PaRS: A Popularity-Aware Redundancy Scheme for In-Memory Stores

Panping Zhou, Jianzhong Huang, Xiao Qin, *Senior Member, IEEE*, and Changsheng Xie

Abstract—In-memory store has become a key component for an increasing number of data-intensive applications like *OLTP* and *OLAP*. To be resilient to data loss incurred by transient failures, redundancy strategies are incorporated into in-memory stores. In-memory datasets are characterized by skewed popularity, because they exhibit varied access frequencies (a.k.a., number of accesses). Therefore, it is prudent to apply customized redundancy schemes with dynamic memory efficiency and access parallelisms to different in-memory datasets. In this work, we propose an adaptive redundancy scheme—*PaRS*—for in-memory datasets. PaRS relies on a re-stripe or replication mechanism to transform involved redundancy groups according to their workload popularity growth. With PaRS in place, a memory-efficient redundancy layout is deployed for data blocks with low access frequencies; a redundancy layout exhibiting high access parallelism is adopted for highly-accessed data blocks. Compared with existing redundancy schemes that employ simple replication or erasure coding, PaRS facilitates a configurable tradeoff between memory efficiency and access parallelism for in-memory data blocks. Quantitative evaluations using YCSB show that PaRS enables in-memory stores to exhibit higher access performance and memory efficiency than the replication scheme. Furthermore, PaRS achieves better load balancing than the erasure coding, while sustaining superb access performance and memory efficiency. In particular, under a double-fault-tolerant in-memory store of limited memory, PaRS improves access latency by 15.1 to 31.5 percent compared to 3-way replication, and PaRS enhances load balancing by more than 3.9x relative to Reed-Solomon coding.

Index Terms—In-memory store, erasure codes, workload popularity, memory efficiency, access parallelism

1 Introduction

It is of vital importance to periodically adjust data-block distributions to coordinate dynamically changing workload for in-memory stores. In this study, we propose an adaptive redundancy scheme—*PaRS*—for in-memory datasets. The overarching goal of PaRS is to promote load balancing and to facilitate an adjustable tradeoff between memory efficiency and access parallelism.

1.1 Motivation

The following three factors motivate us to investigate an adaptive redundancy scheme for in-memory stores.

- *high access performance of in-memory stores,*
- *tradeoff between memory efficiency and access parallelism,*
- *load imbalance induced by popularity skew.*

Motivation 1. Nowadays, high-performance network interconnects make remote memories appear as local ones [1]. In particular, the zero-copy data transfer protocol RDMA over InfiniBand enables memories pre-allocated by multiple servers to form an in-memory store, which speeds up I/O-bound applications (e.g., database queries) by the virtue of low-latency and high-throughput accesses.

In-memory store has increasingly gained popularity in data-intensive applications like *OLTP* [2] and *OLAP* [3]. The volatility of DRAM makes in-memory stores fail to resist data losses in the presence of transient failures such as power outages, system crashes, and just name a few. Therefore, redundancy solutions (e.g., replication and erasure coding) are introduced to in-memory stores to shield against failures. Replication provides a simple but expensive form of fault tolerance. For example, Repcached [4], Redis [5], Bigmemory [6], Coherence [7], and Websphere [8] keep two replicas to memories among nodes. Compared to replication, erasure coding achieves the same level of fault tolerance with much less space store [9]; for instance, EC-cache [10], Cocytus [11], and MemEC [12] adopt erasure coding to construct reliable cluster caches.

Motivation 2. Two imperative metrics are closely associated with redundancy solutions applied to in-memory stores. The first one is *memory efficiency*, which is a ratio of user data to the total of user data plus redundancy data [13]. Since DRAM is a constrained and expensive resource, memory efficiency is an important assessing metric. In particular, DRAM favors erasure coding relative to replication, because erasure coding achieves higher memory efficiency than replication. The second metric is *access parallelism*, which mainly depends on the number of replicas (i.e., *replication factor*) of user data. A high replication factor indicates an enhanced probability of locating a block.

- *P. Zhou, J. Huang, and C. Xie are with Wuhan National Lab. for Optoelectronics, Key Laboratory of Information Storage System, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: {zhoupanping, hjzh, cs_xie}@hust.edu.cn.*
- *X. Qin is with the Department of Computer Science and Software Engineering, Shelby Center for Engineering Technology, Samuel Ginn College of Engineering, Auburn University, Auburn, AL 36849-5347. E-mail: xqin@auburn.edu.*

Manuscript received 27 Mar. 2018; revised 18 Sept. 2018; accepted 15 Oct. 2018. Date of publication 21 Oct. 2018; date of current version 19 Mar. 2019. (Corresponding author: Jianzhong Huang.)
Recommended for acceptance by Z. Shao.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2018.2876827

0018-9340 © 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:37:07 UTC from IEEE Xplore. Restrictions apply.

TABLE 1
Comparisons of Three Double-Fault-Tolerant Redundancy Solutions

Solutions	Blocks	Efficiency	Parallelism
3-way replication	$\{D_1, D_2, D_3, D_4\} \times 3$	1/3	3
Mirrored RAID-5	$\{D_1, D_2, D_3, D_4\} \times 2, P_1$	4/9	2
(k + r = 6, k = 4) RS	$\{D_1, D_2, D_3, D_4\}, P_1, P_2$	2/3	1

Intuitively, replication supports higher access parallelisms than erasure coding, because there is only a copy of source data block in erasure coding, whereas there are two or more copies for each data block in the replication case.

Table 1 summarizes the discrepancies among the three double-fault-tolerant redundancy solutions (i.e., 3-way replication, mirrored RAID-5, and (6, 4) RS codes), where user data blocks are denoted as $\{D_1, D_2, D_3, D_4\}$. It is observed that high access parallelism is achieved at the cost of low memory efficiency. For example, 3-way replication delivers the highest access parallelism while suffering the lowest memory efficiency.

Motivation 3. There exists a large variation among on-disk files or in-memory blocks in terms of accesses. Dryad job logs collected from Microsoft Bing's clusters indicate that 2.5 percent of the files are accessed more than 10 times, and 26 percent of the files are accessed by no more than once [14]. Therefore, dataset popularity varies over time, leading to popularity skew. More often than not, access popularity skew is a common characteristic of workloads in cluster caches [10], [14], [15], [16], [17], [18], [19], [20].

In popularity skew, the majority of data blocks are accessed only a few times (i.e., cold data), and a large portion of accesses are issued to a small fraction of data (i.e., hot data). As a result, such popularity skew in cluster caches induces significant load imbalance across nodes in a cluster. Load imbalance causes two-fold drawbacks: i) *over-provisioning* to accommodate peak workloads, resulting in severe waste in hardware and energy costs [10], [17]; ii) *performance degradation* in terms of access latency and throughput. Evidence shows that access performance declines as popularity skewness increases [18].

1.2 Challenges and Strategies

We are facing the following two challenges while designing an adaptive redundancy scheme for in-memory stores.

Challenge 1. How to achieve a good tradeoff between memory efficiency and access parallelism? Table 1 shows that memory efficiency and access parallelism are two contradicting factors. That is, high access parallelisms are achieved at the cost of poor memory efficiency.

Challenge 2. How to make data-block distributions cater to current access popularity? Access popularity skew is a prevailing characteristic of workloads in cluster caches. Load imbalance occurs when data are uniformly distributed among servers and the popularity skew is ignored.

Recall that each redundancy scheme (e.g., *N-way* replication and *(k + r, k)* erasure codes) exhibits distinctive memory efficiency and access parallelism; it is unwise to adopt a single redundancy scheme for all in-memory datasets regardless of the fluctuation of access frequency. Therefore, it is an alternative to integrate two redundancy schemes or more in an in-memory store, where each redundancy scheme is applied to a specific portion of in-memory datasets. A typical example is selective replication [14], [19], which creates or removes replicas for highly or lowly accessed datasets, respectively. Apart from selective replication, a hybrid redundancy scheme chooses replication or erasure codes for various datasets according to their access frequencies. For example, an adaptive replication management system or *ARM* replicates popular files while applying an erasure code to store unpopular files [21].

A uniform data distribution tends to deliver poor performance to in-memory datasets characterized by fluctuation in access popularity. In this paper, we propose a popularity-aware redundancy scheme—*PaRS*—to dynamically adapt to changing access popularity. PaRS employs hybrid-coding schemes (e.g., a variant of Mirrored RAID-5) for in-memory datasets; PaRS relies on a re-stripe or replication mechanism to adjust access parallelism according to popularity growth. Furthermore, both load balancing and data locality are taken into consideration when selected redundancy groups are transformed. Compared to redundancy schemes based on pure replication or erasure coding, PaRS provides in-memory data blocks with a configurable tradeoff between memory efficiency and access parallelism.

1.3 Contributions

This work makes three main contributions:

- We propose an adaptive redundancy scheme (i.e., PaRS) for in-memory datasets. PaRS relies on a re-stripe or replication mechanism to change both redundancy degrees and access parallelisms of data blocks in a redundancy group according to popularity growth.
- We thoroughly analyze the evolution behaviors (e.g., create, re-stripe, replicate, and reclaim) of in-memory datasets. Key design issues of PaRS elaborated in this paper include popularity levels, transformation of redundancy groups, a mechanism for guaranteeing data consistence, and load balancing through data placement.
- We implement a proof-of-concept prototype, in which the three redundancy schemes (i.e., Rep, PaRS, and EC) are deployed and quantitatively evaluated. Furthermore, we investigate the impacts of configuration factors (i.e., parameters k, l, and μ) on PaRS. The experimental results illustrate that PaRS enables in-memory stores to deliver higher access performance and memory efficiency than Rep. Compared to EC, PaRS accomplishes better load balancing in-memory stores while sustaining superb access performance and memory efficiency.

1.4 Roadmap

The rest of the paper is organized as follows. We introduce the background information in Section 2. The design of PaRS is detailed in Section 3. We present quantitative performance evaluation in Section 4. Section 5 outlines differentiated fault-tolerance services and in-memory load balancing techniques. Section 6 discusses implementation issues. We conclude this paper in Section 7.

