Revisiting Updating Schemes for Erasure-coded In-Memory Stores

Jie Xia[1], Jianzhong Huang[1]*, Xiao Qin[2], Qiang Cao[1], Changsheng Xie[1]
[1]*Wuhan National Lab. for Optoelectronics, Key Lab. of Information Storage System (Ministry of Education of China), Huazhong University of Science & Technology, Wuhan, China 430074*
[2]*Department of Computer Science and Software Engineering, Auburn University, Auburn, AL 36849, USA*
{*xjie, hjzh, caoqiang, cs_xie*}*@hust.edu.cn, xqin@auburn.edu.* **Corresponding Author*

***Abstract*—Erasure coding has been gradually adopted by existing data-intensive in-memory stores for 'hot' data; small writes lead to expensive updating overheads in such in-memory stores characterized by update-heavy workloads. There is a pressing demand to address the issue of data updates for erasure-coded in-memory stores. We revisit existing updating schemes in erasure-coded storage clusters by investigating the applicability of these updating schemes to erasure-coded in-memory stores. After an intensive analysis, we propose a grouping-update mechanism - *GU* - to handle small writes in in-memory stores. With *GU* in place, requests in an updating window are categorized into several updating groups, where multiple small updates in the same stripe can be executed concurrently. Furthermore, we bring forward a hybrid-updating scheme - *Hybrid-U* - to minimize total updating I/Os over network under common writes (e.g., small and large writes). We evaluate four dedicated updating schemes, four *PU*-based updating schemes and *Hybrid-U*. Our experiments illustrate that *PU*-based updating schemes and *Hybrid-U* outperform the four dedicated updating schemes in terms of updating time.**

***Keywords*-Erasure codes, In-memory store, Data update, Balanced I/Os;**

I. Introduction

It is a trend to keep a large amount of data in main memory to achieve high access performance. For example, scientific computing produces intermediate results from huge raw data sets; keeping data sets in memories rather than disks accomplishes low-latency accesses. When DRAM becomes inexpensive, in-memory stores are increasingly gaining popularity (see, for example, Tachyon [7], Cocytus [15], MemEC [13]).

The volatility of DRAM makes in-memory stores fail to resist data losses in the presence of power outages, program crashes, server reboots, and the like. More often than not, in-memory stores place one or multiple copies of data to memory while keeping at least one copy onto non-volatile media. For example, Tachyon writes one copy of data to remote memory and writes three replicas to HDFS systems [7].

Erasure coding achieves the same fault tolerance with much lower redundancy than replications [12]. When it comes to in-memory stores, erasure coding has higher memory efficiency compared to replications. For example, the memory efficiency of triplication tolerating double failures is 33.3%, whereas it is 66.7% for double-fault-tolerant (k+r=6, k=4) Reed-Solomon codes. Hence, one may leverage erasure coding to keep data in memory (see, for example, EC-Cache [10], MemEC [13]).

It is indispensable to deal with data updates in-memory stores, because erasure coding is applied to data-intensive in-memory stores for 'hot' (a.k.a., frequently accessed) data, and small writes cause high updating overhead. Furthermore, a handful of studies investigated data updates in erasure-coding-based in-memory stores. For example, Zhang *et al.* and Yiu *et al.* respectively evaluate Cocytus [15] and MemEC [13] under both read-heavy (i.e., 95% reads and 5% updates) and update-heavy (i.e., 50% reads and 50% updates) workloads.

As mentioned above, one or multiple data-block copies are placed in memories with at least one copy stored on disks to achieve high availability. For simplicity, we assume that there is just one in-memory copy in a node and one on-disk replica on anther node. Apart from DRAM accesses and network transfers, disk writes are involved when a new in-memory data block is created or an in-memory data block is modified by a new data block. The reason why such disk writes are required is that the new data block should be synchronized to underlying disks to maintain data consistency.

