LogECMem: Coupling Erasure-Coded In-Memory Key-Value Stores with Parity Logging

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

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

Zhaokang Ke
Huazhong University of Science and Technology
Wuhan, China
2020612167@hust.edu.cn

Jia Xu
Huazhong University of Science and Technology
Wuhan, China
helenxu@hust.edu.cn

Qiaori Yao
Huazhong University of Science and Technology
Wuhan, China
yaoqr@hust.edu.cn

Dan Feng
Huazhong University of Science and Technology
Wuhan, China
dfeng@hust.edu.cn

Weichun Wang
HIKVISION
Hangzhou, China
wangweichun@hikvision.com

Wei Chen
HIKVISION
Hangzhou, China
chenwei7@hikvision.com

ABSTRACT

In-memory key-value stores are often used to speed up Big Data workloads on modern HPC clusters. To maintain their high availability, erasure coding has been recently adopted as a low-cost redundancy scheme instead of replication. Existing erasure-coded update schemes, however, have either low performance or high memory overhead. In this paper, we propose a novel parity logging-based architecture, HybridPL, which creates a hybrid of in-place update (for data and XOR parity chunks) and log-based update (for the remaining parity chunks), so as to balance the update performance and memory cost, while maintaining efficient single-failure repairs. We realize HybridPL as an in-memory key-value store called LogECMem, and further design efficient repair schemes for multiple failures. We prototype LogECMem and conduct experiments on different workloads. We show that LogECMem achieves better update performance over existing erasure-coded update schemes with low memory overhead, while maintaining high basic I/O and repair performance.

CCS CONCEPTS

• **Computer systems organization** → **Redundancy**; • **Information systems** → **Distributed storage**.

KEYWORDS

Erasure coding, Key-value stores, Update, Parity logging

ACM Reference Format:
Liangfeng Cheng, Yuchong Hu, Zhaokang Ke, Jia Xu, Qiaori Yao, Dan Feng, Weichun Wang, and Wei Chen. 2021. LogECMem: Coupling Erasure-Coded In-Memory Key-Value Stores with Parity Logging. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3458817.3480852

1 INTRODUCTION

Today's parallel filesystem on modern HPC clusters [52] often rely on in-memory key-value (KV) stores for high-performance data analytics [35, 45, 61]. For instance, in-memory KV stores like Memcached [9] are often used to keep primary data in memory as KV items (called *objects*) to improve performance significantly compared to on-disk solutions [3, 10, 41].

To maintain availability, many distributed KV stores replicate data into multiple copies distributed across nodes (e.g., Dynamo [21] and Cassandra [32]) to tolerate failures, which yet incurs high redundancy. Recent studies [18, 34, 48, 59, 62] leverage erasure coding for in-memory KV stores so as to provide availability guarantees at much lower cost compared to replication. Erasure coding encodes original *data chunks* into *parity chunks* such that a subset of a group of data/parity chunks (which collectively form a *stripe*) can reconstruct the original data chunks.

For many big data analytics workloads, updates are often indispensable [58, 62] or even heavy (e.g., 50%:50% ratio of read to update [29]), and updates are expensive in erasure-coded storage as any updated data chunk will cause all parity chunks of the same stripe to be updated. Recently, the erasure-coded updates are considered in a large-scale cluster (called *wide stripes* [25]) where each stripe has a very large size and all its chunks are dispersed over dozens or even hundreds of nodes (see §2.2.1 for details).

There are three major ways of performing erasure-coded updates [53]: 1) *in-place update*, where the old data and parity chunks are replaced with the new ones, 2) *full-stripe update*, where the new data chunks are encoded into new stripes directly, and 3) *parity logging*, where parity updates are inserted to the log nodes. Among these three schemes, in-place update costs significant network transfers and full-stripe update costs high storage overhead (see §2.2 and §2.3 for details), while parity logging that eliminates the reads of parity chunks during updates with low storage overhead is state-of-the-art.

However, existing in-memory KV stores with erasure coding only apply in-place update [18, 62] or full-stripe update [34], while none has considered parity logging, which can actually reduce network transfers for high update performance with low memory footprint. The reason is that to deploy parity logging under in-memory KV stores, it is challenging to handle the performance gap between memory and disk-based log nodes, since the latter has much lower transfer rate than the former. Specifically, for erasure-coded in-memory KV stores, even if parity logging can reduce the network transfers during updates, the log nodes will still greatly degrade the update performance; moreover, the single-failure repair performance will also be lowered significantly via parity chunks residing in log nodes.

In this paper, we propose a new architecture for erasure-coded in-memory KV stores, called HybridPL, which takes a hybrid of in-place update and parity logging. First, HybridPL allows all data and some parity chunks to perform in-place update in DRAMs for fast single-failure repairs while maintaining low memory overhead. Second, HybridPL enables the other parity chunks to perform parity logging to reduce chunk transfers during update, and moreover, HybridPL develops a buffer-logging [42] based scheme to accelerate updates of parity chunks in log nodes. We realize the above architecture as an in-memory KV store called LogECMem, and further design efficient multi-failure repair schemes atop LogECMem. The contributions of this work include:

- We observe via workloads that the existing update schemes for erasure-coded in-memory KV stores either incur many chunk transfers or require high memory overhead, which motivates us to develop a parity logging based scheme to balance the cost of both update and memory (§2).
- We show via reliability analysis that improving single-failure repairs can significantly increase the reliability, so we propose HybridPL which performs in-place update for fast single-failure repairs with low memory footprint, while leveraging parity logging as well as buffer logging for fast updates (§3).
- We build an in-memory KV store LogECMem atop HybridPL, which realizes the basic requests (including single-failure repairs), parity logging based updates, and buffer logging. We also improve buffer logging by merging parity updates in batch (§4).
- We design efficient repair schemes atop LogECMem for multi-failure repairs to reduce disk IOs over state-of-the-art parity logging based repair schemes, while still maintaining high repair performance (§5).
- We prototype LogECMem and compare it with in-place update and full-stripe update via Amazon EC2 experiments. We show that LogECMem reduces the update time by up to 37.8% and 58.0%, and the memory overhead by up to 22.2% and 49.0%, respectively, while maintaining high basic I/O and repair performance (§6). The source code of our prototypes is now available at **https://github.com/yuchonghu/logecmem**.

2 BACKGROUND AND OBSERVATIONS

We provide the background details of erasure coding basics (§2.1) and describe existing erasure-coded update schemes (§2.2). We provide observations to show that full-stripe update cannot balance the costs of update and memory based on YCSB workloads, which motivates our main idea that introduces parity logging to reduce the parity chunk transfers as well as the memory overhead (§2.3).

2.1 Primer on Erasure Coding

An erasure coding scheme, denoted by a (k, r) code, can be constructed by two configurable parameters k and r. A (k, r) code organizes KV objects as fixed-size *data chunks*. For every set of k data chunks, the KV store encodes them into additional r equal-size *parity chunks*, such that any k out of the $k + r$ data/parity chunks (collectively called a *stripe*) suffice to rebuild the original k data chunks.

Recent studies (see survey [44] and §8) have proposed many erasure coding constructions, among which *Reed-Solomon (RS) codes* [49] remain the most popular erasure codes and are recently studied in in-memory KV stores [18, 34, 48, 59, 62], so we focus on (k, r) RS codes in this paper. Specifically, for a stripe composed of data and parity chunks denoted by D_i $(1 \leq i \leq k)$ and P_j $(1 \leq j \leq r)$ respectively, the parity chunks are calculated from a linear combination of the data chunks as $P_j = \sum_{i=1}^{k} \alpha_j^{i-1} D_i$, where α_j^{i-1} $(1 \leq i \leq k$ and $1 \leq j \leq r)$ are encoding coefficients. For example, for a $(3, 1)$ code, the parity chunk $P_1 = D_1 + \alpha_1 D_2 + \alpha_1^2 D_3$.

Based on the definition of RS codes, we can have the following definitions.

XOR parity chunks: refer to the first parity chunk of each stripe (i.e., P_1) is often the XORing of all data chunks (i.e., $\alpha_1 = 1$), which can simply help repair a data chunk, e.g., $D_1 = P_1 - D_2 - D_3$.

Delta: refers to the change between the old and new data chunks, e.g., $\Delta D_1 = D_1' - D_1$ when an old data chunk D_1 is updated to a new data chunk D_1'.

Parity delta: refers to the change between the old and new parity chunks, e.g., $\Delta P_1 = P_1' - P_1$ when an old parity chunk P_1 is updated to a parity chunk P_1'.

Note that RS codes are basically based on linear encoding, so when updating data chunks we can have two properties as follows:

- Property 1: The parity delta can be generated from the delta; the parity deltas of all parity chunks of the same stripe can be computed based on the *same* delta. For example, for a $(3, 2)$ code, when D_2 is updated to D_2', we can compute $\Delta P_1 = P_1' - P_1 = (D_1 + \alpha_1 D_2' + \alpha_1^2 D_3) - (D_1 + \alpha_1 D_2 + \alpha_1^2 D_3) = \alpha_1 \Delta D_2$, where $\Delta D_2 = D_2' - D_2$. Similarly, we have $\Delta P_2 = \alpha_2 \Delta D_2$; ΔP_1 and ΔP_2 have the same delta ΔD_2.
- Property 2: For multiple data chunk updates of the same stripe, a parity chunk's corresponding parity deltas can be reduced to one. For example, for a $(3, 1)$ code, when D_1 is first updated D_1' and then D_2 is updated to D_2', we can compute P_1's two corresponding parity deltas as $D_1' - D_1$ and $\alpha_1(D_2' - D_2)$, which can be combined into one parity delta $D_1' - D_1 + \alpha_1(D_2' - D_2)$ such that the up-to-date parity chunk can be computed by P_1 and $D_1' - D_1 + \alpha_1(D_2' - D_2)$.

We will leverage the two properties in §3.3, §4.2, §4.3, and §5.2.

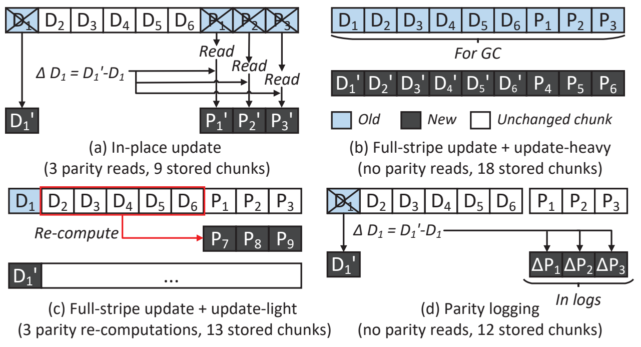

Read
Read
Read
Δ D1 = D1'-D1
(a) In-place update
(3 parity reads, 9 stored chunks)
For GC
Old
New
Unchanged chunk
(b) Full-stripe update + update-heavy
(no parity reads, 18 stored chunks)
Re-compute
(c) Full-stripe update + update-light
(3 parity re-computations, 13 stored chunks)
In logs
(d) Parity logging
(no parity reads, 12 stored chunks)

Figure 1: Illustration of different parity update schemes

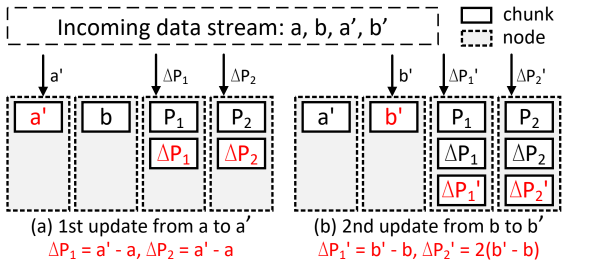

Incoming data stream: a, b, a', b'
chunk
node
(a) 1st update from a to a'
ΔP1 = a' - a, ΔP2 = a' - a
(b) 2nd update from b to b'
ΔP1' = b' - b, ΔP2' = 2(b' - b)

Figure 2: Illustration on parity logging in $(2, 2)$ code. We assume an incoming data stream: a, b, a', b', where a and b are original data chunks, a' is updated from a, and b' is updated from b. a and b form a stripe with $P_1 = a + b$ and $P_2 = a + 2b$.