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:37:07 UTC from IEEE Xplore. Restrictions apply.

TABLE 2
Data Organization of Lin-Memory Systems

Scheme	In-memory Data	On-disk Data
Redis [5], Repcached [4]	$\{D_1, D_2, \ldots, D_k\} \times 2$	$\{D_1, D_2, \ldots, D_k\}$
BigMemory [6], Websphere [8]	$\{D_1, D_2, \ldots, D_k\} \times 2$	{D}
Coherence [7]	$\{D_1, D_2, \ldots, D_k\}$,{D}	{D}

2 Background

Since RAM is volatile, it is unable to store data when it is powered off. Fault tolerance for in-memory stores may be accomplished through replication or erasure coding.

2.1 Replication for In-Memory Store

Replication is a simple yet effective redundancy approach to tolerating failures. When a data block is lost, only one replica block is adequate to repair it, resulting in minimum data movements over network. A high replication factor (e.g., 2 or 3 in the case of replication or triplication) supports increased access parallelisms.

A handful of in-memory systems (e.g., Redis [5], Repcached [4], Bigmemory [6], Websphere [8], and Coherence [7], etc.) keep two replicas in memory among nodes to obtain memory-level redundancy. Table 2 shows that two copies of k sub-blocks reside in memories for the Redis, Repcached, BigMemory, and Websphere schemes; whereas both data block D and its k sub-blocks are kept in memories in the Coherence scheme. Here, data block D is divided into k sub-blocks $\{D_1, D_2, \ldots, D_k\}$.

2.2 Erasure Coding for In-Memory Store

Erasure codes are widely applied to storage clusters deployed in large-scale data centers [22], [23]. Among various erasure codes, Reed-Solomon (a.k.a., *RS*) codes become the most popular choices owing to the RS codes' optimal storage efficiency as well as high level of fault tolerance [9].

In particular, (k + r, k) RS codes encode source data with a $k \times (k + r)$ *Generator Matrix*, which involves a $k \times k$ *Identity Matrix* and a $k \times r$ *Redundancy Matrix* [24]. The source data are embedded in encoded data because of the *Identity Matrix*. The RS encoding adopts the simple linear algebra, where parity strips are generated by multiplying k data strips with the $k \times r$ Redundancy Matrix. In *Vandermonde Reed Solomon* codes [25], coefficient $\alpha_{j,i}$ is set to i^{j-1}, where $i \in \{1, 2, \ldots, k\}$ and $j \in \{1, 2, \ldots, r\}$.

Both k data strips and r parity strips constitute a stripe [26]. Each strip can be partitioned into multiple blocks [27]. For simplicity, a block is viewed as a strip hereinafter, unless otherwise specified.

Let us consider a (k + r, k) RS-coded in-memory store, in which k data blocks and r parity blocks in a stripe are exclusively placed to the main memories of $k+r$ storage nodes. In practice, a parity block in a stripe can either (1) be cached on a separate node or (2) be mixed with a data block in another stripe across all nodes. Here, a block may be sealed from multiple small-sized objects [12], or be a segment of large-sized objects in the range of tens to hundreds of megabytes [10].

Since memory is a constrained resource, high memory efficiency is demanding when one designs reliable in-memory stores. Erasure codes achieve higher memory efficiency than the replication counterpart. For example, the memory efficiency of triplication tolerating double failures is merely 33.3 percent, whereas double-fault-tolerant (6, 4) RS codes push the memory efficiency up to 66.7 percent. Therefore, erasure coding is gradually adopted by in-memory stores. For example, both *Cocytus* and *MemEC* are highly available in-memory key-value stores by using erasure coding [11], [12]. Cocytus tolerates node failures with a fast on-line recovery capability; MemEC accomplishes fast transitions between normal and degraded modes.

3 Popularity-Aware Redundancy Scheme

This section describes a popularity-aware redundancy scheme—*PaRS*—for in-memory stores to dynamically reclaim, replicate, and replace in-memory data blocks according to changing access frequencies.

3.1 Overview

When the access frequency of a data block changes, an usual counter-measure is to place the data block to an appropriate redundancy group[1] to gain a matching access parallelism. There are two main approaches illustrated as below:

Approach 1: reclamation or replication in a replication group. Low access frequency indicates data unpopularity. It is a conventional wisdom to reduce the number of replicas for unpopular data, thereby increasing memory efficiency and decreasing access parallelism. For example, it is irrational to preserve a specific data block in memory when the block's access frequency is zero; such block should be reclaimed. On the contrary, replication is adopted when the data becomes popular.

Approach 2: transformation of a redundancy group. It is wise to apply a redundancy layout exhibiting low access parallelisms and high memory efficiency to data blocks with low access frequency, and vice versa. For example, when a single-fault-tolerant replication group (e.g., $\{D_1, D_1, D_2, D_2\}$) is transformed into a single-fault-tolerant erasure-coding group (e.g., $\{D_1, D_2, P_1 = D_1 \oplus D_2\}$), the access parallelism of block D_1 or D_2 is decreased from 2 to 1, whereas the memory efficiency of redundancy group is increased from 1/2 to 2/3.

PaRS accomplishes a popularity-aware redundancy adjustment as follows: (i) adopting a reclamation mechanism to reduce the replication factor of a data block with decreasing popularity; (ii) using a replication mechanism to raise the replication factor when a data block's popularity goes up; (iii) re-striping a redundancy group by replacing an unpopular data block with a newly-requested popular data block.

3.2 Architecture

In-memory stores are constituted from pre-allocated memories on storage nodes in storage clusters. Logically, primary memories and secondary disks in a storage cluster can be organized into both in-memory store and on-disk storage.

1. Three redundancy patterns are involved in this study: replication (e.g., 2-way replication, 3-way replication, etc.), (k + r, k) erasure coding, and a mixture (e.g., a (k + r, k) erasure coding and replicas of specific data blocks). Here we denote 'all blocks in a stripe and/or replicas of specific data blocks' as a redundancy group. When it comes to replication or erasure coding, a redundancy group becomes a replication group or erasure-coding group, respectively.

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:37:07 UTC from IEEE Xplore. Restrictions apply.

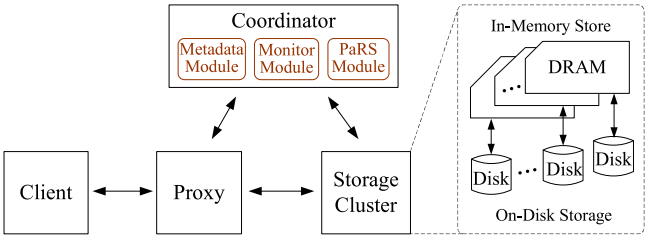

Fig. 1. An architecture of erasure-coding-based in-memory stores, with PaRS module is hosted at the coordinator node.

Fig. 1 illustrates the architecture of a storage cluster, which consists of three types of nodes, namely, storage nodes, proxy nodes, and coordinator nodes.

Each storage node allocates a memory region for in-memory store while applying local disks to guarantee data persistence. The proxy node is an entry point for front-end clients to access data in storage nodes. The coordinator node manages the entire cluster. In particular, the coordinator node (i) maintains all meta-data (e.g., block distribution on disk and memory, block's access frequency, redundancy group's logical mapping) in main memories, (ii) monitors the health state of storage nodes and the change of data access pattern, (iii) coordinates access requests in the presence of node failures, and (iv) manipulates reclamation, replication, and erasure encoding/decoding operations involved in PaRS.

3.3 Design

3.3.1 Definition of Popularity Levels

Each data block exhibits its access popularity in terms of access frequency; in-memory data blocks may be classified into arbitrary number of groups according to access frequencies. For simplicity, we organize in-memory data blocks into three popularity levels {i.e., PL-0, PL-1, PL-2}. The association between popularity levels and access frequencies is highlighted in Fig. 2, where top y% of popular data blocks are characterized by popularity level 'PL-2', and bottom x% of unpopular data blocks are classified into popularity level 'PL-0'. As a result, a data block of PL-2, PL-1, and PL-0 indicates its data is hot, warm, and cold, respectively.

Usually, the value of parameter y% is heavily influenced by workload characteristics. For instance, the top 12 percent of popular data is accessed 10× more than the bottom third in the Bing cluster; while the top 5 percent of blocks are 7x more popular than the bottom three-quarters within the Facebook cluster [28].

The bottom x% of unpopular data blocks tend to be replaced. The replacement rate of an in-memory store is influenced by many configuration factors, e.g., request arrival rate, request distribution, available space of cluster memory,

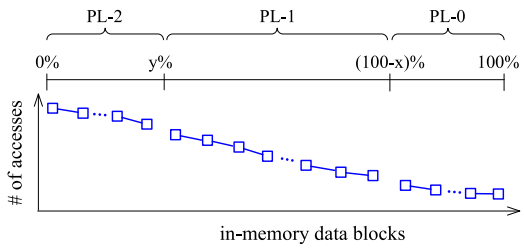

Fig. 2. Association between popularity level and access frequency.

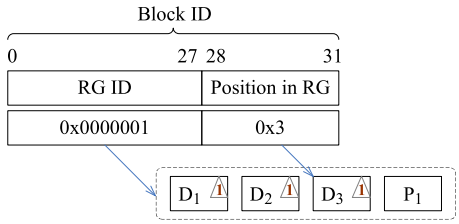

Fig. 3. Diagram of structure 'Block ID'.

redundancy parameter, etc. The following equation can be used to roughly estimate the replacement rate (i.e., x%) of an in-memory store based on its configuration factors. High replacement rate means high miss penalty [29], so the value of x% should not be large, which guides system designers to reasonably configure an in-memory store in accordance to specific workload conditions. Please refer to Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TC.2018.2876827/, for the derivation of the replacement-rate equation and the relevant parameters.

$$x\% = \frac{T_w}{2N_{total}} \div \left(\frac{N_{access} N_{queue}}{\lambda_s (N_{access} - N_{avail})(N_{queue} + 1)} - \frac{S_{blk}}{B_{net}} \right) \times 100\%. \quad (1)$$

Recall that it is prudent to respectively apply high- and low-replication-factor layouts to popular and unpopular data blocks (see also Section 3.1). Thus, it is sensible to grand double replication-factor to data blocks of PL-2. We also assign single and zero replication-factor to data blocks of PL-1 and PL-0, respectively. Data blocks of PL-0 are candidates to be evicted or replaced.

