Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

A reliable and energy-efficient storage system with erasure coding cache*

Ji-guang WAN†‡, Da-ping LI†, Xiao-yang QU, Chao YIN, Jun WANG, Chang-sheng XIE†
(*Wuhan National Laboratory for Optoelectronics, Department of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China*)
†E-mail: jgwan@mail.hust.edu.cn; ldplcp@qq.com; cs_xie@mail.hust.edu.cn

Received Jan. 31, 2016; Revision accepted May 3, 2016; Crosschecked Sept. 23, 2017

Abstract: In modern energy-saving replication storage systems, a primary group of disks is always powered up to serve incoming requests while other disks are often spun down to save energy during slack periods. However, since new writes cannot be immediately synchronized into all disks, system reliability is degraded. In this paper, we develop a high-reliability and energy-efficient replication storage system, named RERAID, based on RAID10. RERAID employs part of the free space in the primary disk group and uses erasure coding to construct a code cache at the front end to absorb new writes. Since code cache supports failure recovery of two or more disks by using erasure coding, RERAID guarantees a reliability comparable with that of the RAID10 storage system. In addition, we develop an algorithm, called erasure coding write (ECW), to buffer many small random writes into a few large writes, which are then written to the code cache in a parallel fashion sequentially to improve the write performance. Experimental results show that RERAID significantly improves write performance and saves more energy than existing solutions.

Key words: Reliability; Energy-efficient; Storage system; Erasure coding; Cache management
https://doi.org/10.1631/FITEE.1600972 **CLC number:** TP316.4

1 Introduction

Along with the rapid growth of data production in information technology (IT) companies, greater requirements for storage space have come. Large-scale parallel I/O systems have been widely used in high performance computer systems and are extremely important for high performance computing (HPC) and other big data applications, such as those used by the U.S. Department of Energy to simulate new energy sources (Department of Energy, 2012), which produce vast amounts of data. On the other hand, due to the continuously increasing scale of parallel I/O systems, the proportion of the energy consumption of I/O systems to the total cost of ownership (TCO) becomes larger and larger. In a typical data center, the storage subsystem can consume more than 27% of the total energy (Zhu *et al.*, 2005) and it grows annually. Power in storage systems is often saved at the expense of reliability and performance (Eom and Hollingsworth, 2000). In replication storage systems, one or more disk groups are powered off to save energy when workloads are light. There are many power layout solutions (Amur *et al.*, 2010), such as EERAID (Li and Wang, 2004), GRAID (Mao *et al.*, 2008), ROLO (Yue *et al.*, 2010; 2016), Rabbit (Department of Energy, 2012), and Sierra (Thereska *et al.*, 2011). However, since the up-to-date writes are stored only in the primary disk

‡ Corresponding author
* Project supported by the National Natural Science Foundation of China (Nos. 61472152, 61432007, 61572209, and 61300047), the Fundamental Research Funds for the Central Universities, China (No. 2015QN069), the Director Fund of Wuhan National Laboratory for Optoelectronics (WNLO), and the MOE Key Laboratory of Data Storage System, China
ORCID: Ji-guang WAN, http://orcid.org/0000-0003-0489-6783

group when the non-primary disk groups are spun down, the reliability of the new data will be degraded until the new data is written back to all the non-primary replicas. To save energy without degrading reliability and performance, we propose to use erasure coding to encode the up-to-date data.

Unlike using replications to improve reliability, storage systems using erasure coding have the disadvantage of high computing overhead. Fortunately, we find that abundant and speedy CPU cores make it possible to perform coding online. For example, a single Intel Xeon E3-1230v3 CPU core can encode data at 5.26 GB/s for Reed-Solomon(3,5) codes, which is faster than even current high-end NIC with 40 Gb/s bandwidth. The performance of the disk has improved much slower than that of CPUs such that the bottleneck of system performance is not the CPU, but the disk. Thus, we can conclude that the effect of computing overhead on a storage system's overall performance is minimal compared with that of the read and write overheads of the disk.

Based on the above considerations, we developed a reliable and energy-efficient replication storage system with erasure coding cache, called RERAID. In our system, we divide all the disks into X groups, which are the primary replica, the second replica, and so on. The erasure coding cache is composed of part of the primary disk group and used as a log to store the up-to-date write requests, and we refer to the erasure coding cache as the code cache for convenience. The up-to-date write data is stored only in the code cache sequentially as a log, so we can spin down all the non-primary disk groups to save energy and use the primary disk group to deal with all the requests. At the same time, the system will have a good write performance. To guarantee the reliability of the new data in the code cache, we also developed the erasure coding writing (ECW) algorithm. The ECW algorithm can optimize small write requests by aggregating them into fixed-size large write chunks and encoding the chunk in memory. Finally, we can obtain a new chunk consisting of data chunk and parity chunk, and then write it into code area.