It is reported that real-world Key-Value storage workloads are dominated by small-sized objects [1]. A data block is most likely to contain one or multiple objects; an object-updating request is mostly issued to a single data block (i.e., a small write). Small writes usually cause expensive updating overhead. Therefore, we propose a group-based updating mechanism - ***GU*** - for small writes in erasure-coded in-memory stores. With **GU** in place, requests in an updating window are reorganized into several updating groups, thereby enabling multiple small updates in the same updating group to be performed in a parallel manner. Furthermore, we bring forward a hybrid updating scheme - ***Hybrid-U***- to address common updating scenarios (i.e., including small and large writes) for erasure-coded in-memory stores. To reduce an overall updating I/Os via networks, the **Hybrid-U** scheme adopts an update-granularity threshold (i.e., $U_{threshold}$) to select an appropriate updating scheme for a specific update. .

The rest of this paper is organized as follows. Section II

978-1-5386-3486-8/17/$31.00 ©2017 IEEE
Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:07:06 UTC from IEEE Xplore. Restrictions apply.

describes the background of erasure codes and updating methods. The GU mechanism and Hybrid-U scheme are elaborated in Section III and Section IV, respectively. Performance evaluation is given in Section V. Section VI outlines the related work. Section VII concludes this paper.

II. Background

A. Erasure Coding

Among various erasure codes, (k+r,k) Reed-Solomon (i.e., *RS*) codes becomes the most popular choice because RS codes have optimal storage efficiency and support high level of fault tolerance. RS codes accomplish encoding with a k×(k+r) *Generator Matrix*, which involves a k×k *Identity Matrix* and a k×r *Redundancy Matrix* [8]. The encoding adopts simple linear algebra, and parity blocks are generated by multiplying k data blocks with the *redundancy matrix*. Fig. 1 shows the encoding process of (6,4) RS codes.

$$\begin{bmatrix} \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} & \alpha_{1,4} \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} \end{bmatrix} \times \begin{bmatrix} D_1 \\ D_2 \\ D_3 \\ D_4 \end{bmatrix} = \begin{bmatrix} P_1 \\ P_2 \end{bmatrix}$$

Transposed Redundancy Matrix — Data Block — Parity Block

Figure 1. Reed-Solomon coding for parameters k=4 and r=2.

Each block can be divided into multiple strips, and the collection of k data strips and r associated parity strips in (k+r,k) RS codes is called a stripe [9]. For simplicity, we assume that blocks and strips are equivalent, k data blocks and r associated parity blocks form a stripe.

B. RS-coded In-memory Stores

Many replication declustering schemes are proposed to distribute data replicas among disks in different nodes, e.g., chained declustering [6]. These replication declustering schemes are designed for 'on-disk primary copy ↔ on-disk secondary copy' data layout. When it comes to in-memory stores, data placement schemes for 'on-disk primary copy ↔ in-memory secondary copy' data layout should be investigated. Fig. 2 shows a data layout for (6,4) RS-coded in-memory stores, where the primary copies of the 4 data blocks are distributed over disks in nodes like a chain, and the secondary copy of one data block is placed to the neighboring memory of its primary copy. Furthermore, as to (k+r,k) RS-coded in-memory stores, both k data blocks and r parity blocks in a stripe are exclusively dispatched among memories in k+r nodes. Therefore, r=2 parity blocks $\{\mathrm{P}_1, \mathrm{P}_2\}$ are exclusively kept to memories in 2 parity nodes $\{\mathrm{PN}_1, \mathrm{PN}_2\}$.

For simplicity, we assume (k+r, k) RS-coded in-memory stores adopt chained declustering to distribute both in-memory data blocks and on-disk replicas among k nodes hereinafter, unless otherwise specified.

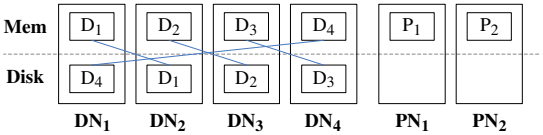


Figure 2. The data layout in a (6,4) RS-coded in-memory store.

C. Updating Methods

There exist two updating methods in RAIDs, namely, reconstruction write (a.k.a., RCW) [11] and read-modify-write (a.k.a., RMW) [2]. It is worth noting that RCW and RMW are equivalent to the *write-using-addition* and *write-using-subtraction* methods [4], respectively. The RCW method yields new parity blocks from both old and new data blocks; while the RMW method generates new parity blocks using old parity blocks and associated data blocks.

Fig. 3 illustrates a small write scenario for (6, 4) RS-coded in-memory stores, where data block D_1 is modified by D'_1. Both RCW and RMW updating procedures are given below.

