WPS: A Workload-aware Placement Scheme for Erasure-Coded In-Memory Stores

Shuang Wang[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*
{*wangshuang, hjzh, caoqiang, cs_xie*}*@hust.edu.cn, xqin@auburn.edu.* **Corresponding Author*

***Abstract*—Data-intensive applications are increasingly depending on in-memory stores to meet high-I/O-performance requirements. To be resilient to server failures and in turn achieve high availability, both replication and erasure codes are introduced to in-memory stores. Since erasure codes have an advantage of memory efficiency over replication, we focus our work on erasure-coded in-memory stores and investigate placement schemes to address the issue of workload fluctuation.**

To mitigate the I/O imbalanced incurred by workload skew and maximize the utilization of all nodes, we proposed a Workload-aware Placement Scheme called WPS for Reed-Solomon-coded in-memory stores. WPS accomplishes balanced I/Os as follows: it divides in-memory data blocks into multiple groups based on access characteristics (e.g., popularity), and classifies all nodes into several groups according to nodes' access performance (e.g., indicated by available bandwidth), and places or migrates high-access-popularity in-memory data blocks to high-performance nodes without violating the essential principle of fault tolerance. The comparative experiments indicate that WPS can significantly improve load balancing for RS-coded in-memory stores exhibiting workload popularity skew; meanwhile, WPS achieves comparable mean, median, and tail latencies relative to two candidate placement schemes.

***Keywords*-In-memory store, Erasure codes, Workload skew, Data placement, Balanced I/Os;**

I. Introduction

A. Motivations

Nowadays, big data facing '4V' challenges (i.e., Volume, Variety, Velocity, and Veracity) induces a growing number of representative applications [39] such as on-line transaction processing (OLTP), high-performance computing (HPC) and the like. OLTP applications are required to accomplish real-time processing during handling data with rapidly-expanding size and heavily-loaded transaction amount. HPC applications are focused on time efficiency, for example, it is desirable for scientific computing to produce intermediate results from huge raw data sets as soon as possible. However, if data required by such applications is read from disks, then disk access will become a performance bottleneck owing to high disk I/O latency. To address this issue, some effective in-memory approaches emerge. In-memory NoSQL databases like Redis [29], MemepiC [5], MongoDB [16] are designed to facilitate low-latency data accesses; whereas in-memory data processing engines like Spark [37], HANA [32], Exalytics [10] process large-volume data at high throughput.

Fault tolerance is imperative for in-memory systems. The volatility of RAM makes in-memory stores fail to resist data losses in the presence of power outages, program crashes, server reboots, etc. Therefore, it is an important issue to guarantee availability of in-memory data, and there exist three general solutions: (1) Persistent storage, e.g, data is re-cached into memory from disks when it cannot be retrieved from memory; (2) Replication mechanism, replication or triplication enables in-memory objects to tolerate single or double failures, respectively; (3) Erasure codes, failed objects can be reconstructed from enough surviving in-memory objects using erasure decoding computations. Different from the first solution, both replication-based and erasure-coded solutions can eliminate disk I/O overheads.

Remote memory accesses over high-speed network achieve far higher read performance than local disk read [21]. Therefore, it is preferred to accomplish fault tolerance by applying redundancy solutions to distributed memory rather than resorting to disk persistence.

Erasure-coded stores suffer from larger network traffic than replication-based ones in the case of data recovery. Take (k+r=6, k=4) Reed-Solomon-codes as an example, k=4 surviving blocks are required to reconstruct a failed block; whereas only one replica block is transferred for the triplication case (i.e., k=1, r=2). Although erasure codes incur higher recovery traffic than replication, such traffic discrepancy can be negligible under existing high-speed networks.

On the other hand, since memory is a constrained and expensive resource, memory efficiency (ratio of user data to the total of user data plus redundancy data [11]) is an important performance metric for redundancy solutions. Compared to replication, erasure codes achieve higher memory efficiency. In a word, it is a good alternative to apply erasure codes to in-memory stores to meet the fault tolerance demand.

Existing reliable in-memory stores mainly investigate the following three aspects [27][40][36]: (1) achieving high throughput and low latency in normal or degraded modes; (2) accomplishing fast recovery in failed modes; and (3) improving network load balancing among nodes. These solutions are focused on pure in-memory data objects, without considering on-disk data. In fact, most data processing engines retrieve original data from disks, and in-memory stores inevitably interact with underlying persistent storage, e.g., on-disk data

978-1-5386-3486-8/17/$31.00 ©2017 IEEE
Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:06:25 UTC from IEEE Xplore. Restrictions apply.

objects should be read to memories in the cases of initialization and memory/cache miss.

Workload popularity skew is a common challenge in data-intensive clusters [2][17]. That is, A small fraction of the data is highly popular, while the rest is less frequently accessed. For example, the top 5% of the blocks are seven times more popular than the bottom three-quarters in the Facebook cluster. It is revealed that data popularity follows a Zipf-like skewed pattern [27]. If popularity skew is not considered, erasure-coded in-memory stores may suffer from imbalanced loads, and load imbalance requires over-provisioning the cluster to accommodate the peaks in the load distribution, thus leading to low resource utilization. Therefore, it is vital to design balanced in-memory systems by taking workload popularity skew into account. In this paper, we proposed a workload-aware placement scheme - **WPS** - for Reed-Solomon-coded in-memory stores. WPS can accomplish balanced I/Os by exploiting workload's access popularity and node's current access performance. In particular, with WPS in place, high-access-popularity in-memory data blocks are placed or migrated to high-available-bandwidth nodes without violating the essential fault-tolerance principle.

B. Contributions

The contributions of this study are summarized as follows:

- We proposed a workload-aware placement scheme for Reed-Solomon-coded in-memory stores, which accomplish balanced I/Os by leveraging block's access frequency and node's available bandwidth. Especially, WPS places or migrates high-access-frequency in-memory data blocks to high-available-bandwidth nodes;
- We quantitatively assess both block's access popularity and node's access performance. In particular, in-memory data blocks are separated into groups of different heat levels in accordance with access frequencies, and all nodes are classified into groups of different bandwidth levels based on available bandwidths;
- We implement and evaluate WPS and two candidate placement schemes in a real-world erasure-coded cluster. Experimental results shows that WPS exhibits the best load balancing, and achieves similar mean, median, and tail latencies among the three schemes.

