level redundancy.

To improve storage efficiency of both indexes, we apply *cuckoo hashing* [42], which has constant lookup and amortized update times. Its idea is to use two hash functions to map a key to two possible locations (called *buckets*). Each

inserted key is stored in either free bucket; if no free bucket is available, we relocate any existing key to make room for the inserted key. We set both indexes to be 4-way set-associative (i.e., each bucket contains four slots, each storing a reference), and it is shown that the space utilization can reach over 90% [26]. Currently, we do not consider range queries, which can be supported by tree-based indexes [49].

Chunk ID: To support updates and recovery, we need to locate the data and parity chunks of the same stripe. We leverage the chunk ID for this purpose. We decompose a chunk ID into three fields: (i) *stripe list ID*, which identifies the set of n data and parity servers that stores the stripe associated with the chunk (§4.2), (ii) *stripe ID*, which identifies a stripe in the storage system, and (iii) *chunk position*, which numbers the position of the chunk in a stripe from 0 to $n-1$. In particular, the data and parity chunks of the same stripe will have the same stripe list ID and same stripe ID. To obtain a stripe ID, each server maintains a local counter (initialized from zero) for each stripe list. Each time when a chunk is sealed, the data server sets the stripe ID as the current counter value, followed by incrementing the counter value by one. Each parity server encodes the data chunks of the same stripe into a parity chunk, which inherits the stripe list ID and stripe ID from the data chunks.

We use the chunk ID and both object and chunk indexes to locate an object. We align all fixed-size chunks (including the 8-byte chunk ID and 4KB chunk content) in the address space of each server. To update an object, we locate the object in the data server via the object index, and then obtain the corresponding chunk ID that is stored at the head of the chunk. Using the chunk ID, we identify each corresponding parity server, in which we can locate the parity chunk of the same stripe via the chunk index.

We also need to maintain the mappings between each key and chunk ID, so that when we reconstruct a failed object during a server failure, we can identify its associated chunk ID and retrieve the chunks of the same stripe in other servers for reconstruction. Nevertheless, the mappings can be stored in secondary storage, since they are only needed for recovery from failures. Thus, the mappings do not add overhead to in-memory storage. We elaborate how we maintain key-to-chunk mappings in §5.3.

Handling large objects: Some workloads can have objects with large value sizes (e.g., 1MB for the ETC workload in Facebook [15]). We can extend our data model to store a large object that cannot fit into a chunk, assuming that the key size remains small. We divide an object into *fragments*, each of which has the same size as a chunk (except the last fragment whose size may be less than the chunk size). We store each fragment as an object, and include an offset field in the object's metadata to specify the position of each fragment. We encode the fragments, together with other chunks in the same stripe, via erasure coding. Note that all fragments keep both key and metadata; in particular, the same key is replicated across fragments. However, since the value size is now large, the storage overhead incurred by the key and metadata becomes low.

3.3 Analysis

We analyze the redundancy overhead of the above data models. Our analysis makes the following assumptions. We only focus on the redundancy incurred for fault tolerance. Also, for all-replication and hybrid-encoding, we only in-

(a) $K=8$, $(n,k)=(10,8)$ (b) $K=32$, $(n,k)=(10,8)$

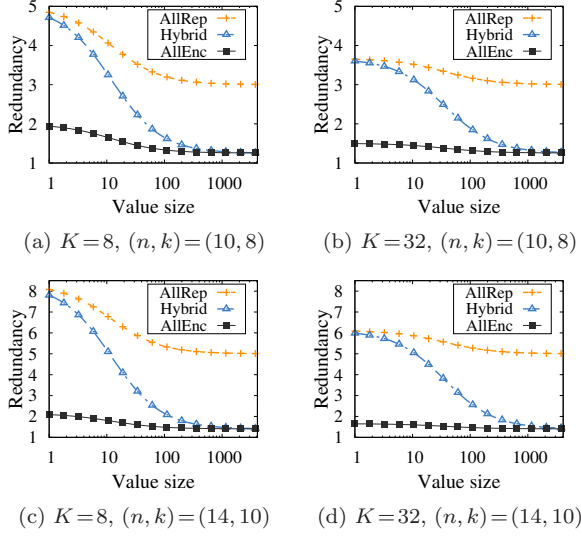(c) $K=8$, $(n,k)=(14,10)$ (d) $K=32$, $(n,k)=(14,10)$

Figure 2: Redundancy of all-replication (AllRep), hybrid-encoding (Hybrid), and all-encoding (AllEnc).

clude the index overhead for locating objects that are stored locally, while excluding their index overhead for maintaining the correlation between the original data and its redundancy since such implementation varies across systems. Thus, our analysis gives *underestimates* for their redundancy overhead. On the other hand, for all-encoding, our analysis includes the index overhead due to chunk IDs and both object and chunk indexes, while excluding the key-to-chunk mappings as the latter can be stored in secondary storage and only used when failures happen (§3.2). For simplicity, our analysis assumes fixed key and value sizes.

Let K, V, M, and R be the key size, value size, metadata size, and the size of a reference, respectively. We quantify the redundancy of a data model as the ratio of the actual storage size of an object with redundancy to the original size of the object without redundancy incurred by fault tolerance (i.e., $K+V+M$).

All-replication: To tolerate $n-k$ failures, we make $n-k+1$ copies for each object. The redundancy is:
$$\frac{(n-k+1)(K+V+M+R)}{K+V+M}.$$

Hybrid-encoding: To tolerate $n-k$ failures, we replicate the key, metadata, and reference of each object with total size $(n-k+1)(K+M+R)$ and encode the value by erasure coding with total size $\frac{nV}{k}$. The redundancy is:
$$\frac{(n-k+1)(K+M+R)+\frac{nV}{k}}{K+V+M}.$$

All-encoding: We encode the key, value, and metadata of each object in a data server and $n-k$ parity servers with total size $\frac{n(K+V+M)}{k}$. The data server also stores a reference to the object in its object index with total size $\frac{R}{O}$, where O is the occupancy of the object index due to cuckoo hashing. Each chunk also has a chunk ID, which we assume has size I, and a reference in the chunk index with total size $\frac{R}{O}$ (also due to cuckoo hashing). Let C be the chunk size, so the n chunks of a stripe store $\frac{kC}{K+V+M}$ objects on average. Thus, the average storage size of the chunk ID and chunk reference

per object is $\frac{n(I+R/O)}{kC/(K+V+M)}$. The redundancy is:
$$\frac{\frac{n(K+V+M)}{k}+\frac{R}{O}+\frac{n(I+R/O)}{kC/(K+V+M)}}{K+V+M}.$$

Numerical results: We analyze the redundancy of all data models for different K, V, and (n,k). We fix $M=4$ bytes (assuming that the metadata only holds a 1-byte key size and 3-byte value size), $R=8$ bytes, $C=4$KB, $I=8$ bytes (for a chunk ID in all-encoding), and $O=0.9$ (§3.2).

