Article
http://justc.ustc.edu.cn
Received: September 06, 2022; Accepted: April 30, 2023

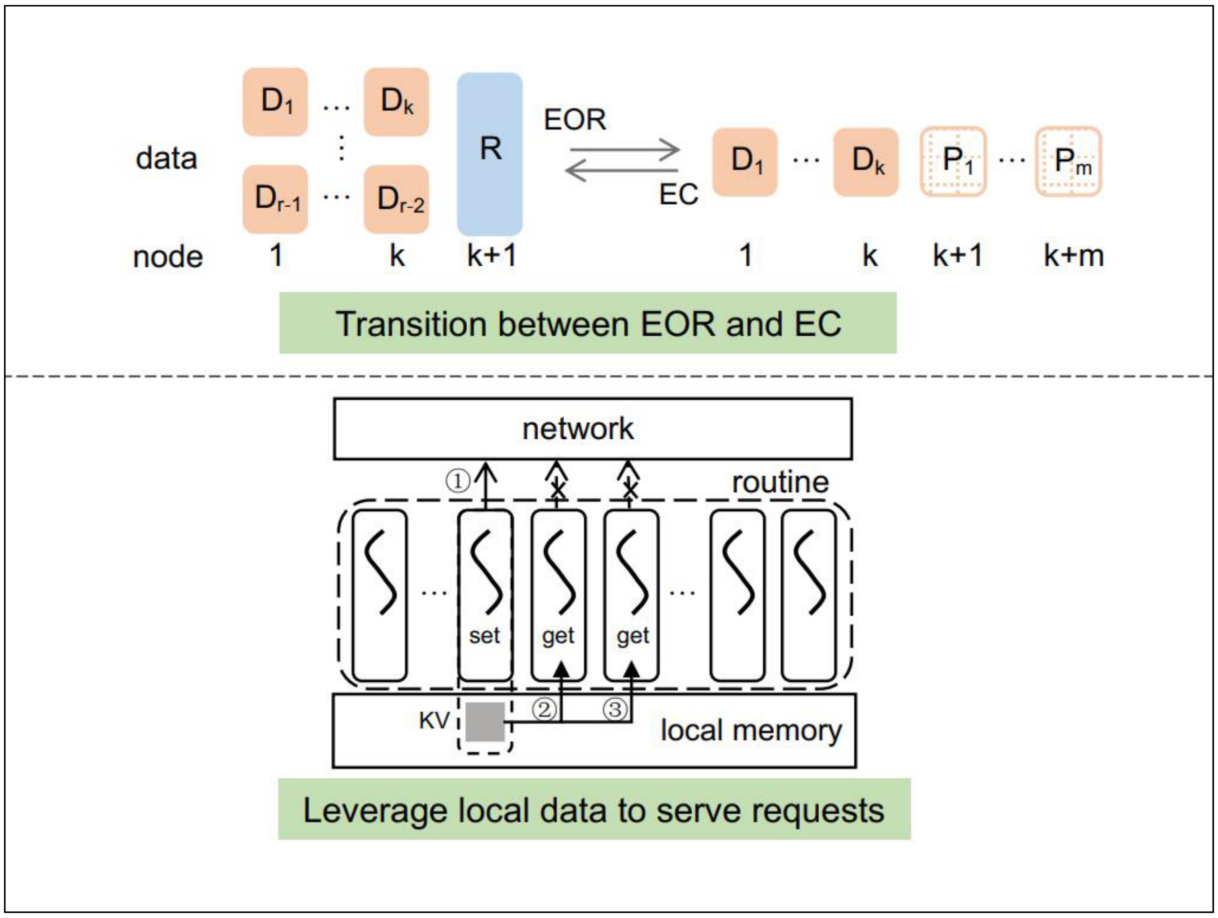
Hybrid fault tolerance in distributed in-memory storage systems

Zheng Gong, Si Wu ✉, and Yinlong Xu

School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China

✉Correspondence: Si Wu, Email: siwu5938@ustc.edu.cn

Graphical abstract



Public summary

■ Replication and erasure code (EC) are used to tolerate faults and have different time and space costs. To obtain the best of both worlds, we support hybrid fault tolerance and dynamic redundancy transition between both schemes.

■ EC-oriented replication is introduced to improve the I/O efficiency of the redundancy transition.

■ Local data could be leveraged to serve coreleased requests while avoiding concurrent consistency problems.

Article

http://justc.ustc.edu.cn

Received: September 06, 2022; Accepted: April 30, 2023

Hybrid fault tolerance in distributed in-memory storage systems

Zheng Gong, Si Wu ✉, and Yinlong Xu

School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China

✉Correspondence: Si Wu, Email: siwu5938@ustc.edu.cn

Cite This: *JUSTC*, **2024**, 54(4): 0406 (10pp)   Read Online

Abstract: An in-memory storage system provides submillisecond latency and improves the concurrency of user applications by caching data into memory from external storage. Fault tolerance of in-memory storage systems is essential, as the loss of cached data requires access to data from external storage, which evidently increases the response latency. Typically, replication and erasure code (EC) are two fault-tolerant schemes that pose different trade-offs between access performance and storage usage. To help make the best performance and space trade-off, we design ElasticMem, a hybrid fault-tolerant distributed in-memory storage system that supports elastic redundancy transition to dynamically change the fault-tolerant scheme. ElasticMem exploits a novel EC-oriented replication (EOR) that carefully designs the data placement of replication according to the future data layout of EC to enhance the I/O efficiency of redundancy transition. ElasticMem solves the consistency problem caused by concurrent data accesses via a lightweight table-based scheme combined with data bypassing. It detects corelated read and write requests and serves subsequent read requests with local data. We implement a prototype that realizes ElasticMem based on Memcached. Experiments show that ElasticMem remarkably reduces the time of redundancy transition, the overall latency of corelated concurrent data accesses, and the latency of single data access among them.

Keywords: In-memory storage system; hybrid fault tolerance; replication; erasure code

CLC number: **Document code:** A

1 Introduction

In-memory storage is becoming increasingly popular as data-driven business is more extensive[1]. Facing emerging data-intensive applications, disk-based schemes[2–3] struggle to serve the massive amount of data access requests with low latency and high concurrency and become the bottleneck of data access flow. As a result, memory-based solutions[4–5] are adopted to alleviate the access load of external storage. In particular, in-memory storage[6–7] is a generic middleware that lies between user applications and external storage. An in-memory storage system caches data from external storage to provide low-latency and high-concurrency data access. Therefore, in-memory storage systems have been widely deployed in production[8,9].

Fault tolerance is a vital promise of in-memory storage. The volatile nature of memory and increasing scale of in-memory systems lead to prevalent failures. In case of failures, the system accesses external storage directly and reloads data into memory, which severely prolongs the access latency[10]. To provide fault tolerance, data redundancy is introduced. Typically, there are two common redundancy schemes, namely, replication[11,12] and erasure code (EC)[13–18]. Replication replicates a data object m+1 times and distributes them into m+1 different nodes to tolerate m node failures. However, replication incurs m times more memory usage. EC splits an object into k data blocks, encodes the data blocks into m additional blocks called parity blocks, and distributes the k+m data and parity blocks to k+m different nodes to tolerate m node failures. The k+m blocks form a stripe. Compared to replication, EC needs only m/k more memory space with the same fault-tolerant ability. However, EC incurs extra CPU overhead for encoding, which inevitably degrades the user write performance.