C. Roadmap

The rest of the paper is organized as follows. We introduce the background in Section II. The overall design of WPS is described in Section III. We present quantitative evaluation in Section IV. Section V outlines prior studies related to in-memory stores. We conclude this study in Section VI.

II. Background

A. Erasure Codes for In-memory Store

Erasure codes are widely applied to storage clusters deployed in large-scale data centers [4][9][13]. Among various erasure codes, Reed-Solomon (i.e., *RS*) codes becomes the most popular choice owing to RS codes' optimal storage efficiency as well as high level of fault tolerance [34].

In particular, (k+r,k) RS codes encode source data with a k×(k+r) *Generator Matrix*, which involves a k×k *Identity Matrix* and a k×r *Redundancy Matrix* [19]. The source data are embedded in encoded data thanks to the *Identity Matrix*. The RS encoding adopts the simple linear algebra; as shown in Fig. 1, parity strips are generated by multiplying k data strips with the k×r Redundancy Matrix. In Vandermonde Reed Solomon codes [28][25], the coefficient $\alpha_{j,i}$ is i^{j-1}, where $i \in \{1, 2, ..., k\}$ and $j \in \{1, 2, ..., r\}$.

$$\begin{bmatrix} \alpha_{1,1} & \alpha_{1,2} & \cdots & \alpha_{1,k} \\ \alpha_{2,1} & \alpha_{2,2} & \cdots & \alpha_{2,k} \\ \vdots & \vdots & & \vdots \\ \alpha_{r,1} & \alpha_{r,2} & \cdots & \alpha_{r,k} \end{bmatrix} \times \begin{bmatrix} D_{row,1} \\ D_{row,2} \\ \vdots \\ D_{row,k} \end{bmatrix} = \begin{bmatrix} P_{row,1} \\ P_{row,2} \\ \vdots \\ P_{row,r} \end{bmatrix}$$

Transposed Redundancy Matrix — Data Block — Parity Block

Fig. 1. Generating r parity blocks in (k+r,k) RS codes

Both k data strips and r parity strips constitute a stripe [24]. Each strip can be partitioned into multiple blocks [26]. 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 associated parity blocks are exclusively placed to the main memories of $k+r$ storage nodes. In practice, each of the $k+r$ storage nodes may cache a data block in a stripe and a parity block in another stripe. That is, 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. As Maximum Distance Separable (i.e., *MDS*) codes, (k+r,k) RS codes have the property that the source data may be reconstructed when any r nodes are failed or powered off.

Object placement plays a significant role in the issue of data reliability. Take replica placement as an example, if two copies of an object are placed to a node, and the node is failed, then the object will be lost. To address this issue, a rack-aware replica placement policy is designed for triplication [3], where one replica is stored on one node in a local rack, another replica is placed on a node in a remote rack, and the last one is on a separate node in the same remote rack. When it comes to replication, each primary block and its replica block are stored on two distinct nodes.

To improve service performance and fault tolerance, several replication declustering schemes were developed. In chained declustering, the primary copies of data are distributed over disks like a chain, and the secondary copy of one data object is dispatched to the neighboring disk of the primary copy [12]. Group rotational declustering partitions all disks into several groups, and the number of groups equals to the number of data copies, and each group stores a complete copy of all data, and the data in the secondary group is distributed in a rotational

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:06:25 UTC from IEEE Xplore. Restrictions apply.

way [6]. Interleaved declustering distributes all primary copies over all disks, and splits the secondary copy of a data object into several pieces and stores them evenly over disks [8]. Shifted declustering is a variant of chained declustering, and it shifts each row of the data objects over all the disks in a circular fashion [41].

Most placement schemes are designed for on-disk data, that is, both data and its replicas are stored to disks. In respect of data location, existing replication declustering can be viewed as 'on-disk primary copy ↔ on-disk secondary copy' data layout; while object placement within in-memory stores should take into account 'on-disk primary copy ↔ in-memory secondary copy' data layout. In addition, as to (k+r,k) RS-coded in-memory stores, the copy of data object kept in memories is grouped into k data blocks, and the k data blocks are encoded to r parity blocks, and then the k+r encoded blocks are distributed among memories in k+r servers.

III. A Workload-aware Placement Scheme

The section describes an architecture of RS-coding-based in-memory stores (InMem-RS for short) and details a workload-aware placement scheme (i.e., WPS). WPS is incorporate to the InMem-RS to dynamically place or migrate high-access-popularity in-memory data blocks to high-performance nodes without compromising the fault tolerance of in-memory stripes.

A. Architecture of InMem-RS

As an in-memory store, InMem-RS manages all data objects in memory organized in RS-coded manner. Fig. 2 illustrates the InMem-RS architecture, which consists of three types of nodes: storage nodes, proxy nodes, and coordinator nodes. Each storage node allocates a memory region for in-memory store and has some local disks to ensure data persistence. The proxy node is an entry point for front-end clients to access data in storage nodes. The coordinator node manages the whole storage cluster, in particular, it maintains all meta-data (e.g., block distribution on disk and memory, block's access frequency, node's available bandwidth, stripe's logical mapping) and provides meta-data to answer to requests from the proxy node, and it constructs RS-coded stripes for in-memory store, and it monitors the health state of storage nodes (e.g., failure detection), and coordinates access requests in the presence of failures or power outages.

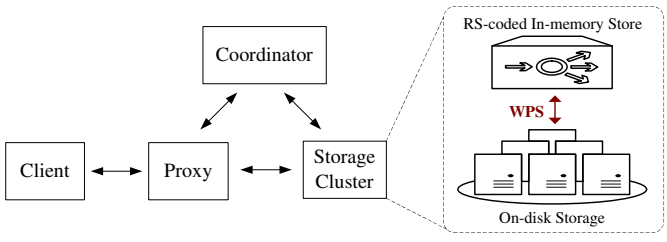


Fig. 2. An architecture of erasure-coding-based in-memory stores

The placement module (i.e., WPS module) is hosted at the coordinator node. The placement module manages the distribution and redistribution of blocks, including data blocks stored to memories, data blocks written to disks, and parity blocks kept to memories. The main goal of placement module in in-memory RS-coded stores is to maintain both in-memory and on-disk data without degrading user-access performance.