Figure 2 presents the results for $K=8$ and 32 bytes, $(n,k)=(10,8)$ and $(14,10)$ (note that $(14,10)$ is used by Facebook's erasure coding [39]), and different values of V (in bytes). All-encoding significantly reduces redundancy especially for small key and value sizes. For example, when $K=8$, $V\le 10$, and $(n,k)=(10,8)$ (Figure 2(a)), all-replication and hybrid-encoding achieve 4.1–4.8× and 3.3–4.7× redundancy, respectively, while all-encoding reduces the redundancy to 1.7–1.9× (up to 60.0% and 58.9% reduction, respectively). We make similar findings for $K=32$ and $(n,k)=(14,10)$. Both hybrid-encoding and all-encoding are approaching $\frac{n}{k}$ of redundancy as V increases, but all-encoding is clearly faster. For example, when $(n,k)=(10,8)$ and $K=8$, the redundancy of all-encoding drops to below 1.3 (4% over $\frac{n}{k}=1.25$) when $V\ge 180$, while hybrid-encoding achieves the same value when $V\ge 890$.

4. MEMEC OVERVIEW

4.1 System Architecture

MemEC is an in-memory KV store that manages the storage of objects with the all-encoding data model. Figure 3 shows the MemEC architecture, which comprises three types of nodes: multiple *servers*, multiple *proxies*, and a *coordinator*. Each server represents a storage node and allocates a memory region for storage. It is attached to persistent secondary storage (e.g., local disks or networked file systems). We aggregate the memories of all servers to a unified in-memory storage pool. Each proxy is a front-end interface for client applications to access objects in servers. The coordinator manages the architecture, detects failures (e.g., by periodic heartbeats), and coordinates I/O requests in the presence of failures. Our current prototype uses a single coordinator for simplicity, yet we can synchronize its state to multiple backup coordinators or external reliable storage for fault tolerance. We can also deploy a distributed coordinator service via Zookeeper [32] for improved performance.

MemEC maintains data availability due to server failures, in which in-memory storage becomes unavailable. When there is no server failure, MemEC operates in *normal mode*; when a server fails, the coordinator notifies all working proxies and servers to switch to *degraded mode* and coordinates the reconstruction of any failed data. In normal mode, MemEC distributes objects from each proxy across servers in a decentralized manner without involving the coordinator, as in existing in-memory KV stores [27, 41]. In contrast, in degraded mode, the coordinator is included into the I/O path and responsible for redirecting read/write requests away from any failed server. Note that if a request does not involve any failed server, it is still executed in a decentralized manner. Our design choice builds on the fact that the common case in practice is that MemEC runs in normal mode and uses decentralized requests for most of the
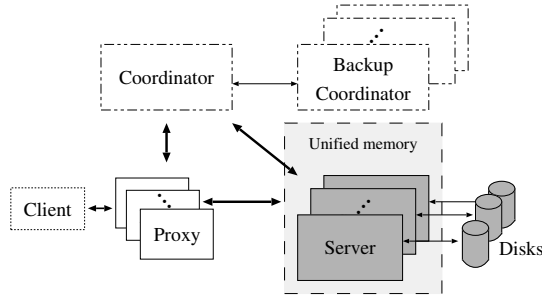
Figure 3: MemEC architecture.

time, while the coordinator is only involved in the I/O path when failures happen. In §5, we elaborate our failure model, and state additional design assumptions for fault tolerance.

MemEC supports linearizability. It keeps each object in only one working server in both normal and degraded modes. All servers process requests in a first-in-first-out manner, so that each read to an object always returns its value of the last write.

4.2 Stripe Management

In normal mode, MemEC distributes objects across servers in a decentralized manner. During bootstrap, MemEC generates a pre-configured number (denoted by c) of *stripe lists*, each of which defines a set of k data servers and $n-k$ parity servers of an erasure-coded stripe. Then a proxy maps an object to a server via two-stage hashing: it first hashes the object's key to one of the stripe lists, followed by hashing the object's key to one of the data servers in the stripe list.

To generate the stripe lists, one concern is that in write-intensive workloads, each write to a data server triggers a write to each parity server of the same stripe, so a parity server receives k times more write load than a data server (assuming that all data servers receive the same amount of write loads). Thus, we consider an algorithm that iteratively generates c stripe lists, with an objective of balancing the write loads across all servers. Specifically, we assign a load variable, initialized to be zero, to each server in MemEC. In each iteration i (where $1 \leq i \leq c$), we select $n-k$ servers with the smallest loads as the parity servers, followed by k servers with the next smallest loads as data servers (we break any tie by choosing servers with smaller IDs). We form the i-th stripe list with the n selected servers. We increment the load of each data server by 1 and that of each parity server by k. Finally, the algorithm outputs c stripe lists, which are installed into all proxies and servers. Note that the algorithm is executed only once during system startup, and runs in polynomial time with respect to the number of servers and the number of stripe lists. Thus, it has limited overhead to overall system performance.

4.3 Basic Requests

MemEC realizes four basic requests of practical KV stores: SET, GET, UPDATE, and DELETE. Here, we focus on small objects, and handle large objects as described in §3.2. Also, we focus on normal mode, and address degraded mode in §5.

SET inserts a new object into storage. A proxy first selects one data server and $n-k$ parity servers as described in §4.2. It sends the object to these servers in parallel, each of which returns an acknowledgement upon receiving the object. The data server appends the object to a data chunk that is not

yet full (i.e., *unsealed*), while each parity server keeps the object in a temporary buffer as a replica. The data server also adds the object reference to its own object index. Note that the data and parity servers do not need to communicate with each other upon a SET request. When the data chunk is full, it is *sealed* (§3.2). The data server then sends the keys of the objects in the sealed data chunk to all parity servers. Upon receiving the keys, each parity server rebuilds the data chunk from the objects in its temporary buffer (as the objects may not arrive in the same order), computes the data delta and updates its parity chunk (§2), and finally discards the objects from its temporary buffer.