2.2 Parity Update Schemes

Parity updates in erasure-coded storage incur significant network transfer, since they need to re-compute parity chunks for consistency. We re-examine existing parity update schemes that fall into four classes: direct reconstruction, in-place update, full-stripe update, and parity logging.

Direct reconstruction: A straightforward way is to first read all the data chunks that are not involved in the update and then reconstruct the new parity chunks using the read chunks and the new data chunks, which clearly costs a large number of chunk transfers.

In-place update: In-place update [12, 18, 62] reads old parities, computes parity deltas via delta (§2.1), generates new parities via parity deltas, and replaces the old parities. However, in-place update incurs additional parity chunk reads, which hinders the update performance. Figure 1(a) updates an old data chunk D_1 to a new one D_1', and generates the corresponding new parity chunks P_1', P_2' and P_3' by merging the old parity chunks P_1, P_2 and P_3 with the *delta* $\triangle D_1 = D_1' - D_1$, incurring three additional parity chunk reads.

Full-stripe update: To eliminate the reads of parity chunks, full-stripe update directly encodes new data chunks into new stripes, and marks the old data chunks as invalid such that they can be released directly via garbage collection (GC). It has been deployed in practical systems, e.g., QFS [43], BCStore [34], and Giza [17]. However, full-stripe update may incur high storage overhead to store stale data chunks, and require parity re-computations for the remaining active unchanged data chunks during GC. Figure 1(b) considers an update-heavy workload which updates multiple old data chunks $D_1, D_2, \ldots, D_6$ to the new data chunks $D_1', D_2', \ldots, D_6'$, which are encoded into a new stripe without parity chunk reads, yet requiring high storage overhead of 18 chunks. Figure 1(c) considers an update-light workload which updates an old data chunk D_1 to a new one D_1', incurring three additional parity chunk re-computations for the active unchanged data chunks $D_2, D_3, \ldots, D_6$.

Parity logging: Parity logging (PL) [30, 56] improves in-place update by logging the parity deltas in log devices, so as to reduce the read overhead of old parity chunks during updates, while maintaining lower storage overhead than full-stripe update. Figure 1(d) logs the parity deltas $\triangle P_1, \triangle P_2, \triangle P_3$ at log nodes, so as to eliminate the parity chunk reads while only storing 13 chunks. Note that parity logging is only adopted in disk-based distributed storage [16, 30, 56], and *our work is the first to study how parity logging works for in-memory KV stores*.

Figure 2 illustrates the details of PL. PL only needs to write the following parity deltas ΔP_1, ΔP_2, $\Delta P_1'$, $\Delta P_2'$ in log nodes, without reading P_1 and P_2. In this way, when we want to read any up-to-date parity chunk, we only need to retrieve its original parity chunk and all its parity deltas, and then combine them into the up-to-date parity chunk. For example, in Figure 2, we can obtain the up-to-date chunk of the first parity via $P_1 + \triangle P_1 + \triangle P_1'$.

2.2.1 Large-scale erasure-coded updates.

Recall that recent studies on erasure-coded updates have begun to focus on a large-scale cluster in §1; i.e., k is very large (e.g., k=128 [25]). In this case, each stripe is so wide that full-stripe update will have a lot of remaining active unchanged data chunks of the stripe during GC, thus incurring significant network transfers for parity re-computations to form a new stripe.

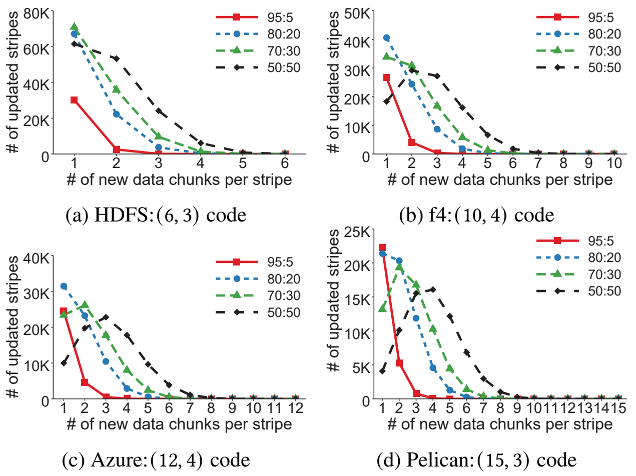
Thus, the recent proposed notion wide stripe [25] motivates us to explore in-place update and parity logging instead of full-stripe updates in a large-scale cluster, since the first two update schemes only update parity chunks based on deltas, regardless of how wide the stripe is. Actually, our main idea is a hybrid of in-place update and parity logging for parity updates, which will be specified in §3.

2.3 Observations

We have illustrated two observations of full-stripe update in Figure 1: high memory overhead for update-heavy workloads and many chunk transfers for re-computations of remaining active data chunks for update-light workloads. Here, we verify these two observations via workloads as follows.

Observation 1: The workloads are generated by Yahoo! Cloud Serving Benchmark (YCSB) [20]. Specifically, we load one million objects with fixed 4KiB-size value, and every k objects forms a stripe in a FIFO mode. We also set one million requests based on Zipfian distribution versus different read/update ratios (e.g., the most update-heavy case refers to read:update = 50%:50%),with different (k, r) codes: $(6, 3)$ as in HDFS [46], $(10, 4)$ as in f4 [40], $(12, 4)$ as in Azure [27], and $(15, 3)$ as in Pelican [15].

Figure 3 shows the number of stripes that are updated for different number of new chunks per stripe. For example, in Figure 3(a), for update-light workload (we consider read:update = 95%:5% to be update-light in this paper), we see that most of the updated stripes have only one new data chunk, meaning that the active unchanged $k - 1$ data chunks of each of those stripes have to be retrieved to re-compute the parity chunks. When the workload becomes more



(a) HDFS: $(6,3)$ code (b) f4: $(10,4)$ code (c) Azure: $(12,4)$ code (d) Pelican: $(15,3)$ code

Figure 3: Observation 1: The number of updated stripes, versus the number of new chunks per stripe.

	95%:5%	80%:20%	70%:30%	50%:50%
In-place update	M	M	M	M
Full-stripe update	$1.05M$	$1.2M$	$1.3M$	$1.5M$

Table 1: Observation 2: Memory overhead of in-place update and full-stripe update.

update-heavy, more updated stripes have more new chunks. For the most update-heavy workload (we consider read:update = 50%:50% to be update-heavy in this paper) some of the updated stripes can have three or more new data chunks, thereby effectively reducing the retrieved chunks. Figure 3(b), (c) and (d) have similar results.

Note that when k is large, the remaining active unchanged chunks per stripe become more, so degrades the efficiency of re-computation of the parity chunks, which is consistent with §2.2.1.

Observation 2: We let the total size of all objects be M, and let the ratio of update requests be p (e.g., read:update = 95% : 5% means p = 5%). We can calculate memory overheads of in-place and full-stripe updates based on details in §2.2. Table 1 shows that full-stripe has only 5% more memory overhead than in-place for the update-light workload (i.e., read:update = 95%:5%), but the additional memory increases when the workload becomes more update-heavy, and even reaches 50% for the most update-heavy workload (i.e., read:update = 50%:50%), which may not be cost-efficient.

Main idea: Based on the above observations, we find that it is difficult for full-stripe update to trade off both the cost of update (in terms of chunk transfers) and memory overhead for any update workload. Also, in-place update always incurs additional chunk transfers to read parity chunks for any update workload (§2.2). In contrast to the above two, we find that parity logging (§2.2) can largely eliminate the parity chunk reads during updates unlike in-place update, and retain in-place update for data chunks to avoid heavy memory overhead unlike full-stripe update.

Therefore, our main idea is to introduce parity logging to in-memory KV stores to reduce the costs of both update and memory. However, the disk-based log devices have pretty lower transfer rate than DRAMs, thereby degrading the update and repair performance, so how to deploy disk-oriented parity logging on in-memory KV stores remains challenging, which will be addressed in §3.

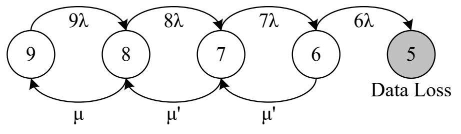


Figure 4: Markov model for (6,3) code.

B (Gb/s)	1	10	40	100
$(6,3)$ code	1.03e+09	9.76e+09	3.89e+10	9.71e+10
$(10,4)$ code	6.41e+08	5.88e+09	2.34e+10	5.83e+10
$(12,4)$ code	5.44e+08	4.91e+09	1.95e+10	4.86e+10
$(15,3)$ code	4.47e+08	3.94e+09	1.56e+10	3.89e+10

Table 2: MTTDLs (in years) for varying B (Gb/s) under different codes and $1/\lambda = 4$ years.

3 HybridPL ARCHITECTURE

Based on observations in §2.3, our main goal is to address the challenge of how to couple in-memory KV stores with parity logging for high repair and update performance. Here, we first analyze the impact of the single-failure repair rate on reliability, and show that improving it can significantly increase the reliability (§3.1). Based on the reliability results, we design the architecture HybridPL, which keeps data and XOR parity chunks of the stripe with in-place update in DRAM nodes for low memory overhead and high single-failure repair performance, while logging the parity deltas of the other parity chunks to disk nodes as well as a leveraging buffer-logging technique for parity deltas in log nodes for high update performance (§3.2).

3.1 Reliability Analysis

We analyze the mean-time-to-data-loss (MTTDL) metric using the Markov model as in prior studies (e.g., [19, 22, 27, 50, 55]). Figure 4 shows the Markov model for $(6,3)$ code used in HDFS. Each state represents the number of available nodes of a stripe. For example, State 9 means that all nodes are healthy, while State 5 means data loss. We also make the models for $(10,4)$ code (f4 [40]), $(12,4)$ code (Azure [27]) and $(15,3)$ code (Pelican [15]) in the same way.

To model failure, we let λ be the failure rate of each node. The state transition rate from State i to State $i-1$, where $6 \le i \le 9$, is $i\lambda$, since any of the i nodes in State i fails independently.