B. Challenges and Strategies

To guarantee high fault tolerance, an essential placement rule is that the original data blocks and associated parity blocks of a stripe must be placed to memories in k+r separate nodes in (k+r,k) RS-coded in-memory stores; meanwhile, an in-memory copy and an on-disk copy of each data block should not be stored to an identical node. As a workload-aware placement scheme, apart from fault tolerance, WPS should address the following three challenges.

Challenge 1: how to efficiently classify data blocks based on access popularity? It is intuitive to place high- and low-access popularity data to high- and low-performance nodes, respectively. In this case, two important issues should be addressed: (1) how to measure data access popularity?; and (2) how to determine the amount of high-access-popularity data to be migrated to high-performance nodes? If all data is only divided into two groups: the top 20% most frequently accessed blocks in a group and all remained blocks in another group, then almost 20% of data will be migrated, resulting in high migration traffic.

Challenge 2: how to dynamically quantify the available resource of nodes? The available bandwidth resource of each node always fluctuates along with the evolution of the whole storage system. If the available bandwidths of nodes are measured in a fine-granularity interval, and the migration of high-access-frequency data blocks are triggered when the measurement is completed, then the migration 'jitter' phenomenon will occur, thereby leading to unnecessary migration traffic. On the contrary, if the available bandwidths of nodes are quantitatively assessed in a coarse-granularity interval, then high-access-frequency data blocks will not be migrated to correct nodes during the interval, thus resulting in imbalanced bandwidth usage and skewed I/O accesses.

Challenge 3: how to make a good tradeoff between locality and load balance? High locality implies that most of user accesses are limited to a small portion of data. For example, the top 20% most frequently accessed blocks account for about 80% of all accesses [20]. In this way, locality leads to imbalanced I/O requests issued to the whole storage system. On the contrary, if high-popularity data is evenly redistributed among nodes to achieve balanced I/Os and high storage utilization, then such redistribution may weaken access locality and incur considerable migration overhead [14] [18].

We propose a workload-aware placement scheme for RS-coded in-memory stores by incorporating the following three strategies to address the above challenges: (1) in-memory data

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:06:25 UTC from IEEE Xplore. Restrictions apply.

blocks are separated into multiple groups (e.g., 5 groups) according to access frequencies, with each group has its heat level, and the amount of migrated data can be flexibly determined by choosing various groups; (2) adaptive measurement strategies should be employed to make an appropriate match between nodes' current available bandwidth and blocks' access frequency; and (3) with both blocks' heat levels and nodes' bandwidth levels in place, it is allowed to reach a dynamic data placement and distribution by migrating hot in-memory blocks from heavily-burdened nodes to low-utilization nodes.

C. Design of WPS

1) Overview

The coordinator node comprehensively manage all the control information in in-memory stores (see, architecture in Fig. 2). To enable our WPS module to manipulate migration of in-memory hot blocks from heavily-burdened nodes to lightly-burdened ones, the coordinator maintains two tables to record both nodes' bandwidth level and in-memory data blocks' heat levels (see details of bandwidth and heat levels in Section III-C1).

Fig. 3 shows the whole data placement process. Both new data blocks issued by proxy node and existing data blocks read from disks are placed to high-bandwidth-level nodes, for example, new block D_2 and on-disk block D_5 are stored to the memory in node SN_2 of bandwidth level BL-3. If data block featured by high heat level is residing in the memory of a low-bandwidth-level node, then it will be migrated to a high-bandwidth-level node, e.g., in-memory data block D_7 of heat level HL-4) is moved from node SN_1 of bandwidth level BL-1 to node SN_4 of bandwidth level BL-4.

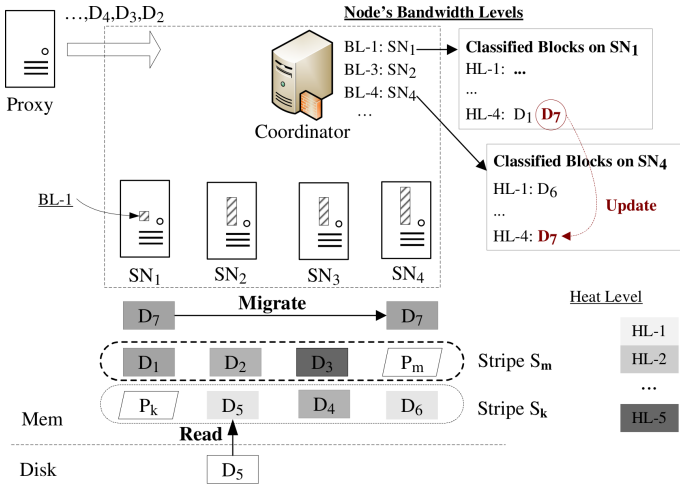


Fig. 3. Diagram of workload-aware data placement processes. Both newly-created data blocks and existing on-disk data blocks are placed to high-bandwidth-level nodes, and existing in-memory data blocks of high heat level is migrated from low-bandwidth-level nodes to high-bandwidth-level ones. The height of a diagonal-filled rectangle in a node indicates the bandwidth level of the node, and the depth of color of a block represents the heat level of the block.

When an in-memory block of high heat level is moved, only object reference (i.e., block location) needs to be updated, its stripe map is kept to be intact (i.e., the block still belongs to its original stripe). In this case, it is not required to perform re-striping operations (e.g., generating new parity blocks), thereby saving bandwidths of involved nodes.

2) Heat Levels and Bandwidth Levels

Usually, access popularity is assessed in term of access frequency. To accomplish balanced I/Os among all nodes in a cluster, it is preferred to migrate high-access-frequency blocks from heavily-burdened nodes to lightly-burdened ones. If all in-memory data blocks in divided into two groups, then it is difficult to determine the amount of high-access-frequency blocks to be migrated, and make a good tradeoff between migration overhead and I/O balance. Therefore, it is inclined to separate in-memory data blocks into multiple groups according to access frequencies. Theoretically, the group number of in-memory data blocks may be very large. For simplicity, we divide access frequencies of in-memory data blocks into 5 heat levels {i.e., HL-1, HL-2, ..., HL-5}, and the association between heat levels and access frequencies is given in Table I.