In essence, our SET operation performs $(n-k+1)$-way replication until a chunk is sealed, and then applies erasure coding to reclaim the space of replicas. The transition from replication to erasure coding only involves the transmissions of object keys, and hence poses limited bandwidth overhead.

Each server is initialized with a pre-configured number of chunks based on the available storage capacity, and maintains a list of currently unsealed data chunks during execution. When it receives a new object via a SET request and is selected as the data server for storing the object, it appends the object to the unsealed data chunk that has the minimum remaining free space sufficient for holding the object, so as to fill up the unsealed data chunk as soon as possible. If all unsealed data chunks cannot store the new object, the server will pick the unsealed data chunk with the least free space to seal, so as to make a free unsealed data chunk available. Since we fix the number of unsealed data chunks, it also ensures that the number of replicas in the temporary buffer (for unsealed data chunks) of each parity server is small, thereby keeping the storage redundancy low.

GET retrieves an object from storage. A proxy uses the key to determine the data server that stores the object, and requests the data server for the object's value.

UPDATE modifies an existing object with a new value. A proxy first identifies the data server that stores the object, and sends the new value to the data server. The data server then updates the object's value and forwards the data delta to all parity servers of the same stripe; the set of parity servers can be determined by the chunk ID (§3.2). Each parity server either updates the corresponding parity chunk if the object is in a sealed data chunk, or the replica in the temporary buffer if the object is in an unsealed data chunk. After all parity servers finish the update, the data server returns an acknowledgement to the proxy.

Here, we allow different value sizes across objects, yet we assume that the value size of an object remains unchanged across updates. This keeps the size of a sealed data chunk unchanged, thereby simplifying storage management. How to adjust the object's value size on-the-fly is a future work.

DELETE removes an object from storage. A proxy requests the data server that stores the object to mark the object as deleted. If the object is in a sealed data chunk, the data server sends the data delta to all parity servers of the same stripe as in UPDATE, by treating the new object's value as zero. Each parity server updates the parity chunk accordingly. Otherwise, the data server notifies each parity server to remove its replica from the temporary buffer. The data server returns an acknowledgement to the proxy after the data server and all parity servers complete the deletion. We can later reclaim the space of the deleted objects.

5. FAULT TOLERANCE

In this section, we describe how MemEC achieves fault tolerance, and in particular, ensures graceful transitions between normal and degraded modes.

5.1 Failure Model

Our failure model focuses on server failures that make data unavailable. Server failures can be *transient* or *permanent*. Transient failures are temporary and do not incur actual data loss, such as due to server overloads and network congestion; in contrast, permanent failures incur actual data loss. When transient failures occur, MemEC reconstructs the unavailable data from other in-memory working servers for fast recovery; in contrast, when permanent failures occur, MemEC can either recover lost data from other working servers or from secondary storage (Figure 3). We require that the coordinator be responsible for detecting server failures, say by periodic heartbeats (§4.1). In this work, we focus on how MemEC reacts when transient server failures happen and performs fast recovery.

We do not consider proxy failures when MemEC is in degraded mode (i.e., server failures exist). In this case, we assume that MemEC reconstructs all available data from secondary storage. Nevertheless, our evaluation shows that resolving inconsistencies is fast (less than 700ms), so it is relatively rare for this special case to happen. On the other hand, when there is no server failure, each proxy only acts as an interface for clients to access servers. If a proxy fails, clients can access objects in servers through a different proxy. We also do not consider coordinator failures, as we can keep backups for the coordinator's states (§4.1).

5.2 Server States

To maintain availability and consistency, MemEC requires that all requests that involve a failed server (called *degraded requests*) be managed by the coordinator, which redirects a request from any failed server to a different working server (called *redirected server*). However, making a graceful transition from normal mode to degraded mode is non-trivial. Specifically, there may be ongoing requests that *inconsistently* update the data and parity chunks of a stripe. For example, in an `UPDATE` request, a data server sends a data delta to all parity servers of the same stripe (§4.3). If the data server fails, we cannot guarantee that all parity servers receive the data delta and update their parity chunks, since the failed data server is the only entity that keeps track of the parity updates. In this case, MemEC needs to resolve any potential inconsistency caused by the incomplete request.

To address the above challenges, in MemEC, the coordinator maintains *server states* to explicitly specify the operational status of each server. Figure 4 shows the server state diagram, which specifies four states:

- **Normal state**: It is the initial state of all servers, in which all requests are decentralized (§4.3).
- **Intermediate state**: It is the state when a server fails while data inconsistency is not yet resolved.
- **Degraded state**: After data inconsistency is resolved, all proxies issue degraded requests in collaboration with the coordinator.
- **Coordinated normal state**: When the failed server is restored, the coordinator instructs the redirected server to migrate any changes made in the degraded state to the
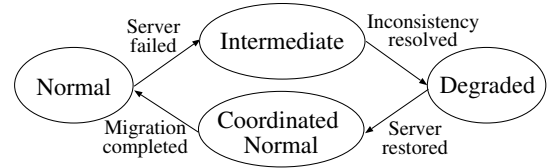

Figure 4: State diagram for failure handling.

restored server. All proxies still issue requests through the coordinator before the migration is completed.

Based on the current state of a server, a proxy decides whether decentralized or degraded requests should be issued to the server; for degraded requests, other working servers are also involved to reconstruct failed data. Thus, we require that *all proxies and working servers share the same view of the server states*. To do this, we need a two-phase protocol when the coordinator detects a server failure: the coordinator first notifies all proxies to complete processing requests in normal mode; after all proxies have confirmed that they have completed all normal requests, the coordinator notifies all proxies to start processing requests in degraded mode. We leverage *atomic broadcast* to realize the two-phase protocol, so that the notifications from the coordinator in both phases are reliably delivered to all proxies.