To model repair, we use the similar assumptions as in Azure [27]: for the single failure, let μ be the single-failure repair rate from State 8 to State 9; for multiple failures, let μ' be the repair rate for each node from State i to State $i+1$, where $6 \le i \le 7$. To calculate μ and μ', we let each node have capacity of S and data transfer rate of B. Thus, the average repair rate of single failures is $\mu = B/(SC)$, where C is the single-failure repair cost. For the (6, 3) RS code, it takes 6 chunks to repair any failed chunk, so the single-failure repair cost $C = 6$. In addition, another repair rate when multiple failures is $\mu' = 1/T$, where T denotes the detection and triggering time of multiple failures.

We configure the parameters as follows. For λ, we assume that the mean-time-to-failure (MTTF) of a node is in the range of a few years [51], and set $1/\lambda = 4$ years as in Facebook [50]. We also set $S = 16TB$ and $T = 30$ minutes as in Azure [27],

To show the impact of the single-failure rate (i.e., μ) on reliability, we show the MTTDL results with different data transfer rate of

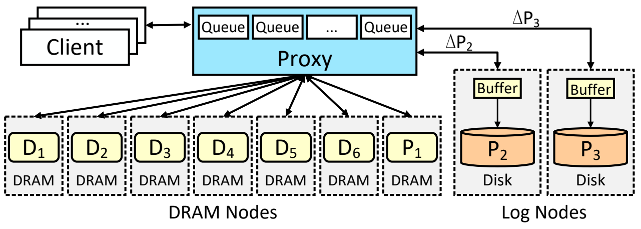


Figure 5: HybridPL **architecture with** $(6,3)$ **code.**

nodes (i.e., B) ranging from 1Gbps to 100Gbps (see Table 2). Here, for disk nodes, the transfer rate can reach nearly 1Gbps [6], so we set $B = 1Gbps$. For DRAM nodes, the transfer rate can achieve up to 17GB per second [4], so we set $B = 100Gbps$, which means the practical DRAM node's transfer rate is limited by 100 Gigabit Ethernet's transfer rate. Similarly, we also set $B = 10Gbps$ (10 Gigabit Ethernet) and $B = 40Gbps$ (40 Gigabit Ethernet).

From Table 2, we observe that for each code, the MTTDL increases significantly with B, which means that the transfer rate of the nodes for repairing the single-node failure has a significant impact on the reliability. For example, for $B = 100Gb/s$, which uses DRAM nodes to perform the single-failure repair, the MTTDL increases by 94.27× under (6,3) code compared to the case for $B = 1Gb/s$, which only uses disk nodes to perform the single-failure repair.

Therefore, the above reliability results inspire us to just keep only one parity chunk of each stripe in DRAM nodes for fast single-failure repairs to obtain decent reliability, which will be leveraged to design our architecture in §3.2.

3.2 Overview

We present the HybridPL architecture, which applies both in-place update and parity logging in a hybrid mode to in-memory KV stores. Figure 5 shows HybridPL with $(6,3)$ code. HybridPL comprises multiple clients, one proxy, multiple DRAM nodes and log nodes. Specifically, 1) the clients interface with user applications; 2) the proxy serves as a front-end interface for the clients to perform various requests on objects (e.g, write, read, degraded read, delete in §4.1 and update in §4.2), as well as multi-failure repair operations in §5; 3) the DRAM nodes store all data chunks (e.g., D_1, D_2, D_3, D_4, D_5 and D_6) and the XOR parity chunks (e.g., P_1); and 4) multiple log nodes store the remaining parity chunks (e.g., P_2 and P_3).

Prior workload studies of in-memory KV stores [14, 41] consider that the size of each object is usually small, which is hard to deploy erasure coding directly. To this end, we organize multiple objects to form a larger fixed-size unit (i.e., data chunks) as [18, 34, 59, 62]. To realize that, the proxy has multiple queues that contain encoding buffers to gather objects that are newly written to form k fixed-size data chunks (4 KiB as the default size [59, 62]) and encodes them into r equal-size parity chunks. In addition, the proxy has to maintain metadata of stripes to manage the clients' requests (e.g., updates), degraded reads and repair operations, mainly including the *Stripe ID* that identifies each stripe, the *Object Index* that maps objects to the Stripe ID with detailed data chunk metadata, and the *Stripe Index* that records all k data chunks and r parity chunks for each stripe.

With a (n,k) code, we can tolerate up to $n-k$ DRAM (or log) node failures, i.e., the objects stored in the HybridPL architecture can still be available even if three DRAM (or log) nodes become offline with $(6,3)$ code in Figure 5. In addition, to avoid the single point of failure of the proxy, we can maintain multiple hot backups of the proxy similar to the prior studies [18, 34], which also could protect the reliability of metadata, including the Stripe Index and Object Index since these data structures are stored in the proxy and replicated in the backup proxies.

3.3 Goals and Approaches

Based on the observations in §2.3 and reliability analysis in §3.1, HybridPL focuses on the following goals:

- **(Goal 1) Low memory overhead:** HybridPL mitigates memory overhead via in-place update for data chunks (§3.3.1).
- **(Goal 2) Efficient updates:** HybridPL performs parity logging for non XOR parity chunks and leverages buffer logging for parity deltas in log nodes to accelerates updates (§3.3.2).
- **(Goal 3) Efficient single-failure repair:** HybridPL keeps XOR parity chunks in DRAM nodes to ensure degraded reads efficiency (§3.3.1).

3.3.1 In-place update for data and XOR parity chunks.

In-place updated data chunks: In-place update does not incur extra memory overhead, while full-stripe update does, as the latter preserves multiple old versions of chunks in memory, especially for update-heavy workloads. Hence, HybridPL can reduce memory overhead via deploying in-place update for data chunks compared to full-stripe update.

DRAM-based XOR parity chunks: Recall that the data transfer rate of the nodes for repairing the single-node failure has a significant impact on the reliability in §3.1, that is, we can only keep XOR parity chunks in DRAMs as well as data chunks to obtain high performance of single-failure repair. Specifically, When one requested data chunk cannot be read directly due to single failure, HybridPL retrieves the remaining $k-1$ data and XOR parity chunks of the same stripe to decode the requested data chunk within the DRAM nodes, which ensures high performance of single-failure repairs.

Thus, HybridPL achieves Goal 1 and 3.

3.3.2 Parity logging for non XOR parity chunks.

Parity logging to eliminate parity chunk transfers: HybridPL updates parity chunks except XOR ones via parity logging, such that HybridPL only needs to store parity deltas into log devices to update parity chunks Clearly, HybridPL neither reads old parity chunks for updating parity chunks, nor retrieves active unchanged data chunks for recomputing parity chunks like full-stripe update.

Buffer logging for parity deltas: However, with parity logging, the low transfer rate of log nodes will become the bottleneck to degrade write (update) performance compared to that of DRAM nodes. To address that, we introduce the *buffer logging* approach proposed in RAMCloud [42], which distributes replicas for both DRAMs and disks. RAMCloud stores the data in DRAMs of the primary server and replicas on the disks of backup servers as shown in Figure 6(a). During write operations, the primary server writes the data in DRAMs and forwards the data replica to all backup servers. RAMCloud considers the write operation completed as soon as the replicas have been written to DRAMs of backup servers, where these disk replicas in DRAMs can be flushed to disks asynchronously.

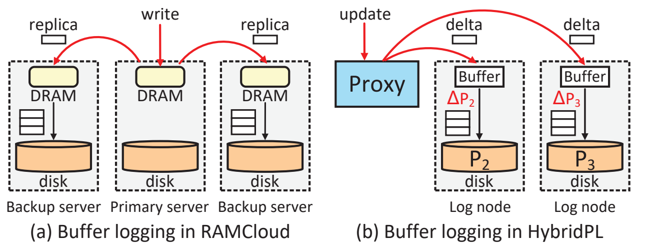


Figure 6: Illustration of the buffer logging in RAMCloud and HybridPL in $(6, 3)$ code as Figure 5.

Note that the asynchronous IOs at the log nodes can accelerate write/update operations, but need to maintain the crash consistency that can reconstruct the data from the disk logs when buffers crash.

We observe that buffer logging can be also applied to HybridPL, as illustrated in Figure 6(b). Specifically, during update operations,the proxy can also send the *same* delta to all log nodes, similar to RAMCloud that forwards the *same* replica to backup servers. Then each log node can compute its parity delta of (e.g., ΔP_2 and ΔP_3) via the delta, based on Property 1 in §2.1. In this way, HybridPL can perform fast writes (updates) operations since they can be completed as soon as the parity deltas have been stored to DRAMs of log nodes, and the parity deltas in DRAMs will be flushed to disks asynchronously in batches (See details in §4.3).

Thus, HybridPL achieves Goal 2.

4 LogECMem DESIGN

We design LogECMem based on the HybridPL architecture, describe its design of basic requests (§4.1), depict the update workflow to show that LogECMem reduces the chunk transfers during updates (§4.2). We also specify in LogECMem how to design buffer logging using merging parity deltas, which can further mitigate the overhead of the disk IOs (§4.3).

4.1 Basic Requests

LogECMem supports four basic requests: write, read, degraded read and delete, where key and value of each object are arbitrary strings.

Write: For a new object to be written into LogECMem, the proxy first selects a DRAM node to store the object via the object's key. To realize erasure coding, the proxy maintains a queue for each DRAM node, where each element of the queue is a fixed-size unit (e.g., 4KiB) that can gather the values of objects via first-come-first-serve for forming data chunks. Note that we can locate an object via its offset and length within a data chunk. When k out of all queues have at least one full unit, these k units can be changed into k data chunks and encoded into r parity chunks. The proxy associates these $k + r$ data and parity chunks of the same stripe with a unique Stripe ID. It then distributes the XOR parity chunk (its key comes from the Stripe ID) into one of the DRAM nodes and the other $r - 1$ parity chunks into different log nodes. Especially, these non XOR parity chunks are not stored in log nodes immediately. Instead, they can be stored in buffers of log nodes temporarily for fast writes (see §4.3).

Here, we organize the metadata of the write operation as follows. First, the proxy maps the written object's key to one element of Object Index, where each element is composed of Stripe ID, sequence number of the data chunk within the stripe, offset and length

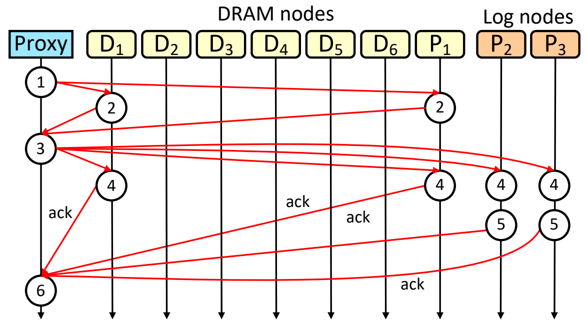


Figure 7: Workflow of the update from D_1 to D_1' in $(6, 3)$ code: (1) the proxy obtains the object's Stripe ID, sequence number, offset and length from the Object Index; (2) D_1's and P_1's nodes return D_1 and P_1 to the proxy; (3) the proxy computes ΔD_1 and P_1', replaces D_1 with D_1', replaces P_1 with P_1', and sends ΔD_1 to P_2's and P_3's nodes; (4) D_1's and P_1's nodes returns acks to the proxy after the writes, P_2's node computes and stores ΔP_2 in buffer, and P_3's node computes and stores ΔP_3 in buffer; (5) P_2's and P_3's nodes returns acks to the proxy; (6) the proxy completes the update after receiving acks from D_1's, P_1's, P_2's and P_3's nodes.

within the data chunk. Second, the proxy records all $k + r$ data/parity chunks in order and all objects' keys of each data chunks via the Stripe Index. In this way, we can decode this written object when it fails to be read directly (see the degraded read operation below).