The performance requirement of in-memory storage and the limited memory resources call for a proper balance between access performance and memory space. Storing all data with replication results in high overhead of the precious memory space, while storing all data with EC degrades access performance. Therefore, it is reasonable to employ a hybrid fault-tolerant scheme that incorporates both replication and EC to help reach the best trade-off between performance and memory usage. The system assigns replication and EC to different data types. When the user demands change, the system conducts a redundancy transition to change the redundancy scheme of an object from replication to EC, or vice versa. For example, as in-memory caching exhibits skewed popularity[19], small-sized or hot data could be stored with replication, while big-sized or cold data could be stored with EC. When the hot data turn cold or the cold data turn hot, the system transitions the redundancy scheme between replication and EC. In this manner, the system provides both high access performance for hot data and high storage efficiency for cold data.

Many studies have focused on the redundancy transition between EC schemes with different parameters. StripeMerge[21] changes from EC(k,m) to EC(2k,m) and performs IO-efficient unilateral redundancy transition by carefully choosing two EC(k,m) stripes and merging them into EC(2k,m). ERS[22] supports the change from EC(k,m) to EC(k',m). It minimizes the transition I/Os by an innovative codesign of encoding matrix construction and data placement design. Cocytus[23] applies three-way replication to metadata of objects and EC (k = 3, m = 2) to values of objects. Here, we emphasize that all these studies do not combine replication and EC to effectively store in-memory data according to data types. In addition, they do not consider the consistency issue caused by concurrent data accesses in distributed in-memory storage. There are still other studies adopting both replication and EC, but they are either based on file systems or consider cross-rack scenarios[31–32].

In this paper, we present ElasticMem, a distributed in-memory storage system combining both replication and EC for hybrid fault tolerance and supporting elastic redundancy transition between replication and EC. ElasticMem addresses the I/O efficiency and consistency issue during redundancy transition. The main contributions include the following:

(Ⅰ) We design ElasticMem, a distributed in-memory storage system that incorporates both replication and EC for hybrid fault tolerance. ElasticMem assigns replication and EC to different data types and adjusts the redundancy scheme according to user demand changes.

(Ⅱ) ElasticMem adopts EC-oriented Replication (EOR), a new replication scheme that determines the data placement of replication according to the future data layout of EC. EOR greatly reduces the I/Os of the redundancy transition while guaranteeing the access performance of replication.

(Ⅲ) ElasticMem addresses the consistency issue triggered by distributed data accesses by a table-based scheme combined with data bypassing. It detects concurrent accesses to the same data and serves the subsequent requests with local data instead of sending redundant requests to the nodes.

(Ⅳ) We implemented ElasticMem as a prototype atop Memcached. Our testbed experiments show that ElasticMem reduces the transition time by up to 35% compared to naive transition. In addition, it remarkably reduces the overall latency of corelated requests and reduces the latency of a single request to at most 6 µs.

2 Background

We present the background of the distributed in-memory storage system. In particular, we introduce the distributed in-memory KV store, Memcached, based on which we will develop our storage prototype. We will introduce the replication mechanism and elaborate the EC mechanism on Memcached and show the process of redundancy transition between replication and EC.

2.1 Architecture of the distributed in-memory storage system

Distributed in-memory storage systems such as Memcached[7] are composed of a server side and client side (Fig. 1). The server side contains several servers called nodes, which store memory objects. The client side provides access interfaces (e.g., *get* and *set*) to users. The client side uses a dispatcher to determine the nodes for storing an object. For example, the client side of Memcached leverages consistent hashing for object distribution.

2.2 Replication in a distributed in-memory storage system

When writing a KV pair <k,v> using replication on Memcached, the client side (i.e., Libmemcached[25]) first applies consistent hashing on the key and derives a main node for storing the object. It then stores some replicas of the objects in successive nodes along the clockwise direction. For example, in Fig. 1, the client side assigns node$_3$, node$_4$, and node$_1$ for storing an object using three-way replication.

2.3 Erasure coding in a distributed in-memory storage system

When storing an object using EC, Memcached first splits an object evenly into k data blocks and then encodes the data blocks into m parity blocks. The k+m blocks together are called a stripe. Memcached also assigns a main node by hashing the key. It then stores the stripe of k+m blocks in successive nodes from the main node on by assigning the same keys to all k+m blocks. Memcached chooses Reed-Solomon (RS) code[26] as the coding scheme of EC, which is broadly studied by academics and industry for in-memory data and KV store[13,15,16,22,23].

For example, Fig. 1 depicts how to store a KV pair using RS(k = 2, m = 1) in Memcached. The client side splits the value into two data blocks v_0 and v_1, encodes them into one parity block v_2, and assigns the same keys to form three new KV pairs. Then, the three KV pairs are stored on node$_3$, node$_4$ and node$_1$.

We call the nodes that store data blocks and parity blocks

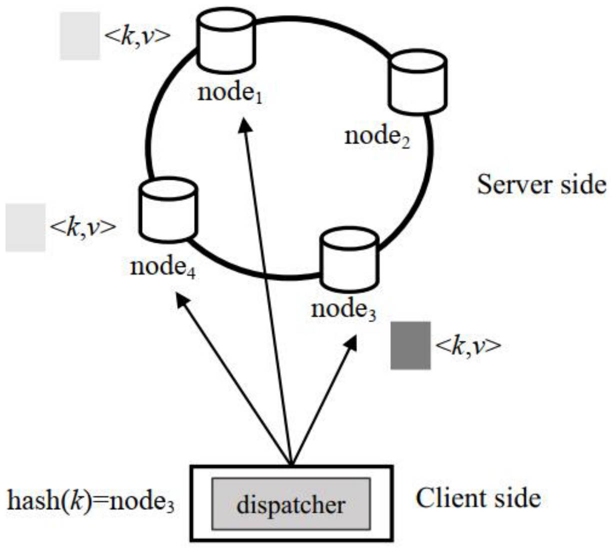


Fig. 1. Store an object with three-way replication on Memcached. One main copy (in node$_3$) and two replicas (in node$_4$ and node$_1$) are stored at the same time.