3.3.2 Data Organization

To support re-stripe and replication, PaRS needs to locate both data and parity blocks in the same redundancy group (*RG*). We leverage the block ID to fulfill this purpose. For each block, we prepend unique fixed-size block IDs to identify blocks in a node. As shown in Fig. 3, we decompose a block ID into two sections: (i) *RG ID*, which identifies the set of in-memory blocks in the same redundancy group, and (ii) *Position in RG*, which identifies the position of a block in a redundancy group. In particular, the blocks in the same redundancy group share the same redundancy group ID. The format of Block ID may be determined according to in-memory dataset requirements. It is prudent to specify a long Block ID for a large in-memory dataset. The number of data blocks in most redundancy groups is fewer than 16 ($16 = 2^4$), implying that only 4 bits are assigned to the field of 'Position in RG'. Suppose that 'Block ID' is of 32 bits, and each RG contains four original data blocks of size 8 KB, then 'RG ID' and 'Position in RG' are of 28 and 4 bits, respectively; and space for in-memory datasets is $2^{28} \times 4 \times 8$ KB = 8 TB.

PaRS maintains a redundancy group list to globally manage all redundancy groups. Fig. 4 depicts that each redundancy group comprises three fields. (i) *RG ID* is a unique identifier assigned to a redundancy group. (ii) *Layout* determines the configuration of a redundancy group. Specifically, N_d and N_p represent the number of original data blocks and associated parity blocks, respectively; $N_r[]$

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:37:07 UTC from IEEE Xplore. Restrictions apply.

RG ID	Layout	Block Address (i.e., block list)
0x0000001	N_d=3, $N_r[]$=[0,1,1], N_p=1	@D_1, @D_2, @D_3, Null, @D_2, @D_3, @P_1
0x0000002	N_d=3, $N_r[]$=[0,0,0], N_p=2	@D_1, @D_2, @D_3, Null, Null, Null, @P_1, @P_2
…	…	…
0xFFFFFF1	N_d=3, $N_r[]$=[1,0,1], N_p=1	@D_1, @D_2, @D_3, @D_1, Null, @D_3, @P_1

Fig. 4. Diagram of structure 'Redundancy Group'.

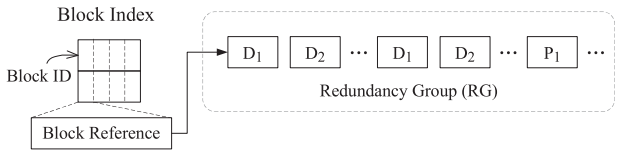

Fig. 5. Diagram of structure 'Block Index'.

Block ID	AF	PL	TM
0x00000011	0x0011	"PL-1"	2017-05-29 00:00:01.123
0x00000012	0x0101	"PL-2"	2017-05-29 00:02:01.001
…	…	…	…
0x000000e4	0x0001	"PL-0"	2017-05-29 00:03:02.012

Fig. 6. Diagram of structure 'Block Metadata'.

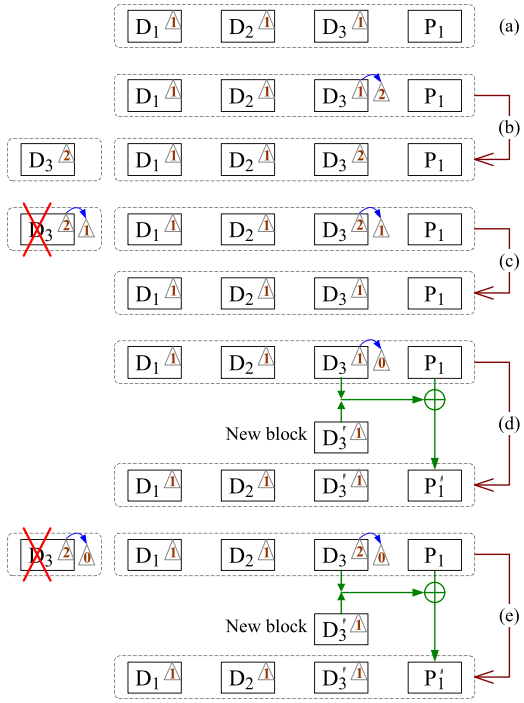

Fig. 7. Transformation of redundancy groups when the popularity level of data block D_3 changes, with encoding parameters k = 3 and r = 1. (a) Initialization; (b) PL-1→PL-2; (c) PL-2→PL-1; (d) PL-1→PL-0; (e) PL-2→PL-0.

shows the number of extra replicas of original data blocks $\{D_1, D_2, \ldots, D_{N_d}\}$. For example, '$N_r[]$ = [0, 1, 1]' signifies that blocks D_1, D_2, and D_3 have 0, 1, and 1 extra replica, respectively. (iii) *Addresses* of all blocks in a redundancy group. If a data block has no extra replica, the address of its replica will be set to 'Null'.

Some studies enhance lookup performance for key-value stores by applying a *cuckoo* hashing scheme [12], [30], [31], which adopts a new dictionary with worst case constant lookup time. Similarly, we also introduce the *cuckoo* hashing scheme to speed up block-indexing operations, in which a block ID is mapped to a block reference (see Fig. 5). When a data block is replaced owing to its deteriorated access frequency, associated parity blocks in the same redundancy group need to be updated. The parity blocks are located via both RG ID of the data block and their positions in the redundancy group (see *Layout* field in Fig. 4). PaRS efficiently locates both data and parity blocks through block indexing.

Fig. 6 shows that three kinds of metadata are affiliated to a specific block, including access frequency (AF), popularity level (PL), and timestamp of metadata modification (TM). Such metadata of blocks is frequently retrieved and updated, it is rational to look up desired blocks using a hash table.

3.3.3 Transformation of Redundancy Groups

A block's popularity level is initialized to PL-1 when it is first placed to memories; its popularity level may change to PL-0 or PL-2 when its access frequency decreases or increases, respectively. Fig. 7 shows several transformations of redundancy groups in the case of '$k = 3, r = 1$'.

Fig. 7a illustrates k = 3 data blocks $\{D_1, D_2, D_3\}$ and r = 1 associated parity block P_1 are placed to memories in four (i.e., $k + r = 4$) nodes in the initialization stage. The popularity level of each data block is initialized to 1 (see tag '1' in triangle).

Fig. 7b demonstrates how to transform a redundancy group that caters to an increasing popularity level of a data block. In particular, an extra in-memory replica is kept in a different node when the popularity level of data block D_3 is shifted from PL-1 to PL-2.

Figs. 7c, 7d, and 7e show the following three cases, in which a redundancy group is reformed due to data block D_3's popularity transition from a high level to a low one.

- *Case 1: PL-2→PL-1*. The replica of data block D_3 is marked 'deleted' and; thus, the replica's occupied memory can be reclaimed. Obviously, such a 'replica removal' process is a reverse operation of the 'replica increment' process shown in Fig. 7b.
- *Case 2: PL-1→PL-0*. The data block of PL-0 trends to be replaced by a new data block. This case is equivalent to a data update of 'data block D_3 that is changed to new data block D_3'' (see Fig. 7d). The Read-Modify-Write (a.k.a, RMW) updating method is applicable to such a single-block-updating case [32], where old data block D_3 is modified by D_3', and old parity block P_1 is changed to new parity block P_1'' (i.e., $P_1' = D_3 \oplus D_3' \oplus P_1$).
- *Case 3: PL-2→PL-0*. This case is a combination of '*Case 1: PL-2→PL-1*' and '*Case 2: PL-1→PL-0*'. Thus, a replica of block D_3 is removed; another replica is modified by new data block D_3'.

We observe from Fig. 7 that the replication factor of a data block in a resulting redundancy group is two when its popularity level is PL-2. In this scenario, PaRS supports high access parallelism for popular data; moreover, the replication factor of a data block is one when it is of PL-1, thereby allowing unpopular data to achieve high memory efficiency. When the popularity level of a data block is PL-0, it is prone to reclaim the data block (i.e., its replication factor comes

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:37:07 UTC from IEEE Xplore. Restrictions apply.

down to 0). Hence, the above transformation process in PaRS lets in-memory stores adaptively apply a redundancy paradigm to a data block according to its access popularity.

Table 3 summarizes all the notations facilitating the presentation of the algorithms deployed in PaRS.

Algorithms Proc(Seq) and Transform(RG_i) outline the pseudo-codes for processing user requests and transforming redundancy groups, respectively.