When the system workload is light or the code cache is going to be full, RERAID will migrate the up-to-date data from the code cache to the old data area of both the primary group and other groups. Thus, the reliability of the old data will also be secured by replications. Owing to the use of the logging technique, the modification of the same data blocks will be stored in different places in the code cache. To make data migration more convenient, we proposed the minimum correlation flush disk (MCFD) algorithm. This algorithm reorganizes data blocks from correlated chunks in the code cache by the order of every block's location in the disk, and then flushes them to all the replicas.

The contributions of this study are summarized as follows:

1. RERAID is capable of providing the reliability of new and old data. RERAID also decreases energy consumption without compromising performance. We have developed the ECW algorithm to improve write performance. With this algorithm, RERAID buffers many small random writes into large writes and then writes them to the code cache, which is a special area established using erasure coding in the primary disk group. The system details will be described in Section 3.2.

2. To migrate data from the code cache to all the replicas in RERAID, we have developed the MCFD algorithm (Section 3.4) to improve the write performance. According to the correlations among different chunks in the code cache, we choose the chunks with higher correlations and reconstitute them into stripes and then flush the stripes to the disk.

3. We have implemented RERAID based on Linux soft RAID. A comprehensive sensitivity evaluation indicates that the higher the write ratio is, the better the RERAID performance will be, especially under random write workloads. For high write-to-read-ratio I/O workloads, RERAID could significantly improve the write performance by 69.1% compared to current energy saving solutions (GRAID, ERAID, etc.) when the requests are all write requests. In addition, it reduces energy consumption by 44.4% compared to RAID10, and 16.7% compared to GRAID. The code cache of RERAID allows the flexibility of choosing any erasure coding level according to storage security demand, hence supporting different disk failure-recovery numbers.

2 Objective and related work

In this section, we first show the objectives of this study. Then, we discuss several popular disk array layouts, such as GRAID. Finally, we discuss

small write technology, which will be used in our system.

2.1 Objective

Reliability is a prerequisite in storage systems, and redundancy is a common method used to ensure a storage system's reliability. Well-known storage systems, such as GFS (Ghemawat *et al.*, 2003), HDFS (Borthaku, 2010), Amazon S3 (Amazon, 2007), Ceph (Weil *et al.*, 2006), and EMC Atmos (EMC, 2008), use replication to provide data redundancy. For example, HDFS, a currently widely used cloud file system, adopts three-way replication by default; EMC Atmos allows the reservation of more replicas with additional payment. Replication has many advantages over erasure coding. Simplicity of implementation is its prime advantage. More importantly, replication that is available in different nodes increases the total disk-read bandwidth, which helps balance loads across the nodes.

With reliability requirements increasing, a single point of fault tolerance is far from meeting the needs of consumers, and thus multi-level fault-tolerant systems come into play. There are many erasure codes. For example, EVENODD (Blaum *et al.*, 1994), RDP (Corbett *et al.*, 2004), X code (Xu and Bruck, 1999), and star code, can tolerate simultaneous failures of two to three disks; RS coding (Plank and Xu, 2006) can tolerate k disk failures. In distributed systems, by erasure coding, data can be divided into m fragments and recoded into n fragments ($n > m$). In this situation, the original data can be reconstructed from any m fragments. Some researchers believe that erasure coding has the potential to replace the N-way replication in the future, as with the updated GFS2, which has imported RS encoding in action. However, erasure coding has its disadvantages, such as extra disk requests when modifying and an additional calculation workload. In this study, we successfully avoid modifying writes by importing write logging technology and designing the ECW algorithm. As a note, we are aware that the development of hardware can help us solve the calculation problem.

Storage system designers should consider how to balance performance, power consumption, and reliability. We design our system to combine the advantages of replications and erasure coding. In the system, we use a multi-mirror fashion to accept more read requests and use a code cache to receive write requests to reduce disk writes. If the read workload is not heavy, we can shut off the nodes containing non-primary replicas to save energy. Our proposal has the main advantage of saving energy and maintaining reliability with seldom performance degradation, while not requiring any additional hardware.

2.2 Related work

2.2.1 Disk array layouts on energy saving

Ever since RAID (Patterson *et al.*, 1988) was invented, energy consumption and reliability have been a concern of researchers. Massive array of idle disks (MAID) (Colarelli and Grunwald, 2002) introduces an energy consumption disk array, which has an alternative tape library as a backup system. It organizes a group of disks into RAID0 form and defines a subset of the disks as 'hot disks' and the others as 'cold disks'. The hot disks are active for a long period of time, while the cold disks remain at a low energy consumption status. This technology has solved the energy-efficiency problem, but the up-to-date data in the hot disks suffers from a reliability problem.