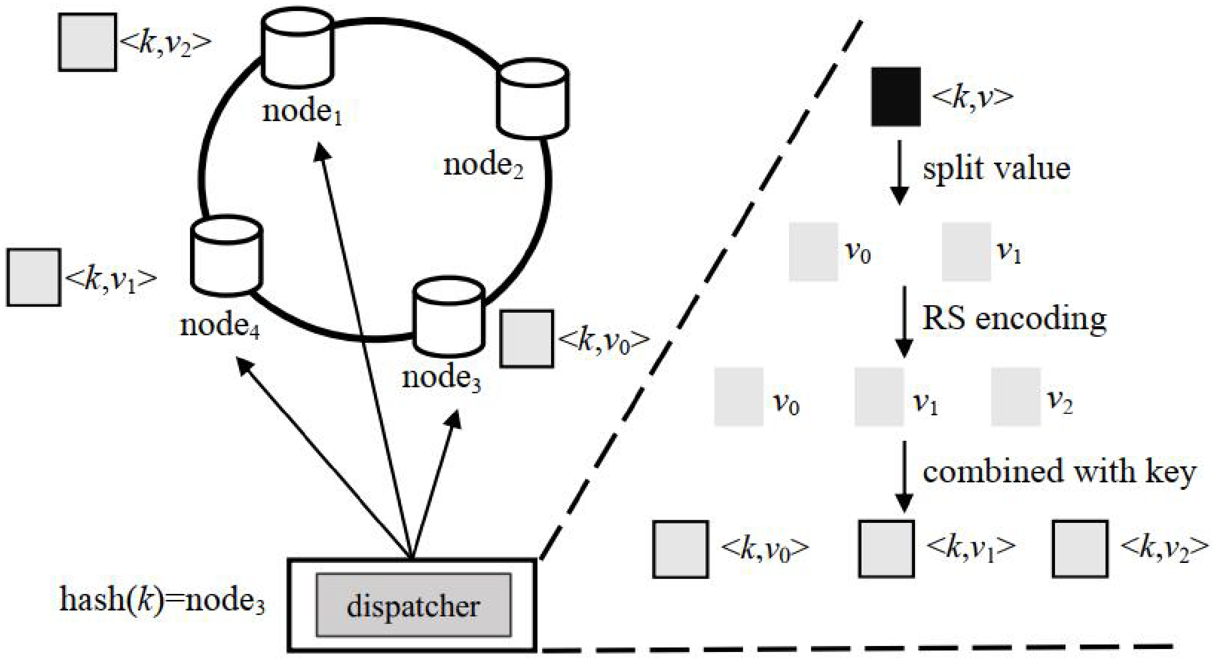


Fig. 2. RS($k = 2$, $m = 1$) on Memcached. A KV pair is split and encoded into multiple data and parity blocks (i.e., $<k, v_0>$, $<k, v_1>$, and $<k, v_2>$), which are separately stored.

data nodes and parity nodes, respectively. In Fig. 1, node$_3$ and node$_4$ are data nodes, while node$_1$ is a parity node. When a *get* request for this KV pair is issued, two concurrent *get* requests are sent to data nodes to retrieve v_0 and v_1. Then, they are merged to construct the original KV pair. However, if node$_3$ is not available, two concurrent *get* requests are sent to the data node (node$_4$) and parity node (node$_1$) to retrieve v_1 and v_2. Then, Memcached applies decoding to obtain v_0 and constructs the original KV data.

2.4 Redundancy transition

When the user demands change, the system conducts a redundancy transition to change the redundancy scheme between replication and EC to achieve the tradeoff between performance and storage overhead. Here, we show how a naive transition policy transits an object from replication to EC.

The client side performs three steps in naive transition: (i) It reads an object from nodes. (ii) It performs erasure coding to construct the stripe of data and parity blocks. (iii) It writes the EC stripe to nodes. For example, in Fig. 3, we show how the naive transition transitions an object from two-way replication to EC ($k = 3$, $m = 1$). The client reads the entire object, splits and encodes it into D_1, D_2, D_3, and P_1, and writes the stripe of four blocks to four nodes.

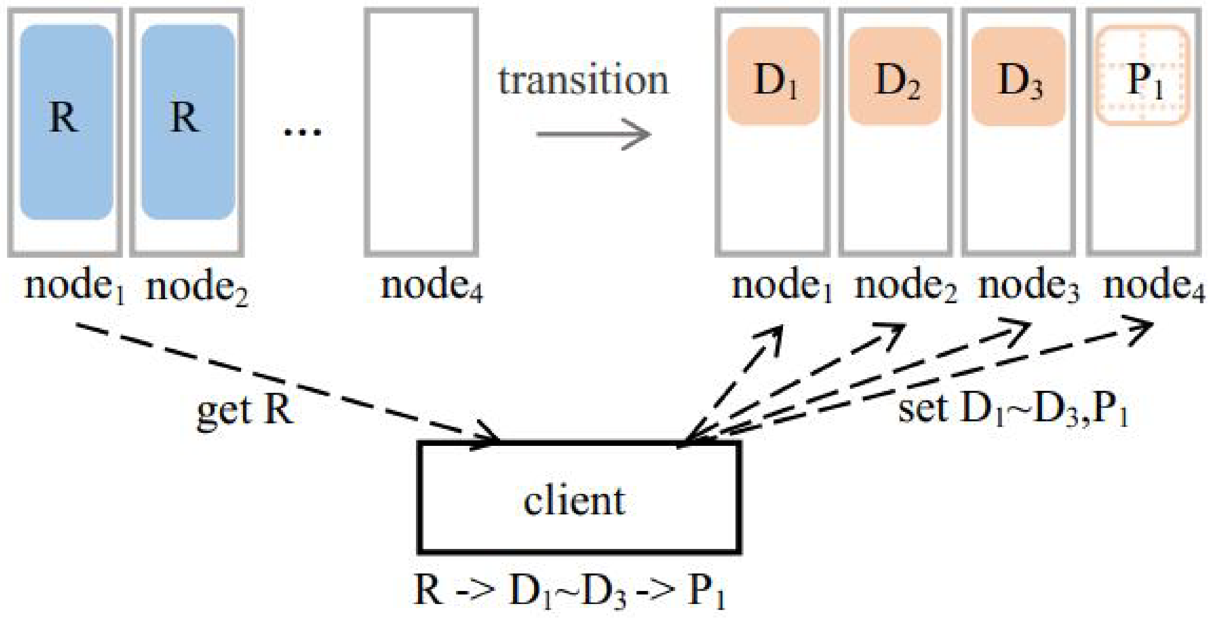


Fig. 3. Transition from two-way replication to EC ($k = 3$, $m = 1$). The naive transition process includes replica read, EC encoding, and EC block writes.

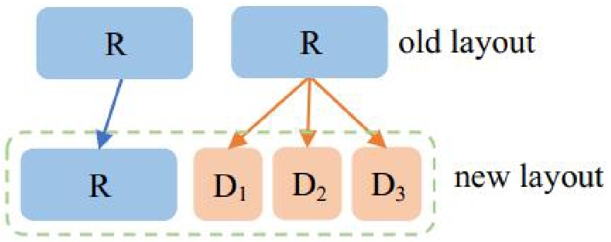


Fig. 4. New data layout of replicas for two-way replication, which is similar to EC($k = 3$,m).

The process from EC to replication is nearly the reverse of the above. The client reads all the data blocks, i.e., D_1, D_2, and D_3 in Fig. 3, constructs the original data R, and writes R to node$_1$ and node$_2$. Both transitions have a similar core; hence, we mainly focus on the elaboration of the former transition. The reverse process is evaluated in our experiments as well.

3 Challenges and motivations

3.1 Challenges

Challenge 1: Naive transition incurs heavy I/O overhead. Suppose the size of an object is S, and we transit it from replication to EC. The amount of I/O for replica read is S, while that for EC block writes is $(k+m)S/k$. Thus, a total IO overhead of $(2k+m)S/k$ (more than 2 times the size of the object) is incurred in the naive transition. For example, in Fig. 3, the total I/O size is $7S/3$. The I/O overhead will further increase for EC with larger m/k and S.