We emphasize that MemEC only uses atomic broadcast to reliably notify all proxies about state transitions in the presence of failures, but not for handling normal and degraded requests. Also, when a server fails, we do not require all proxies to transit from normal mode to degraded mode atomically (even they have all received the atomic broadcast message); instead, the failed server switches to the intermediate state, in which the coordinator notifies all proxies via atomic broadcast that they no longer need to issue decentralized requests to the failed server. After all proxies resolve inconsistencies of incomplete requests (§5.3) and notify the coordinator, the coordinator will issue another atomic broadcast to notify all proxies that the server is safe to switch to the degraded state, in which all requests are now handled through the coordinator. We elaborate the details in the following discussion. Our evaluation shows that the state transition overhead is small (§7).

5.3 Backups

MemEC resolves inconsistency when a failed server is in the intermediate state. To achieve this, it makes backups for ongoing requests, so as to revert inconsistent changes or replay incomplete requests. We emphasize that such backups are only temporarily kept, and will be cleared when data consistency is achieved. MemEC keeps three types of backups, as explained below.

Each proxy makes backups for unacknowledged requests, so that incomplete requests due to the failed server can be replayed later as degraded requests. Specifically, a proxy locally buffers each request that is sent to a server, and clears the buffered request upon receiving an acknowledgment.

Each proxy also makes backups for the mappings between keys and chunk IDs generated by servers to facilitate the reconstruction of failed objects. We require that each data server periodically checkpoints the mappings between keys and chunk IDs to the coordinator for recovery (§3.2). When each data server returns an acknowledgement to a proxy request, it also piggybacks the mapping between the ob-

ject's key and the chunk ID associated with the request. The proxy then buffers the mapping. When a server issues a new checkpoint, it acknowledges each proxy to clear its buffered mappings. During a server failure, each proxy sends its buffered mappings to the coordinator for reconstructing the mappings from the latest checkpoint.

Furthermore, each parity server makes backups for the data deltas for both UPDATE and DELETE requests, so as to revert any changes on parity chunks made by incomplete (unacknowledged) requests. We adopt an idea similar to that in LH*RS [38], in which we let each parity server store the received data deltas in a temporary buffer for possible rollbacks. When a data server fails, the proxy that initiates an incomplete UPDATE or DELETE request receives the updated server state from the coordinator (§5.2). It notifies the parity servers to revert the changes on parity chunks for the incomplete request using the buffered data deltas. To facilitate the removal of buffered data deltas, when a proxy issues an UPDATE or DELETE request, it also attaches a *local sequence number* (i.e., independent of other proxies) for the latest acknowledged request, so a parity server can decide if any buffered data delta can be removed.

5.4 Degraded Requests

When a failed server switches to the degraded state (i.e., after data inconsistency is resolved), the coordinator redirects all degraded requests from the failed server to a redirected server. The following explains how each type of degraded requests works.

Degraded SET: Recall that a SET request sends an object to a data server and $n - k$ parity servers (§4.3). If a proxy finds that a SET request for a new object is mapped to a failed data or parity server, it issues a degraded SET through the coordinator. The coordinator first identifies the stripe list for the object. It then finds a working server in the stripe list as the redirected server, and instructs the proxy to write the object to the redirected server. Upon receiving the object, the redirected server stores the object in a temporary buffer. Later when the failed server is restored, the redirected server migrates the object to the restored server.

Degraded GET: If a proxy issues a GET request for an object in a failed data server, the coordinator first maps the object to its stripe list and selects a working server in the stripe list as the redirected server (which can be the same as in degraded SET). The proxy then asks the redirected server for the object. If the object is stored in an unsealed data chunk, the redirected server simply retrieves its replica from a working parity server; if the object is stored in a sealed data chunk, the redirected server reconstructs the whole data chunk via erasure coding to retrieve the failed object; if the object is written via a degraded SET, the redirected server retrieves the object from its temporary buffer.

Here, the redirected server performs *on-demand* reconstruction for the data that is requested by the proxy, instead of reconstructing all data stored in the failed server. This avoids unnecessary decoding overhead, especially in the presence of transient failures in which data is only temporarily unavailable. Moreover, the redirected server reconstructs a failed object at the granularity of chunks (i.e., the whole chunk containing the failed object will be reconstructed). If subsequent GET requests access the same object or other objects within the reconstructed chunk, no extra reconstruction is needed.

Degraded UPDATE and DELETE: If a proxy issues an UPDATE or DELETE to an object whose associated stripe list contains a failed server, the request becomes a degraded request. In this case, the coordinator first identifies a working server in the stripe list as the redirected server, which first reconstructs the failed chunk (as in degraded GET) before the object is modified or deleted. Note that we reconstruct the failed chunk even though the object is available in a working server (belonging to the same stripe list as the failed server). The reason is that each update or delete to an object also triggers updates to all parity chunks. We ensure that the failed chunk is reconstructed before its corresponding parity chunks are updated, thereby eliminating the need of resolving any inconsistency. After the failed chunk is reconstructed, the data server (which may be the redirected server if the original data server fails) handles the request as in §4.3. Note that if the failed server is a parity server, the data server sends the data delta to the redirected server.

5.5 After Failures

When a failed server is back to normal, it is in the coordinated normal state. All redirected servers migrate any reconstructed objects back to the restored server, while new proxy requests are still managed by the coordinator. Specifically, if a proxy accesses an object that has been redirected before and not yet migrated, the coordinator instructs the proxy to access the redirected server; if the redirected object has been migrated, the coordinator instructs the proxy to send the request to the restored server. When all data migration is completed, the restored server switches to the normal state, and proxies can issue decentralized requests without involving the coordinator.

6. IMPLEMENTATION

MemEC is deployable on commodity hardware and operating systems. We have implemented a MemEC prototype in C++ on Linux in about 34K lines of code. We use the Spread toolkit [11] to implement atomic broadcast for server state synchronization (§5.2), and use Intel's Storage Acceleration Library [6] to realize erasure coding.

MemEC is optimized for high network performance. It maintains persistent TCP connections between the coordinator, proxies, and servers to save connection establishment overhead. It also uses non-blocking socket I/O based on epoll [33], a Linux-specific I/O event notification mechanism that polls multiple sockets with $O(1)$ complexity.