Another energy-saving technique named PDC (Pinheiro and Bianchini, 2004) migrates popular data to a subset of disks dynamically. Thus, though the load will be unbalanced, the majority of disks will be switched to a low power mode. In realization, PDC places the files visited most frequently on the first disk and places the second most frequently visited files in the second disk, and so on. ERAID is similar to PDC. ERAID (Li and Wang, 2006; Wang *et al.*, 2008) saves energy by spinning down partial disk groups. Through its time-window control scheme, it can control the trade-off between energy conservation and performance degradation. All schemes mentioned above have the same defect: they do not address reliability and aim only for energy conservation.

Auto RAID (Wilkes *et al.*, 1996) combines RAID1 (Lu *et al.*, 2007) and RAID5 to achieve a tradeoff between performance and energy. GRAID and ROLO add a separate log disk on the basis of RAID10, so it has two replicas for writing new data and reading old data, and thus its reliability can be guaranteed. However, hardware added in GRAID leads to an increase in loading time and uncertainty, so it is not advisable.

Rabbit and Sierra address power-proportional distributed storage. When a primary node fails, the non-primary node will be activated and the data in the primary node will be restored. Each replica is grouped into gear groups. When one of the primary servers fails, non-primary groups will be spun up to recover the files. These two models have used gear to guarantee system reliability in low power modes, in which most of the replicas are shut down. In essence, the gear groups also use replica technology to protect data from loss, and data is stored with at least two replicas.

2.2.2 Small write technology

Parity Logging (Stodolsky *et al.*, 1993) is the first that introduced the logging technique in disk arrays. It is used to overcome the small write problem of RAID5. Log-Structured Array (LSA) (Menon, 1995) was proposed by combining LFS, RAID5, and a non-volatile cache. LSA writes the updated data into new disk locations instead of writing in place to improve the write performance of RAID5. As with Parity Logging, Logging RAID (Chen *et al.*, 2000) was also proposed to solve the small write problem of RAID5. Parity Logging, LSA, and Logging RAID are all based on RAID5. The main idea of DCD (Hu and Yang, 1996) is to use a small log disk, referred to as a cache disk, as a secondary disk cache to optimize write performance. While the cache disk and normal data disk have the same physical properties, the access speed of the former differs dramatically from that of the latter because of different data units and the different ways in which data is accessed. Its objective is to exploit this speed difference by using the log disk as a cache to build a reliable and smooth disk hierarchy. In this study, we have extended these technologies into RERAID.

3 RERAID

3.1 Design

The power consumption of disks in standby mode is far less than that in active mode. However, switching a disk from a standby mode to an active mode requires a great deal of energy. For example, a Western Digital RE4 1 TB WD1003FBYX (3.5SATA/64 MB Cache 3 Gb/S, 7200 r/min) disk's idle power, standby power, and active power are about 5.9, 0.7, and 7.9 W, respectively. The time for spinning up the disk is about 5.6 s. Therefore, our strategy is to put non-primary replications in standby mode as long as possible, and reduce the number of transitions from standby mode to active mode as much as possible.

The architecture of RERAID is shown in Fig. 1. Disks are divided into X groups, which are the primary replica, the second replica, and so on. The disks in each replica are organized in the form of RAID0. In the example in Fig. 1, each disk group has four disks. In the primary group, a part of the unused space is separated by erasure coding as a code cache, similar to log. Usually, there exists unused storage space that can be used as a code cache to cache the new data. The non-primary replicas are usually put in standby mode to save energy. As shown in Fig. 1, the code cache stores all the up-to-date data, while the data areas of the primary replica and other non-primary replicas have the same old data.

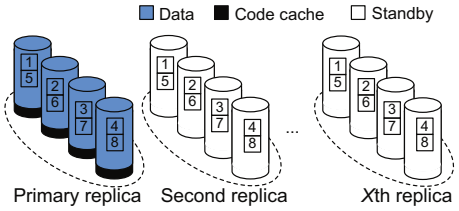

Fig. 1 The architecture of RERAID

In detail, upon receiving write requests, RERAID will encode the data and send the requests to the code cache first. When the code cache is almost full or the system has a low workload, RERAID will wake up all the replica disks and flush the new data in the code cache to all the replicas, including the data area of the primary replica. To make the destage more quickly, we develop a flush algorithm, called MCFD, to migrate data from the code cache to all the replicas (see Section 3.4). When receiving read requests, only the code cache and the primary replica group are used. When a disk of the primary replica is damaged, the new data in the code cache can be restored by erasure coding and the old data not in the code cache can be restored from the other replicas. So, our scheme can guarantee the reliability of the system.

Here, we compare erasure coding with multi-replication. The greatest benefit of erasure coding is that its write overhead is less than that of replication.