Challenge 2: Concurrent requests may cause inconsistency. While redundancy transition is being conducted, the system needs to serve normal user requests. As shown in Fig. 3, EC changes a KV pair into $k+m$ smaller KV pairs, thus introducing distributed accesses. At one time point, there may be user write, user read, and transition write to the $k+m$ smaller KV pairs. Such concurrent access may cause inconsistency of the EC data.

3.2 Motivations

First, the reason for challenge 1 is that the data layout of replication is not suitable for being transmitted to the EC. Thus, naive transition needs to reproduce a totally new data layout. For example, in Fig. 4, one main copy of an object as well as a replica initially split into k data blocks are stored in two-way replication. During transition, we read the main copy, perform EC encoding, and only write m parity blocks. This eliminates the writes of k data blocks in transition.

Second, the reason for challenge 2 is that distributed accesses to multiple blocks split from an object are not atomic. Hence, concurrent requests may arrive at different nodes in different orders. We can detect these corelated requests and allow only one request to be executed at one time. Furthermore, we leverage the request being executed to serve subsequent read requests more efficiently. For example, a current write request can be cached such that subsequent read requests for the same object can be immediately responded to.

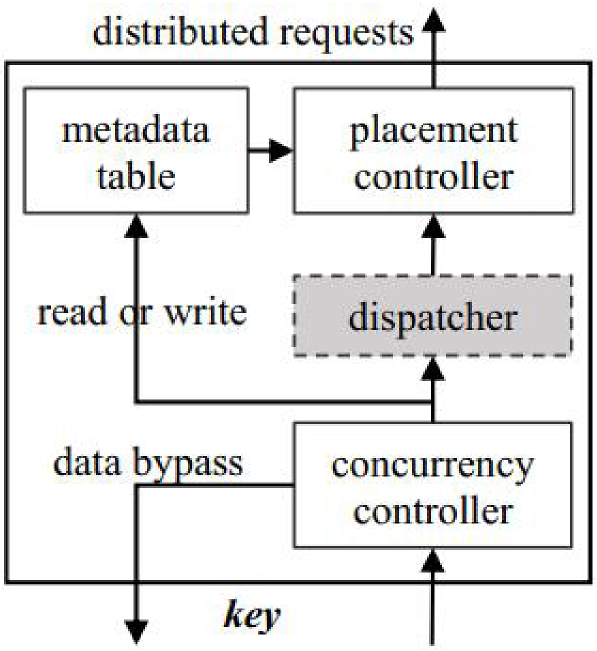


Fig. 5. Overall architecture. Modules identified by solid boxes are introduced to support our design.

4 Design

4.1 Overall architecture

The overall architecture of our design is shown in Fig. 5. Apart from the dispatcher module inherited from the in-memory store, we add a placement controller, concurrency controller, and metadata table. The placement controller determines the data placement of the objects in the nodes. It adopts EC-oriented replication to facilitate redundancy transition. The concurrency controller is a gate for all requests. It schedules the processing of requests and protects multiple concurrent accesses from triggering the consistency problem. It also coordinates concurrent transition requests and user requests. The metadata table records the redundancy scheme and the necessary metadata (e.g., key, value length) of each object.

4.2 Placement control

For an in-memory store using r-way replication and EC (k,m), we denote the corresponding EC-oriented replication (EOR) as EOR (r,k). The data placement of EOR is determined by the placement controller as follows. One main copy of the object is kept, while other r-1 replicas are all split into k data blocks as EC does. The main copy is stored in the first parity node, while k data blocks of one redundant replica are stored in k data nodes, and placement for data blocks of different redundant replicas are interleaved by employing a rotated strategy.

Due to the rotated placement strategy, EOR can tolerate any r-1 node failures. During redundancy transition, the system reads the main copy of the object, splits and encodes to generate the parity blocks. Later, the system only writes the parity blocks, as k data blocks of the first redundant replica are already stored in k data nodes. It avoids the writes of k

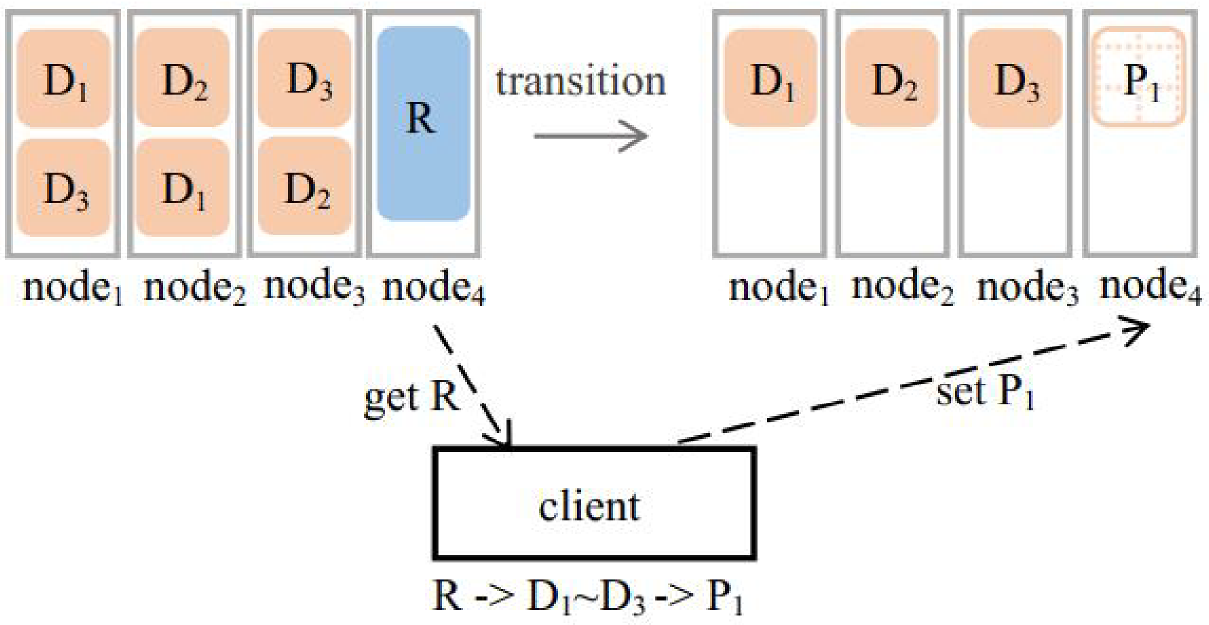


Fig. 6. Transition from EOR(3,3) to EC(3,1). IO overhead is reduced due to the new data placement of EOR.

data blocks in transition, and so it greatly saves the transition I/Os. It has achieved the lowest IO overhead.

We take EOR(3,3) as an example (refer to Fig. 6). Suppose the main node is computed as node$_1$; then, the main copy of the object is stored in node$_4$ (parity node), while the other two replicas are both split into k = 3 data blocks. The 3 data blocks of the first redundant replica are stored in node$_1$, node$_2$, and node$_3$. The 3 data blocks of the second redundant replica are stored in node$_2$, node$_3$, and node$_1$. Since one main copy is in node$_4$ and the other two replicas are split and stored in rotation in node$_1$, node$_2$, and node$_3$, EOR(3,3) tolerates any two node failures.