In addition, MemEC exploits multi-threading to parallelize I/O operations. Each entity (i.e., the coordinator, proxy, or server) uses a dedicated thread to poll I/O events and multiple worker threads to process the received events. It also uses multiple worker threads to send data to different destinations in parallel (e.g., in SET request). To mitigate context switching in multi-threading, MemEC adopts an event-driven architecture. Specifically, each entity consists of a multi-producer, multi-consumer event queue and a fixed number of worker threads. When the polling thread receives a new I/O event, it dispatches the event, via the event queue, to one of multiple worker threads for processing. Each worker thread may also insert one or multiple events to the event queue for further processing (e.g., sealing a data chunk after a SET request). As multiple worker threads process the I/O events in parallel, the overall latency of request handling is significantly reduced.

Workload	Proportions of requests
A (Update heavy)	50% GET, 50% UPDATE
B (Read mostly)	95% GET, 5% UPDATE
C (Read only)	100% GET
D (Read latest)	95% GET, 5% SET
F (Read-modify-write)	50% GET, 50% read-modify-write (GET and UPDATE)

Table 1: YCSB workloads used in our experiments.

7. EVALUATION

We conduct extensive testbed experiments on MemEC under commodity settings. We run MemEC on a cluster of 21 machines, including one coordinator, four proxies, and 16 servers. Each machine runs Ubuntu 12.04 LTS with the Linux kernel version 3.5.0-54-generic or higher. All servers are equipped with Intel Core i5-3570 3.4GHz CPU, while the coordinator and all proxies are equipped with Intel Core i5-2400 3.1GHz CPU. Each server pre-allocates 8GB memory to form a KV store of 128GB capacity in total. All machines are interconnected via a Gigabit full-duplex switch.

We use workloads derived from YCSB version 0.7.0 [23] to evaluate MemEC in both normal and degraded modes. We focus on the read-heavy and update-heavy workloads, namely A, B, C, D, and F, as shown in Table 1, while we omit Workload E since MemEC does not support range queries. We co-locate both YCSB clients and MemEC proxies in the same machine. Each of the YCSB clients connects to its co-located MemEC proxy via the loopback interface. Before running each workload, we load MemEC with 10M objects using SET requests (called the *load phase*). Each of the four clients then issues 5M requests via YCSB (i.e., 20M requests in total), while the access pattern follows a heavy-tailed Zipf distribution with the shape parameter 0.99.

Our experiments focus on the workloads with small objects. We divide the set of objects into two halves, whose value sizes are set to 8 bytes and 32 bytes. Two clients issue requests with 8-byte values, and the other two clients issue requests with 32-byte values. In Experiment 3, we also consider larger value sizes. Also, we set the key size as 24 bytes for all objects, since the minimum configurable key size in YCSB is 23 bytes and we use one extra byte to distinguish between two value sizes. For MemEC, we fix $c = 16$ stripe lists (§4.2) and the chunk size as 4KB.

To fully exploit multi-threading, we tune the numbers of threads in YCSB and MemEC for maximum possible performance. Finally, we fix 64 threads in YCSB for workload generation and 12 worker threads in each MemEC node (i.e., the coordinator and each of the proxies and servers).

We plot the average results over 10 runs with 95% confidence intervals based on the student's t-distribution. Some intervals may be invisible due to small variations.

7.1 Performance in Normal Mode

We start with evaluating MemEC in normal mode when no failure happens.

Experiment 1 (Comparisons with existing systems): We compare MemEC with two existing in-memory KV stores: Redis 3.0.7 [8] and Memcached 1.4.25 [7]. Our objective here is *not* to show which system is better than others, since all systems are implemented with different functionalities; in fact, Redis and Memcached support more features (e.g., object eviction) than MemEC. Instead, our objective is to validate the implementation of our MemEC prototype, such
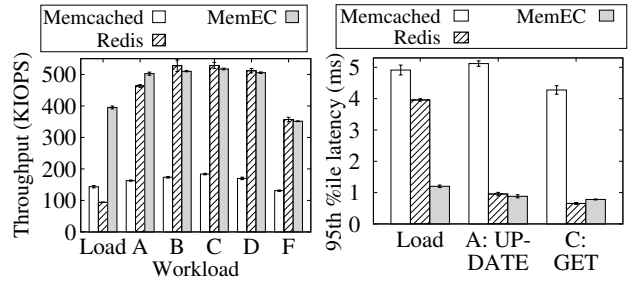
(a) Aggregate throughput (b) 95th percentile latencies

Figure 5: Experiment 1: Comparisons of MemEC, Redis, and Memcached.

that its performance matches that of state-of-the-arts.

We use the default settings for both Redis and Memcached, except that we configure 12 worker threads in Memcached as in MemEC. Memcached does not support fault tolerance, while Redis uses replication for fault tolerance. We disable the replication in Redis to maximize its performance. For MemEC, we disable erasure coding and set $n = 10$, such that all stripe lists contain data servers only.

Figure 5(a) shows the aggregate throughput over all objects for different systems. MemEC's throughput is 2.7–3.1× of Memcached's in all cases. The low throughput of Memcached may be explained by its poor scalability on multi-core CPUs [29]. Compared to Redis, MemEC's throughput is 4.2× of Redis's in the load phase. MemEC is much faster since it leverages multi-threading to handle SET requests on multiple servers in parallel, while Redis uses a *mostly* single-threaded design to serve requests (some slow I/O requests are served via multi-threading) [9]. On the other hand, MemEC's throughput is close to Redis's in Workloads A–F: MemEC is faster in Workload A (which is update-heavy) by 8.4%, while Redis is slightly faster in Workloads B–F (which are read-heavy) by 1.2–3.3%. Since reading an object only involves a single server, there is no significant performance difference between single- and multi-threading.

Figure 5(b) depicts the 95th percentile latencies of different systems. In the interest of space, we only consider three specific types of requests: SET in the load phase, UPDATE in Workload A, and GET in Workload C. We find that both 8-byte and 32-byte values have very similar latency results. Thus, we directly report the average latency over all objects (same for other experiments). Memcached has the highest latency in all cases. MemEC achieves 70% lower latency than that of Redis for SET, while both MemEC and Redis have comparable latencies for UPDATE and GET.

We point out that MemEC has comparable performance to existing in-memory storage systems proposed in the literature (e.g., [19,30,37]). For example, in a commodity cluster over a Gigabit network, their access latencies are reportedly in the range of few milliseconds.