When using triple replication, if we want to write seven blocks of new data, we must write 21 blocks (7 blocks to write data and 14 blocks for replication). In contrast, if we use erasure coding, such as two parities, writing 7 blocks of new data requires writing only 9 blocks.

However, the calculation overhead of erasure coding is larger than that of multi-replication, especially when the data is modified. This is because erasure coding requires calculation updating rather than the simply overwriting in multi-replication. Fortunately, CPU performance has been developing rapidly; however, disk performance is developing slowly. This indicates that disk performance becomes the bottleneck in storage servers. As CPU performance can benefit the calculations, however, the calculation overhead of erasure coding is not an impact on the overall storage performance.

Another workload in erasure coding is additional disk requests in modifying writes. When data is modified, the system must update the parity. Since it needs to read unmodified data blocks in a check chain separated on several disks, more read requests are involved.

To shield the above disadvantages, we design the ECW algorithm for RERAID, which will be described in detail in Section 3.2.

3.2 Write buffer and code cache

To improve the write performance of our system, we developed an algorithm called ECW. The up-to-date writing requests are stored in the magnetic random access memory (MRAM) buffer provisionally and every modification is treated as a new write. The MRAM write buffer is divided into a few fixed-size chunks (about 4 chunks) and the size of MRAM is just about 4 MB in our experiment. So, the MRAM writer buffer is small enough and it will not affect the reliability of RERAID. In the system, each replica has n disks and $n = k + m$, where m represents the number of data blocks and k the number of parity blocks. Note that data is divided and encoded in the system memory, not in MRAM.

There are four steps for ECW (Fig. 2):

1. Cache the data: The system receives write requests and caches the data in a fixed-size chunk in MRAM sequentially. There are about four chunks in MRAM.

2. Split the chunk: When a chunk is full, the system will divide it logically into m fixed-size blocks $(D_1, D_2, \cdots, D_m)$ in the system memory.

3. Code the chunk: m data blocks are encoded to obtain k parity blocks $(P_1, P_2, \cdots, P_k)$ in the system memory.

4. Write to the code cache: RERAID writes the data blocks $(D_1, D_2, \cdots, D_m)$ and parity blocks $(P_1, P_2, \cdots, P_k)$ into the code cache in the disks of the primary replica in parallel. Encoded data is distributed on disks using the common left-symmetric mode, to facilitate load balance.

Here, the type of erasure coding that we choose depends on the system requirements. If we choose one parity block, we can use erasure coding similar to RAID5 XOR; if we need more reliability, we can use two or more parity blocks. Currently, only the RS algorithm can tolerate multiple faults and is easy to configure, so we use this algorithm in our system. The data to be encoded is stored in the write buffer, so we can configure flexibly the number of redundancies to adjust the reliability. The ECW algorithm is described in Algorithm 1.

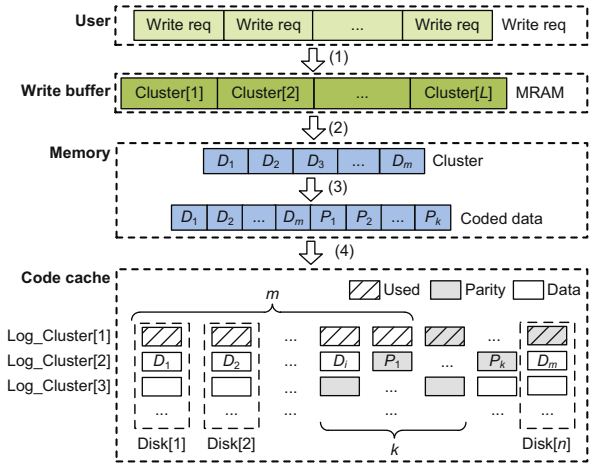

Fig. 2 The erasure coding writing (ECW) algorithm of RERAID

3.3 System data structure

RERAID uses write buffer to temporarily store small-size data that is accessed by host I/O requests. In addition, as a result of write logging, prime data and changes are stored separately, which makes data recovery a concern. Our solution is to employ a simple log structure as seen in prior studies (Bhadkamkar *et al.*, 2009; Soundararajan *et al.*, 2010).