In redundancy transition, the client retrieves the main copy from node$_4$. It then splits the object into D_1–D_3 and encodes them into P_1. The client finally writes only P_1 to node$_4$. The k = 3 data blocks of the first redundant replica are kept in k = 3 data nodes, so we do not need any writes of data blocks. During transition, the system needs to delete some data (e.g., k = 3 data blocks in the second redundant replica and the main copy in node$_4$). The I/O overhead of deletion is small compared to read and write, as only I/O overhead of key size is caused. In addition, its latency can be hidden by immediately emitting asynchronous deletion requests after the read phase.

4.3 Concurrency control

With asynchronization, users could submit multiple requests concurrently. When it meets data blocks produced by EC and EOR, however, a risk of inconsistency occurs, which we define as the concurrent consistency problem.

4.3.1 Concurrent consistency problem

KV stores usually use an optimistic strategy and send multiple read requests to data nodes concurrently and individually. In our system, the optimistic strategy becomes error-prone in the presence of block storage. For example, the user issues read and write requests concurrently for an object (corelated requests) stored in blocks in Fig. 7. The subrequests for different blocks may interleave with each other due to the disorder during their executions. New data block D_2 may arrive at node$_2$ first, and before the other new blocks arrive, read requests all arrive. Thus, old data blocks D_1 and D_3 and new data block D_2 are returned to the client, resulting in inconsistency of the obtained data blocks. Generally, three categories of scenarios may encounter the problem of disorder, i.e., read after write, write after read, or write after write.

4.3.2 Work table and data bypassing

In our design, before a request is executed, it is passed to the concurrency controller first to be scheduled. Controller detects out corelated requests by work table, a dynamic table recording requests in progress and their contexts. It leverages temporary local data in memory to serve corelated read requests without actually executing the net request, which we denote as data bypassing.

As Scheme 1 shows, each row of the work table indicates a read or write request in progress for a unique key, and crucially, it records the context data for the request, which is the basis of how data bypassing works. For writing <k_1,v_1>, the context data are exactly v_1. While the write request is blocked on network processing, v_1 still resides in local memory and could be leveraged to serve subsequent read requests for k_1. For reading k_2, the context data is a future, which is a placeholder for incoming data. Suppose the data to be obtained is v_2; we can fetch it from the corresponding future after v_2 is in place. Before the arrival of v_2, an attempt to fetch data from the future would result in synchronization with v_2.

The concurrency controller registers a request with its context data in the work table when no request for the same key exists and deletes the corresponding table entry at the end of the request. When a request for the same key already exists, the controller judges the correlation between the registered request and each subsequent request and then takes different actions. We now describe how to make use of work tables and data bypassing to work correctly and efficiently in three cat-

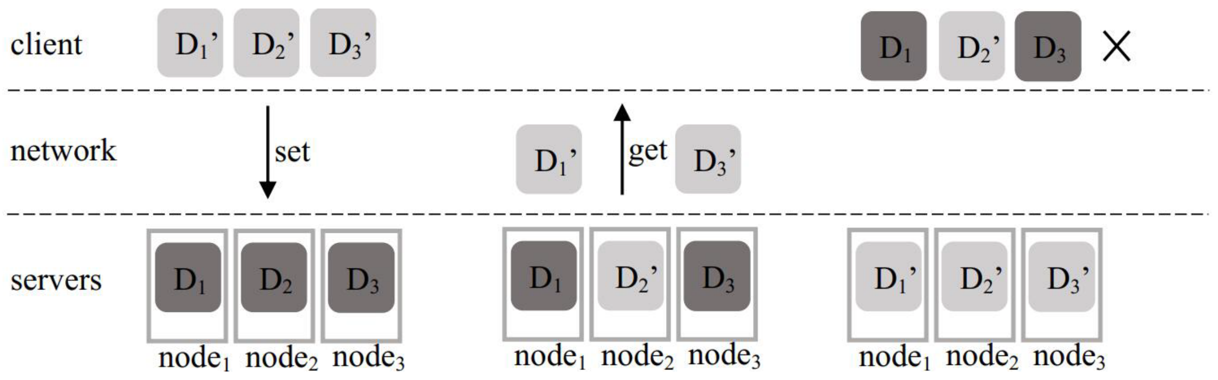


Fig. 7. Consistency problem triggered by writing and reading the same KV concurrently. Wrong data are returned.

key	IO state	context
k_1	set	v_1
k_2	get	future<v_2>
…	…	…

Scheme 1. Structure of work table, IO state is get or set, context indicates address of data to be written or read in local memory. Future means the data may not be in place, and synchronization is needed until data arrives.

egories of scenarios where the concurrent consistency problem occurs.

Read after write. When a write request for <k,v> has been registered in the work table with context data being v, followed by which a concurrent read request for k is issued, the concurrency controller simply fetches v via the work table and returns it to the read request. Just a table lookup is required and no subrequest interleaving is caused in the network.

Write after read. When a read request for k has been registered in the work table with context data being *future*<v_1> and a write request for <k,v_2> is issued concurrently, as the read request is already sent to the network, the concurrency controller cannot forward the write request right now to refrain from interleaving requests. It establishes the synchronization between two requests through *future*<v_1> and suspends write requests. Then, it modifies the table entry for k to state of writing <k,v_2> to provide data bypass and serve the subsequent corelated read requests correctly.

Write after write. When a write request for <k,v_1> has been registered in the work table, the concurrency controller only lets subsequent corelated write requests fail. Two reasons account for this. First, semantics for write requests are overwriting, which means that among concurrent corelated write requests, only one of them succeeds as a result. Second, it is difficult to clarify the exact order of concurrent requests. If users do want the request to succeed, they can repeat the request themselves after being informed of the failure of the request.

Although the read-after-read scenario does not cause inconsistency, the concurrency controller can still leverage data bypassing to accelerate the performance of corelated read requests. As a *future* means, a network request has been sent for data that subsequent corelated read requests want. Therefore, it is not necessary to execute subsequent read requests, and they synchronize with the *future* until the data they want arrives.

The work table combined with data bypassing not only solves the concurrent consistency problem but also exploits the correlation between requests and takes advantage of temporary data in local memory to improve concurrent performance.

We need to emphasize that the above consistency problem caused by concurrent access to objects stored in blocks is different from traditional consensus issues. If necessary, consensus algorithms[38,39] could be integrated.

4.3.3 Coordinate redundancy transition and access request

We could coordinate redundancy transition and access requests based on the same idea as work table and data bypassing. An additional transition table is maintained by the concurrency controller to detect transition requests.

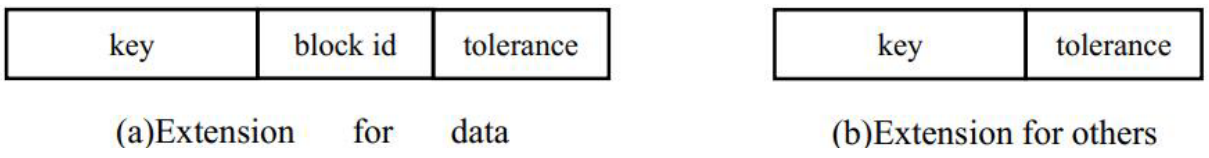
In our design, we do not support concurrent redundancy transition and corelated write requests. Because overwriting after the transition makes the transition meaningless, the transition after the write should be replaced by a write with a destined redundancy scheme. Therefore, we do not regard concurrent transition and write as reasonable in application. We simply let the subsequent transition and write requests fail.