Read: For obtaining an existing object, the proxy selects the DRAM node and retrieve it from LogECMem via the object's key.

Degraded read: For an object that is failed to read from LogECMem normally, the proxy triggers a decoding process to re-obtain it, so the read operation is performed in a degraded mode. We study that both the transient network congestion and permanent node failure can result in chunk unavailability. Recall that HybridPL architecture realizes efficient single-failure repair (See §3.2) since it stores k data and XOR parity chunks in DRAM nodes.

To degraded read an object that belongs to the unavailable data chunk, LogECMem first obtains the Stripe ID, sequence number of the data chunk, offset and length via looking up the Object Index, reconstructs $k - 1$ available data chunks via gathering all contained objects of this stripe's data chunks via the Stripe Index with the Stripe ID, as well as the XOR parity chunk, decodes the unavailable data chunk from these $k-1$ data chunks (based on sequence number) and one XOR parity chunk, and re-obtains the object by its offset and length from the decoded data chunk.

Delete: For an existing object to be removed from LogECMem, the proxy can update the object's value (See §4.2) to zero-bytes straightforwardly as the delete request, but we need to deploy garbage collection method to reclaim these zero-bytes space.

4.2 Updates

The update operation changes an existing object's old value into a new value, where we also need to update the parity chunks of the stripe that contains this object.

To update an object, the proxy first obtains the object's Stripe ID, sequence number, offset and length from the Object Index. Then, the

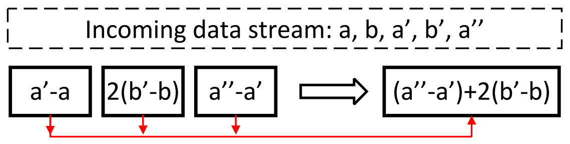


Figure 8: Merge-based buffer logging for a parity chunk $a + 2b$ where a and b are original data chunks. We assume an incoming data stream: a, b, a', b' and a'', where a and b are original data chunks, a' and a'' are updated from a, and b' is updated from b.

proxy retrieves the old value of the object and the stripe's XOR parity chunk (note that its key can be-obtained by the Stripe ID; see §4.1) by read operations. Here, the proxy first computes the delta from the old and new data chunks, uses the delta and a corresponding coefficient (determined by the sequence number) to compute the parity delta of the XOR parity chunk via Property 1 in §2.1, and then computes the new XOR parity chunk via merging the old parity chunk with the parity delta. The proxy also sends the delta to each log node with the corresponding coefficient (determined by the sequence number), and each log node computes its parity deltas in the similar way. Finally, LogECMem writes the new object and the XOR parity chunk to the DRAM nodes, and writes the parity deltas with offsets and lengths to the log nodes. Figure 7 depicts the update workflow of an object that belongs to D_1 in $(6, 3)$ code, where D_1, D_2, D_3, D_4, D_5 and P_1 are stored in DRAM nodes and P_2, P_3 in log nodes. Note that for XOR parity chunks, LogECMem performs in-place update for the entire chunk, but for non XOR parity chunks, records log-file-based parity deltas with the object's offset and length.

4.3 Merge-based Buffer Logging

Based on HybridPL, LogECMem stores data and XOR parity chunks as objects in in-memory KV stores, while storing the other parity chunks and parity deltas as files in log nodes. Here, LogECMem create a log file for each parity chunk and its corresponding parity deltas, where its filename is generated via its Stripe ID.

We find that there are often multiple parity deltas of the same stripe in each log node's buffer. Thus, based on Property 2 in §2.1, LogECMem improves the buffer logging method of HybridPL by merging multiple parity deltas (called *merge-based buffer logging*) to reduce them into one chunk. In this way, LogECMem can reduce disk IOs during updates at log nodes. As illustrated in Figure 8, with this incoming data stream a, b, a', b', a'', the parity chunk $a + 2b$ in HybridPL has to store three parity deltas, which can be reduced to only one parity delta $(a'' - a) + 2(b' - b)$ in LogECMem.

5 MULTIPLE CHUNK FAILURE REPAIR

As stated in §4.1, LogECMem can repair the single-chunk failure by simply performing fast degraded reads in DRAM nodes, but multi-chunk failures sometimes occur in real systems. There are two cases of multiple chunk failures: (1) multiple chunks of the same stripe fail at the same time, like correlated failures [19, 22, 26], and (2) a single node failure contains multiple failed chunks, each of which belongs to different stripes. In this section, we first introduce a state-of-the-art PL method in §5.1 and then propose efficient recovery for the above two cases in §5.2 and §5.3 respectively.

5.1 Parity Logging with Reserved Space

To handle multi-chunk failures, we need to use more parity chunks besides XOR parity ones, so LogECMem need to use log nodes to do the multi-failure repair, but parity logging in HybridPL will incur significant I/O overhead during repair at log nodes. The reason is that during repair, PL may incur a lot of disk seeks to all parity deltas, since PL may make multiple parity deltas of the same parity chunk dispersed due to the append-only policy, so computing the up-to-date parity chunk has to cost multiple random disk IOs.

A prior work [16] studies a state-of-the-art PL scheme – parity logging with reserved space (PLR), which keeps parity deltas next to the old parity chunks to mitigate disk seeks when computing up-to-date parity chunks. Figure 9(a) shows that each parity chunk and its corresponding parity deltas are placed in the contiguous reserved space, e.g., for the parity chunk $c + 2d$, its parity deltas $c' - c$ and $c'' - c'$ are together in the disk, thereby ensuring PLR can compute the up-to-date parity chunk $c'' + 2d$ via one disk seek by $(c + 2d) + (c' - c) + (c'' - c')$ (see §2.2).

However, PLR needs to write parity deltas into different specific reserved spaces of different stripes, thereby incurring heavy random write disk IOs. As illustrated in Figure 9(a), $P_2's$ log node of PLR incurs eight disk writes for writing parity deltas in two different reserved spaces. Thus, PLR trades write (update) performance for the repair performance.

5.2 Repairing Multi-chunk Failures of a stripe

As stated in §5.1, PLR cannot be used in LogECMem, which aims for high performance for both writes (updates) and repairs. Recall the merging method of buffer logging in §4.3, we observe that multiple parity deltas of the same stripes can be merged via linear combination. Thus, we can simply enhance PLR with merging (called *PLR-m*), which merges parity deltas of the same stripe in memory immediately before flushing in the disk. Figure 9(b) shows that PLR-m incurs five disk IOs. It merges parity deltas three times, which generate merged parity deltas $a' + 2b$ and $c + 2d$ after the first merging, $2(b' - b)$ and $c'' - c$ after the second merging, and $(a'' - a) + 2(b'' - b)$ after the third merging.

Obviously, PLR-m only can merge closely incoming parity deltas due to the limited size of memory space. To address that, we use a continuous disk space to do lazy merging (called *parity logging with merging*, or *PLM* for short) instead, which first writes parity deltas into an extra continuous disk space sequentially, and reads them back for merging deltas of the same stripes later, and finally writes merged deltas into specific parity chunks' reserved spaces. In this way, PLM can merge more parity deltas than PLR-m because of the larger size of disk space, so as to reduce more disk IOs during updates than PLR and PLR-m. Figure 9(c) shows that PLM only incurs four disk IOs. It first flushes the buffer containing all parity chunks and deltas into disks sequentially, reads them back via one sequential disk read for merging, and writes two merged parity deltas $a'' + 2b''$ and $c'' + 2d$ into specific reserved spaces.

5.3 Repairing A Single Node Failure

We now study how to realize the node repair. Straightforwardly, we can perform multiple degraded reads to re-obtain all unavailable chunks and migrate them to a new node.

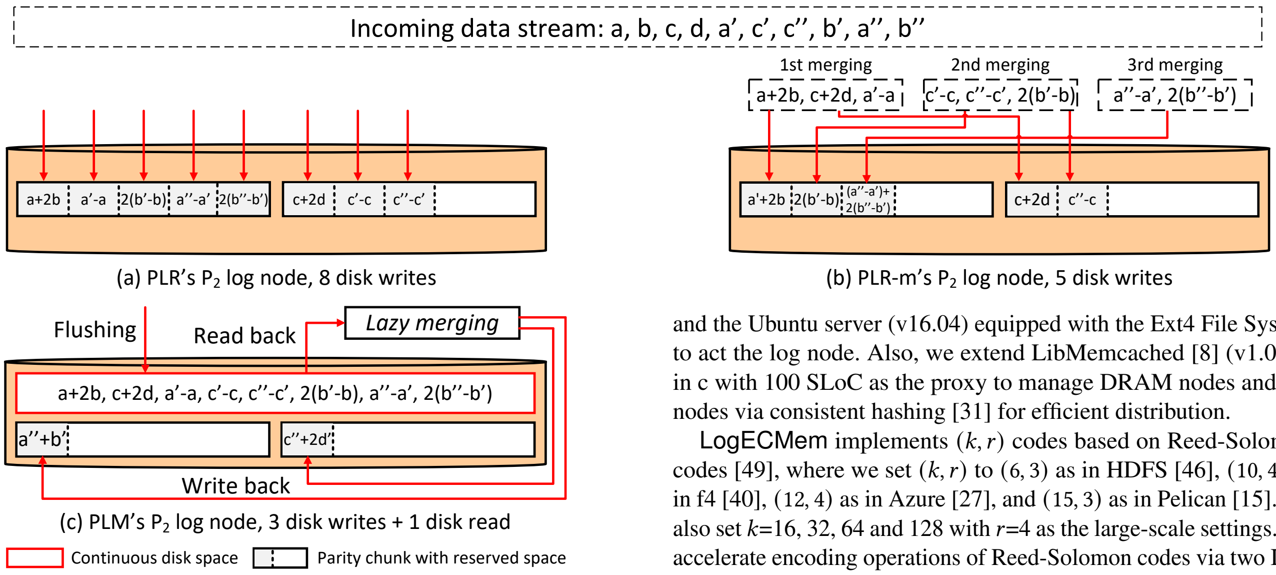

Incoming data stream: a, b, c, d, a′, c′, c′′, b′, a′′, b′′
a+2b
a′-a
2(b′-b)
a′′-a′
2(b′′-b′)
c+2d
c′-c
c′′-c′
(a) PLR's P₂ log node, 8 disk writes
1st merging
a+2b, c+2d, a′-a
2nd merging
c′-c, c′′-c′, 2(b′-b)
3rd merging
a′′-a′, 2(b′′-b′)
a′+2b
2(b′-b)
(a′′-a′)+2(b′′-b′)
c+2d
c′′-c
(b) PLR-m's P₂ log node, 5 disk writes
Flushing
Read back
Lazy merging
a+2b, c+2d, a′-a, c′-c, c′′-c′, 2(b′-b), a′′-a′, 2(b′′-b′)
a′′+b′
c′′+2d
Write back
(c) PLM's P₂ log node, 3 disk writes + 1 disk read
Continuous disk space
Parity chunk with reserved space

Figure 9: PLR, PLR-m and PLM in $(2, 2)$ code. We assume a and b, c and d form two stripes respectively, where the two stripes' second parity chunks $a + 2b$ and $c + 2d$ are stored in one log node. Next, we update a to a', c to c', c' to c'', b to b', a' to a'', and b' to b''.