Algorithm 1 Erasure coding writing algorithm for write operation of RERAID

Input: Req, incoming request; O, offset (default value is 0); L_{r}, size of Req; p, pointer pointing to the available cluster in buffer; L_{c}, available capacity of cluster p

1: **while** $L_{\mathrm{c}} < L_{\mathrm{r}}$ **do**
2: Write data with size L_{c} of request to cluster p
3: $O += L_{\mathrm{c}}$
4: $L_{\mathrm{r}} -= L_{\mathrm{c}}$
// Pointing to the next available cluster in buffer
5: $p = p \rightarrow$ next
6: $L_{\mathrm{c}} =$ available_Length(p)
// Destaging processing
7: **if** $p =$ NULL **then**
8: Calculate the recently least visited cluster p using the LRU algorithm
9: Split cluster p into m chunks and calculate k parities using Reed-Solomon coding
10: Write the data into the log_cluster list
11: **end if**
12: **end while**
13: Write data with a size of L_{r} of request into cluster p

Since RERAID writes new data in the write buffer and code cache, the system must ensure that the reads are always of the up-to-date versions after a clean shutdown or a system crash. RERAID implements a persistent log_table, which is updated whenever RERAID is reconfigured or a new map entry is added. To minimize the write cost, we do not maintain dirty data when flushing disks. After an unclean shutdown, all entries in the persistent map are marked as dirty and the I/O operations to these blocks are directed to RERAID.

The log_table mentioned above is a structure that saves the RERAID metadata information. It is a concrete realization of the system that conserves disk space in a hash table. The hash table will remain in memory and will be written separately into the corresponding cluster in the code cache; thus, the submitted requests can safely remain in the code cache and do not lose the primary destination address. Since metadata is stored not only in the memory log_table, but also separately in every cluster on the disks, the system can rebuild the log_table to recover the updated data after a crash.

The log_table in memory uses hashing search to provide an efficient block query service. It can quickly inform the system whether a block has been fetched into memory or not. Furthermore, using the memory based log_table will reduce update costs. Since metadata is visited and updated frequently, avoiding metadata operations on disks will reduce I/O load and improve performance.

The log_table of RERAID is shown in Fig. 3 for a sequence of I/O requests accessing data blocks at addresses addr[1], addr[2], $\cdots$, addr[i], where $i > 1$ is the sum of the number of requests in the cluster. To save space, we take 1 kb space from the cluster to construct a log_table to record the location of each request. A logical block addressing (LBA) table indicates the location of each request block so that queries can be made easily. We define the data structure with an offset of 2 bytes, a length of 2 bytes, and an LBA of 6 bytes. In this arrangement, we are guaranteed to have enough space to point to the requested information's destination address. Each block's information is stored in one address. Some blocks may be very large and the others may be smaller because the size of the request is not defined. If there is enough space in the cluster, the next request will join this cluster. Otherwise, we will create a new cluster to write the request.

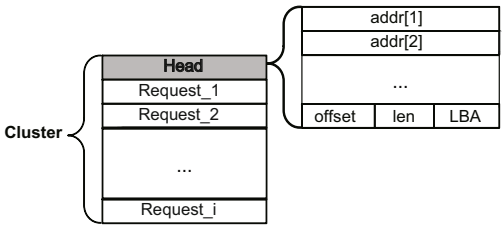

Fig. 3 The log_table of RERAID

When a system crash occurs, the system will search the log_table to update the data. Then, we can retrieve the lost data from LBA. LBA will be updated when a new request arrives. Meanwhile, the recovered data can be written to all the replicas.

3.4 Flush disk

3.4.1 Idea of flush disk

As mentioned in Section 3.1, we need to migrate data from the code cache in the primary replica to the data area of the primary replica and the second replica (or more if available). Also, we will write the data in MRAM to disks. Before that, we should merge the up-to-date data in MRAM with the data in the code cache (if they are related) and then write the merged data to the data area of all the replicas

in parallel. Then, we will delete the data in the code area and MRAM after we have completed the migration. We call this operation 'flush disk (destage)'. The principle of flush disk is described as follows:

1. Release all the code cache space. Once we wake up the second replica, we should flush all the data in the code cache as quickly as possible to all replicas, including the primary replica. After flushing the disks, we restore the sleep state for the non-primary replicas.

2. Flush the disk for the best performance. The code cache is organized in the form of clusters, which are divided into small blocks. Data is written to all the replicas by the address. The written data in a cluster is dispersed in different locations; as a result, if we use the general algorithm, the writing will be inefficient. Thus, we should flush disks according to stripes in replicas. If we can decrease the number of writes, then the write performance will be better.

From the analysis above, we design a minimum correlation flush disk (MCFD) algorithm, which repackages the stripes in accordance with the correlation of the different clusters in the code cache. The idea of the flush disk is shown in Fig. 4.