Table I. Association between heat level and access frequency.

HL	HL-5	HL-4	HL-3	HL-2	HL-1
Frequency	top 5%	6%-15%	16%-30%	31%-50%	51%-100%

Memory-level access performance is usually constrained by network bandwidths of nodes, so we adopt node's available bandwidth to indicates its performance. The available bandwidths of nodes always are varied since I/Os issued to each node are not constant. That is, the available bandwidth of each node is dynamic.

There are mainly two methods to evaluate the available bandwidth of a node: (1) experimental method – send a certain amount of data to the node and record the elapsed time, and calculate its quantitative bandwidth using data amount and elapsed time; and (2) statistical method – record the amount of requested blocks in all nodes in a certain duration, and rank the bandwidth usage of each node, e.g., the more requested blocks denote the lower available bandwidth as far as homogeneous nodes are concerned. We adopt the statistical method, because the experimental method causes extra network traffic. According to ranked bandwidths, all involved nodes are divided into several groups. For simplicity, all nodes are equally separated into 5 groups, and nodes in each group have a bandwidth level BL, with $BL \in \{BL\text{-}1, BL\text{-}2, ..., BL\text{-}5\}$ (see Table II).

Table II. Association between bandwidth level and available bandwidth rank.

BL	BL-5	BL-4	BL-3	BL-2	BL-1
Rank	top 20%	21%-40%	41%-60%	61%-80%	81%-100%

3) Algorithm

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:06:25 UTC from IEEE Xplore. Restrictions apply.

Table III summarizes all notations facilitating the presentation of the algorithms deployed in WPS.

Table III. Symbols and definitions.

Symbols	Definition
k, r	Redundancy parameters used to encode in-memory blocks
Seq	User request sequence
S_{row}	row^{th} stripe in memory
TW	Time window used to trigger migration, e.g., 200ms
HL-i	i^{th} heat level, e.g., i∈{1,2,...,5}
BL-j	j^{th} bandwidth level, e.g., j∈{1,2,...,5}
$Freq[Blk]$	Access frequency of data block Blk
$HL[Blk]$	Heat level of data block Blk
$BL[SN]$	Bandwidth level of storage node SN
$SN_{Blk,m}$	Node holding an in-memory copy of data block Blk
$SN_{Blk,d}$	Node holding an on-disk copy of data block Blk
Set_{SN}	Set of high-bandwidth-level nodes
Set_{Blk}	Set of high-heat-level in-memory data blocks
EN_{row}	Encoding node of stripe S_{row}

Procedure WPS(Seq, HL_t, BL_t) outlines the pseudo-code of workload-aware block placement. As mentioned above, a fine-granularity interval used to trigger the procedure of data migration may bring about migration 'jitter', and cause unnecessary migration traffic. To address this issue, a time window (i.e., TW) is introduced to initiate the procedure of data migration and the time window can be re-configured. When the time window is timeout, both heat and bandwidth levels are generated for the quantification processes for blocks' access frequencies and nodes' available bandwidths) and procedure Migration(HL_t, BL_t) is launched (see Lines 4, 5, 6). With the migration procedure in place, hot blocks tend to be moved to lowly-utilized node, thus accomplishing I/O balancing.

If the requested block has been already in memory, then its access frequency increases by 1 (see Line 10); otherwise, the block is an existing on-disk one or a new data block. If it is an on-disk block, then the block will be read from node $SN_{Blk,d}$. If it is a new data block, then a node is selected from node set Set_{SN}, and the block will be written to disk in the selected node. The set Set_{SN} contains storage nodes of high bandwidth level (e.g., $\geq BL$-4). The on-disk data block or the new data block is sent to the memory in a high-bandwidth-level node SN, and node SN is not the node keeping an on-disk copy of the block (see Lines 17, 18), thus guaranteeing high fault tolerance; additionally, the data block is designated to be a source block in stripe S_{row}, and sent to the corresponding encoding node EN_{row}. Since the data block is first placed to memory, its access frequency is set to 1 (see Line 20). When encoding node EN_{row} has received k data blocks in stripe S_{row}, it generates r parity blocks, and then keeps a copy in local memory and delivers other r-1 copies to memories in r-1 nodes over network.

Procedure Migration(HL_t,BL_t) is employed to move high-access-frequency blocks from low-bandwidth-level nodes to high-bandwidth-level nodes. Here, notations 'HL_t and BL_t' mean that if the heat level HT of an in-memory data block

Procedure 1: WPS(Seq, HL_t, BL_t)

Input:
Seq: user request sequence
HL_t: threshold of block heat level
BL_t: threshold of node bandwidth level

```
1  Initialize T ;                                              // set timer
2  while Seq is not over do
3      if T ≥ TW then
4          Set new heat levels (i.e.HL[]) for in-memory blocks;
5          Set new bandwidth levels (i.e.,BL[]) for nodes;
6          Migration(HL_t, BL_t) ;
7          Reset T;
8      end
9      if Requested block Blk is in memory then
10         Freq[Blk]←Freq[Blk]+1 ;          // update frequency
11     else
12         if Blk is on disk then
13             Read Blk from disk in node SN_{Blk,d};
14         else                                  // Blk is a new data block
15             Write Blk to disk in a node selected from Set_SN;
16         end
17         Select a node SN from Set_SN, with SN ≠ SN_{Blk,d} ;
18         Send Blk to SN;
19         Send Blk to encoding node EN_row of stripe S_row;
20         Freq[Blk] ← 1;
21     end
       /* parity generation for stripe S_row                      */
22     if EN_row has receives k data blocks in stripe S_row then
23         Generate r parity blocks;
24         Keep a parity block in local memory;
25         Dispatch other r-1 parity blocks among remote memories;
26         Discard the k data blocks;
27         row ← row+1;
28     end
29 end
```

Blk is greater than or equal to HL_t, and the bandwidth level BL of its hosted node is less than or equal to BL_t (i.e., $HL[Blk] \geq HL_t$, and $BL[SN_{Blk,m}] \leq BL_t$), then the data block will be migrated.