Algorithm 1. Proc(Seq)

Input:
Seq: user request sequence

```
1  Set a timer T;
2  while Seq is not over do
3    if T ≥ TW then
4      Update popularity levels for all in-memory data blocks;
5      Update current loads for all involved nodes;
6      foreach Redundancy Group RG_i do
7        Transform(RG_i);
8      Reset T;
9    if Requested block Blk is in memory or in queue then
10     if T ≥ (TM[Blk]%TW) then
11       AF[Blk] = AF[Blk] + 1;        // update frequency
12       Get Blk from SN[Blk] and deliver it to proxy;
13   else                              // Blk is a new data block
14     Retrieve Blk from disk in node SN and send it to proxy;
15     Insert Blk to queue Q[SN];
```

Algorithm 2. Transform(RG_i)

Input:
RG_i: ith redundancy group to be transformed

```
1  foreach original data block Blk in RG_i do
2    if OPL[Blk]=PL-2 and NPL[Blk]=PL-1 then
                                       // Reclaiming
3      Determine list List_src of nodes containing replicas of
       Blk;
4      Select the heaviest-load node SN in List_src;
5      Remove Blk in SN from block list of RG_i;
6    if OPL[Blk]=PL-1 and NPL[Blk]=PL-2 then
                                       // Replicating
7      Determine List_tgt, with List_tgt excludes nodes in RG_i;
8      Select the least-load node SN in List_tgt;
9      Deliver a copy of Blk to SN;
10   if NPL[Blk]=PL-0 and Blk ∈ Set_old then      // Replacing
11     Determine List_pdr, with nodes in List_pdr contain new
       data blocks in Set_new;
12     Determine List_tgt, with each node in List_tgt does not
       keep partner blocks of Blk in RG_i;
13     Select the least-load node in List_tgt as target node SN_tgt;
14     if SN_tgt ∈ List_pdr then
15       SN_pdr = SN_tgt;            // SN_pdr is a provider node
16     else
17       Select the heaviest-load node in List_pdr to be SN_pdr;
18     Move Blk_new in Set_new from node SN_pdr to node SN_tgt;
19     Update parities in RG_i using 'RMW' updating method;
20     Assign popularity level PL-1 to block Blk_new;
21     Remove all in-memory replicas of Blk;
22   Update the block list of RG_i;
23   Update timestamp (i.e., TM[ ]) of the new block/replica in
     RG_i
```

TABLE 3
Notations Used for the PaRS Algorithms

Notation	Description
k, r	Redundancy parameters used to encode in-memory blocks
Seq	User request sequence
RG_i	ith in-memroy redundancy group
TW	Time window for triggering group transformation, e.g., 100 ms
PL-i	ith popularity level, e.g., i ∈ {0, 1, 2}
$AF[Blk]$	Access frequency of data block Blk
$OPL[Blk]$	Old popularity level of data block Blk
$NPL[Blk]$	New popularity level of data block Blk
$TM[Blk]$	Timestamp of updating metadata for data block Blk
$SN[Blk]$	Storage node holding an in-memory copy of data block Blk
$Q[SN]$	Queue of new data blocks in node SN
Set_{old}	Set of in-memory data blocks to be replaced
Set_{new}	Set of in-queue data blocks replacing blocks in Set_{old}
$List_{src}$	List of source nodes containing replicas
$List_{tgt}$	List of target nodes keeping extra replicas or new data blocks
$List_{pdr}$	List of provider nodes supplying new data blocks

3.3.4 Data Distribution

To guarantee fault tolerance for in-memory datasets, all blocks in a redundancy group are exclusively distributed to memories in storage nodes. When the popularity level of a data block increases, PaRS duplicates another copy for the data block; thus, a new replica will be placed to the memory in a distinct node. When the popularity level of an existing data block is decreased to PL-0, PaRS prepares to replace the block with a new data block, and the new block may be placed to the memory in a separate node or in the original node keeping the existing data block. If data blocks are randomly distributed without taking node's load into account, then a load imbalancing issue is likely to incur. To address this issue, we consider three aspects below.

i) *Node Load*. Assuming all nodes are homogeneous, we measure a node's load using the aggregated access load of requests issued to the node. We sort all the nodes in an ascending order of the aggregated load of each node; thus, the first node in the sorted list is the least-loaded one. If the size of each request to all nodes is identical, then a node's load can be readily measured by the number of accesses issued to the node.

ii) *Load Balancing*. When a replica of an old data block is removed, PaRS selects the heaviest-load node containing the old data block (see Line 4 in Algorithm Transform(RG_i)). When an extra replica of an old data block is added, PaRS chooses the least-load node as the target node (see Line 8 in Algorithm Transform (RG_i)). When a new data block is placed to the in-memory store, PaRS selects the most lightly-loaded node from a list of candidate target nodes that are non-members of the redundancy group (see Lines 12 and 13 in Algorithm Transform(RG_i)). In the long run, the data-block load is expected to be uniformly distributed across all nodes.

iii) *Data Locality*. Each node maintains a queue for new data blocks in its local memory, acting a new-data-block provider node. All nodes containing a

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:37:07 UTC from IEEE Xplore. Restrictions apply.

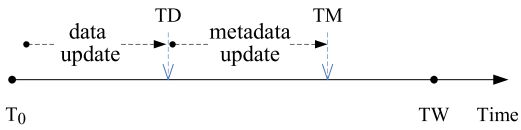

Fig. 8. Creating timestamps (i.e., $TD[\,]$ and $TM[\,]$) for updated data and metadata.

non-empty 'new-data-block' queue form a list of provider nodes. To reduce network traffic, PaRS strives to select a target node from the list of provider nodes (see Line 15 in Algorithm Transform(RG_i)). In this way, data locality can be exploited to minimize data migration over network.

3.4 Optimization

3.4.1 Metadata Update

While group transformation is in progress, it is indispensable for in-memory stores to serve user I/O requests. If some redundancy groups have been transformed but corresponding mapping metadata associated with block addresses has not yet been updated, user requests will be redirected to incorrect nodes, resulting in a data-inconsistency problem.

During the course of group transformation, two candidate operations are updating mapping metadata and updating data blocks (e.g., reclaiming/replicating/replacing data blocks). Usually, updating a data block occurs prior to updating its metadata (see Fig. 8). Stale data blocks are caused by the fact that requested data blocks may be replaced by another new blocks. To prevent incoming user requests from accessing the stale data blocks, PaRS carries out two steps: (i) assigning a timestamp to block Blk when its mapping metadata is successfully updated (see Line 23 in Algorithm Transform(RG_i)); (ii) checking whether current timestamp is greater than the metadata timestamp of block Blk (i.e., $TM[Blk]$); if this is the case, then block Blk is the legitimate one requested by the user (see Line 11 in Algorithm Proc(seq)).

3.4.2 Transformation Scheduling

Fig. 7 illustrates that there exist three main paradigms for redundancy-group transformations: i) Replicating new copies when its popularity level increases; ii) Reclaiming memory space when its popularity level declines to PL-1; iii) Replacing it with a new data block when the popularity level becomes PL-0. Not surprisingly, 'Reclaiming' is the fastest paradigm, because it is accomplished by simply setting a 'free' tag to the block and deleting its block address from the redundancy group list. 'Replacing' is the slowest paradigm as it may trigger both data migration and parity update.

To speed up the overall transformation process and accelerate metadata updating, we enforce the following order to redundancy-group transformations for all involved in-memory data blocks: 'Reclaiming a data block of PL-1' $\Rightarrow$ 'Replicating a data block of PL-2' $\Rightarrow$ 'Replacing a data block of PL-0' (see Algorithm Transform(RG_i)).

3.4.3 Determining Replacement Sets

If an in-memory data block of 'PL-0' has a higher access number than that of an in-queue data block, then there is no need to replace the in-memory data block with the in-queue one. Therefore, it is required to determine the set Set_{old} of in-memory data blocks to be replaced and the set Set_{new} of in-queue data blocks replacing blocks in Set_{old}. We refer to Set_{old} and Set_{new} as *replacement sets*.

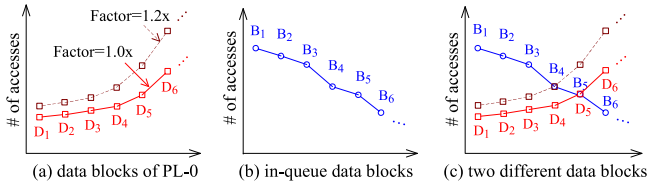

Fig. 9. Determining two replacement sets. (a) All in-memory data blocks of 'PL-0' are sorted in an ascending order of the access number. (b) All in-queue data blocks are sorted in a descending order of the access number. (c) Comparing in-memory and in-queue blocks to construct the two replacement sets.

As shown in Figs. 9a and 9b, all in-memory data blocks of 'PL-0' are sorted in an ascending order of the access number, and all in-queue data blocks are sorted in a descending order of the access number. If the access number of the first data block (i.e., D_1) in the sorted in-memory data blocks is smaller than that of the first data block (i.e., B_1) in the sorted in-queue data blocks, then data blocks D_1 and B_1 will be inserted to sets Set_{old} and Set_{new}, respectively. Such a filtering process is repeatedly performed until (1) the access number of an in-memory data block is greater than or equal to that of an in-queue data block (see blocks D_5 and B_5 in Fig. 9c), or (2) all the sorted in-memory or in-queue blocks have been processed. In this way, the two replacement sets are finalized.

To minimize data migration caused by replacement operations and to mitigate the occurrence of *memory churns* [33], we apply an amplification factor (e.g., 1.2x) to the access number of all in-memory data blocks (see Fig. 9a). In doing so, a part of borderline in-memory data blocks of PL-0 (e.g., block D_4 in Fig. 9c) will not be replaced.

With replacement sets Set_{old} and Set_{new} in place, a portion of unpopular in-memory data blocks are replaced by popular in-queue data blocks.

4 Experiments

In this section, we comparatively evaluate our PaRS scheme and two candidate redundancy schemes, i.e., replication (Rep for short) and erasure-coding (EC for short). Furthermore, we quantitatively examine the sensitivity of PaRS to various factors, e.g., number k of logical data blocks in a redundancy group, length l of queues, and distribution parameter μ of workloads.

4.1 Experimental Environment

4.1.1 System Setup

We build an experiment cluster, which consists of one coordinator node, one proxy, and 12 storage nodes {SN_1, SN_2, . . . , SN_{12},}. All the nodes are connected through a Cisco Catalyst 2960-S series Ethernet switch. The coordinator and proxy node are two Dell R720 servers, each Dell server is equipped with two Xeon E5-2609 @2.40 GHz CPUs, and 32 GB DDR3 memory. Each storage node contains an Intel E5800 @3.2 GHz CPU, 3 GB DDR3 memory, and a Western Digital Enterprise SATA2.0 disk with model WD1003FBYX.

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:37:07 UTC from IEEE Xplore. Restrictions apply.