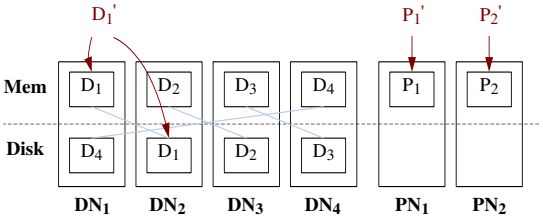


Figure 3. A small write scenario, with data block D_1 is modified by D'_1.

Procedure using RCW method. (1) new parity blocks P'_1 and P'_2 are generated from new data block D'_1 and 3 old data blocks $\{\mathrm{D}_2, \mathrm{D}_3, \mathrm{D}_4\}$ read from memories in nodes $\{\mathrm{DN}_2, \mathrm{DN}_3, \mathrm{DN}_4\}$ using the RCW method; (2) new data block D'_1 and new parity blocks $\{\mathrm{P}'_1, \mathrm{P}'_2\}$ are written to memories in corresponding nodes, and new data block D'_1 is written to disk in node DN_2.

Procedure using RMW method. (1) new parity blocks P'_1 and P'_2 are calculated using old in-memory data block D_1, old in-memory parity blocks P_1 and P_2, and new data block D'_1 using the RMW method; (2) new parity blocks P'_1 and P'_2 are respectively written to memories in nodes PN_1 and PN_2, and the new data block D'_1 are simultaneously written to memory in node DN_1, and disk in node DN_2.

III. GU: A Group-based Updating Mechanism

A. Applying Existing Schemes to In-memory Stores

Zhang *et al.* introduced two in-place updating schemes (i.e., *DUM*, and *PUM*) for erasure-coded storage clusters according to the *RCW* and *RMW* methods, and further proposed two new partial-updating schemes (i.e., PUM-P, and PDN-P) [14]. Fig. 4 shows the updating diagrams of the four updating schemes in the arena of erasure-coded in-memory stores. Fig. 4 shows that apart from r in-memory

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:07:06 UTC from IEEE Xplore. Restrictions apply.

parity blocks, two copies of data block D_1 (i.e., an in-memory copy in node DN_1, and another on-disk copy in node DN_2) need to be updated. The updating manager is a module residing in a dedicated node (a.k.a., a coordinator node), and new data blocks are issued by a proxy node. The coordinator node manages the entire in-memory store; the proxy node acts as an entry point for clients [13].

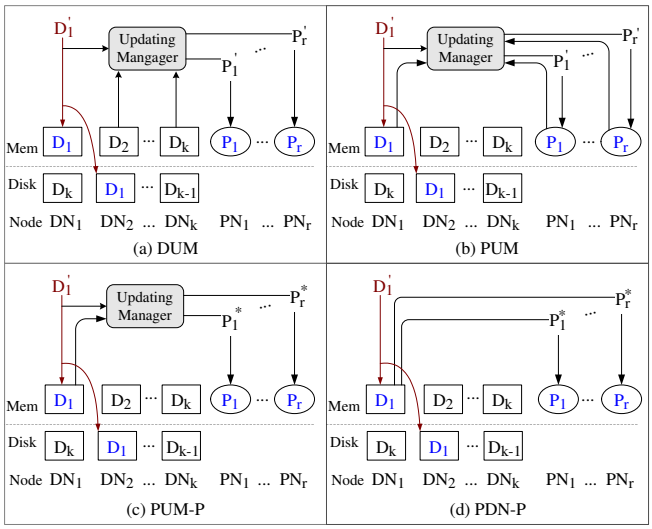


Figure 4. Updating diagrams under erasure-coded in-memory stores, with data block D_1 is modified by D_1'.

In PUM-P and PDN-P, intermediate parity block P_j^* is calculated using (1) below:

$$P_j^* = \alpha_{j,i}(D_i'\text{-}D_i), \quad j \in \{1, 2, ..., r\} \tag{1}$$

B. GU-based Updating Schemes

Because real-world KV storage workloads are usually dominated by small-sized objects [1], each write operation is located to a single record or block, meaning that the operation is a small write. To handle small writes, an updating method that are similar to PDN-P is adopted by Cocytus [15] and MemEC [13] to handle each small writes, and all small writes are processed in a 'one-by-one' manner.