Experiment 2 (Erasure coding in MemEC): We study the erasure coding overhead in MemEC. Here, we consider RDP codes [24] and Reed-Solomon (RS) codes [44]. RDP codes are double-fault tolerant. They perform only bitwise XOR-based coding operations and are computationally efficient. RS codes perform Galois Field arithmetic (§2) and are generally more computationally expensive than RDP codes, but they can tolerate a general number of server failures. We set $(n, k) = (10, 8)$. We also evaluate MemEC when no

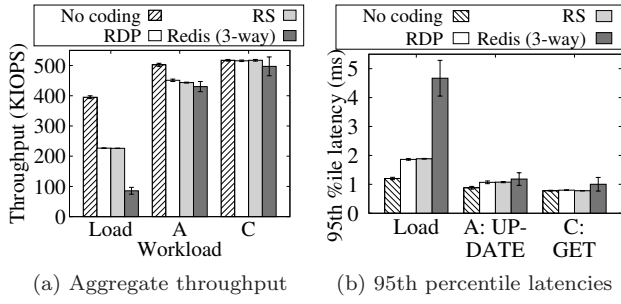(a) Aggregate throughput　　(b) 95th percentile latencies

Figure 6: Experiment 2: Performance of different erasure coding schemes in MemEC and 3-way replication in Redis.
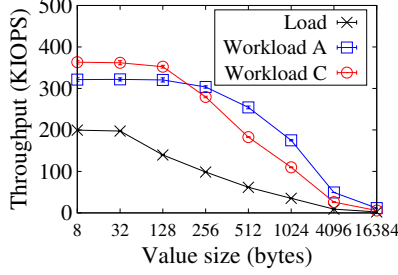

Figure 7: Experiment 3: Aggregate throughput of the load phase, Workload A, and Workload C versus value size.

erasure coding is used (as in Experiment 1), referred to as *No coding*, as well as Redis with 3-way replication.

Figure 6(a) shows the aggregate throughput over all objects for different coding schemes. We focus on the load phase, Workload A, and Workload C. In the load phase, the throughput of RDP and RS drops to 57.3% and 57.2% of that without coding, respectively, mainly due to the extra traffic generated to parity servers in `SET` requests. Also, in Workload A, the throughput values of RDP and RS codes are 90.0% and 88.2% of that without coding, respectively. Thus, erasure coding does not significantly reduce the performance for `UPDATE` requests. Furthermore, in Workload C, both coding schemes achieve similar throughput to that without coding, as `GET` requests only access data servers. Overall, both RDP and RS codes have very similar results, so the coding schemes do not significantly change the performance.

Compared to 3-way replication in Redis, we observe that the throughput of MemEC with RDP and RS is at least 164%, 4.0%, and 3.9% higher in the load phase, Workload A, and Workload C, respectively. Our results are consistent with those in Experiment 1.

Figure 6(b) shows the 95th percentile latencies. For `SET` requests, both RDP and RS codes have 55.8% higher latencies than without coding; for `UPDATE` requests, they are 21.7% higher than without coding; for `GET`, the latencies are almost identical across coding schemes.

Experiment 3 (Workloads of larger value sizes): We study how MemEC performs for larger value sizes, even though it is designed for small objects. Specifically, in each test, we configure each of the four clients to send objects of the same value size, which we vary from 8 bytes to 16KB across different tests. Each client first loads 250K objects (i.e., 1M objects in total). Each of the four clients then issues 500K requests (i.e., 2M requests in total) in both Workload A and Workload C. Recall that if an object has size larger than the 4KB chunk size, it will be split into fragments
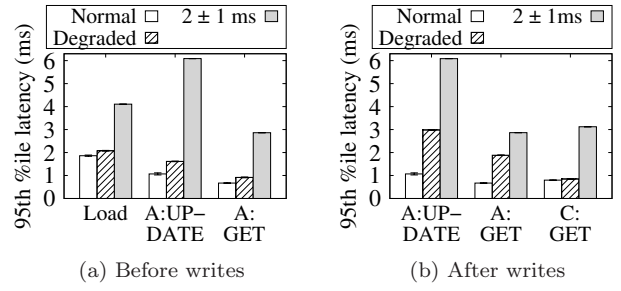

(a) Before writes　　(b) After writes

Figure 8: Experiment 4: Impact of transient failures.

before being stored (§3.2). We use RDP with $(n, k) = (10, 8)$ for erasure coding.

Figure 7 shows the aggregate throughput versus different value sizes. When the value size is below 1KB, the performance is bottlenecked by the processing overhead of small objects. As the value size is at least 1KB, the throughput, in terms of the amount of data being processed per second, is saturated at 36.5MB/s, 181MB/s, and 89.0MB/s for the load phase, Workload A, and Workload C, respectively. Note that the network bandwidth is not fully utilized for large objects. For example, for the load phase, we expect that four proxies can theoretically achieve an effective network speed of at most $\frac{4 \times 125}{3} = 167$MB/s (recall that a proxy first writes data via `SET` with $(n - k + 1)$-way replication (§4.3)). We plan to optimize the implementation for large objects as future work.

7.2 Performance in Degraded Mode

We now evaluate MemEC in degraded mode and examine the impact of transient failures. We use RDP [24] with $(n, k) = (10, 8)$ for erasure coding. We simulate a transient failure as network congestion, by injecting an artificial link delay to the network interface of a failed server for each outgoing packet via the Linux traffic controlling tool `tc` [1]. We configure the delay to follow a normal distribution with mean 2ms and standard deviation 1ms. Note that the actual delay for determining a transient server failure varies depending on the deployment environment, and we do not explore this issue in this work.

Experiment 4 (Impact of transient failures): We evaluate the degraded requests of MemEC in the presence of transient failures. We only consider a single-server failure, while the results are similar for a double-server failure. We consider two independent cases:

- **Before writes:** A transient failure happens before MemEC stores any data. We then run the load phase followed by Workload A. Each proxy issues degraded `SET` requests that redirect writes from the failed server to a redirected server in the load phase. Subsequent `GET` and `UPDATE` requests are also redirected in Workload A.
- **After writes:** We run the load phase and trigger a transient failure after the load phase is completed. We then run Workloads A and C to evaluate degraded `GET` and degraded `UPDATE`.