Procedure 2: Migration(HL_t, BL_t)

Input:
HL_t: threshold of block heat level
BL_t: threshold of node bandwidth level

```
   /* blocks to be migrated are inserted to set Set_Blk          */
1  foreach in-memory data block Blk do
2      if HL[Blk] ≥ HL_t and BL[SN_{Blk,m}] ≤ BL_t then
3          Set_Blk ← Blk;
4      end
5      Freq[Blk] ← 0 ;                  // initialize access frequency
6  end
   /* migrate candidate blocks to nodes in set Set_SN            */
7  foreach Blk ∈ Set_Blk do
       /* target node has no on-disk copy of Blk                 */
8      Select a node SN from Set_SN, with SN ≠ SN_{Blk,d};
9      Move block Blk from node SN_{Blk,m} to node SN;
10     Update mapping between block Blk and node SN;
11 end
```

Once the migration procedure is started, all in-memory data blocks' access frequencies is set to 0 (see Line 5). Since

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:06:25 UTC from IEEE Xplore. Restrictions apply.

heat levels of all in-memory data blocks are re-assessed at a time granularity of TW, high-access-frequency blocks can be dynamically placed to appropriate nodes. To accomplish high fault tolerance, an in-memory copy and another on-disk copy should be stored to two nodes (see Line 8). When a data block is successfully migrated to a destination node, the coordinator node updates the mapping between the block and its destination (see Line 10).

IV. Performance Evaluation

To comparatively evaluate our workload-aware scheme WPS, we run WPS and two candidate placement schemes on a real-world cluster. In this section, we describe the experiment environment and methodology; then we quantitatively examine the impact of various parameters (e.g., $Size_{blk}$, and r) on the three schemes.

A. Experimental Environment

1) System Setup

We build an experiment cluster, which consists of one coordinator node, one proxy, and 8 storage nodes {SN_1, SN_2, ..., SN_8,}. 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 @2.40GHz CPUs, and 32GB DDR3 memory. Each storage node contains an Intel® E5800 @3.2GHz CPU, 3GB DDR3 memory, and a Western Digital Enterprise SATA2.0 disk with model WD1003FBYX. We install the Linux with Kernel 2.6.28.10 (x86-64) on all the machines.

2) Prototype Design

We implement a prototype system, where the coordinator node embraces one out of the three placement schemes to guide the cluster nodes to organize in-memory blocks, and maintains all meta-data (e.g., stripe mapping, access frequency, bandwidth statistics, etc.); the proxy node simulates client to issue requests derived from YCSB benchmark [7]; storage nodes are responsible for data placement (i.e., delivering on-disk blocks to remote memories), stripe formation, and data migration (i.e., redistributing in-memory blocks among nodes).

3) Testing Methodology

MemEC selects k data servers and r parity servers for a stripe according to servers' loads [36]. Here we denote such load-based placement scheme to LPS. Trinity uses a hashing mechanism to locate a given key's corresponding value in the memory cloud [31]. Here we refer to this hash-based placement scheme as HPS. In the following tests, we comparatively evaluate WPS, LPS, and HPS schemes.

We use read-only workloads generated from the YCSB benchmark to examine the validity of WPS. Both YCSB client and InMem-RS proxy are hosted in the same machine. The number of unique data blocks is 1K, and the YCSB client issues total 1M requests, and the access pattern follows a Zipf distribution with the shape parameter 0.8. On-disk data blocks are randomly distributed across the storage nodes and the distribution is identical for the three schemes during initialization phase. In our scheme WPS, node bandwidths are divided into 5 levels, and each node's bandwidth level is initialized to BL-5. Data blocks are grouped into 5 heat levels based on access frequency, where data blocks of HT-5 occupy 1% of total data blocks.

Request response time is measured according to the time interval between issuing request and returning requested data blocks. We calculate the average response time over 5 runs for each test item. We conduct four groups of experiments to examine the sensitivity of the three schemes to both size $Size_{blk}$ of request data block and number r of parity blocks in a stripe.

- Parameters k=4, r=1, and $Size_{blk}$=32KB;
- Parameters k=4, r=1, and $Size_{blk}$=16KB;
- Parameters k=4, r=1, and $Size_{blk}$=64KB;
- Parameters k=4, r=2, and $Size_{blk}$=32KB.

B. Experimental Results

1) Parameters k=4, r=1, and $Size_{blk}$=32KB

Fig. 4 shows both access latency and access distribution across nodes of the three schemes, with parameters k, r, and $Size_{blk}$ are set to 4, 1, and 32KB, respectively. Access distribution across nodes are measured in terms of the amount of data blocks read from each node.

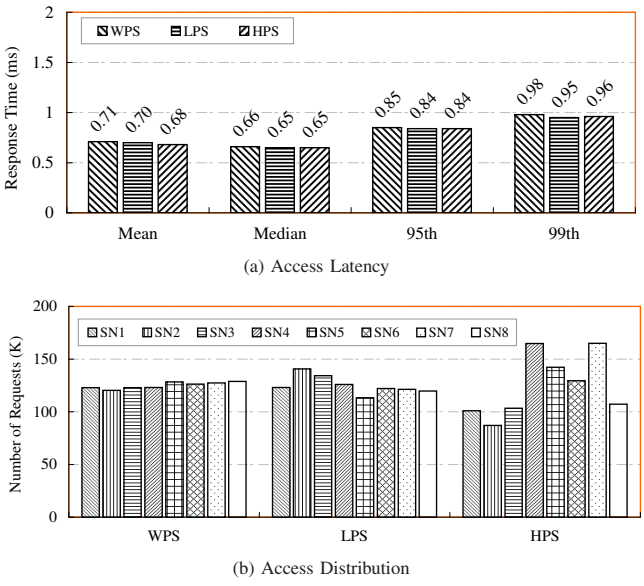


Fig. 4. Both access latency and access distribution across nodes, with parameters k=4, r=1, and $Size_{blk}$=32KB.

From Fig. 4(a), we observe that, compared to LPS and HPS, WPS accomplishes similar mean, median, and tail latencies (i.e., 95th and 99th latencies). This observation indicates that the data migration operations in WPS have a marginal impact on user access performance.

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:06:25 UTC from IEEE Xplore. Restrictions apply.