After the read step and before the end of the transition, direct data bypass is available (no synchronization is needed). If corelated read requests appear during the read step of the transition, the controller synchronizes subsequent requests with the *future*.

4.4 Labeling data blocks of different replicas

Data placement of EOR introduces a new problem, distinguishing data blocks of different redundant replicas. If r is greater than two, the number of redundant replicas is no less than two; as a result, each data node stores at least two data blocks, and the two data blocks are different for the interleaving placement of redundant replicas. Data blocks are KVs stored in data nodes in essence; thus, different data blocks on the same node should be distinguished from each other by different keys.

We propose a key management protocol to resolve key conflicts. Key management extends a key to a new key by appending a 1–2 byte suffix identifying the redundancy scheme



Scheme 2. Key management is composed of two extension schemes for data blocks and others.

and an optional block ID (Scheme 2).

For data block splitting from <*k*,*v*>, the placement controller extends *k* by appending 1 byte identifying different blocks and 1 byte identifying a redundancy scheme. The placement controller uses this extension for EC and EOR and sets their tolerance field equal to keep data placement alike.

For <*k*,*v*> stored with another redundancy scheme without blocks, the placement controller simply extends *k* by appending 1 byte identifying the redundancy scheme. Specifically, the primary replica of EOR also uses this extension.

5 Implementation

We prototype ElasticMem, a hybrid fault-tolerant distributed in-memory KV store atop Memcached. We implement the clustered architecture of ElasticMem atop aiomcache —a Memcached client implemented in Python. ElasticMem supports many fault-tolerant schemes, including replication, EC, EOR, and None (i.e., no redundancy and fits for transient data[19,27,28]).

Thread model. ElasticMem is able to issue concurrent IO requests in a single-thread environment due to IO multiplexing. Under single-threading, it would be better to keep computation efficient to saturate NIC; thus, we use Python/C api to leverage Jerasure and gf-complete[30] for Galois-Field operations in RS code, and as a requirement of gf-complete, the size of data blocks is aligned to 16 bytes.

Metadata. On the server side, each key is appended a 1~2 byte suffix owing to key mangling. On the client side, a metadata table is maintained to record the redundancy scheme and value length of each KV pair. We show that the metadata storage overhead can be neglected. We consider objects with a common key size (30 bytes[33]) and a large value size (256 KB). Metadata in total are 35 bytes on the client side or approximately 2 bytes on the server side for an object. Hence, for a 100 GB data volume, metadata account for 12 MB for the client or 800 KB for the servers. In addition, metadata lookup overhead is already contained in read or write operations and shows no obstacle in the following experiments.

6 Evaluation

We carry out numerical analysis and testbed experiments. From numeric analysis, ElasticMem reduces the amount of I/Os of naive transition by 25% – 40%. From testbed experiments, ElasticMem reduces the transition time of naive transition by up to 35% and reduces the latency of a single access request to 6 μs at most.

6.1 Numerical analysis

We analyze the I/O overhead of naive transition and improved transition under EOR. Suppose the size of a KV pair is *S*, and the redundancy schemes are *r*-way replication (i.e., Rep(*r*)), EC(*k*,*m*), and EOR(*r*,*k*). We use *rep_to_ec* to denote the transition from Rep(*r*) to EC(*k*,*m*) (others are similar). We define *rep_to_ec* and *eor_to_ec* as forward transitions and *ec_to_rep* and *ec_to_eor* as reverse transitions. In the read step, the I/O overhead of *S* is produced as complete data are needed. In the write step, I/O overhead varies under different transitions.

Table 1 lists the numerical results. Usually, *k* is much greater than *m*, and we can suppose *k* is greater than 2*m*; thus, the improved forward transition can reduce the I/O overhead by more than 40%. Typically, *r* is 2 or 3, and the improved reverse transition can reduce I/O overhead by 33% or 25%.

6.2 Testbed experiments

Setup. We conduct experiments on a local cluster comprised of 7 nodes, each of which has a 40-core 2.4 GHz Intel Xeon Gold 5115, 128 GB of RAM and 10 Gbps network. We run 6 of 7 nodes as ElasticMem servers and the remaining node as the client.

Methodology. We adopt several sets of (*k*,*m*,*r*) to configure the fault-tolerant scheme. We consider different value sizes. We measure the normal read and write latency of an object as well as the latency of forward transition and reverse transition under various settings. We also measure the latency of a single request and the overall latency in the presence of corelated concurrent read and write requests.

Experiment 1 (Normal read and write latency under different redundancy schemes). We evaluate the normal I/O performance of ElasticMem under (*k*,*m*,*r*) = (4,2,3). Rep(*r*) if the scheme in vanilla Memcached. We let *m* = *r*–1replication and EC can tolerate the same node failures. Fig. 8 shows the results.

As Fig. 8 shows, Rep and EOR have almost the same read performance. It is reasonable, as both schemes handle read requests with the same process. In regard to write performance, EOR incurs higher latency when the value size is less than 64 KB and has a similar write performance to Rep when the value size is larger than 64 KB. In summary, EOR incurs slightly more I/O overhead than Rep. As read requests take up a larger proportion in a cache system, we can deduce that EOR has similar access performance to Rep.

As the value size increases, the read performance of EC shows a trend to outperform Rep. The reason is that EC can

Table 1. Numerical results of I/O overhead for redundancy transition and improvement.

transition	I/O overhead	reduced I/O ratio under EOR
rep_to_ec	$\frac{2k+m}{k}S$	$\frac{k}{2k+m}$
eor_to_ec	$\frac{k+m}{k}S$	
ec_to_rep	$(r+1)S$	$\frac{1}{r+1}$
ec_to_eor	rS	

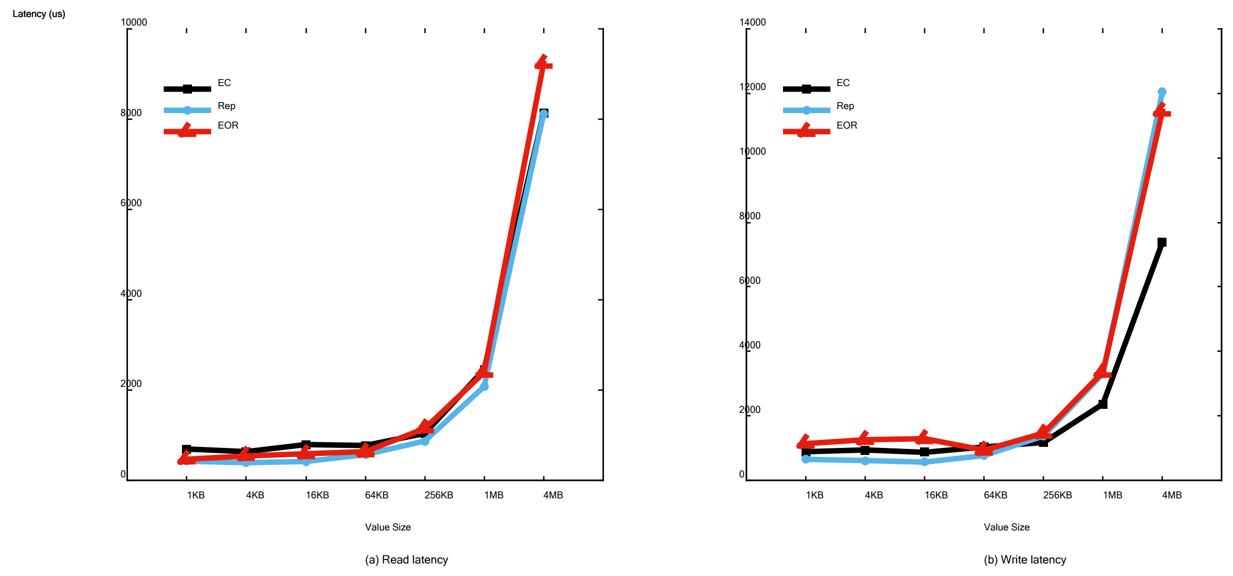


Fig. 8. Normal read and write performance with different redundancy schemes under (k,m,r) = (4,2,3)