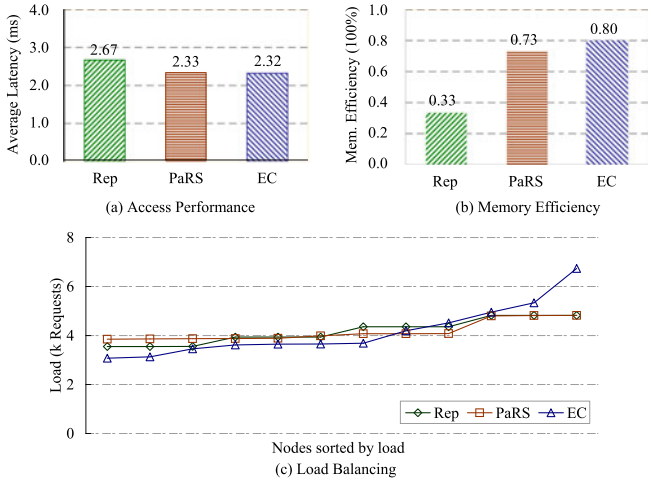

Fig. 10. Comparison of the three redundancy schemes in terms of access performance, memory efficiency, and load balancing under limited memory (i.e., the in-memory store keeps 8.0, 17.6, and 19.2 percent of dataset for Rep, PaRS, and EC, respectively). Parameter configuration: '$k = 8$, $r = 2$, and $\mu = 0.8$'.

We install the Linux with Kernel 2.6.28.10 (x86-64) on all the machines.

4.1.2 Prototype Design

We implement a prototype system, where the coordinator node embraces one out of the three redundancy schemes to guide the cluster nodes to manage in-memory blocks, and maintains all metadata (e.g., block mapping in a redundancy group, block address, access frequency, etc.); the proxy node simulates concurrent clients to issue requests by replaying I/O workloads generated from the YCSB benchmark [34]. Each of storage node not only provides pre-allocated memory region to in-memory store, but also maintains an individual queue for new data blocks.

4.1.3 Testing Methodology

We make use of read-only workloads generated from the *YCSB* benchmark to examine the validity of PaRS. The trace replayer is hosted on the proxy node, and issuing meta-data and data requests to the coordinator node and certain storage nodes, respectively; and 5 worker threads are launched for the trace replayer. The number of unique data blocks is 1K, and a total of 50K requests in total are replayed; the access pattern follows a Zipf distribution with a shape parameter μ (e.g., 0.8). Since a few in-memory stores are featured with double fault-tolerance [11], [12], redundancy parameter r is set to 2 for both the PaRS and EC schemes, and 3-way replication is adopted by the Rep scheme. In PaRS, queue length l is set to 10 blocks. Data blocks are grouped into 3 popularity levels, where PL-2 blocks account for up to 10 percent of total data blocks.

To accomplish high coding performance, we leverage Intel's open-source Intelligent Storage Acceleration Library (Intel ISA-L) [35], which accelerates erasure-code-related linear algebra calculations by exploiting advanced hardware instruction sets like SSE, AVX, and AVX2.

We conduct six categories of experiments as follows:

- limited memory constraints (Rep versus PaRS versus EC);
- varying memory resource (Rep versus PaRS versus EC);
- impact of parameter k (PaRS);
- impact of parameter l (PaRS);
- impact of parameter μ (PaRS);
- impact of parameter PL (PaRS);

4.2 Experimental Results

4.2.1 Limited Pre-Allocated Memory

Recall that DRAM is a constrained and expensive resource, it is a common practice to keep a small portion of dataset to an in-memory store. To quantitatively evaluate the three redundancy schemes (i.e., Rep, PaRS, and EC) under limited memory resource (e.g., the in-memory store can keep 8.0, 17.6 and 19.2 percent of dataset for Rep, PaRS, and EC, respectively). Fig. 10 shows the experimental results of the three redundancy schemes in terms of access performance, memory efficiency, and load balancing.

We draw two observations from Fig. 10a. First, the PaRS and EC schemes outperform the Rep counterpart, because PaRS and EC enable keep more essential data blocks in memory than Rep, reducing the probability of retrieving missed data blocks from disks. Second, PaRS and EC exhibit similar user-response performance, because a positive factor shortens response time, which is enlarged by a negative factor. On one hand, PaRS keeps high-frequency data blocks in memory in a replicated manner, thereby accelerating the popular blocks. On the other hand, unlike EC, PaRS has low memory efficiency (see Fig. 10b), meaning that fewer essential data blocks are kept in memory than the EC case.

Fig. 10b indicates that the Rep scheme is of the lowest memory efficiency among the three redundancy schemes; the memory efficiency of PaRS is marginally lower than that of EC. The basic reason behind the decreasing memory efficiency is contributed by the existence of data-block replicas.

Load imbalance can be measured via an imbalance metric λ [10], with $\lambda = ((L_{max} - L_{avg})/L_{avg})$, where L_{max} and L_{avg} are the maximal and average load among all the nodes, respectively. In this group of experiments, the imbalance metric λ for Rep, PaRS, and EC is 15.7, 15.6, and 61.7 percent, respectively. That is, PaRS and Rep exhibit similar load balancing, and the reason is two-fold: First, the replication mechanism is adopted in both Rep and PaRS, allowing access workload to be dispatched in a balanced manner. Second, compared to Rep, PaRS performs hot-data-block-replication and new-data-block-insertion operations according to node's current load. Fig. 10c reveals that the Rep and PaRS schemes achieve better load balancing than EC, because EC merely keeps one replica for each data block, and skewed-popularity requests inevitably induce significant load imbalance across the nodes in a cluster. Especially, Rep and PaRS enhance load balancing by more than a factor of 3.9 compared to the EC scheme.

4.2.2 Different Pre-Allocated Memories

To examine the impact of pre-allocated memory on access performance, we investigate the three redundancy schemes under four cases: *Case-1={8.0, 17.6, 19.2 percent}*, where a part of dataset is kept in memory for each redundancy scheme, and the ratio of dataset kept to the in-memory store

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:37:07 UTC from IEEE Xplore. Restrictions apply.

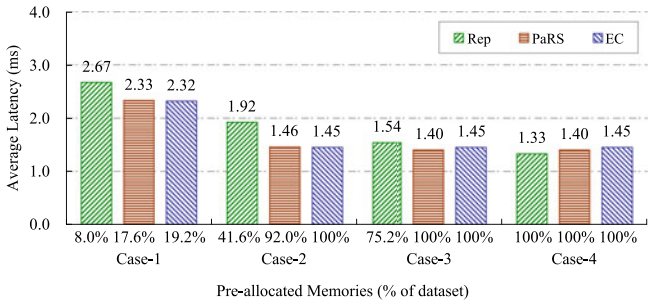

Fig. 11. Comparison of the three redundancy schemes under different pre-allocated-memory cases, with parameters k, r, and μ are set to 8, 2, and 0.8, respectively.

is 8.0, 17.6, and 19.2 percent for Rep, PaRS, and EC, respectively; *Case-2) = {41.6, 92.0, 100 percent}*, where a part of dataset is kept in memory for Rep and PaRS, all of dataset is stored in memory for EC; *Case-3) = {75.2,100,100 percent}*, where a part of dataset is kept in memory for Rep, all the datasets are placed to memory for PaRS and EC; and *Case-4) = {100, 100, 100 percent}*, where all the datasets are kept in memory for all redundancy schemes. Fig. 11 illustrates average access latencies of the three redundancy schemes in the aforementioned four cases.

From Fig. 11, we observe two performance trends. First, the access performance is improved along with the increasing pre-allocated memory for each redundancy scheme, because a big pre-allocated memory keeps a large number of in-memory data blocks, quickly serving user requests. Second, when the entire dataset is stored in memory, the access performance of Rep is higher than that of PaRS, and the access performance of PaRS is higher than that of EC. This trend is reasonable, because Rep supports higher access parallelism than PaRS, and PaRS also exhibits higher access parallelism than EC. In particular, in the limited-memory case, PaRS improves access latency by 15.1 to 31.5 percent compared against the Rep scheme.

4.2.3 Different Numbers of Data Blocks—k

A redundancy group in PaRS consists of k logical data blocks. To examine the sensitivity of PaRS to parameter k, we vary the k value in a PaRS-enabled in-memory store.

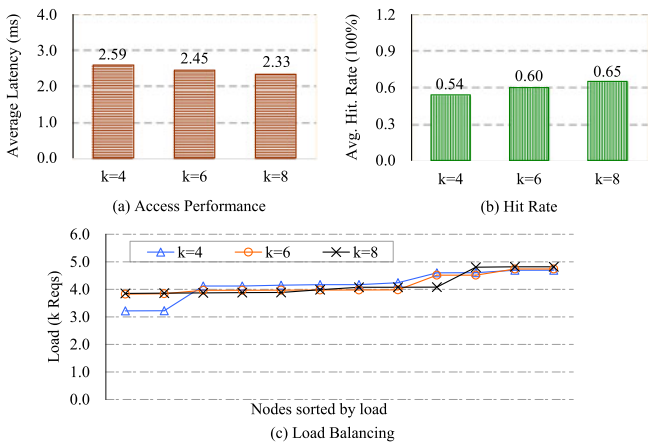

Fig. 12. Impacts of the parameter k on both access performance, hit rate, and load balancing of a PaRS-enabled in-memory store, with k = {4, 6, 8}, and parameters r, l, and μ are set to 2, 10 blocks, and 0.8, respectively.

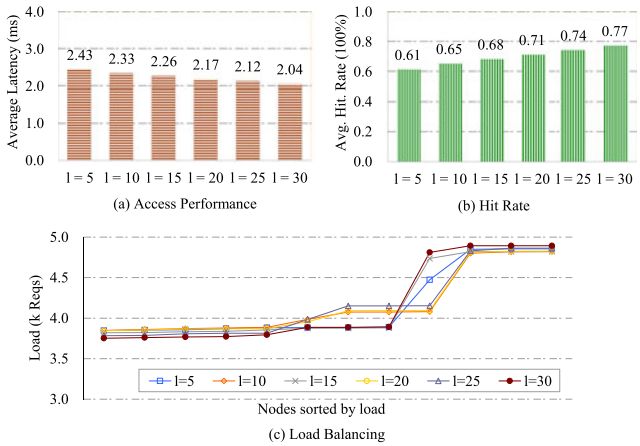

Fig. 13. Impacts of queue length l on access performance, hit rate, and load balancing of PaRS, with l = {5, 10, 15, 20, 25, 30}. Parameters k, r, and μ are set to 8, 2, and 0.8, respectively.