Load imbalance can be measured via a percent imbalance metric λ [27], with λ=$((L_{max}-L_{avg})/L_{avg})\times 100$, where L_{max} is the maximal load among all nodes and L_{avg} is the average load of all nodes. In this group of experiments, the percent imbalance metric λ for WPS, LPS, and HPS are 3.10%, 12.46%, and 31.96%, respectively. That is, WPS scheme can achieve the best load balancing among the three schemes, the reasons are three-fold: (i) WPS dynamically migrates high-heat-level data blocks from low-bandwidth-level nodes to high-bandwidth-level ones, that is, WPS considers both block access frequencies and node available bandwidths; (ii) Different from WPS, the LPS scheme does not take access frequencies into account; (iii) the HPS scheme randomly selects exact k+r nodes to keep k data blocks and r parity blocks in a stripe, ignoring both block access frequencies and node available bandwidths.

2) Parameters k=4, r=1, and $Size_{blk}$={16KB, 64KB}

To examine the performance sensitiveness of the three schemes to parameter $Size_{blk}$, we set $Size_{blk}$ to 16KB and 64KB, with k=4 and r=1. Note that Fig. 4 has reported the experimental results in the case of 'k=4, r=1, and $Size_{blk}$=32KB'. Fig. 5 and Fig. 5 show both access latency and access distribution across nodes of the three schemes, with parameter $Size_{blk}$ is set to 16KB and 64KB, respectively. Similar to experimental results in the case of '$Size_{blk}$=32KB', the three schemes accomplish similar mean, median, and tail latencies under '$Size_{blk}$=16KB' or '$Size_{blk}$=64KB'; and WPS achieves the best load balancing among the three schemes.

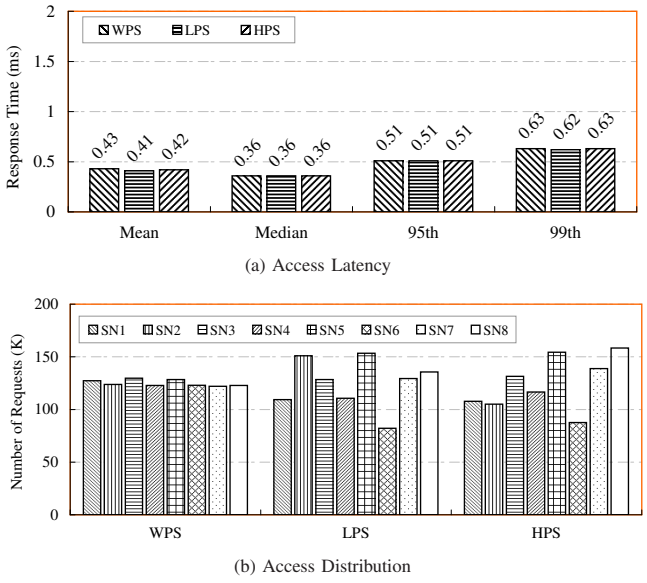


Fig. 5. Both access latency and access distribution across nodes, with parameters k=4, r=1, and $Size_{blk}$=16KB.

From Figs. 5(a), 4(a), and 6(a), it is observed that, compared to case '$Size_{blk}$=16KB', the mean latencies of WPS, LPS, and HPS increase by a factor of 1.65, 1.70, and 1.62 in the case of '$Size_{blk}$=32KB', respectively; and the mean latencies

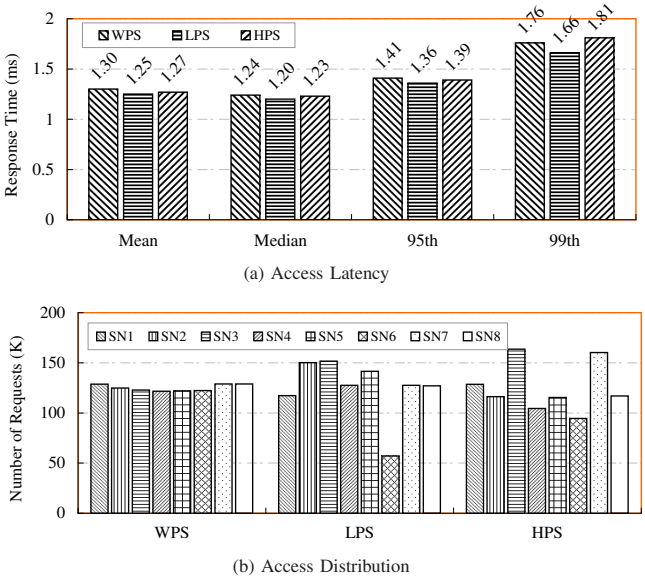


Fig. 6. Both access latency and access distribution across nodes, with parameters k=4, r=1, and $Size_{blk}$=64KB.

of WPS, LPS, and HPS increase by a factor of 3.02, 3.05, and 3.02 under '$Size_{blk}$=64KB', respectively. The reason lies in the fact that, compared to small blocks, large blocks usually incur higher access latencies. However, the factor is not 2.0x (32KB/16KB=2.0) or 4.0x(64KB/16KB=4.0), the reasons are two-fold: (i) Data accesses in in-memory stores may involve three steps, i.e., reading from disks/memories, transferring over networks, and writing to memories. (ii) A large-block access triggers a single request of 'reading/transferring a block of large size', or launches multiple requests of 'reading/transferring a block of small size', and the latter case incurs more access overhead. For example, A 64KB-block access triggers a single request of 'reading/transferring a block of 64KB' in the case of '$Size_{blk}$=64KB', or four requests of 'reading/transferring a block of 16KB' in the case of '$Size_{blk}$=16KB'.

3) Parameters k=4, r=2, and $Size_{blk}$=32KB

The above two groups of tests are conducted under parameter r=1. To explore the performance impact of redundancy parameter r, we conduct a new group of experiment using settings k=4, r=2, and $Size_{blk}$=32KB; and the experimental results is plotted in Fig. 7.

From Fig. 7, it is indicated that the three schemes obtain similar mean, median, and tail latencies; and the WPS scheme still achieves the best load balancing among the three schemes. In particular, the percent imbalance metric λ for WPS, LPS, and HPS are respectively 3.05%, 18.61%, and 27.65%. These observations are consistent with the experimental results in Figs. 5, 4, and 6.