We advocate for parallel updates to handle small writes in erasure-coded in-memory stores, because there is sufficient updating-improvement space when disk writes are embraced by data updates. As reported in Cocytus and MemEC [13], updating time is around 1ms for in-memory data in erasure-coded in-memory stores, where disk writes are ignored; otherwise, updating time is exceeding 1ms (e.g., 6ms) due to disk accesses. Therefore, we design a group-based updating mechanism - ***GU*** - for small writes in erasure-coded in-memory stores.

GU is comprised of two fundamental phases. First, a list of updating requests in updating window ΔT are partitioned into several updating groups, ensuring that (1) updates in a group belong to an identical stripe, and (2) updates in a group are not associated with an identical data block. Second, some updating groups are combined into a new updating group, with all updates in the new updating group are uncorrelated with an identical nodes. In this way, all updates in the new updating group can be concurrently executed.

Algorithm 1 outlines the pseudo-code of GU performing three steps. First, N_u update requests in updating window ΔT are categorized into N_g updating groups $\{G_1, G_2, ..., G_{N_g}\}$, with updates belonged to a specific stripe are inserted to an updating group. Then, the N_g updating groups are sorted in time order according to arrival times of the earliest updating request in a group. Second, if an identical data block is updated by two or multiple updates in an updating group; that is, *write-write conflicts* [5], then all the updates other than the earliest one are regarded as conflicted updates. In this case, all conflicted updates are removed from the updating group, followed by being prepended to the request list in next updating window. Third, if there is no an identical node involved in two or multiple adjacent updating groups, then the stripes within these updating groups will be concurrently updated.

Algorithm 1: The GU-based Updating Procedure

```
/* There are N_u updates in an updating window ΔT. */
foreach updating window ΔT do
    Categorize N_u updates into N_g updating groups {G_1, G_2, ..., G_Ng},
    with updates in a stripe are inserted to an updating group;
    Sort N_g updating groups according to arrival time of the earliest
    updating request in a group;
    foreach group G_i ∈ {G_1, G_2, ..., G_Ng} do
        Search conflicted updates;
        Prepend conflicted updates to request list in next updating window;
        Remove conflicted updates from updating group;
    end
    foreach group G_i ∈ {G_1, G_2, ..., G_Ng} do
        if There is no identical node involved in groups G_i and G_{i+1} then
            Combine groups G_i and G_{i+1}, and denote them as G_{i+1};
        else
            Perform parallel updating for each stripe in group G_i using a
            method (e.g., DUM);
        end
    end
end
```

Existing updating schemes can be adapted to small writes in in-memory stores (see Fig. 4). With the GU mechanism in place, the updating schemes not only handle small writes in a stripe in parallel, but also concurrently update multiple uncorrelated stripes. Updating schemes DUM, PUM, PUM-P, and PDN-P adopting the GU mechanism are referred to as $DUM_{[gu]}$, $PUM_{[gu]}$, $PUM\text{-}P_{[gu]}$, and $PDN\text{-}P_{[gu]}$, respectively.

C. Analysis of Overhead

We quantitatively analyze the number of I/Os incurred by updates. Table I presents the amount of updating I/Os over network when u data blocks are modified in (k+r,k)

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:07:06 UTC from IEEE Xplore. Restrictions apply.

erasure-coded in-memory stores, with $1 \leq u \leq k$. Table I reveals that PUM-P$_{[gu]}$ and PDN-P$_{[gu]}$ have lower updating I/O load over network than DUM$_{[gu]}$ when the value of parameter u is small (e.g., u=1); on the contrary, DUM$_{[gu]}$ has lower updating I/O load over network than PUM-P$_{[gu]}$ and PDN-P$_{[gu]}$ when parameter u's value is large (e.g., u=k).

Table I. Number of network I/Os incurred in (k+r,k) RS-coded in-memory stores, with u data blocks are updated. Notations '[w,D]', '[w,M]', and '[r,M]' indicate 'write to Disk', 'write to Memory', and 'read from Memory', respectively.