Fig. 12 plots the access performance, hit rate, and load balancing of the PaRS-enabled in-memory store with respect to different k, with k = {4, 6, 8}.

Fig. 12a indicates that the user access performance of a PaRS-enabled in-memory store goes up with the increasing value of parameter k. The reason lies in the fact that high memory efficiency—attributed by the k value—leads to a large number of data blocks stored the in-memory store. The memory efficiency is approximately k/(1.1 × k + r) when the percentage of popular data is about 10 percent.

We learn from Fig. 12b that the hit rate of the PaRS-enabled in-memory store rises when the value of k increases, because the in-memory store can hold more data blocks when the k value is larger.

Fig. 12c reveals that the imbalance metric λ is 12.7, 13.8, and 15.6 percent when parameter k is set to 4, 6, and 8, respectively. That is, parameter k has a slight impact on the load balancing of PaRS-enabled in-memory stores. The reason is that PaRS can dynamically adjust load among involved nodes as follows: (i) PaRS replicates a popular data block to another light-load node; and (ii) PaRS replaces an unpopular data block by inserting a newly-requested data block to a lightly-loaded node.

4.2.4 Different Lengths of Queue—l

Recall that each node maintains a queue for new data blocks in its local memory in a PaRS-enabled in-memory store. The queue performs as a bridge between disks and memories. In-queue data blocks not only serve user requests, but also act as candidate blocks to replace unpopular in-memory blocks. To evaluate the impact of queue length l on access performance, hit rate, and load balancing of PaRS, we conduct a set of tests by varying the values of parameter l, with l = {5, 10, 15, 20, 25, 30}. Fig. 13 depicts the experimental results with respect to various queue lengths.

We observe from Fig. 13a that the average access latency of the PaRS-enabled in-memory store decreases when the queue length is enlarged. The reason is two-fold: (i) more data blocks are kept in a queue in case of a longer queue; and (ii) more in-queue data blocks directly serve missed user requests, potentially reducing the probability of retrieving requested blocks from disks. In other words, the

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:37:07 UTC from IEEE Xplore. Restrictions apply.

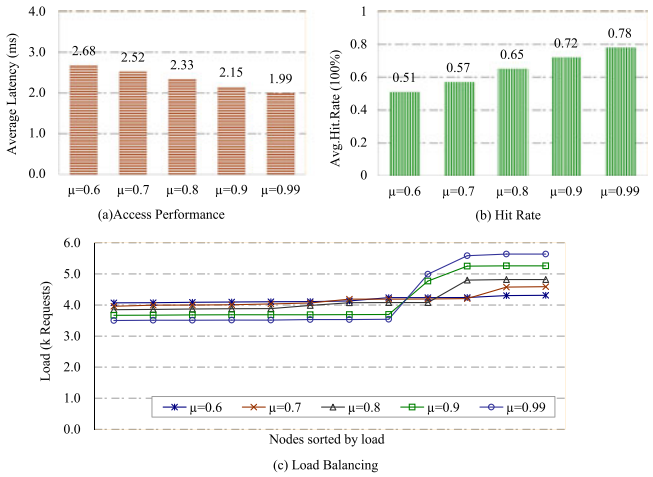

Fig. 14. Impacts of the Zipf parameter μ on access performance, hit rate, and load balancing of PaRS, with μ = {0.6, 0.7, 0.8, 0.9, 0.99}. Parameters k, r, and l are set to 8, 2, and 10 blocks, respectively.

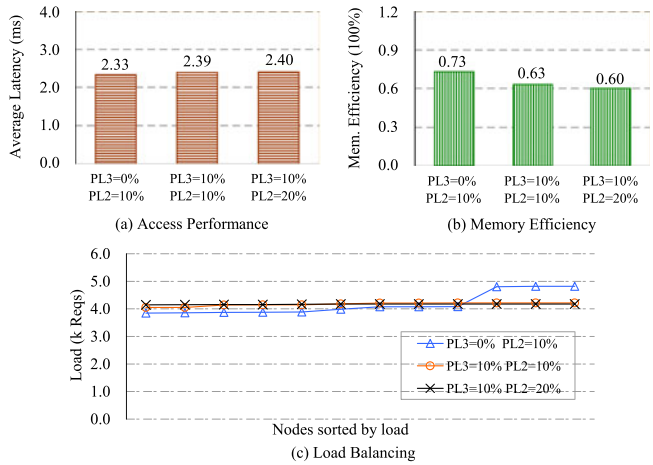

Fig. 15. Impacts of the number of popularity levels on both access performance, memory efficiency, and load balancing of a PaRS-enabled in-memory store, with {'PL-3 = 0 PL-2 = 10', 'PL-3 = 10 PL-2 = 20', and 'PL-3 = 10 PL-2 = 20 percent'}, and parameters k, r, l, and μ are set to 8, 2, 10 blocks, and 0.8, respectively.

new-data-block queue can be envisioned as an extension of the in-memory store.

Fig. 13b reveals that the hit rate of the PaRS-enabled in-memory store is enhanced with the increasing queue length, because more in-queue data blocks can serve more user missed requests. Furthermore, the in-queue blocks can serve subsequent user requests owing to temporal locality after they are migrated to the in-memory store.

Fig. 13c shows that the queue length imposes slight effect on load balancing of the PaRS-enabled in-memory store, where the imbalance metric λ is measured as 16.5, 15.6, 15.7, 15.8, 16.7, and 17.4 percent when the value of l is set to 5, 10, 15, 20, 25, and 30 blocks, respectively. The reason can be described from two angles. (i) The queue length merely affects the probability that 'an in-queue data block is requested by a missed request'. In other words, the probability is high when the queue is long, and vice versa. (ii) No matter the requested data block is an in-queue or on-disk data block, its access pattern is relied on the characteristic of application workload after it is moved to the in-memory store. In other words, the queue length is independent of the distribution of newly-fetched data blocks.

4.2.5 Different Distribution Parameter—μ

Zipf parameter μ captures the popularity distribution of a workload. A large value of parameter μ means highly skewed popularity. To examine the sensitivity of PaRS to parameter μ, we conduct a set of experiments by varying values of parameter μ, with μ = {0.6, 0.7, 0.8, 0.9, 0.99}. Fig. 14 shows the access performance, hit rate, and load balancing with respect to Zipf parameter μ.

We observe from Fig. 14a that the access performance of the PaRS-enabled in-memory store is improved when parameter μ increases. With the increasing value of parameter μ, more accesses are issued to a small portion of data blocks (i.e., hot data blocks), implying that an increasing number of subsequent requests are directly served by the hot data blocks after being fetched by PaRS into the in-memory store. In this way, the average latency per request is reduced.

Fig. 14b reveals that the hit rate of the PaRS-enabled in-memory store increases along with the increasing value of parameter μ. This is because in the PaRS case, a portion of in-memory data blocks become hot data, which answer a large number of subsequent requests due to a high popularity skewness.

From Fig. 14c, we observe that a large value of parameter μ leads to high load imbalance. In particular, the imbalance metric λ is 3.5, 10.1, 15.6, 26.2, and 35.3 percent when parameter μ is set to 0.6, 0.7, 0.8, 0.9, and 0.99, respectively. The reason is that only an extra replica is given to data blocks of *PL-2* regardless the distribution parameter μ and; thus, the load discrepancy among different nodes becomes more pronounced when more accesses are issued to a smaller portion of data blocks. This observation implies that it is indispensable to re-configure the number of popularity levels according to the popularity skewness of dataset (e.g., 4 popularity levels for dataset with '$\mu \geq 0.9$').

4.2.6 Different Levels of Popularity—PL

As to the case of 3 popularity levels, PaRS assigns single-fault-tolerant PL-1 to a majority of warm data blocks, and assigns double-replication-factor PL-2 to hot data blocks to improve their access parallelisms. When the number of popularity levels goes up, taking 4 popularity levels {i.e., PL-0, . . . ,PL-3} as an example, double-replication-factor PL-2 and triple-replication-factor PL-3 are assigned to extremely-hot and relatively-hot data blocks, respectively.

To examine the sensitivity of PaRS to number of popularity levels, we adopt three cases of popularity levels {'PL-3 = 0 PL-2 = 10', 'PL-3 = 10 PL-2 = 20', and 'PL-3 = 10 PL-2 = 20 percent'}. Fig. 15 plots the access performance, memory efficiency, and load balancing of the PaRS-enabled in-memory store with respect to different popularity levels. Here 'PL-3 = 0 PL-2 = 10 percent' is equivalent to the case of 3 popularity levels in Section 3.3.1

Fig. 15a indicates that the access latency of the PaRS-enabled in-memory store increases with the increasing number of popularity levels and percentage of popular blocks. The reason lies in the fact that more popularity levels and higher percentage of popular blocks lead to fewer different blocks stored the in-memory store.

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:37:07 UTC from IEEE Xplore. Restrictions apply.

TABLE 4
Comparison of Redundancy Strategies for Differentiated Fault-Tolerance Services

Scheme	1st Service	2nd Service
Cocytus [11]	replication for small-sized data	erasure coding for large-sized data
Scarlett [14]	more-replicas for higher-access-popularity data	less-replicas for lower-access-popularity data
Facebook [19]	more-replicas for small and frequently-accessed data	less-replicas for large and infrequently-accessed data
ARM [21]	replication for popular files	erasure codes for unpopular files
HACFS [37]	fast erasure coding for frequently-accessed data	compact erasure coding for less-frequently-accessed data
Ring [38]	replication for hot data	erasure codes for cold data
Our PaRS	high-replication-factor hybrid-coding for high-frequency data blocks	low-replication-factor hybrid-coding for low-frequency data blocks

Fig. 15b shows that the memory efficiency of the PaRS-enabled in-memory store decreases with the increasing number of popularity levels and percentage of popular blocks, because there is a trade-off between memory efficiency and replication factor.