From Figs. 4(a) and 7(a), we observe that, the three schemes have almost identical access performance in both cases of 'r=1' and 'r=2' in terms of mean, median, and tail latencies. There

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:06:25 UTC from IEEE Xplore. Restrictions apply.

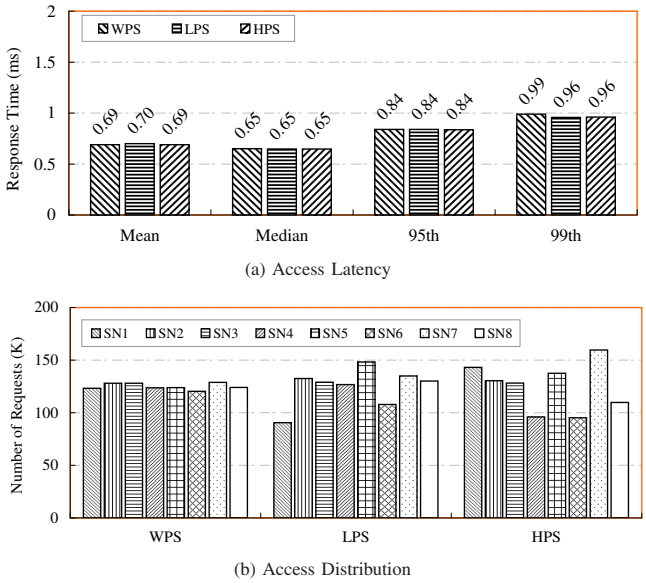


Fig. 7. Both access latency and access distribution across nodes, with parameters k=4, r=2, and $Size_{blk}$=32KB.

are two reasons for this phenomenon: (i) As to newly-created stripes, the parity-generation overhead is close because existing CPUs have sufficiently high computation capability to produce parity blocks and resulting parity blocks are delivered to r unique nodes in parallel; (ii) As for existing stripes, migrations in WPS are aimed at data-block redistribution, without leading to parity updates; both LPS and HPS are unconcerned with data redistribution.

C. A Summary of Observations

We summarize the important observations drawn from Figs. 4, 6, and 7 as follows.

- Our WPS scheme accomplishes the best load balancing among the three schemes, while it achieves comparable access performance in terms of mean, median, and tail latencies relative to the LPS and HPS schemes.
- The size $Size_{blk}$ of requested data block helps to improve access performance in erasure-coded in-memory stores, while the number r of parity blocks in a stripe has slight impact on access latencies of data placement schemes.

V. Related Work

A. In-memory Store

Just like caching in storage systems, a wide range of in-memory stores have been proposed for the low-access-latency purpose. For example, Spark persists iteratively-used dataset in memory at coarse-grained resilient distributed datasets (RDDs), and outperforms Hadoop by 10× in iterative machine learning jobs [38]. Memcached is a simple but effective in-memory caching solution, and Facebook adopts memcached to construct a distributed in-memory key-value store that handle billions of requests per second [22].

Usually, in-memory stores place one or multiple copies of data to memory in a remote server across network and writes at least one copy onto non-volatile media to survive cluster-wide failures (e.g., power outages). For example, Tachyon keeps one copy of data to remote memory and writes three replicas to HDFS systems [17]; RAMcloud is a DRAM-based storage system that stores a data block in DRAM, and scatters a replica of the data block across hundreds or thousands of disks [23].

B. Fault-tolerant In-memory Store

Since RAM is volatile, it can not maintain its data while it is powered off. Some in-memory stores accomplish memory-level fault tolerance using replication or erasure codes.

1) Replication-based In-memory Store

Replication has the following advantage: First, it provides increased parallelisms, for instance, selective replication keeps more replicas in RAM for hot data blocks [1]. Second, it needs less data movement over network in the recovery case. When a data block is lost, only one replica block is adequate to repair it.

To obtain memory-level redundancy, Repcached [15], Redis [29], Bigmemory [35], Coherence [30], and Websphere [33] schemes keep two replicas to memory among nodes. Assume data block D is divided into k sub-blocks $\{d_1, d_2, ..., d_k\}$. As shown in Table. IV, two copies of k sub-blocks reside in the memory for Redis, BigMemory, and Websphere; whereas one copy of data block D and its k sub-blocks are kept in memory in the Coherence scheme.

Table IV. Data organization of in-memory systems

Scheme	In-memory Data	On-disk Data
Redis, Repcached	$\{d_1, d_2, ..., d_k\}$,$\{d_1, d_2, ..., d_k\}$	$\{d_1, d_2, ..., d_k\}$
BigMemory	$\{d_1, d_2, ..., d_k\}$,$\{d_1, d_2, ..., d_k\}$	$\{D\}^*$
Coherence	$\{d_1, d_2, ..., d_k\}$,$\{D\}^*$	$\{D\}^*$
Websphere	$\{d_1, d_2, ..., d_k\}$,$\{d_1, d_2, ..., d_k\}$	$\{D\}^*$

$\{D\}^*$: Data block D can be divided into k sub-blocks $\{d_1, d_2, ..., d_k\}$.

2) Erasure-coded In-memory Store

Since memory is an expensive resource, it is necessary to achieve high memory efficiency when designing reliable in-memory stores. Compared to replication, erasure codes achieve higher memory efficiency. For example, the memory efficiency of triplication tolerating double failures is 33.3%, while it is 66.7% for double-fault-tolerant (6, 4) RS codes. Therefore, erasure coding is gradually adopted by in-memory stores.

EC-cache is a load-balanced, low latency cluster cache using object splitting, online erasure coding, and late binding [27]. Cocytus makes the first case of building highly available in-memory key-value store by using erasure coding. Cocytus not only incurs low overhead for latency and throughput, but also tolerates node failures with fast online recovery [40]. Similar to Cocytus, MemEC is an erasure-coding-based in-memory key-value store, in which workloads are dominated by small objects.

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:06:25 UTC from IEEE Xplore. Restrictions apply.

MemEC not only achieves high throughput and low latency in both normal and degraded modes, but also accomplishes fast transitions between the two modes [36].