Updating Scheme	Number of I/Os over network
DUM$_{[gu]}$	u[w,D] + u[w,M] + k[w,M] + r[w,M], i.e., k + 2u + r
PUM$_{[gu]}$	u[w,D] + u[w,M] + (2u+r)[w,M] + r[w,M], i.e., 4u + 2r
PUM-P$_{[gu]}$	u[w,D] + u[w,M] + 2u[w,M] + r[w,M], i.e., 4u + r
PDN-P$_{[gu]}$	u[w,D] + u[w,M] + u×r[w,M], i.e., u×(r+2)

IV. Hybrid-U: A Hybrid Updating Scheme

A. Idea of Hybrid-U

Table I shows that PUM-P$_{[gu]}$ and PDN-P$_{[gu]}$ are tailored for small updates; whereas DUM$_{[gu]}$ is conducive to handling large updates. Intuitively, we bring forward a hybrid updating scheme - **Hybrid-U** - to process mixed writes (i.e., small and large writes) in erasure-coded in-memory stores. For example, a Hybrid-U can be 'DUM$_{[gu]}$ + PDN-P$_{[gu]}$', where DUM$_{[gu]}$ and PDN-P$_{[gu]}$ handle large updates and small updates, respectively.



Figure 5. The schematic diagram of the ' DUM$_{[gu]}$ + PDN-P$_{[gu]}$' updating schemes.

Fig. 5 depicts the schematic diagram of the DUM$_{[gu]}$ + PDN-P$_{[gu]}$ hybrid updating scheme, where U$_{\text{threshold}}$ is a threshold governing the selection of an appropriate updating scheme. If the number of to-be-modified data blocks is smaller than U$_{\text{threshold}}$, then the PDN-P$_{[gu]}$ scheme is employed; otherwise, the DUM$_{[gu]}$ scheme is applied.

B. Configuring Threshold $U_{threshold}$

Parameter U$_{\text{threshold}}$ governs the selection of an appropriate updating scheme. Theoretically, the optimal value of U$_{\text{threshold}}$ could be determined if a design goal of the hybrid updating scheme is to minimize total data transmission. When it comes to the 'DUM$_{[gu]}$ + PDN-P$_{[gu]}$' hybrid updating scheme, the U$_{\text{threshold}}$ value can be close to u if DUM$_{[gu]}$ and PDN-P$_{[gu]}$ have the same amount of data transfers. We learn from Table I that the transmission I/Os of DUM$_{[gu]}$ and PDN-P$_{[gu]}$ are k+2u+r and u×(r+2), respectively. We obtain two expressions: $U_{threshold}$=u and k+2u+r=u×(r+2). Therefore, we calculate $U_{threshold}$ using (2).

$$\mathrm{V}_{\text{threshold}} = 1+\frac{\mathrm{k}}{\mathrm{r}} \tag{2}$$

In summary, if parameter u is greater than or equal to $1+k/r$, then the DUM$_{[gu]}$ scheme is adopted; otherwise, the PDN-P$_{[gu]}$ scheme is deployed.

Recall that small updates usually refer to single block updates (see Section IV-A). More generally, in this study small updates refer to a case where the number u of updated data blocks is smaller than $U_{threshold}$; it becomes a large update when the number u of updated data blocks is greater than or equal to $U_{threshold}$.

V. Performance Evaluation

A. Experimental Environment

The RS-coded in-memory store evaluated in our experiments consists of 10 commodity-based storage nodes, one updating manager, and one client node. All the nodes are connected through a 1000Mbps switch. Each node contains 3GB DDR3 memory, and all the disks attached in the nodes are Western Digital Enterprise WD1003FBYX SATA2.0 disks. The operating systems installed on all the nodes are Linux with kernel 2.6.32.

B. Prototype and Methodology

A trace player is running on the client node, emulating multiple concurrent users. The trace player issues write requests to associated nodes according time stamps. Upon arrivals of write requests, both requested data blocks (i.e., in-memory data blocks and their on-disk copies) and the associated r parity blocks are updated, where the r new parity blocks are generated via an updating scheme (e.g., DUM$_{[gu]}$). To validate disk-write steps in the updating schemes, file-system option 'Direct I/O' is enabled to synchronize new data blocks to disks. We replay 4k updating requests generated by the YCSB benchmark [3] to evaluate the above-mentioned updating schemes. We choose the update-heavy workload (i.e., Workload A) followed by a uniform distribution, with 50% read and 50% update operations.