DRAM nodes repair: The straightforward way for repairing the failed DRAM node will cause heavy bandwidth cost, as it traverses the Stripe Index to re-construct k available chunks for all stripes, so as to degraded read each stripe's chunk that belongs to the failed node. Besides, the existing DRAM nodes need to provide continuous service via the proxy for requests, which may make DRAM nodes' network congested. Actually, the bandwidth of disk nodes is only served for writes and updates of parity chunks, which may not be fully utilized and thus can be used to help repair single-node failure.

To this end, we propose a *log-assist* node repair to obtain one non XOR parity chunk from the log node for each stripe to participate in repairing the lost chunks of the failed node in parallel with another $k - 1$ chunks retrieved from DRAM nodes. Note that similar to the repair in §5.2, log nodes can read old parity chunks with all parity deltas via one disk IO and re-compute the up-to-date parity chunks.

Log nodes repair: When incurring the log node failure, we can also deploy log-assist node repair to accelerate node repair similar to DRAM nodes repair, where we need to re-obtain the non XOR parity chunk from one of the available log nodes instead.

6 EVALUATION

We describe LogECMem's implementation (§6.1), provide testbed and workloads configurations (§6.2), and show evaluation results compared to in-place and full-stripe updates in terms of basic I/O, updates, memory overhead and repair performance (§6.3).

6.1 Implementation

We build LogECMem atop the Memcached protocol [9] that is widely used in academia and industry [3, 18, 41, 62], which is implemented with about 1000 SLoC in C++ on Linux. In particular, we use the memcached instance [9] (v1.4) to act as the DRAM node and the Ubuntu server (v16.04) equipped with the Ext4 File System to act the log node. Also, we extend LibMemcached [8] (v1.0.18) in c with 100 SLoC as the proxy to manage DRAM nodes and log nodes via consistent hashing [31] for efficient distribution.

LogECMem implements (k, r) codes based on Reed-Solomon codes [49], where we set (k, r) to $(6, 3)$ as in HDFS [46], $(10, 4)$ as in f4 [40], $(12, 4)$ as in Azure [27], and $(15, 3)$ as in Pelican [15]. We also set k=16, 32, 64 and 128 with r=4 as the large-scale settings. We accelerate encoding operations of Reed-Solomon codes via two Intel ISA-L APIs [7]: *ec_init_tables* and *ec_encode_data*, which splits each data chunks into slices and accelerates slice-based encoding by pre-fetching slices ahead via the CPU cache.

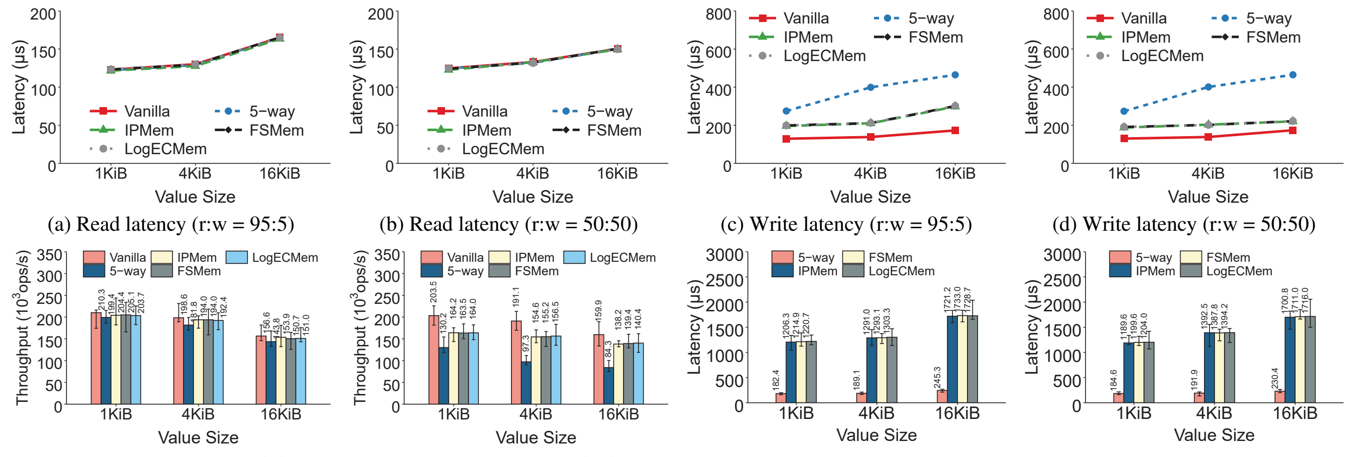
In addition, we use *dd* command in the log nodes to create a large empty file for each newly written parity chunk to realize the reserved space in PLR, PLR-m and PLM, where the following parity deltas of the parity chunk can be stored sequentially.

To show LogECMem's performance improvements, we extend Memcached and implement the erasure-coded update schemes in-place and full-stripe updates. We call the Memcached implementation with in-place update scheme IPMem, and that with full-stripe update scheme FSMem. Also, we deploy a traditional replication way (called $(r + 1)$-way replication) that uses r replicas to ensure data reliability for at most r failures, and vanilla Memcached without erasure coding called Vanilla which has no reliability assurance.

6.2 Experimental Setup

Testbed and configurations: We conduct our experiments on Amazon EC2 [2] in US-West (North California) with k+r+2 m5d.2xlarge instances, which evaluates Vanilla, $(r + 1)$-way replication, IPMem, FSMem and LogECMem in the cloud environment. $k + r$ instances represent the storage nodes including $k + 1$ DRAM nodes and $r - 1$ log nodes, and two instances represent the proxy and the client respectively. Each DRAM node runs a memcached instance, and each log node with Ubuntu-16.04 has a DRAM-based buffer for fast storage and an additional Amazon EBS [1] volume to serve as a 1 TiB disk. We report average results of experiments about the latency and throughput over ten runs, and provide the variance of these results due to fluctuating cloud network environment.

Workloads: We use YCSB [20] to generate workloads to evaluate Vanilla, 5-way (or 4-way) replication, IPMem, FSMem, and LogECMem. We set the size of the key for each object around 20 bytes by using the default setting in YCSB. We also set the size of the value of an object as 1 KiB, 4 KiB and 16 KiB [34, 62], and we consider each object as a data chunk for simplicity. we load LogECMem with one million objects using write requests, so the total data sizes are 1 GiB, 4 GiB, and 16 GiB, respectively. We evaluate these systems



(a) Read latency (r:w = 95:5) (b) Read latency (r:w = 50:50) (c) Write latency (r:w = 95:5) (d) Write latency (r:w = 50:50)

(e) Throughput (r:w = 95:5) (f) Throughput (r:w = 50:50) (g) Degraded read latency (r:w = 95:5) (h) Degraded read latency (r:w = 50:50)

Figure 10: Experiment 1: Comparison of read, write, degraded read latency, and throughput with different workloads in $(10, 4)$ **code.**

with one million requests via YCSB with different read/write (i.e., r:w) ratios including 95%:5% (read-mostly) and 50%:50%, and different read/update (i.e., r:u) ratios including 95%:5% (update-light), 80%:20%, 70%:30% and 50%:50% (update-heavy) [18, 58, 59, 62]. We also set these requests to follow the default Zipf distribution in YCSB to mimic the popularity distribution.

6.3 Performance

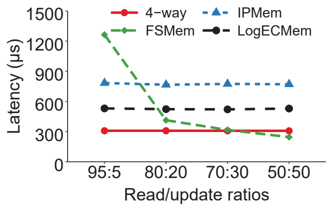
Experiment 1 (Basic requests performance): We compare Vanilla, 5-way replication, IPMem, FSMem, and LogECMem with PLM in terms of read, write, and degraded read latency, and throughput (in the number of operations per second). We consider different value sizes and read/write ratios. Figure 10(a) and (b) show that LogECMem, IPMem and FSMem have similar read latency as Vanilla and 5-way replication. Figure 10(c) and (d) show that LogECMem, IPMem and FSMem have similar write latency for read-mostly and write-heavy workloads, but all of them incur higher write latency compared to Vanilla due to a long I/O path for the additional encoding operation. Also, 5-way replication needs to write multiple copies to the storage system and incurs significant write latency, which is higher than that of LogECMem, IPMem, FSMem and Vanilla. Figure 10(e) and (f) show that the throughputs of LogECMem, IPMem, FSMem and 5-way replication have a small degradation compared to Vanilla, since Vanilla does not perform additional encoding for parity chunks (as LogECMem, IPMem and FSMem) nor store multiple replicas (as 5-way replication). Here, although Vanilla has similar and even better read, write and throughput performance compared to LogECMem, IPMem and FSMem, it does not provide data availability. Thus, we do not consider Vanilla in the following experiments.

Figure 10(g) and (h) LogECMem keeps the similar degraded read latency to IPMem and FSMem, since all of them re-obtain the object belongs to the unavailable chunk in the same way via retrieving and decoding the available chunks stored in DRAM nodes of the same stripe. It addresses the repair performance degradation of disk-oriented parity logging, which is consistent with our theoretical findings in §3.1. Note that 5-way replication has a low degraded read latency compared to others since its degraded read operation is just to read another replica triggered by the failed read operation.

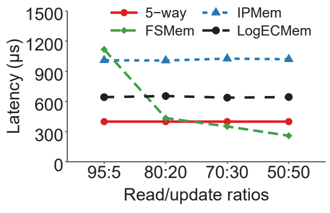
Experiment 2 (Update latency): We compare 5-way (or 4-way) replication, IPMem, FSMem and LogECMem with PLM in terms of update latency. We consider different (k, r) codes and read/update ratios. Figure 11 shows that LogECMem outperforms IPMem because the former mitigates the number of parity reads from r to one during updates, since it only needs to read the XOR parity chunk back for in-place update. Further, IPMem in $r = 4$ (e.g., $(10, 4)$ code) incurs higher update latency than that in $r = 3$ (e.g., $(6, 3)$ code), since the former needs to update one more parity chunk than the latter; similarly, LogECMem in $r = 4$ also outperforms that in $r = 3$. For example, compared to IPMem in Figure 11(a) in $r = 3$ and (b) in $r = 4$, we see LogECMem reduces up to 32.7% and 37.8%, when r:u = 70%:30%, respectively. Figure 11(c) and (d) have similar results.