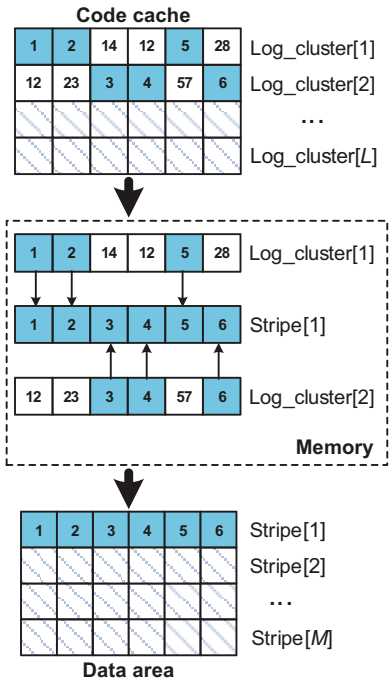

Fig. 4 The destaging process of RERAID

At first, we suppose that there are L clusters in the code cache. As shown in Fig. 4, the data is stored in log_clusters 1 to L. If we construct the stripe in the flush disk, we can obtain data from correlative log_clusters that have blocks belonging to the same stripe. Then, we can write the stripe to the replicas. Here, we suppose that both the log_cluster and stripe have six blocks. Applying the write logging system, data can be logically arranged in one stripe of the replica, and stored in separated clusters in the code cache. To flush that stripe to disks, the system needs to read correlative log_clusters to memory, restructure the whole data of the stripe, and write it to replicas.

3.4.2 MCFD algorithm

Before we discuss the process of the flush disk, we give the definition of correlation:

Definition (Correlation) As part of the data in a log_cluster in the code cache will be stored in a stripe in the data area of the disk, we say the stripe having some data of the log_cluster is correlative with the log_cluster and the correlation between them is 1.

For example, as shown in Fig. 4, log_cluster 1 has blocks named 1, 2, 14, 5, 12, and 28, and blocks 1, 2, and 5 belong to stripe 1. In this case, we say that there is a correlation between log_cluster 1 and stripe 1, and the value is 1; otherwise, the value of correlation is null. When the log_cluster has been read into memory, we will change the available value of log_cluster from 1 to 0.

To construct a stripe, the related log_clusters should be loaded into memory first. A simple solution is to load every log_cluster from the code cache into memory. However, since the code cache may be large, the memory requirement could be a challenge. So, we use a dynamic release solution to solve the problem. In our algorithm, allocated memory can store only part of the code cache. Every time MCFD memory has a free space, it locates the stripe with the lowest correlation, reads its correlative log_clusters into memory, and constructs the stripe in memory. When log_clusters related to the stripe are all read into memory, the constructed stripe will be written to replicas, and then the corresponding blocks of these log_clusters in memory will be released. This process is repeated until every log_cluster is disposed, which makes it possible to run MCFD with the minimal memory requirements, and MCFD is as efficient as fully loading the whole data in the code cache. The following describes an example of MCFD:

1. Calculate the correlation table. To construct a stripe, we construct the correlation table among stripes and log_clusters. We give the correlation table among clusters and stripes of this example in Table 1. The columns are stripes 1 to 8 and the rows are log_clusters 1 to 7. The values in the last row show the number of correlated log_clusters to each stripe.

2. Construct the first stripe with related clusters. From Table 1, we can see that stripe 8 has the lowest correlation to the log_clusters. So, we choose stripe 8 and the correlated log_cluster, log_cluster 2. We read log_cluster 2 into memory, construct stripe 8, and write it to all the replicas. After stripe 8 is flushed, the memory-cached data blocks related with stripe 8 in log_cluster 2 will be freed, and the correlated value will be changed to null. This is the first step of flush disk. After these operations, we change the correlation table. We erase the column of stripe 8 and change the available value of log_cluster 2 from 1 to 0, since we have read log_cluster 2 into memory. When all the blocks' values of a log_cluster are null (not 0), we will delete the log_cluster. At the same time, the value of the last row will be changed to show the new correlation in total. The table is now smaller as shown in Table 2.

3. Choose the second stripe. After retrieving the first stripe and correlated log_clusters, we check the correlation values. Now, we find that stripe 3 has the lowest correlation value. So, we choose stripe 3 and the correlation log_cluster 5. Then we perform the same operations as stripe 8 and log_cluster 2. Thus, we can erase the row of log_cluster 2.

4. Obtain other stripes. Repeat steps 1–3. Then, we will obtain other stripes and flush them one by one.

It is possible that many log_clusters and stripes may be visited, but each stripe is associated only with a few of the log_clusters. As a result, many correlations between log_clusters and stripes are null. To save storage space, a cross list storage technology that uses a sparse array is adopted.

3.4.3 Read and write when flushing disk

As we have already activated all the replicas when flushing disks, we can write and read them directly. In addition, we will write the data cached in MRAM (if available) to all the replicas, but some up-to-date data may still be in MRAM for a short time, so we should take this scenario into account, even though its possibility of occurring is low. Given all that, here we describe read and write processes in detail when flushing disks:

1. Read. When a read request first goes to the MRAM, if a read miss occurs, the request will refer to the code cache; if this request misses in the code cache, it will be redirected to all the replicas in parallel to obtain the required data.