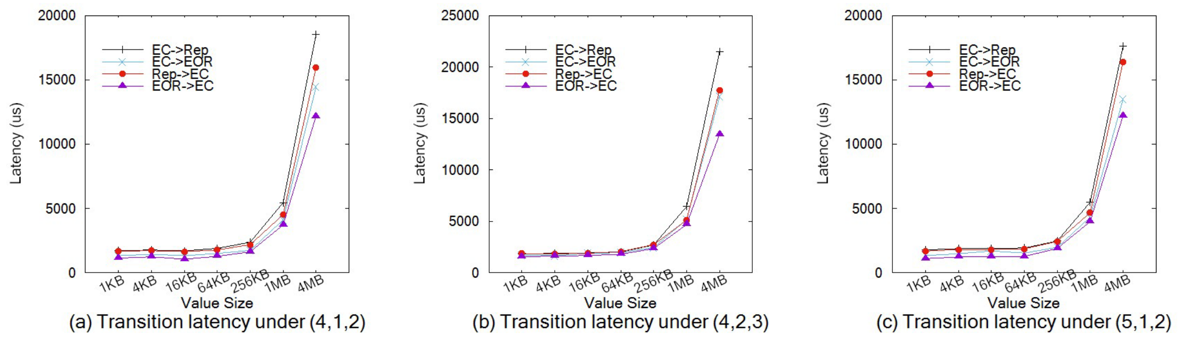


Fig. 9. Transition performance under different (k,m,r).

better exploit read parallelism and the pipelined operations of CPU decoding and NIC transfers. As the value size increases, the write performance of EC is notably better than that of Rep. The reason is that the I/O overhead incurred by the EC write is much smaller.

Experiment 2 (Performance of redundancy transition). We evaluate the improvement of transition performance under EOR. We consider three sets of (k,m,r), i.e., (4,1,2), (4,2,3), and (5,1,2). The value size is varied from 1 KB to 4 MB. Fig. 9 shows comparisons of naive transition and improved transition.

We can see that the improved transition constantly outperforms the naive transition, and the proportion of improved time shows a trend to increase with value size. EOR reduces the forward transition time by 35% at most and 8% at least and the reverse transition time by 26% at most and 3% at least. Generally, the improved performance of forward transition is slightly more than that of reverse transition, as the saved I/O overhead takes up a larger proportion in forward transition.

By comparing the transition performance under (4,1,2) and (4,2,3), we can see that with a greater m/k and r, the improved transition has a smaller improvement over the naive transition. This is because the saved I/O overhead occupies a smaller proportion in transition, and it is consistent with the numerical results in Table 1. In Fig. 9c, there is no more performance improvement. We also observe that the improvement of the improved transition under EOR hardly increases with a large k.

Experiment 3 (Performance of corelated concurrent read, write). We issue corelated get or set requests to measure the improvement brought by data bypassing. We adopt (k,m,r) = (4,1,2) as in experiment 1. We consider a value of size 1 KB. Roughly we know from Fig. 8 that the read and write latency for a 1 KB value employing Rep(2) are approximately 500 µs and 600 µs, respectively.

We first test the get-after-set scenario. We issued a set request followed by 3 corelated get requests and recorded the overall latency and latency of every single request. Table 2 shows the results.

The write performance fluctuates and is 350 µs more than we just learned, but we can see the subsequent get requests return responses in approximately 10 µs only, as each corelated get request only performs a table lookup operation and simply returns local data in memory. The overall latency of 4 requests is 1038 µs, which is much lower than dealing with requests in sequence.

Table 2. Performance of corelated get and set requests.

	set	get	get	get	overall
latency (µs)	952	13	7	6	1038

Table 3. Performance of corelated get requests.

	get	get	get	get	overall
latency (μs)	445	404	396	393	567

Then, we test the get-after-get scenario, which is more likely to appear than the previous case. Table 3 shows the results**.**

From Table 3, we find that the latency of each subsequent get request decreases, as the get request issued later waits for less time until the requested data are transferred over the network owing to data bypassing. On average, each subsequent get request incurs only 41 μs latency for overall latency, which is a significant improvement for read performance.

7 Conclusion and future work

We design ElasticMem, which implements a hybrid fault-tolerant scheme adopting replication and erasure code to make a better trade-off of performance and memory usage in providing data availability. ElasitcMem supports per-object redundancy and dynamically adjusts the redundancy of each object to adapt to changing user demands. ElasticMem exploits EC-oriented Replication to accelerate redundancy transition while promising the same I/O performance as normal replication. By detecting the correlation of concurrent access requests, ElasticMem leverages data bypassing to solve the concurrent consistency problem and significantly improves the performance of corelated requests at the same time.

We can extend our work in the following ways. First, we can make the server side aware of hybrid fault tolerance to gain performance improvement in the transition from EOR to EC. Currently, we only reimplement the client side to support hybrid fault tolerance on clustered KV stores, reusing the memcached network communication protocol. If not limited to this lightweight design, we could enable the server side to perform RS encoding and send parity blocks among servers. In this way, when performing the transition from EOR to EC, as the first parity node holds the monolithic object, which is all we need for encoding, I/O overhead could be further reduced.

In addition, memory pooling based on CXL[34,35] or RDMA[36,37] has recently attracted much attention. In such architecture, memory is disaggregated from computational resources to improve resource utilization and elasticity. Building a high-performance hybrid fault-tolerant system atop limited interfaces provided by the memory pool is worth trying.

Acknowledgments

This work was supported by the Fundamental Research Funds for the Central Universities (WK2150110022), Anhui Provincial Natural Science Foundation (2208085QF189), and NSFC (62202440).

Conflict of interest

The authors declare that they have no conflicts of interest.

Biographies

Zheng Gong is currently a master candidate. His research mainly focuses on fault-tolerant storage systems.

Si Wu received B.E. and Ph.D. in computer science from the University of Science and Technology of China (USTC) in 2011 and 2016. He is currently an Associate Researcher with the School of Computer Science and Technology, USTC. His research interests include storage reliability and distributed storage.

References

[1] Reinsel D, Gantz J, Rydning J. The Digitization of The World from Edge to Core. Framingham, MA: International Data Corporation, **2018**.