Also, we see that LogECMem outperforms FSMem in update-light scenarios (i.e., r:u = 95%:5%) by 58.0%, 42.4%, 37.8% and 37.3% in $(6, 3)$, $(10, 4)$, $(12, 4)$, and $(15, 3)$ codes respectively. The reason is that in the update-light scenarios, most of the updated stripes in FSMem may have only one new data chunk, thus leading to heavy re-computing overhead (See Observation 1 in § 2.3). When the update ratio becomes higher (i.e., r:u = 80%:20%) we see that compared to FSMem, LogECMem performs worse in Figure 11(a) and (b) (when $k = 6$ and 10), but better in Figure 11(c) and (d) (when $k = 12$ and 15). The reason is that for a larger k, the average number of new chunks per stripe will be fewer, which means that FSMem has to retrieve more active unchanged data chunks of those stripes to re-compute the new parity chunks. That can be more critical for even larger k in the wide stripe [11, 25]. When the update ratio continues to increase (i.e., r:u = 50%:50%), FSMem outperforms LogECMem in update-heavy scenarios, but incurs heavy memory overhead. Besides, although replication has a low update latency compared to the others in different (k, r) codes and read/update ratios, it incurs significant memory overhead (See Experiment 3 and 4), which is unaffordable.

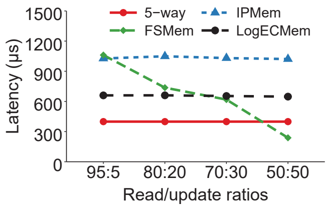
Experiment 3 (Memory consumption): We compare 5-way (or 4-way) replication, IPMem, FSMem and LogECMem with PLM



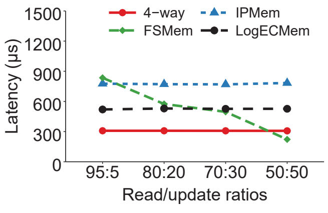
(a) $(6, 3)$ code



(b) $(10, 4)$ code

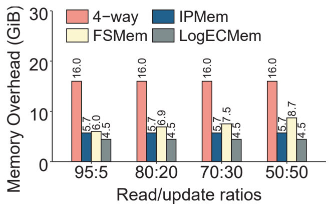


(c) $(12, 4)$ code

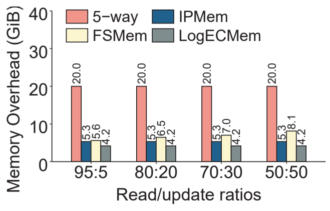


(d) $(15, 3)$ code

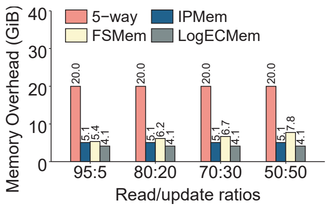
Figure 11: Experiment 2: Comparisons of update latency in different (k, r) codes and read/update ratios with 4 KiB value size.



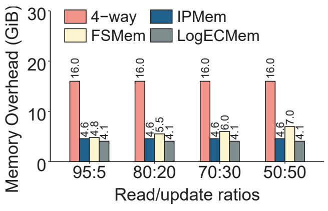
(a) $(6, 3)$ code



(b) $(10, 4)$ code

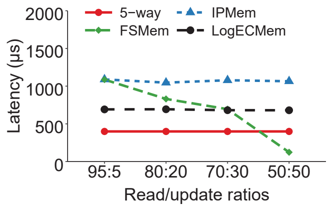


(c) $(12, 4)$ code

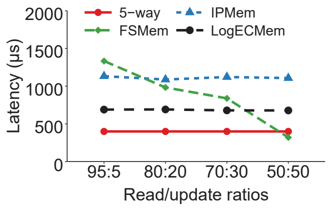


(d) $(15, 3)$ code

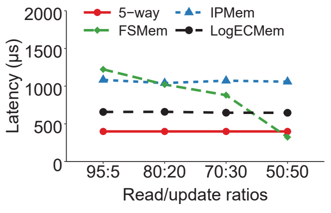
Figure 12: Experiment 3: Comparisons of memory overhead in different (k, r) codes and read/update ratios with 4 KiB value size.



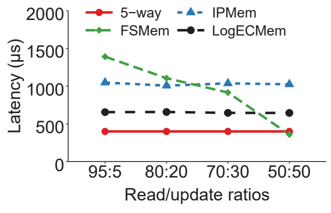
(a) $(16, 4)$ code



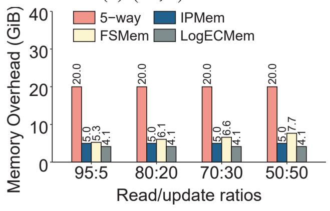
(b) $(32, 4)$ code



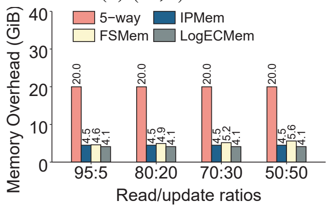
(c) $(64, 4)$ code



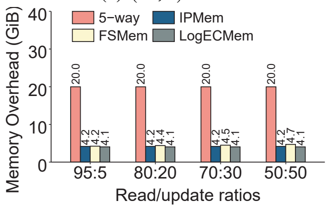
(d) $(128, 4)$ code



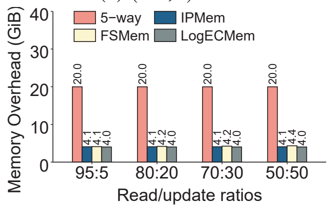
(e) $(16, 4)$ code



(f) $(32, 4)$ code



(g) $(64, 4)$ code



(h) $(128, 4)$ code

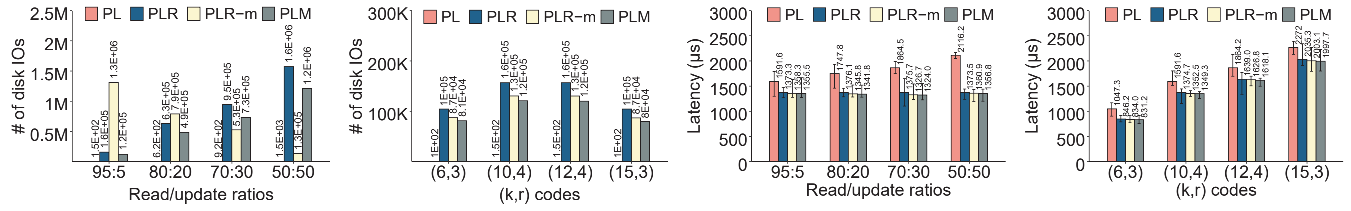
Figure 13: Experiment 4: Repeating Experiment 2 and 3 with a large k, where $k = 16, 32, 64$ and 128 with $r = 4$.

in terms of memory overhead including data and parity chunks. We consider different (k, r) codes and read/update ratios. Figure 12 shows that LogECMem reduces significant memory overhead up to 79.3% compared to 5-way replication (when in $(12, 4)$ code), because the former stores 4 parity chunks for 12 data chunks while the latter stores 4 replicas for each object; up to 22.2% compared to IPMem (when in $(6, 3)$ code), because it only maintains XOR parity chunks in DRAM nodes, and stores remaining parity chunks and parity deltas in disks instead. Further, LogECMem reduces memory overhead up to 49.0% compared to FSMem (when in $(6, 3)$ code), since FSMem additionally keeps multiple versions of old data and parity chunks in DRAMs. We also see that LogECMem outperforms IPMem and FSMem when the update ratio increases, which is consistent with the Observation 2 in § 2.3.

Experiment 4 (Performance in a large-scale setting): We repeat the experiment 2 and 3 in a large-scale setting with a large k. Figure 13(a)~(d) confirm that LogECMem outperforms IPMem similar to Experiment 2. Further, we find that with a large k, LogECMem outperforms FSMem in r:u=95%:5%, 80%:20% and r:u=70%:30%, where in 70%:30% the former reduces update latency by 1.9%, 19.0%, 36.4% and 29.4% in $(16, 4)$, $(32, 4)$, $(64, 4)$ and $(128, 4)$ codes, respectively. Figure 13(e)~(f) also confirm that LogECMem has the lowest memory overhead similar to Experiment 3, which reduces up to 79.8% (when in $(128, 4)$ code), 17.2% (when in $(16, 4)$ code), and 46.0% (when in $(16, 4)$ code) compared to 5-way replication, IPMem and FSMem, respectively.

Figure 12 and Figure 13(e)~(f) show that the memory overhead of 5-way (or 4-way) replication is much more than that of the others. Here, although $(r + 1)$-way replication has decent read, degraded read and update performance compared to IPMem, FSMem and LogECMem, it does incur significant memory overhead. Thus, we do not consider $(r + 1)$-way replication in the following experiments.

Experiment 5 (Disk IOs during updates): We compare PL, PLR, PLR-m and PLM schemes of LogECMem in terms of the number of disk IOs during updates caused by buffer logging. We consider different (k, r) codes and read/update ratios. Figure 14(a) shows that all of PLR, PLR-m and PLM incur more disk IOs than PL when flushing buffers in all four read/update ratios. The reason is that PL flushes the full buffer containing all parity chunks and deltas via one disk IO, but PLR, PLR-m and PLM need to write parity chunks



(a) $(10, 4)$ code (b) Impact of codes in r:u =95:5 (c) Read/update ratios in $(10, 4)$ code (d) (k, r) code in r:u = 95:5

Figure 14: Experiment 5 and Experiment 6: Comparisons of disk IOs during updates and multiple chunk failures repair performance in different (k, r) codes and read/update ratios with 4 KiB value size.

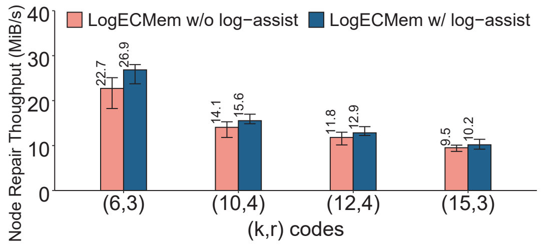


Figure 15: Experiment 7: Comparisons of node repair performance in different (k, r) codes with 4 KiB value size.

and deltas into specific reserved spaces one by one, thus resulting in heavy disk IOs. Figure 14(b) shows that PLM reduces disk IOs compared to PLR up to 23.7% (when in $(15, 3)$ code) in r:u = 95%:5%, because PLM can merge multiple parity deltas of the same stripe via linear combination. Also, PLM reduces disk IOs compared to PLR-m up to 8.2% (when in $(15, 3)$ code) in r:u = 95%:5%, because PLM can merge multiple parity deltas via continuous disk space rather than PLR-m that has limited memory space. Note that the number of disk IOs increases with r and update ratios since both of them generate more parity deltas.

Experiment 6 (Multiple chunk failures repair performance): We compare PL, PLR, PLR-m and PLM schemes in LogECMem in terms of degraded read latency when incurring multiple chunk failures, which is equal to the average latency of all degraded reads. We consider different (k, r) codes and read/update ratios. We mimic two chunks failures by manually killing memcached processes in their DRAM nodes. Figure 14(c) and (d) show that all of PLR, PLR-m and PLM have the similar degraded read performance but outperform PL. The reason is that PL's degraded reads have to incur many random disk reads to retrieve all old parity chunks and deltas to compute the up-to-date parity chunks, while PLR, PLR-m and PLM leverage the reserved space to reduce the number of the disk reads. Note that PLM performs a little better than PLR and PLR-m, since PLM has fewer deltas stored in the reserved space after merging. For example, PLM incurs lower degraded read latency up to 35.9% (when r:u = 50%:50%) compared to PL. In addition, we see that the improvement of PLM decreases with k from 20.3% (when k = 6) to 11.8% (when k = 15), because when k becomes large, retrieving chunks from DRAM nodes dominates the degraded read operation rather than parity chunks in the disk.