2. Write. When a write request comes, as the up-to-date data in MRAM will also be flushed to the disks, we can just store the write request in MRAM first and wait for the destage.

Table 1 Correlation between log_clusters 1–7 and stripes 1–8

Log_cluster	Correlation							
	1	2	3	4	5	6	7	8
1	1	1		1			1	
2			1					1
3		1		1	1	1		
4		1		1			1	
5			1	1		1		
6	1			1				
7		1			1		1	
Sum	2	4	2	5	2	2	3	1

Table 2 New correlation between log_clusters 1–7 and stripes 1–7

Log_cluster	Correlation						
	1	2	3	4	5	6	7
1	1	1		1			1
2			0				
3		1		1	1	1	
4		1		1			1
5			1	1		1	
6	1			1			
7		1			1		1
Sum	2	4	1	5	2	2	3

4 Prototype and implementation

4.1 Prototype system

Our prototype system runs on a Linux server (Fig. 5). The module between the iSCSI-target module and multiple-device (MD) module is the RERAID module that we designed, and it is responsible for energy saving for the multi-mirror system. For simplicity, we use RAID10 as a two-mirror system to realize the RERAID system.

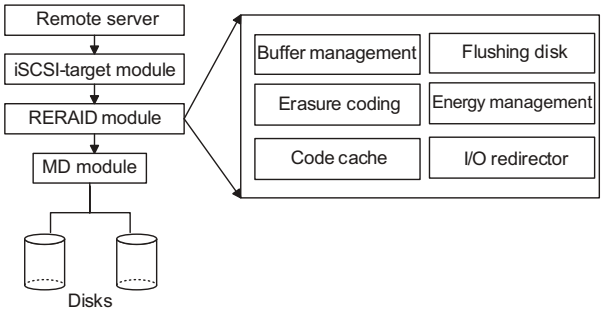

Fig. 5 The prototype of RERAID

The iSCSI-target module is responsible for the transformation of iSCSI and the construction of the target. This module is made by the open source project, iSCSI Enterprise Target (version 0.4.17). The RERAID module keeps the primary replica active, takes part of the primary disks to make up a RAID array, and turns non-primary replicas into standby mode to save energy. When the second replica (or more if available) goes into standby mode, read requests of the mirror disks will be redirected to the matching primary replica, while write requests will be written to the code cache, which will wait until the other replicas switch to an active mode for writing. The MD module manages the disk array and realizes soft RAID with virtual block equipment.

As shown in Fig. 5, there are six modules in the RERAID module, which are the buffer management module, the flushing disk module, the erasure coding module, the energy management module, the code cache module, and the I/O redirector module. The buffer management module controls the write buffer, receives data, and writes data to the code cache. The erasure coding module is used to encode and decode data through the specific code. The code cache module is an important module which deals with all write requests and part of the read requests for the system. When the size of data in the code cache reaches a threshold, the flushing disk module will be invoked to move data with the MCFD algorithm. The energy management module is responsible for controlling the second replica group in standby mode or active mode. To ease the implementation, RERAID keeps the second replica group in standby mode until the code cache is almost full and we have to do the destage. When the destage is finished, RERAID will switch the second replica to standby mode. The I/O redirector module will control the I/O requests to particular replica groups by judging the state of the replica groups.

4.2 Hash table data structure

RERAID uses a hash table to maintain the relationship between the data in the code cache and the data disks (Fig. 6). The main variables are explained below:

1. cluster_id records the cluster index of each block.

2. clu_offset records the block offset in the cluster. It can fix the position of the data in the cache through the cluster index and the block offset of the cluster for a convenient search.

3. Map is an optional unit of eight mappings. Each mapping records a 1-bit data that represents one sector in a block. When the value is 1, the mapping is valid and the data in the represented sector is up-to-date; if the value is 0, the mapping is invalid.

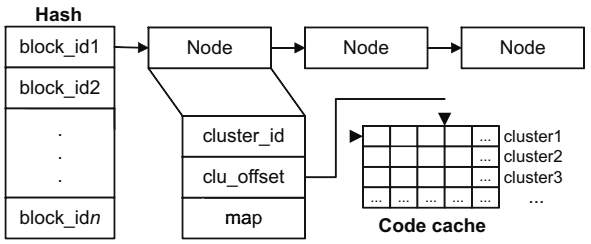

Fig. 6 The data structure of the hash table

5 Reliability analysis

In this section, we will compare the reliability of RERAID with those of GRAID and RAID10. We use the Markov model to analyze the reliability of RERAID. We use the mean time to data loss (MTTDL) as the standard of comparison. At first, we assume temporarily that the failure probability of each disk is independent and obeys the index distribution to establish the model conveniently. We denote μ as the repair rate and λ the failure rate. We suppose $\lambda \ll \mu$ because the mean time between failures is much longer than the mean time to repair a disk. When a disk fails, a repair process is immediately initiated for data recovery.