Fig. 15c reveals that the load-balancing performance of the PaRS-enabled in-memory store improves when both the number of popularity levels and percentage of popular blocks increase. The reason is that more replicas of popular blocks are replicated to light-load nodes.

4.3 A Summary of Observations

We summarize the important observations drawn from Figs. 10, 12, 13, and 14 as follows.

- PaRS-enabled in-memory stores demonstrate similar access performance and memory efficiency compared to EC-based counterparts. In the limited-memory case, PaRS is superior to the Rep scheme in terms of access performance and memory efficiency.
- PaRS enables in-memory stores to accomplish comparable load balancing to Rep; PaRS delivers better load balancing than EC.
- PaRS-enabled in-memory stores exhibit similar trends in both access performance and hit rate with respect to number k of logical data blocks in a redundancy group, length l of queues in nodes, and distribution parameter μ. Specifically, the access performance and hit rate of a PaRS-enabled in-memory store are improved with increasing value of parameter k, l, or μ.

5 Related Work

Hybrid coding and popularity skewness are relevant to PaRS and therefore, we conduct a literature survey in the areas of both differentiated fault-tolerance services and in-memory load balancing techniques.

5.1 Differentiated Fault-Tolerance Services

Differentiated services or *DiffServ* is a computer networking architecture that specifies a scalable mechanism for managing network traffic and providing quality of service (QoS) on IP networks [36]. DiffServ can, for example, offer low-latency to critical network transmissions while providing best-effort services to non-critical network traffic.

Similar to DiffServ offering differentiated transmission services in networks, a few redundancy-based stores are focused on achieving differentiated fault-tolerance services for different data.

Study Case 1. Scattered metadata in an in-memory KV-store may incur a large number of coding operations and updates. To address such a challenge, Cocytus leverages replication for small-sized and scattered data (e.g., metadata and key), while applying erasure coding to relatively large data (e.g., value) [11]. Thus, Cocytus enables differentiated redundancy services to data of different sizes.

Study Case 2. Analysis of job logs from a large production cluster suggests that nodes storing popular blocks become performance bottlenecks. To address this issue, Scarlett adopts a redundancy scheme called *Selective Replication*, which replicates blocks based on access popularity [14]. That is, Scarlett accomplishes differentiated replication services for blocks with diverse popularity.

Study Case 3. Facebook leverages *memcached* as a building block to construct a distributed key-value store [19], where a replicated memcache pool is configured for small and frequently accessed data, and a regional memcache pool is used for large and infrequently accessed data. Depending on both data size and access popularity, Facebook's key-value store offers differentiated replication services for data.

Study Case 4. Owing to different access rates of data files, the replication mechanism maintained by HDFS for every data file leads to detrimental effects on the performance. To solve this problem, an adaptive replication management system or *ARM* employs replication and erasure coding for popular and unpopular files, respectively [21]. ARM implements differentiated fault-tolerance services according to access potentials.

Study Case 5. HACFS is a new erasure-coded storage system that utilizes two different erasure codes from the same code family to dynamically adapts to workload changes [37]. In particular, HACFS uses the fast code to encode frequently accessed data, and applies the compact code to encode less frequently accessed data. In other words, HACFS provides differentiated erasure-coding services in the light of access frequencies.

Study Case 6. Ring is an in-memory key-value store using a novel redundancy scheme called SRS encoding, which combines hash key distributions of replication and erasure coding schemes [38]. With SRS codes in place, Ring can support changing resilience. That is, Ring places hot data in high-performance replicated storage while keeping cold data in low-overhead erasure-coded storage.

Table 4 summarizes that Cocytus leverages replication and erasure coding for small- and large-sized data, respectively;

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:37:07 UTC from IEEE Xplore. Restrictions apply.

Scarlett applies more- and less-replicas to higher- and lower-access-popularity data blocks, respectively; Facebook employs more- and less-replicas to 'small and frequently-accessed' and 'large and infrequently-accessed' data, respectively; ARM and Ring adopt replication and erasure codes for popular and unpopular files, respectively; HACFS uses the fast code and the compact code to encode frequently- and less-frequently-accessed data, respectively.

From Table 4, we learn that all the above-described redundancy schemes and systems assign a single redundancy scheme (i.e., replication or erasure coding) to a specific redundancy group. That is, the replication factor of data blocks in the same redundancy group is identical. On the contrary, PaRS incorporates both erasure coding and replication schemes (i.e., a variant of mirrored RAID-5) into a redundancy group, thus the replication factor of different data blocks in a redundancy group may be different. In this way, PaRS allows in-memory datasets to support adaptive memory efficiency and access parallelism. On one hand, PaRS deploys a high-replication-factor scheme to high-access-frequency data blocks, improving access parallelisms; on the other hand, PaRS invokes a low-replication-factor scheme to deal with low-access-frequency data blocks, achieving high memory efficiency.

5.2 In-Memory Load Balancing

Recall that access popularity skew is a common characteristic of workloads in cluster caches (see Section 1.1). When workload is concentrated on a small number of data (e.g., keys), an issue of load imbalance occurs. Load imbalance across servers leads to two disadvantages: i) low resource utilization (e.g., CPU, memory) owning to over-provisioning of resources necessary to withstand peak workloads [10], [17]; ii) degradation of access performance in terms of latency and throughput due to hotspots in workloads [18].

Traditionally, there are two ways to improve load balancing. i) In *hotspot migration*, systems dynamically move data from busy nodes to less busy nodes to alleviate hefty load [17]. ii) As for *hotspot replication*, systems replicate two or more replicas for hotspots, and direct accesses to the least-load nodes keeping replicas [14].

Zhang et al. proposed an load-balancing network middle-box to automatically detect hotspots and to balance load across memcached servers through request redirection and replication. The key-hotspot-redirection strategy forwards a portion of requests to high-powered servers; the key replication scheme prevents overload [17].

Cheng et al. developed a design of an in-memory caching tier called *MBal*, which employs the key replication and data migration techniques to improve cache performance and load balancing. MBal accomplishes adaptive load balancing by adopting both server-local and cross-server data migration to re-distribute and mitigate hotspots [18].

Hong et al. designed an augmented memcached variant—*SPORE*—by using a popularity-based key replication mechanism to proactively monitor and replicate hot objects across multiple cache servers, thereby mitigating the load imbalance effects [39].

Facebook adopts a key replication system to mitigate high request rates in a cache pool, thus reducing the load per server. Furthermore, a system named *Cold Cluster Warmup* accomplishes across-cluster data replication by allowing clients in a 'cold cluster' to retrieve data from a 'warm cluster' [19].

EC-Cache applies erasure coding and object splitting to individual large objects in the range of tens to hundreds of megabytes during writes [10]. Compared to selective replications, online erasure coding enables EC-Cache to achieve better load balancing, because read requests are dispatched across both data and parity splits when an object is read, whereas the object can not be accessed in parallel in the case of object replication.

PaRS simultaneously deploys two load-balancing mechanisms. First, PaRS judiciously adjusts the replication factor of a data block according to access popularity. Specifically, PaRS replicates one or more replicas of high-popularity blocks to lightly-loaded nodes; PaRS removes one or more replicas of unpopular blocks from heavily-loaded ones. Second, PaRS re-stripes a redundancy group by replacing a rarely-accessed data block with a newly-requested data block. Such a re-striping approach can be envisioned as a variant of 'hot migration', since it not only delivers newly-requested data blocks to a lightly-loaded node, but also triggers parity updates for redundancy groups.

6 Further Discussion

When redundancy parameter k in (k + r, k) RS codes is set to 1, a redundancy group in PaRS is equivalent to a replication group, and PaRS is degraded to the existing selective replication scheme [14], [19].

As described earlier, erasure coding exhibits low access parallelism (i.e., 1) due to single-copy source data block. In fact, with its fault-tolerance feature, (k + r, k) RS coding can decode a source data block from any other k blocks. Let's take (4, 3) RS coding as an example, apart from reading data block D_1 directly, D_1 can be indirectly retrieved by decoding k = 3 other blocks $\{\mathrm{D}_2, \mathrm{D}_3, \mathrm{P}_1\}$ using $\mathrm{D}_1 = \mathrm{D}_2 \oplus \mathrm{D}_3 \oplus \mathrm{P}_1$. From this aspect, the access parallelism of erasure coding is higher than 1.

The idea of hybrid-encoding has been adopted by Cocytus [11], Atlas [40], and LH*$_{RS}$ [41], where erasure coding is applied to objects' value, and replication is used for objects' key and meta-data. Unlike these schemes, our PaRS advocates hybrid-encoding (e.g., erasure-coding & replication) for user data (i.e., value of object in KV stores) in in-memory stores.

7 Conclusion and Future Work

In this paper, we proposed a popularity-aware redundancy scheme—*PaRS*—for in-memory datasets to address the issue of popularity skewness. PaRS relies on the re-stripe or replication mechanism to adjust the redundancy factor and access parallelism according to workload popularity growth. With PaRS in place, a redundancy layout with high memory efficiency is deployed to low-access-frequency data, whereas a redundancy layout exhibiting high access parallelism is used to highly-accessed data.

The comparative experiments show that, compared to redundancy schemes powered by either replication or erasure coding, PaRS provides in-memory data blocks with an adjustable configuration between memory efficiency

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:37:07 UTC from IEEE Xplore. Restrictions apply.

and access parallelism. In particular, experimental results illustrate that PaRS-enabled in-memory stores accomplish higher access performance and memory efficiency than replication-based ones in the limited-memory case, and better load balancing than erasure-coding-based ones. Additionally, PaRS exhibits enhanced access performance and hit rate when number k of logical data blocks, length l of queues, or distribution parameter μ grows up.