In-memory stores like MemEC [13] and Cocytus [15] adopt double-fault-tolerant erasure codes; similarly, we set redundancy parameter r to 2.

C. Data Updates

1) Non-GU-based Updates

Fig. 6 the shows the impact of parameter k (i.e., k={4, 6, 8}) on average updating time of the four updating schemes {DUM, PUM, PUM-P, and PDN-P}. We draw two observations from Fig. 6.

- Given a specific parameter k, all the updating schemes (i.e., DUM, PUM, PUM-P, or PDN-P) share similar updating performance. The reason is two fold. First, disk writes dominate the majority of updating overhead.

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:07:06 UTC from IEEE Xplore. Restrictions apply.

Second, only one new data block is written to a disk for an updating request under all the updating schemes.

- Each updating scheme (i.e., DUM, PUM, PUM-P, or PDN-P) exhibits similar updating time when parameter k varies; that is, the performance of each updating scheme is insensitive to parameter k under small updates;

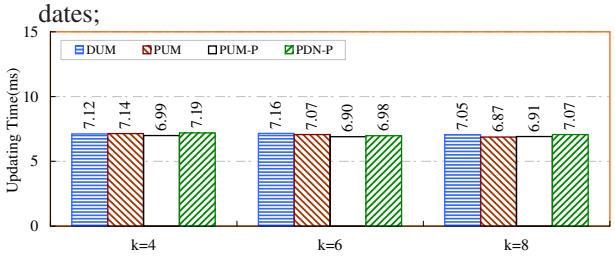


Figure 6. Average updating times of non-GU-based updating schemes under different parameter k, with k=$\{4, 6, 8\}$, r=2.

2) GU-based Updates

Fig. 7 plots average updating time four GU-based updating schemes $\{\text{DUM}_{[\text{gu}]}$, $\text{PUM}_{[\text{gu}]}$, $\text{PUM-P}_{[\text{gu}]}$, and $\text{PDN-P}_{[\text{gu}]}\}$, where parameter k is set to 4, 6, and 8, respectively. We make the following observations from Fig. 7.

- For a specific k, each GU-based updating scheme (i.e., $\text{DUM}_{[\text{gu}]}$, $\text{PUM}_{[\text{gu}]}$, $\text{PUM-P}_{[\text{gu}]}$, and $\text{PDN-P}_{[\text{gu}]}$) exhibit similar updating time due to updating overhead dominated by disk writes. All the updating schemes trigger the same total number of disk writes.
- Each GU-based updating scheme achieves higher updating performance when the value of parameter k is higher, because the number of updating groups decreases when parameter k increases, and less updating groups lead to smaller updating time.

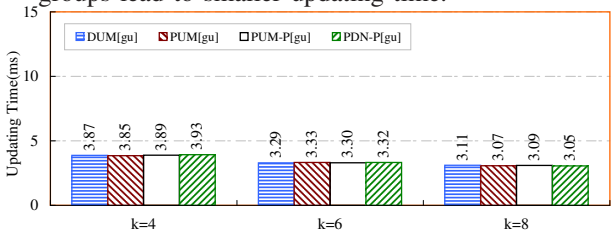


Figure 7. Average updating times of GU-based updating schemes under different parameter k, with k=$\{4, 6, 8\}$, r=2.

From Figs. 6 and 7, it is reached that the GU updating mechanism improves the updating performance of small writes for erasure-coded in-memory stores, because GU combines multiple small writes into a single write to concurrently perform multiple small writes.

3) Hybrid Updates

To evaluate the impact of threshold on the hybrid updating scheme '$\text{DUM}_{[\text{gu}]}$ + $\text{PDN-P}_{[\text{gu}]}$', we set threshold $\text{U}_{\text{threshold}}$ to 0, 4, and 6, with k=6. When $\text{U}_{\text{threshold}}$ is set to 0, the hybrid updating scheme is downgraded to the $\text{DUM}_{[\text{gu}]}$ scheme; when $\text{U}_{\text{threshold}}$ is set to 6, the hybrid updating scheme is equivalent to $\text{PDN-P}_{[\text{gu}]}$. Table II shows the request distribution of the tested workload.