We compare the performance of MemEC for three settings: (1) there is no failure (labeled as *Normal*); (2) there is a failure and MemEC enables degraded request handling (labeled as *Degraded*); and (3) there is a failure but degraded request handling is disabled (i.e., a request contacts a failed

State transition		Elapsed time (ms)	
		Single failure	Double failure
$T_{N \to D}$	With req.	4.77 ± 0.79	9.24 ± 0.78
	No req.	1.74 ± 0.09	4.91 ± 0.89
$T_{D \to N}$	With req.	628.5 ± 43.9	667.5 ± 27.2
	No req.	0.91 ± 0.46	1.10 ± 0.19

Table 2: Experiment 5: Average elapsed times of state transitions with 95% confidence intervals.

server that is under network congestion and waits for its delayed response).

Figure 8(a) shows the 95th percentile latencies of SET, UP-DATE, and GET requests when transient failures occur before writes. Compared to normal mode, the latency of SET in the load phase increases by 11.6% with degraded request handling enabled. For Workload A, the latencies of UPDATE and GET increase by 50.9% and 36.9%, respectively. However, with degraded request handling disabled, the latencies are much higher; for example, they increase by 469% and 326% in UPDATE and GET over normal mode, respectively.

Figure 8(b) shows the 95th percentile latencies of GET requests in Workloads A and C, and UPDATE requests in Workload A when transient failures occur after writes. The latencies of GET and UPDATE increase by at most 180% and 178%, respectively. Note that the latencies of degraded GET in Workloads A and C differ, which can be explained by the need of maintaining consistency within the same stripe. Specifically, if a redirected server receives an UPDATE before GET, it reconstructs all failed chunks of the same stripe before modifying the data and parity chunks so as to avoid inconsistency caused by concurrent access to parity chunks by UPDATE and reconstruction of chunks by GET. The following GET requests to the failed data chunks are blocked until the reconstructed chunks are available on the redirected server. Thus a higher latency is observed in Workload A, but not in Workload C, which involves no UPDATE.

Note that the impact of reconstructing failed data on a read-heavy workload is very small. For example, the latency of GET in Workload C only increases by 6.0% when compared to normal mode. MemEC performs reconstruction at the granularity of chunks, and the reconstruction overhead can be amortized over subsequent GET requests to the same reconstructed chunks (§5.4).

Experiment 5 (State transition overhead): We further study the state transition overhead in the presence of a transient failure. We consider the transitions when a failed server switches from the normal to degraded states (denoted as $T_{N \to D}$) and later when it switches from the degraded to normal states (denoted as $T_{D \to N}$). In addition to a single-server failure, we also consider the case of a double-server failure, in which two servers fail simultaneously.

We compare the cases with and without ongoing requests during the transitions. For the former, we run Workload A to generate ongoing requests until the latency becomes stable, and trigger a transient failure. We then restore the failure 5s later. We measure both the elapsed times for the state transitions and the average latencies of UPDATE and GET requests over the workload.

Table 2 first shows the elapsed times of state transitions, averaged over 10 runs with 95% confidence intervals. For a single-server failure, $T_{N \to D}$ takes 4.77ms and 1.74ms on average with and without requests, respectively. The difference between the two elapsed times is mainly caused by reverting parity updates of incomplete requests. The aver-

age elapsed times for $T_{D \to N}$ are 628.5ms and 0.91ms with and without requests, respectively. The elapsed time for $T_{D \to N}$ with ongoing requests includes data migration from the redirected server to the restored server, so it is significantly larger than that without any request. For a double-server failure, the transitions $T_{N \to D}$ and $T_{D \to N}$ with ongoing requests spend 9.24ms and 667.5ms, respectively. Note that the elapsed times for $T_{N \to D}$ are higher than that for $T_{D \to N}$ even though when there is no request. The reason is that the atomic broadcast for the transition $T_{N \to D}$ involves each failed server, which experiences network congestion. In all cases, a state transition incurs less than 1s, so we expect that its overhead is limited.

8. RELATED WORK

Erasure coding has been studied in large-scale KV stores for low-redundancy fault tolerance, such as distributed hash tables (DHTs) [20, 25, 45] and cloud storage [5, 13, 36]. However, the above systems store large-size objects and perform erasure coding on a per-object basis, which is inadequate for in-memory KV stores in which small objects dominate as we show in this paper.

Some in-memory KV stores combine erasure coding and replication, such as LH*RS [21, 22, 38] and Cocytus [50]. They store keys and metadata by replication, while storing values of multiple objects by erasure coding. MemEC differs from LH*RS and Cocytus by applying erasure coding across objects in entirety to further reduce redundancy based on the all-encoding data model. We also design MemEC to explicitly support efficient workflows of degraded requests and transitions between normal and degraded modes.

Ensuring parity consistency is critical in any erasure-coded storage system. Myriad [17] leverages two-phase commit to consistently update both data and parity chunks. LH*RS [38] uses a variant of two-phase commit to retain data delta backups in a temporary buffer for possible rollbacks. Aguilera et al. [12] propose to keep a recent list of the previous data update requests to resolve consistency of multiple chunks. Pahoehoe [13] associates each version of object with a globally unique timestamp to guarantee eventual consistency. Cocytus [50] piggybacks the commit message of each previous request in the current parity update and avoids the two rounds of message exchanges in two-phase commit. MemEC keeps data delta backups as in LH*RS [38] for reverting any inconsistent change when a server fails.

9. CONCLUSION

This paper presents MemEC, which makes a case of how to efficiently apply erasure coding to small objects in in-memory KV storage. We propose an all-encoding data model that effectively reduces redundancy for fault tolerance by up to 60% over existing approaches. We design and implement MemEC to efficiently operate in both normal and degraded modes. We evaluate our MemEC prototype running on commodity hardware, and demonstrate its efficiency in both normal and degraded modes under YCSB benchmarks.

10. REFERENCES

[1] Advanced Traffic Control. https://wiki.archlinux.org/index.php/Advanced_traffic_control.

[2] Aerospike. http://www.aerospike.com/.

[3] Amazon ElastiCache. http://aws.amazon.com/elasticache/.

[4] Couchbase Server. http://www.couchbase.com/nosql-databases/couchbase-server.

[5] Erasure Code - Ceph Documentation. http://docs.ceph.com/docs/master/rados/operations/erasure-code/.

[6] Intel® Storage Acceleration Library (open source version). https://01.org/intel%C2%AE-storage-acceleration-library-open-source-version.

[7] Memcached. http://memcached.org.

[8] Redis. http://redis.io.

[9] Redis Latency Problems Troubleshotting. http://redis.io/topics/latency.