In our future study, we will further investigate PaRS in the arena of heterogeneous in-memory stores. Specifically, we plan to measure the load of nodes by taking into account access amount, request size, and configuration heterogeneity. Furthermore, we plan to conduct comparative evaluations using read-write workloads (e.g., real-world workloads obtained from SPC [42].

Acknowledgments

This work is supported by the National Science Foundation of China under Grants 61572209, 61872413, and 61702004. Xiao Qin's work is supported by the U.S. National Science Foundation under Grants CCF-0845257, CNS-0917137, and CCF-0742187, and the 111 Project under Grant B07038. We would like to thank the anonymous reviewers for their comments.

References

[1] H. Mühleisen, R. Gonçalves, and M. Kersten, "Peak performance: Remote memory revisited," in *Proc. 9th Int. Workshop Data Manage. New Hardware*, 2013, Art. no. 9.

[2] T. Suganuma, A. Koseki, K. Ishizaki, Y. Ueda, K. Mizuno, D. Silva, H. Komatsu, and T. Nakatani, "Distributed and fault-tolerant execution framework for transaction processing," in *Proc. 4th Annu. Int. Conf. Syst. Storage*, 2011, pp. 2:1–2:12.

[3] A. K. Goel, J. Pound, N. Auch, P. Bumbulis, S. MacLean, F. Färber, F. Gropengiesser, C. Mathis, T. Bodner, and W. Lehner,"Towards scalable real-time analytics: An architecture for scale-out of OLxP workloads," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1716–1727, 2015.

[4] KLab Inc., "Repcached [Online]," (2017, Dec.). [Online]. Available: http://repcached.lab.klab.org

[5] Salvatore Sanfilippo, "Redis [Online]," (2017, Dec.). [Online]. Available: https://www.redis.io

[6] D. Wind, *Instant Effective Caching with Ehcache*. Birmingham, U.K.: Packt Publishing Ltd, 2013.

[7] A. Seovic, M. Falco, and P. Peralta, *Oracle Coherence 3.5*. Birmingham, U.K.: Packt Publishing Ltd, 2010.

[8] M. E. Taylor, et al., *WebSphere MQ V7.1 and V7.5 Features and Enhancements*. Armonk, NY, USA: IBM Redbooks, 2013.

[9] H. Weatherspoon and J. D. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," in *Proc. 1st Int. Workshop Peer-to-Peer Syst.*, 2002, pp. 328–337.

[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. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 401–417.

[11] H. Zhang, M. Dong, and H. Chen, "Efficient and available in-memory KV-store with hybrid erasure coding and replication," in *Proc. 14th Usenix Conf. File Storage Technol.*, 2016, pp. 167–180.

[12] 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. 10th ACM Int. Syst. Storage Conf.*, 2017, pp. 14:1–14:12.

[13] J. L. Hafner, "WEAVER codes: Highly fault tolerant erasure codes for storage systems," in *Proc. 4th Conf. USENIX Conf. File Storage Technol.*, 2005, pp. 211–224.

[14] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett: Coping with skewed content popularity in MapReduce clusters," in *Proc. 6th Conf. Comput. Syst.*, 2011, pp. 287–300.

[15] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li, "An analysis of Facebook photo caching," in *Proc. 24th ACM Symp. Operating Syst. Principles*, 2013, pp. 167–181.

[16] Q. Huang, H. Gudmundsdottir, Y. Vigfusson, D. A. Freedman, K. Birman, and R. van Renesse, "Characterizing load imbalance in real-world networked caches," in *Proc. 13th ACM Workshop Hot Topics Netw.*, 2014, pp. 8:1–8:7.

[17] W. Zhang, T. Wood, H. H. Huang, J. Hwang, and K. K. Ramakrishnan, "Load balancing of heterogeneous workloads in memcached clusters," in *Proc. 9th Int. Workshop Feedback Comput.*, 2014, pp. 17–24.

[18] Y. Cheng, A. Gupta, and A. R. Butt, "An in-memory object caching framework with adaptive load balancing," in *Proc. 10th Eur. Conf. Comput. Syst.*, 2015, pp. 4:1–4:16.

[19] 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. 10th USENIX Conf. Netw. Syst. Des. Implementation*, 2013, pp. 385–398.

[20] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," *ACM SIGMETRICS Perform. Eval. Rev.*, 2012, pp. 53–64.

[21] D.-M. Bui, S. Hussain, E.-N. Huh, and S. Lee, "Adaptive replication management in HDFS based on supervised learning," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 6, pp. 1369–1382, Jun. 2016.

[22] D. Ford, F. Labelle, F. Popovici, M. Stokely, V. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in globally distributed storage systems," in *Proc. 9th USENIX Symp. Operating Syst. Des. Implementation*, 2010, pp. 61–74.

[23] 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. USENIX Conf. Annu. Tech. Conf.*, 2012, pp. 15–26.

[24] M. S. Manasse, C. A. Thekkath, and A. Silverberg, "A reed-solomon code for disk storage, and efficient recovery computations for erasure-coded disk storage," in *Proc. 6th Workshop Distrib. Data Structures (WDAS'2004)*, pp. 51–61, 2004.

[25] I. Reed and G. Solomon, "Polynomial codes over certain finite fields," *J. Soc. Ind. Appl. Math.*, vol. 8, no. 2, pp. 300–304, 1960.

[26] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1988, pp. 109–116.

[27] J. Plank, J. Luo, C. Schuman, L. Xu, and Z. Wilcox-O'Hearn, "A performance evaluation and examination of open-source erasure coding libraries for storage," in *Proc. 7th Conf. File Storage Technol.*, 2009, pp. 253–265.

[28] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "PACMan: Coordinated memory caching for parallel jobs," in *Proc. 9th USENIX Conf. Netw. Syst. Des. Implementation*, 2012, pp. 20–33.

[29] M. Zahran, "Cache replacement policy revisited," in *Proc. 6th Annu. Workshop Duplicating, Deconstructing, and Debunking (WDDD) held in conjunction with the Int. Symp. Comput. Arch. (ISCA)*, 2007, pp. 1–8.

[30] R. Pagh and F. F. Rodler,"Cuckoo hashing," *J. Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[31] J. Jang, Y. Cho, J. Jung, and G. Jeon, "Enhancing lookup performance of key-value stores using cuckoo hashing," in *Proc. Res. Adaptive Convergent Syst.*, 2013, pp. 487–489.

[32] P. Chen, E. Lee, G. Gibson, R. Katz, and D. Patterson, "RAID: High-performance, reliable secondary storage," *ACM Comput. Surv.*, vol. 26, no. 2, pp. 145–185, 1994.

[33] R. Hariharan and N. Sun, "Workload characterization of specweb 2005," in *Proc. SPEC Benchmark Workshop, in conjunction with the Standard Performance Evaluation Corporation (SPEC)*, pp. 10–18, 2006.

[34] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 143–154.

[35] Intel, "Intel storage acceleration library (open source version)," (2017, Dec.). [Online]. Available: https://goo.gl/zkVl4N

[36] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "An architecture for differentiated services," Internet Society, Reston, Virginia, Tech. Rep. RFC 2475, 1998.

[37] M. Xia, M. Saxena, M. Blaum, and D. Pease, "A tale of two erasure codes in HDFS," in *Proc. 13th Usenix Conf. File Storage Technol.*, 2015, pp. 213–226.

[38] K. Taranov, G. Alonso, and T. Hoefler, "Fast and strongly-consistent per-item resilience in key-value stores," in *Proc. 13th Eur. Conf. Comput. Syst.*, 2018, pp. 39–53.

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:37:07 UTC from IEEE Xplore. Restrictions apply.

[39] Y.-J. Hong and M. Thottethodi, "Understanding and mitigating the impact of load imbalance in the memory caching tier," in *Proc. 4th Annu. Symp. Cloud Comput.*, 2013, pp. 13:1–13:17.
[40] C. Lai, S. Jiang, L. Yang, S. Lin, G. Sun, Z. Hou, C. Cui, and J. Cong, "Atlas: Baidu's key-value storage system for cloud data," in *Proc. 31st Symp. Mass Storage Syst. Technol.*, 2015, pp. 1–14.
[41] D. Cieslicki, S. Schackeler, and T. Schwarz, "Efficient updates in highly available distributed random access memory," in *Proc. 12th Int. Conf. Parallel Distrib. Syst.*, 2006, pp. 9–17.
[42] S. P. Council, "I/O and Search Engine I/O. UMass Trace Repository," 2007. [Online]. Available: http://traces.cs.umass.edu/



Panping Zhou received the BS degree in computer science and technology from Central South University (CSU), China, in 2016. She is now working toward the MS degree at the Huazhong University of Science and Technology (HUST). Her research interests include erasure-coded memory systems and file systems.



Jianzhong Huang received the PhD degree in computer architecture, in 2005 and completed the post doctoral research in information engineering, in 2007 from the Huazhong University of Science and Technology (HUST). He is currently an associate professor with the Wuhan National Laboratory for Optoelectronics, HUST. His research interests include storage architecture, dependable storage systems, and in-memory stores. He is a member of China Computer Federation (CCF).



Xiao Qin (S'00-M'04-SM'09) received the BS and MS degrees in computer science from the HUST, China, and the PhD degree in computer science from the University of Nebraska-Lincoln, Lincoln, in 1992, 1999, and 2004, respectively. Currently, he is a professor with the Department of Computer Science and Software Engineering, Auburn University. His research interests include parallel and distributed systems, storage systems, fault tolerance, real-time systems, and performance evaluation. He received the US NSF Computing Processes and Artifacts Award and the NSF Computer System Research Award in 2007 and the NSF CAREER Award in 2009. He is a senior member of the IEEE.



Changsheng Xie received the BS and MS degrees in computer science both from the HUST, China, in 1982 and 1988, respectively. He is currently a professor with the Department of Computer Engineering, HUST. He is also the director of the Data Storage Systems Laboratory of HUST. His research interests include computer architecture, disk I/O system, and networked storage system. He is the vice chair of the expert committee of Storage Networking Industry Association (SNIA), China.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:37:07 UTC from IEEE Xplore. Restrictions apply.