Experiment 7 (Node repair performance): We compare within and without log-assist method in LogECMem in terms of node repair performance , which is equal to the storage capacity of a node over the repair time. Here, we consider the failed node is DRAM-based, which stores data and XOR parity chunks. Based on the

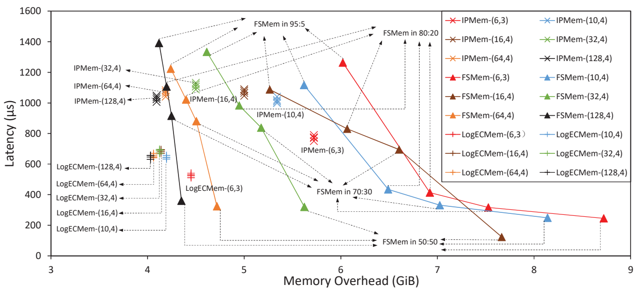


Figure 16: Tradeoff analysis of memory overhead and update latency in different (k, r) codes and read/update ratios with 4 KiB value size.

workload, we set the total data size to 4 GiB (i.e., one million 4 KiB value size objects), so the total memory overhead of the repaired node equals $\frac{4}{k}$ GiB. Figure 15 shows that LogECMem within log-assist method outperforms that without one by up to 18.2% (when in $(6, 3)$ code) since the proxy can pre-repair the non XOR parity chunks via the free bandwidth of log nodes. Note that the node repair performance decreases with k, which is similar to Experiment 6.

6.4 Tradeoff Analysis

Figure 16 plots the memory overhead and update latency of IPMem, FSMem and LogECMem for different (k, r) codes (i.e., $(6, 3)$, $(10, 4)$, $(16, 4)$, $(32, 4)$, $(64, 4)$ and $(128, 4)$ codes) and read/update ratios, where each point represents a trade-off between memory overhead and update latency. Note that LogECMem has very close update latencies for different update ratios under the same (k, r) code as well as IPMem, while FSMem has different update latencies for different update ratios under the same (k, r) code. Overall, LogECMem's curves are closer to the origin than IPMem's and FSMem's, indicating that LogECMem has a better trade-off between update latency and memory overhead than IPMem and FSMem, which can be more significant with a larger k.

Specifically, Table 3 gives a thorough comparison of IPMem, FSMem, and LogECMem in terms of update latency as well as memory overhead under different k and read/update ratios. Here, we use "best", "low" and "high" outside the brackets to indicate the comparison results of update latency between IPMem, FSMem and LogECMem, and the results inside the brackets for memory overhead. First, we see that when k is not large (k = 6, 10, 12 and 15), FSMem often has the best update performance, due to its direct parity updates without any reads, while LogECMem requires parity reads which limit its update performance (see §2.2). However, when k becomes large ($k \geq 16$), LogECMem has more cases with

	r:u ratios	IPMem	FSMem	LogECMem
k = 6 and 10	95:5	low (low)	high (high)	***best*** **(*best*)**
	80:20	high (low)	***best*** (high)	low (***best***)
	70:30	high (low)	***best*** (high)	low (***best***)
	50:50	high (low)	***best*** (high)	low (***best***)
k = 12 and 15	95:5	low (low)	high (high)	***best*** **(*best*)**
	80:20	high (low)	low (high)	***best*** **(*best*)**
	70:30	high (low)	***best*** (high)	low (***best***)
	50:50	high (low)	***best*** (high)	low (***best***)
k = 16, 32, 64, and 128	95:5	low (low)	high (high)	***best*** **(*best*)**
	80:20	high (low)	low (high)	***best*** **(*best*)**
	70:30	high (low)	low (high)	***best*** **(*best*)**
	50:50	high (low)	***best*** (high)	low (***best***)

Table 3: Comparison results of IPMem, FSMem and LogECMem in terms of "update latency (memory overhead)".

the best update performance, since it still maintains stable update performance (see Figure 16) due to its delta-based update scheme regardless of k, while FSMem's update performance degrades significantly due to its many parity re-computations during GC under a large k (see §2.2.1). In addition, LogECMem always has the lowest memory overhead due to its HybridPL architecture (see §3.3).

7 ORIGINALITIES AND LIMITATIONS

Originalities: Although full-stripe update is state-of-the-art in in-memory KV stores for fast erasure-coded updates [34], we are the first to discover that its update performance will be degraded seriously in a large-scale setting (i.e., a large k), while delta-based update schemes suit well with the large-scale setting (§2.2.1 and Experiment 4), since the delta is generated regardless of k (§2.1).

In addition, parity logging is state-of-the-art in delta-based update schemes [16], but we are the first to successfully deploy parity logging under in-memory KV stores, via addressing the major challenge that comes from the performance gap between memory and log nodes, with the help of our new architecture HybridPL (§3).

Further, we propose three new methods that improve existing works for better performance: (1) merge-based buffer logging (§4.3) that enables parity deltas to be merged in the classical buffer logging approach [42], (2) parity logging with merging, i.e., PLM (§5.2) for better update and repair performance than the state-of-the-art scheme PLR [16], and (3) log-assist node repair that leverages both memory and disk nodes (§5.3) for better node repair performance than those that only use memory nodes [62].

Limitations: LogECMem mixes in-place update and parity logging, both of which use delta for parity updates, thereby leading to limitations in some cases. (1) *Update-heavy cases*: In update-heavy scenarios (e.g., 50%:50%), LogECMem incurs too many read IOs for old data chunks to compute delta and thus performs worse than FSMem in terms of update latency. (2) *Slow-read cases*: For LSM-tree based KV stores which have the read amplification problem [47], LogECMem's delta-based update performance may be degraded due to the slow reads of old data chunks to compute delta.

8 RELATED WORK

Erasure coding in distributed storage: Erasure coding has been widely adopted in distributed file systems [15, 16, 25–27, 33, 36, 39, 46, 50, 63] and KV stores [17, 18, 34, 38, 48, 54, 57, 59, 60, 62]. Most of studies mainly focus on repair performance [26, 27, 33, 39, 50, 54, 55], improving storage efficiency [40, 46, 62], memory management [26, 38, 48, 59, 62], scaling performance [18, 28, 57, 63], and update performance [16, 34, 53, 59].

To improve update performance in erasure-coded storage, prior studies deploy the in-place update to minimize extra storage overhead [12, 62] and form new stripes instead of updates to reduce network bandwidth [17, 34, 43]. Specifically, BCStore [34] organizes the newly updated chunks into new stripes, marks old data chunks as invalid and reclaims them via GC. Also, Parity logging [56] is a well-known approach of mitigating parity update traffic by eliminating reads of parity chunks and recording parity deltas to log devices. CodFS [16] performs the parity logging by placing parity deltas next to original parity chunks to reduce disk seeks during recovery. In contrast, LogECMem realizes a novel HybridPL architecture, which takes a hybrid of in-place update for data and XOR parity chunks and parity logging for the remaining parity chunks.

In-memory KV stores: Many studies in recent years worked on in-memory KV stores [9, 10, 13, 18, 21, 37, 41, 42, 48, 57, 59, 62], which focus on different aspects: memory efficiency [48, 57, 59, 62], availability [18, 34, 62], scaling [13, 18, 41], and load balancing [23, 24, 48].

Some closely related studies to ours include: Cocytus [62], which uses a hybrid scheme to use primary-backup replication for small-sized and scattered data and only apply erasure coding to relatively large data; ECHash [18] combines the consistent hashing with erasure coding, decouples the relation between data chunks and nodes to significantly reduced parity updates during scaling nodes; Fatcache [5] leverages SSDs to expend its storage capacity based on Memcached [9] for a higher basic request performance. In contrast, LogECMem stores data and parity in both DRAM and log nodes, which stores data and XOR parity chunks in DRAM node for high repair performance, and stores the other parity chunks and deltas in disks of log nodes for update efficiency.

9 CONCLUSION AND FUTURE WORK

We propose HybridPL, a novel parity logging based architecture that takes a hybrid of in-place updates for data and XOR parity chunks in DRAMs and log-based updates for the remaining parity chunks in logs, such that the memory cost and performance of updates and single-failure repairs can be balanced well. We prototype HybridPL as LogECMem atop Memcached, and design two schemes (PLM and log-assist) to improve multi-failure repair performance. Cloud-based experiments demonstrate the efficiency of LogECMem in basic I/O, updates and repair with low memory overhead.

We first plan to investigate how LogECMem can be applied to NVRAM-based and SSD-based KV stores and parallel file systems. We also plan to re-organize HybridPL's architecture to proactively identify the popularity of incoming data for better update efficiency.

Acknowledgments: We thank our shepherd and the anonymous reviewers for their comments. This work was supported by the National Natural Science Foundation of China (61872414), Key Laboratory of Information Storage System Ministry of Education of China. The corresponding author is Yuchong Hu.

REFERENCES

[1] Amazon Elastic Block Store. http://aws.amazon.com/ebs.

[2] Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2.

[3] Amazon Elasticache. https://docs.aws.amazon.com/elasticache.

[4] Ddr4 sdram. https://en.wikipedia.org/wiki/DDR4_SDRAM.

[5] Fatcache. https://github.com/twitter/fatcache.

[6] Hard disk drive performance characteristics. https://en.wikipedia.org/wiki/Hard_disk_drive_performance_characteristics.

[7] Intel ISA-L. https://github.com/intel/isa-l.

[8] LibMemcached. https://libmemcached.org.

[9] Memcached. https://memcached.org.

[10] Twittercache. https://github.com/alexpghayes/twittercache.

[11] Vastdata. https://vastdata.com/providing-resilience-efficiently-part-ii/.

[12] M. K. Aguilera, R. Janakiraman, and L. Xu. Using erasure codes efficiently for storage in a distributed system. In *Proc. of IEEE/IFIP DSN*, pages 336–345. IEEE, 2005.

[13] A. Anwar, Y. Cheng, H. Huang, J. Han, H. Sim, D. Lee, F. Douglis, and A. R. Butt. Bespokv: Application tailored scale-out key-value stores. In *Proc. of IEEE SC*, pages 14–29. IEEE, 2018.

[14] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proc. of ACM SIGMETRICS*, pages 53–64, 2012.

[15] S. Balakrishnan, R. Black, A. Donnelly, P. England, A. Glass, D. Harper, S. Legtchenko, A. Ogus, E. Peterson, and A. Rowstron. Pelican: A building block for exascale cold data storage. In *Proc. of USENIX OSDI*, 2014.

[16] J. C. Chan, Q. Ding, P. P. Lee, and H. H. Chan. Parity logging with reserved space: Towards efficient updates and recovery in erasure-coded clustered storage. In *Proc. of USENIX FAST*, pages 163–176, 2014.