Table II. Statistic of updating requests with k=6.

Statistical Item	**Updating Granularity** (Number of requests in a group)					
	1	2	3	4	5	6
Percentage	14.9%	28.0%	29.7%	18.8%	7.3%	1.3%

Fig. 8 shows the average updating time of the hybrid updating scheme, with parameters 'k=6, r=2'.

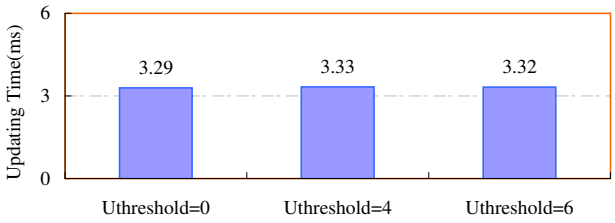


Figure 8. The impacts of the threshold value $\text{U}_{\text{threshold}}$ on the average updating time of the hybrid updating scheme '$\text{DUM}_{[\text{gu}]}$ + $\text{PDN-P}_{[\text{gu}]}$'.

Fig. 8 indicates that different thresholds (e.g., $\text{U}_{\text{threshold}}$=0, 4, or 6) have a marginal affect on updating performance of the hybrid updating scheme '$\text{DUM}_{[\text{gu}]}$ + $\text{PDN-P}_{[\text{gu}]}$', because disk writes dominate the majority of updating overhead for both $\text{DUM}_{[\text{gu}]}$ and $\text{PDN-P}_{[\text{gu}]}$ schemes, and both schemes trigger the same total amount of disk writes.

Table III shows the total updating I/Os over network of the updating schemes. We observe that the hybrid updating scheme '$\text{DUM}_{[\text{gu}]}$ + $\text{PDN-P}_{[\text{gu}]}$' achieves the least updating I/Os over network compares to DUM, PDN-P, $\text{DUM}_{[\text{gu}]}$, and $\text{PDN-P}_{[\text{gu}]}$ schemes.

Table III. Total updating I/Os over network, with k=6.

Scheme	DUM	PDN-P	$\text{DUM}_{[\text{gu}]}$	$\text{PDN-P}_{[\text{gu}]}$	Hybrid-U
Amount	40000	16000	19448	16000	15716

VI. Related Work

We outline updating schemes designed for erasure-coded in-memory stores.

Cocytus [15] and MemEC [13] address small updates in erasure-coded in-memory stores. They adopt an updating method similar to PDN-P. Concretely, a proxy/client node first sends a new value of an object to a data node; then the data node updates the object's value and forwards the data delta (e.g., 'D_i'-D_i' when D_i is changed) to all parity nodes in parallel (e.g., broadcasting the delta to all parity nodes in Cocytus); finally, each parity node adds the data delta to parity with a predefined coefficient.

Different from updating methods in existing in-memory stores only consider in-memory data updates and in-memory parity updates, our GU-based schemes take on-disk data updates, in-memory data updates, and in-memory parity updates into account. Furthermore, our GU mechanism

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:07:06 UTC from IEEE Xplore. Restrictions apply.

maximizes updating parallelisms by avoiding write-write conflicts in an updating group, and accomplishing concurrent updates among stripes in multiple updating groups.

There are three distinctions between our Hybrid-U updating scheme and the above-mentioned updating schemes: First, Hybrid-U is aimed at common writes, while existing updating schemes are mainly focused on small updates; second, Hybrid-U takes into account balanced updating I/Os, while updating performance is the primary goal of existing updating schemes; third, Hybrid-U combines both RCW and RMW updating methods (i.e., $\mathrm{DUM}_{[\mathrm{gu}]}$ + $\mathrm{PDN\text{-}P}_{[\mathrm{gu}]}$), whereas the existing in-memory stores only adopt the RMW-based updating scheme.

VII. Conclusion and Future Work

We designed a group-based updating mechanism (i.e., ***GU***) to speed up small writes in erasure-coded in-memory stores, and developed four GU-based updating schemes (i.e., $\mathrm{DUM}_{[\mathrm{gu}]}$, $\mathrm{PUM}_{[\mathrm{gu}]}$, $\mathrm{PUM\text{-}P}_{[\mathrm{gu}]}$, and $\mathrm{PDN\text{-}P}_{[\mathrm{gu}]}$). To address common writes in erasure-coded in-memory stores, we proposed a hybrid updating scheme (i.e., ***Hybrid-U***) to judiciously handle small and large writes. The experimental results confirm that the GU mechanism boosts updating performance of small updates; the Hybrid-U scheme is conducive to minimizing total updating I/Os over network.