Apart from eviction mechanism and load-balancing strategy, existing erasure-coded in-memory stores primarily investigate performance improvements with respect to throughput, access latency, and recovery. These solutions are focused on pure in-memory data objects, without considering on-disk data. In fact, in-memory stores are tightly coupled to on-disk storage (e.g., on-disk data objects should be read to memories in the cases of initialization and memory/cache miss.), it is necessary to take into consideration both network transfers and disk I/Os. Our WPS scheme investigates data placement for erasure-coded in-memory stores by exploiting access popularity and available bandwidth.

VI. Conclusions and Future Work

In-memory stores encounter a challenge of load imbalance due to access popularity skew, and load imbalance trends to incur low resource utilization. To address such an issue of load imbalance, we proposed a workload-aware placement scheme - *WPS* - for erasure-coded in-memory stores. WPS accomplishes balanced I/Os by taking both block's access frequency and node's available bandwidth into consideration. Assessing metrics 'heat level' and 'bandwidth level' are introduced to quantify in-memory data block's access popularity and node's access performance, respectively, and the WPS scheme places or migrates high-heat-level in-memory data blocks to high-bandwidth-level nodes. The experimental results show that WPS can significantly improve load balancing for erasure-coded in-memory stores exhibiting workload popularity skew; on the other hand, WPS achieves comparable access performance in terms of mean, median, and tail latencies relative to two candidate placement schemes.

In our future study, we will further investigate the WPS scheme in the arena of heterogeneous erasure-coded in-memory stores. Specifically, we plan to accomplish load balancing by taking into account workload popularity and configuration heterogeneity.

Acknowledgment

This work is supported 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 CCF-0845257, CNS-0917137, and CCF-0742187, and the 111 Project under Grant B07038.

References

[1] 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 *Proceedings of the sixth conference on Computer systems (EuroSys'11)*, pages 287–300. ACM, 2011.

[2] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. Pacman: Coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 20–33. USENIX Association, 2012.

[3] D. Borthakur. The Hadoop Distributed File System: Architecture and Design. *Hadoop Apache Project. http://hadoop.apache.org/common/docs/current/hdfs_design.html*, 2009.

[4] D. Borthakur, R. Schmidt, R. Vadali, S. Chen, and P. Kling. HDFS RAID, 2010. Technical Talk. Yahoo! Developer Network.

[5] Q. Cai, H. Zhang, G. Chen, B. C. Ooi, and K.-L. Tan. Memepic: Towards a database system architecture without system calls, 2014.

[6] S. Chen and D. Towsley. A performance evaluation of raid architectures. *IEEE Transactions on computers*, 45(10):1116–1130, 1996.

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

[8] G. Copeland and T. Keller. A Comparison of High-availability Media Recovery Techniques. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data (SIGMOD'89)*, pages 98–109, New York, NY, USA, 1989. ACM.

[9] D. Ford, F. Labelle, F. Popovici, M. Stokely, V. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, pages 61–74. USENIX, 2010.

[10] G. Gligor, S. Teodoru, et al. Oracle exalytics: engineered for speed-of-thought analytics. *Database Systems Journal*, 2(4):3–8, 2011.

[11] J. L. Hafner. WEAVER Codes: Highly Fault Tolerant Erasure Codes for Storage Systems. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies (FAST'05)*, pages 211–224, Berkeley, CA, USA, 2005. USENIX Association.

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

[13] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure Coding in Windows Azure Storage. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (ATC'12)*, pages 15–26. USENIX Association, 2012.

[14] J. Hwang and T. Wood. Adaptive performance-aware distributed memory caching. In *ICAC*, volume 13, pages 33–43, 2013.

[15] K. Inc. Repcached[online]. http://repcached.lab.klab.org.

[16] M. Inc. Mongodb[online]. https://www.mongodb.com.

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:06:25 UTC from IEEE Xplore. Restrictions apply.

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

[18] C. Lu, G. A. Alvarez, and J. Wilkes. Aqueduct: Online data migration with performance guarantees. In *FAST*, volume 2, page 21, 2002.

[19] 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.

[20] A. Miranda and T. Cortes. Craid: online raid upgrades using dynamic hot data reorganization. In *FAST*, pages 133–146, 2014.

[21] H. Mühleisen, R. Gonçalves, and M. Kersten. Peak performance: Remote memory revisited. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware (DaMoN'13)*, page 9. ACM, 2013.

[22] 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 *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation (NSDI'13)*, volume 13, pages 385–398, 2013.

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

[24] D. A. Patterson, G. Gibson, and R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data (SIGMOD'88)*, pages 109–116, New York, NY, USA, 1988. ACM.

[25] J. Plank et al. A Tutorial on Reed-Solomon Coding for Fault-tolerance in RAID-like Systems. *Software Practice and Experience*, 27(9):995–1012, 1997.

[26] 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 *Proccedings of the 7th conference on File and Storage Technologies (FAST'09)*, pages 253–265. USENIX, 2009.

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

[28] I. Reed and G. Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[29] S. Sanfilippo. Redis[online]. https://www.redis.io.

[30] A. Seovic, M. Falco, and P. Peralta. *Oracle Coherence 3.5*. Packt Publishing Ltd, 2010.

[31] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*, pages 505–516. ACM, 2013.

[32] V. Sikka, F. Färber, A. Goel, and W. Lehner. Sap hana: The evolution from a modern main-memory data platform to an enterprise application platform. *Proceedings of the VLDB Endowment*, 6(11):1184–1185, 2013.

[33] M. E. Taylor et al. *WebSphere MQ V7.1 and V7.5 Features and Enhancements*. IBM Redbooks, 2013.

[34] H. Weatherspoon and J. D. Kubiatowicz. Erasure Coding vs. Replication: A Quantitative Comparison. In *Peer-to-Peer Systems*, pages 328–337. Springer, 2002.

[35] D. Wind. *Instant Effective Caching with Ehcache*. Packt Publishing Ltd, 2013.

[36] M. M. Yiu, H. H. Chan, and P. P. Lee. Erasure Coding for Small Objects in In-Memory KV Storage. *arXiv preprint arXiv:1701.08084*, 2017.

[37] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[38] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'10)*, 10(10-16):95, 2010.

[39] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1920–1948, 2015.

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

[41] H. Zhu, P. Gu, and J. Wang. Shifted declustering: a placement-ideal layout scheme for multi-way replication storage architecture. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 134–144. ACM, 2008.

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:06:25 UTC from IEEE Xplore. Restrictions apply.