[2] Apache. HDFS Design. **2021.** https://hadoop.apache.org/docs/r3.3.1/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html. Accessed July 01, 2022.

[3] Ghemawat S, Gobioff H, Leung S T. The Google file system. *In: Proceedings of the nineteenth ACM symposium on Operating systems principles. New York: ACM*, **2003**, 29–43.

[4] Apache. Spark. **2021**. https://spark.apache.org/. Accessed July 01, 2022.

[5] Apache. Storm. **2021**. https://storm.apache.org/index.html. Accessed July 01, 2022.

[6] J. Zawodny. Redis: Lightweight key/value store that goes the extra mile. **2009**, 79. www.linux.com/news/redis-lightweight-keyvalue-store-goes-extra-mile. Accessed July 01, 2022.

[7] B Fitzpatrick. Distributed caching with memcached. *Linux Journal*, **2004** *124*: 5.

[8] Nishtala R, Fugal H, Grimm S, et al. Scaling memcache at Facebook. In: 10th USENIX Symposium on Networked Systems Design and Implementation. Lombard, IL: USENIX Association, **2013**, 385–398.

[9] Twitter Inc. Twemcache is the twitter memcached. **2012**. https://github.com/twitter/twemcache. Accessed July 01, 2022.

[10] Goel A, Chopra B, Gerea C, et al. Fast database restarts at facebook. In: ProcIn: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. New York: ACM, **2014**, 541–549.

[11] Budhiraja N, Marzullo K, Schneider F B, et al. The primary-backup approach. In: Distributed systems, New York: ACM Press/Addison-Wesley Publishing Co. **1993**: 199–216.

[12] van Renesse R, Schneider F. Chain replication for supporting high throughput and availability. In: 6th Conference on Symposium on Operating Systems Design & Implementation. San Francisco, CA: USENIX Association, **2004**, 91–104.

[13] Lai C, Jiang S, Yang L, et al. Atlas: Baidu's key-value storage system for cloud data. In: 2015 31st Symposium on Mass Storage Systems and Technologies (MSST). Santa Clara, USA: IEEE, **2015**, 1–14.

[14] Li S, Zhang Q, Yang Z, et al. BCStore: Bandwidth-efficient in-memory KV-store with batch coding. In: 33nd International Conference on Massive Storage Systems and Technologies. Santa Clara, USA: MSST, **2017**.

[15] Rashmi K V, Chowdhury M, Kosaian J, et al. EC-Cache: Load-balanced, low-latency cluster caching with online erasure coding. In: Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation. New York: ACM, **2016**, 401–417.

[16] Yiu M M T, Chan H H W, Lee P P C. Erasure coding for small objects in in-memory KV storage. In: Proceedings of the 10th ACM International Systems and Storage Conference. New York, USA: ACM, **2017**, 14.

[17] Xu L, Lyu M, Li Q, et al. SelectiveEC: Towards balanced recovery load on erasure-coded storage systems. *IEEE Transactions on Parallel and Distributed Systems*, **2022**, *33* (10): 2386–2400.

[18] MacWilliams F J, Sloane N J A. The theory of error-correcting codes. In: North-Holland Mathematical Library. Murray Hill, USA: Bell Laboratories, **1977**, 16.

[19] Yang J, Yue Y, Rashmi K V. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), Berkeley CA, USA: USENIX Association, **2020**, 191–208.

[20] Xia M, Saxena M, Blaum M, et al. A tale of two erasure codes in HDFS. In: 13th USENIX conference on file and storage technologies (FAST' 15). Santa Clara, USA: USENIX Association, **2015**, 213–226.

[21] Yao Q, Hu Y, Cheng L, et al. StripeMerge: Efficient wide-stripe generation for large-scale erasure-coded storage. In: 2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS). DC, USA: IEEE, **2021**, 483–493.

[22] Wu S, Shen Z, Lee P P C. Enabling I/O-efficient redundancy transitioning in erasure-coded KV stores via elastic reed-solomon codes. In: 2020 International Symposium on Reliable Distributed Systems (SRDS). Shanghai, China: IEEE, **2020**, 246–255.

[23] Chen H, Zhang H, Dong M, et al. Efficient and available in-memory KV-store with hybrid erasure coding and replication. *ACM Transactions on Storage*, **2017**, *13* (3): 25.

[24] M. Kerrisk. The Linux Programming Interface. San Francisco, USA: No Starch Press, **2010**.

[25] Libmemcached. https://libmemcached.org/libMemcached.html. Accessed July 01, 2022.

[26] Plank J S. A tutorial on Reed–Solomon coding for fault‐tolerance in RAID‐like systems. *Software:Practice and Experience*, **1997**, *27* (9): 995–1012.

[27] Stuedi P, Trivedi A, Pfefferle J, et al. Unification of temporary storage in the nodekernel architecture. In: Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference. New York: ACM, **2019**, 767–781.

[28] Klimovic A, Wang Y, Stuedi P, et al. Pocket: Elastic ephemeral storage for serverless analytics. In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), Carlsbad,USA: 2018: 427–444.

[29] Plank J S, Simmerman S, Schuman C D. Jerasure: A library in C/C++ facilitating erasure coding for storage applications. Knoxville, TN, USA: University of Tennessee, **2007**.

[30] Plank J S, Miller E L, Houston W B. GF-Complete: A comprehensive open source library for Galois Field arithmetic. Knoxville, TN, USA: University of Tennessee, **2013**.

[31] Xu B, Huang J, Cao Q, et al. TEA: A traffic-efficient erasure-coded archival scheme for In-memory stores. In: Proceedings of the 48th International Conference on Parallel Processing. New York: ACM, **2019**, 24.

[32] Li R, Hu Y, Lee P P C. Enabling efficient and reliable transition from replication to erasure coding for clustered file systems. *IEEE Transactions on Parallel and Distributed Systems*, **2017**, *28*: 2500–2513.

[33] Atikoglu B, Xu Y, Frachtenberg E, et al. Workload analysis of a large-scale key-value store. In: Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems. New York: ACM, **2012**, 53–64.

[34] Li H, Berger D S, Hsu L, et al. Pond: CXL-based memory pooling systems for cloud platforms. In: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. New York: ACM, **2023**, *2*: 574–587.

[35] Guo Z, Shan Y, Luo X, et al. Clio: A hardware-software co-designed disaggregated memory system. In: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. New York: ACM, **2022**: 417–433.

[36] Cai Q, Guo W, Zhang H, et al. Efficient distributed memory management with RDMA and caching. *Proceedings of the VLDB Endowment*, **2018**, *11*: 1604–1617.

[37] Zuo P, Sun J, Yang L, et al. One-sided RDMA-conscious extendible Hashing for disaggregated memory. In: Proceedings of the 2021 USENIX Annual Technical Conference. Berkeley, CA, USA: USENIX, **2021**, 15–29.

[38] Ongaro D, Ousterhout J. In search of an understandable consensus algorithm. In: Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference. New York: ACM, **2014**, 305–320.

[39] Wang Z, Li T, Wang H, et al. CRaft: An erasure-coding-supported version of raft for reducing storage cost and network cost. In: FAST'20: Proceedings of the 18th USENIX Conference on File and Storage Technologies. New York: ACM, **2020**, 297–308.