This paper only addresses the advantage of the Hybrid-U scheme from aspect of 'minimizing updating I/Os over network'. In the future, we plan to investigate its load-balancing potential of the hybrid updating scheme.

Acknowledgment

This work is supported in part by the National Science Foundation of China under Grants 61572209, 61300047, 61472152, and the Fundamental Research Funds for the Central Universities under HUST:2015MS006. Xiao Qin's work is supported by the U.S. National Science Foundation under Grants IIS-1618669, CCF-0845257 (CAREER), and CNS-0917137, and the 111 Project under Grant B07038.

References

[1] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[2] P. Chen, E. Lee, G. Gibson, R. Katz, and D. Patterson. RAID: High-performance, Reliable Secondary Storage. *ACM Computing Surveys (CSUR)*, 26(2):145–185, 1994.

[3] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proc. of the 1st ACM Symposium on Cloud Computing (SoCC'10)*, pages 143–154, New York, NY, USA, 2010. ACM.

[4] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-diagonal Parity for Double Disk Failure Correction. In *Proc. of the 3rd USENIX Conf. on File and Storage Technologies*, pages 1–14, 2004.

[5] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The bayou architecture: Support for data sharing among mobile users. In *Proc.First Workshop on of Mobile Computing Systems and Applications(WMCSA'94)*, pages 2–7. IEEE, 1994.

[6] H.-I. Hsiao and D. J. DeWitt. Chained declustering: A new availability strategy for multiprocessor database machines. In *Data Engineering, 1990. Proc.. Sixth Int. Conf. on*, pages 456–465. IEEE, 1990.

[7] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proc. of the ACM Symposium on Cloud Computing (SoCC'14)*, pages 1–15. ACM, 2014.

[8] M. Manasse, C. Thekkath, and A. Silverberg. A Reed-solomon Code for Disk Storage, and Efficient Recovery Computations for Erasure-coded Disk Storage. *Proceeding in Informatics*, pages 1–11, 2009.

[9] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proc. of the 1988 ACM SIGMOD Int. Conf. on Management of Data*, pages 109–116, Chicago, Illinois, USA, 1988. ACM Press.

[10] K. V. 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 the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, pages 401–417, Savannah, GA, USA, 2016. USENIX Association.

[11] S. Savage and J. Wilkes. AFRAID: A Frequently Redundant Array of Independent Disks. In *Proc. of the 1996 Annual Conf. on USENIX Annual Technical Conf.*, pages 3–12, San Diego, CA, USA, 1996. USENIX.

[12] H. Weatherspoon and J. Kubiatowicz. Erasure Coding vs. Replication: A Quantitative Comparison. In *Proc. of the First Int. Workshop on Peer-to-Peer Systems (IPTPS'02)*, pages 328–337, Cambridge, MA, USA, 2002. Springer.

[13] M. M. T. Yiu, H. H. W. Chan, and P. P. C. Lee. Erasure Coding for Small Objects in In-memory KV Storage. In *Proc. of the 10th ACM Int. Systems and Storage Conf. (SYSTOR'17)*, pages 14:1–14:12, New York, NY, USA, 2017. ACM.

[14] F. Zhang, J. Huang, and C. Xie. Two Efficient Partial-Updating Schemes for Erasure-Coded Storage Clusters. In *Proc. of the 7th IEEE Int. Conf. on Networking, Architecture and Storage (NAS'12)*, pages 21–30, Xiamen, China, 2012. IEEE.

[15] H. Zhang, M. Dong, and H. Chen. Efficient and Available In-memory KV-store with Hybrid Erasure Coding and Replication. In *Proc. of the 14th Usenix Conf. on File and Storage Technologies (FAST'16)*, pages 167–180, Berkeley, CA, USA, 2016. USENIX Association.

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:07:06 UTC from IEEE Xplore. Restrictions apply.