[17] Y. L. Chen, S. Mu, J. Li, C. Huang, J. Li, A. Ogus, and D. Phillips. Giza: Erasure coding objects across global data centers. In *Proc. of USENIX ATC*, pages 539–551, 2017.

[18] L. Cheng, Y. Hu, and P. P. Lee. Coupling decentralized key-value stores with erasure coding. In *Proc. of ACM SoCC*, pages 377–389, 2019.

[19] A. Cidon, R. Escriva, S. Katti, M. Rosenblum, and E. G. Sirer. Tiered replication: A cost-effective alternative to full cluster geo-replication. In *Proc. of USENIX ATC*, pages 31–43, 2015.

[20] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. of ACM SoCC*, pages 143–154, 2010.

[21] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. of ACM SOSP*, pages 205–220, 2007.

[22] 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*, Oct 2010.

[23] Y.-J. Hong and M. Thottethodi. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *Proc. of ACM SoCC*, page 13, 2013.

[24] X. Hu, X. Wang, Y. Li, L. Zhou, Y. Luo, C. Ding, S. Jiang, and Z. Wang. LAMA: Optimized locality-aware memory allocation for key-value cache. In *Proc. of USENIX ATC*, pages 57–69, 2015.

[25] Y. Hu, L. Cheng, Q. Yao, P. P. C. Lee, W. Wang, and W. Chen. Exploiting combined locality for wide-stripe erasure coding in distributed storage. In *Proc. of USENIX FAST*, Feb. 2021.

[26] Y. Hu, X. Li, M. Zhang, P. P. Lee, X. Zhang, P. Zhou, and D. Feng. Optimal repair layering for erasure-coded data centers: From theory to practice. *ACM Trans. on Storage*, 13(4):33, 2017.

[27] 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*, Jun 2012.

[28] J. Huang, X. Liang, X. Qin, P. Xie, and C. Xie. Scale-RS: An efficient scaling scheme for RS-coded storage clusters. *IEEE Trans. on Parallel and Distributed Systems*, 26(6):1704–1717, 2015.

[29] Z. Jia, J. Zhan, L. Wang, C. Luo, W. Gao, Y. Jin, R. Han, and L. Zhang. Understanding big data analytics workloads on modern processors. *IEEE Trans. on Parallel and Distributed Systems*, 28(6):1797–1810, 2017.

[30] C. Jin, D. Feng, H. Jiang, and L. Tian. Raid6l: A log-assisted raid6 storage architecture with improved write performance. In *Proc. of IEEE MSST*, pages 1–6. IEEE, 2011.

[31] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. of ACM STOC*, pages 654–663, 1997.

[32] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[33] R. Li, X. Li, P. P. Lee, and Q. Huang. Repair pipelining for erasure-coded storage. In *Proc. of USENIX ATC*, pages 567–579, 2017.

[34] S. Li, Q. Zhang, Z. Yang, and Y. Dai. BCStore: Bandwidth-efficient in-memory KV-store with batch coding. In *Proc. of IEEE MSST*, 2017.

[35] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proc. of ACM EuroSys*, page 27, 2014.

[36] X. Li, R. Li, P. P. Lee, and Y. Hu. Openec: Toward unified and configurable erasure coding management in distributed storage systems. In *Proc. of USENIX FAST*, pages 331–344, 2019.

[37] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *Proc. of USENIX NSDI*, pages 429–444, 2014.

[38] W. Litwin, R. Moussa, and T. Schwarz. LH* RS: A highly-available scalable distributed data structure. *ACM Trans. on Database Systems*, 30(3):769–811, 2005.

[39] S. Mitra, R. Panta, M.-R. Ra, and S. Bagchi. Partial-parallel-repair (PPR): a distributed technique for repairing erasure coded storage. In *Proc. of ACM Eurosys*, page 30. ACM, 2016.

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

[41] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling Memcache at Facebook. In *Proc. of USENIX NSDI*, pages 385–398, 2013.

[42] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, et al. The case for ramclouds: scalable high-performance storage entirely in dram. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[43] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The Quantcast File System. *Proc. of VLDB Endowment*, 6(11):1092–1101, 2013.

[44] J. S. Plank. Erasure codes for storage systems: A brief primer. *The magazine of USENIX & SAGE*, 38(6):44–50, 2013.

[45] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *Proc. of USENIX OSDI*, volume 10, pages 293–306, 2010.

[46] R. Li, Z. Zhang, K. Zheng, and A. Wang. Progress report: Bringing erasure coding to apache hadoop. https://blog.cloudera.com/blog/2016/02/progress-report-bringing-erasure-coding-to-apache-hadoop, 2016.

[47] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proc. of ACM SOSP*, pages 497–514, 2017.

[48] K. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran. EC-Cache: Load-balanced, low-latency cluster caching with online erasure coding. In *Proc. of USENIX OSDI*, pages 401–417, 2016.

[49] I. Reed and G. Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 1960.

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

[51] B. Schroeder and G. A. Gibson. Disk Failures in the Real World: What does an MTTF of 1,000,000 Hours Mean to You? In *Proc. of USENIX FAST*, page 1, 2007.

[52] D. Shankar. Designing high-performance, resilient and heterogeneity-aware key-value storage for modern hpc clusters. *IEEE SC Doctoral Showcase*, 2018.

[53] Z. Shen and P. P. Lee. Cross-rack-aware updates in erasure-coded data centers. In *Proc. of ICPP*, pages 1–10, 2018.

[54] H. Shi and X. Lu. Inec: fast and coherent in-network erasure coding. In *Proc. of IEEE SC*, pages 924–940. IEEE Computer Society, 2020.

[55] M. Silberstein, L. Ganesh, Y. Wang, L. Alvisi, and M. Dahlin. Lazy means smart: Reducing repair bandwidth costs in erasure-coded distributed storage. In *Proc. of ACM SYSTOR*, pages 1–7. ACM, 2014.

[56] D. Stodolsky, G. Gibson, and M. Holland. Parity logging overcoming the small write problem in redundant disk arrays. *ACM SIGARCH Computer Architecture News*, 21(2):64–75, 1993.

[57] K. Taranov, G. Alonso, and T. Hoefler. Fast and strongly-consistent per-item resilience in key-value stores. In *Proc. of ACM EuroSys*, page 39, 2018.

[58] J. Yang, Y. Yue, and K. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *Proc. of USENIX OSDI*, pages 191–208, 2020.

[59] M. M. Yiu, H. H. Chan, and P. P. Lee. Erasure coding for small objects in in-memory KV storage. In *Proc. of ACM SYSTOR*, page 14, 2017.

[60] Y. Yu, R. Huang, W. Wang, J. Zhang, and K. B. Letaief. Sp-cache: Load-balanced, redundancy-free cluster caching with selective partition. In *Proc. of IEEE SC*, pages 1–13. IEEE, 2018.

[61] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. of USENIX NSDI*, pages 15–28, 2012.

[62] 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, 2016.

[63] X. Zhang, Y. Hu, P. P. Lee, and P. Zhou. Toward optimal storage scaling via network coding: From theory to practice. In *Proc. of IEEE INFOCOM*, pages 1808–1816, 2018.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

This appendix describes the information of the prototype and experimental details of the SC'21 paper "LogECMem: Coupling Erasure-Coded In-Memory Key-Value Stores with Parity Logging". More precisely, we explain how to prepare, install, configure and benchmark LogECMem to re-run the experiments in this paper.

ARTIFACT DESCRIPTION

Our LogECMem is tested on Ubuntu 16.04 and now available at *https://github.com/yuchonghu/logecmem.*

Preparation

These are the required libraries that users need to download separately, e.g., gcc (v5.18.7), g++ (v5.4.0); make (v4.1), cmake (v3.5.1), autogen (v5.18.7), autoconf (v2.69), automake (v1.14); yasm (v1.3.0), nasm (v2.11); libtool (v2.4.6); boost libraries (libboost-all-dev) (v1.58); libevent (libevent-dev) (v2.0.21):

```
$ sudo apt-get install gcc g++ make cmake autogen autoconf automake yasm nasm libtool libboost-all-dev libevent-dev
```

Users can install the following library manually: Intel-storage-acceleration-library (ISA-l) (v2.14.0):

```
$ tar -zxvf isa-l-2.14.0.tar.gz
$ cd isa-l-2.14.0
$ sh autogen.sh
$ ./configure; make; sudo make install
```

LogECMem Installation

Memcached Servers (v1.4.25)

```
$ sudo apt-get install memcached
```

For standalone setup, users can use "bash cls.sh" to re-set memcached instances with IPs and Ports; For distributed setup, users can use multiple nodes with different IPs to run memcached instances.

LogECMem Proxy

Users can use source code to install libmemcached (extented from v1.0.18) in the proxy:

```
$ cd libmemcached-1.0.18
$ sh configure; make; sudo make install
$ export LD_LIBRARY_PATH=/usr/local/lib:$LD_LIBRARY_PATH
```

Users can setup passwordless SSH login, and use "SSH root@node bash cls.sh" to re-set memcached instances in the proxy.

Users can use g++ to compile *update.cpp* and *repair.cpp*:

```
$ bash make.sh
```

Workloads

Users can use the provided workloads ("*ycsb_set.txt*" and "*ycsb_test.txt*") in each directory to do demo tests, and further more workloads via YCSB with *basic* parameter in *https://github.com/brianfrankcooper/YCSB/wiki/Running-a-Workload.*

ARTIFACT EVALUATION

The paper mainly compares In-place, Full-stripe and LogECMem in terms of update latency, memory overhead and multiple chunks failures repair performance.

1. Update latency and Memory overhead

Users can configure *N, K, workload dir, and server IP* parameters, and gain all three In-palace, Full-stripe and LogECMem results. "./update [1|2|3] dir N K IP > /dev/null", 1|2|3 indicates In-place|Full-stripe|LogECMem respectively, *N* indicates the number of all data and parity chunks, *K* indicates the number of all data chunks, *dir* indicates the path of workloads and *IP* indicates the DRAM node's IP. Note that users can configure more IPs and Ports in *update.cpp* and *run.sh* for distributed setup.

```
$ cd update
$ bash run.sh
```

2. Multiple chunks failures repair performance

Users can configure *N, K, workload dir and server IP* parameters, and gain all schemes' repair performance. " ./repair dir N K > /dev/null", *N* indicates the number of all data and parity chunks, *K* indicates the number of all data chunks, *dir* indicates the path of workloads and *IP* indicates the DRAM node's IP. Note that users can configure more IPs and Ports in *repair.cpp* and *run.sh* for distributed setup.

```
$ cd repair
$ bash run.sh
```

Author-Created or Modified Artifacts:

```
Persistent ID:
↪  https://zenodo.org/badge/latestdoi/355192108,
↪  https://github.com/yuchonghu/logecmem
Artifact name: LogECMem
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: N/A

Operating systems and versions: Ubuntu 16.04

Compilers and versions: gcc v5.18.7, g++ v5.4.0

Applications and versions: N/A

Libraries and versions: Memcached v1.4, Libmemcached v1.0.18, isa-l v2.14

Key algorithms: N/A

Input datasets and versions: N/A