[10] Repcached. http://repcached.lab.klab.org.

[11] Spread Toolkit. http://www.spread.org.

[12] M. Aguilera, R. Janakiraman, and L. Xu. Using Erasure Codes Efficiently for Storage in a Distributed System. In *Proc. of IEEE DSN*, pages 336–345, Jun 2005.

[13] E. Anderson, X. Li, A. Merchant, M. A. Shah, K. Smathers, J. Tucek, M. Uysal, and J. J. Wylie. Efficient Eventual Consistency in Pahoehoe, an Erasure-Coded Key-Blob Archive. In *Proc. of IEEE/IFIP DSN*, pages 181–190, Jun 2010.

[14] C. Aniszczyk. Caching with Twemcache. https://blog.twitter.com/2012/caching-with-twemcache.

[15] 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, Jun 2012.

[16] J. C. W. Chan, Q. Ding, P. P. C. Lee, and H. H. W. Chan. Parity Logging with Reserved Space: Towards Efficient Updates and Recovery in Erasure-Coded Clustered Storage. In *Proc. of USENIX FAST*, pages 163–176, Feb 2014.

[17] F. Chang, M. Ji, S.-T. A. Leung, J. MacCormick, S. E. Perl, and L. Zhang. Myriad: Cost-effective Disaster Tolerance. In *Proc. of USENIX FAST*, pages 103–116, Jan 2002.

[18] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-Performance, Reliable Secondary Storage. *ACM Comput. Surv.*, 26(2):145–185, Jun 1994.

[19] Y. Cheng, A. Gupta, and A. R. Butt. An In-Memory Object Caching Framework with Adaptive Load Balancing. In *Proc. of ACM EuroSys*, pages 4:1–4:16, Apr 2015.

[20] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient Replica Maintenance for Distributed Storage Systems. In *Proc. of USENIX NSDI*, pages 45–58, May 2006.

[21] D. Cieslicki, S. Schäckeler, and T. Schwarz. Efficient Updates in Highly Available Distributed Random Access Memory. In *Proc. of IEEE ICPADS*, pages 49–57, Jul 2006.

[22] D. Cieslicki, S. Schäeckeler, and T. Schwarz. Maintaining and Checking Parity in Highly Available Scalable Distributed Data Structures. *The Journal of Systems and Software*, 83(4):529–542, 2010.

[23] 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, Jun 2010.

[24] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-diagonal Parity for Double Disk Failure Correction. In *Proc. of USENIX FAST*, pages 1–14, Mar 2004.

[25] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a DHT for Low Latency and High Throughput. In *Proc. of USENIX NSDI*, pages 85–98, Mar 2004.

[26] 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, Apr 2013.

[27] B. Fitzpatrick. Distributed Caching with Memcached. *Linux Journal*, 2004(124), Aug 2004.

[28] D. Ford, F. Labelle, F. I. Popovici, M. Stokel, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *Proc. of USENIX OSDI*, pages 61–74, Oct 2010.

[29] N. Gunther, S. Subramanyam, and S. Parvu. Hidden Scalability Gotchas in Memcached and Friends. In *VELOCITY Web Performance and Operations Conference*, Jun 2010.

[30] Y.-J. Hong and M. Thottethodi. Understanding and Mitigating the Impact of Load Imbalance in the Memory Caching Tier. In *Proc. of ACM SoCC*, pages 13:1–13:17, Oct 2013.

[31] 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, Jun 2012.

[32] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proc. of USENIX ATC*, pages 11–11, 2010.

[33] M. Kerrisk. *The Linux Programming Interface*. No Starch Press, 2010.

[34] O. Khan, R. Burns, J. Plank, W. Pierce, and C. Huang. Rethinking Erasure Codes for Cloud File Systems: Minimizing I/O for Recovery and Degraded Reads. In *Proc. of USENIX FAST*, pages 251–264, Feb 2012.

[35] F. Klein, K. Beineke, and M. Schottner. Memory Management for Billions of Small Objects in a Distributed In-Memory Storage. In *Proc. of IEEE CLUSTER*, pages 113–122, Sept 2014.

[36] 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, May 2015.

[37] C. Li and A. L. Cox. GD-Wheel: A Cost-Aware Replacement Policy for Key-Value Stores. In *Proc. of ACM EuroSys*, pages 5:1–5:15, Apr 2015.

[38] W. Litwin, R. Moussa, and T. Schwarz. LH*RS - A

Highly-Available Scalable Distributed Data Structure. *ACM Trans. Database Syst.*, 30(3):769–811, 2005.

[39] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar. f4: Facebook's Warm BLOB Storage System. In *Proc. of USENIX OSDI*, pages 383–398, Oct 2014.

[40] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proc. of USENIX NSDI*, pages 385–398, Apr 2013.

[41] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The RAMCloud Storage System. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, Aug 2015.

[42] R. Pagh and F. F. Rodler. Cuckoo Hashing. *Journal of Algorithms*, 51(2):122–144, May 2004.

[43] J. Plank, J. Luo, C. Schuman, L. Xu, and Z. Wilcox-O'Hearn. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage. In *Proc. of USENIX FAST*, pages 253–265, Feb 2009.

[44] I. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society of Industrial and Applied Mathematics*, 8(2):300–304, Jun 1960.

[45] R. Rodrigues and B. Liskov. High Availability in DHTs: Erasure coding vs. Replication. In *Proc. of IPTPS*, pages 226–239, 2005.

[46] M. Sathiamoorthy, M. Asteris, D. S. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing Elephants: Novel Erasure Codes for Big Data. In *Proc. of the VLDB Endowment*, pages 325–336, Aug 2013.

[47] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber. Extending SSD Lifetimes with Disk-based Write Caches. In *Proc. of USENIX FAST*, pages 101–114, Feb 2010.

[48] H. Weatherspoon and J. D. Kubiatowicz. Erasure Coding Vs. Replication: A Quantitative Comparison. In *Proc. of IPTPS*, pages 328–338, Mar 2002.

[49] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang. In-Memory Big Data Management and Processing: A Survey. *IEEE Trans. on Knowledge and Data Engineering*, 27(7):1920–1948, Jul 2015.

[50] H. Zhang, M. Dong, and H. Chen. Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication. In *Proc. of USENIX FAST*, pages 167–180, Feb 2016.