According to the previous conclusions on the MTTDL of RAID10 (Xin *et al.*, 2003), the MTTDL

of RAID10 consisting of eight disks is

$$\mathrm{MTTDL}_{\mathrm{RAID10-8}} \approx \frac{19\lambda + 2\mu}{8\mu^2}, \tag{1}$$

and the MTTDL of GRAID consisting of eight data disks and one dedicated log disk is (Mao *et al.*, 2008)

$$\mathrm{MTTDL}_{\mathrm{GRAID-9}} \approx \frac{17\lambda + 2\mu}{10\mu^2}. \tag{2}$$

As for RERAID, we take two-disk failure tolerance erasure coding as an example. Fig. 7 shows the simplified state transition probability diagram for a RERAID disk array consisting of four primary data disks and four mirrored data disks. State $\langle 0\rangle$ represents the normal state of the system when all eight disks are operational. In the normal state, four mirrored disks are spun down. It is reasonable to assume that, for simplicity of analysis, the spun down disks are not likely to fail or do not incur data loss even if they do fail. A failure of any of the four primary data disks would bring the disk array to state $\langle 1\rangle$. A failure of a second disk would bring the array to state $\langle 2\rangle$. Any third failure occurring while the array is in state $\langle 2\rangle$ will result in a data loss.

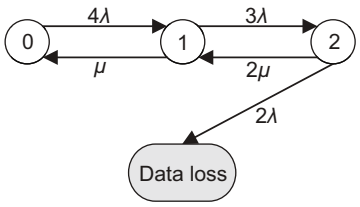

Fig. 7 State transition probability diagram with average disk repair time (day) for a RERAID disk array consisting of four data disks and four mirrored disks

Repair transitions recover the array from state $\langle 2\rangle$ to state $\langle 1\rangle$ and then from state $\langle 1\rangle$ to state $\langle 0\rangle$. The state changing rate is equal to the multiple of the number of disks that fail and the disk repair rate μ.

The Kolmogorov system of differential equations describing the behavior of RERAID is

$$\begin{cases} \dfrac{\mathrm{d}p_0(t)}{\mathrm{d}t} = -4\lambda p_0(t) + \mu p_1(t), \\ \dfrac{\mathrm{d}p_1(t)}{\mathrm{d}t} = -(3\lambda + \mu)p_1(t) + 4\lambda p_1(t) + 2\mu p_2(t), \\ \dfrac{\mathrm{d}p_2(t)}{\mathrm{d}t} = -(2\lambda + 2\mu)p_2(t) + 3\lambda p_1(t), \end{cases} \tag{3}$$

where $p_i(t)$ $(i = 1, 2)$ is the probability that the system is in state $\langle i\rangle$ with the initial conditions $p_0(0) = 1$ and $p_i(0) = 0$. The Laplace transforms of these equations are

$$\begin{cases} sp_0^*(s) - 1 = -4\lambda p_0^*(t) + \mu p_1^*(t), \\ sp_1^*(s) = -(3\lambda + \mu)p_1^*(t) + 4\lambda p_1^*(t) + 2\mu p_2^*(t), \\ sp_2^*(s) = -(2\lambda + 2\mu)p_2^*(t) + 3\lambda p_1^*(t). \end{cases} \tag{4}$$

Observing that the MTTDL of the array is given by (Xin *et al.*, 2003)

$$\mathrm{MTTDL} = \sum_i p_i^*(0), \tag{5}$$

we solve the system of Laplace transforms for $s = 0$ and use this result to compute the MTTDL of our system:

$$\mathrm{MTTDL}_{\mathrm{RERAID}} = \frac{13\lambda^2 + 5\lambda\mu + 4\mu^2}{12\lambda^2(\lambda + \mu)}. \tag{6}$$

Fig. 8 shows MTTDL as a function of MTTR (mean time to repair) for RERAID, RAID10, and GRAID. The disk failure rate λ is assumed to be one failure every one hundred thousand hours, which is slightly less than one failure every eleven years. These values come from the failure rates observed by Pinheiro *et al.* (2007). MTTR is expressed in days and MTTDL in years. Fig. 8 shows that MTTDL of RERAID is slightly better than those of RAID10 and GRAID. As the disk repair time increases, MTTDL of RERAID will be almost equal to those of RAID10 and GRAID. When the total number of disks increases, we can obtain similar results. As to ERAID, the recently up-to-date data will be lost, even if just one disk fails. Thus, compared with these architectures, our RERAID is more reliable.

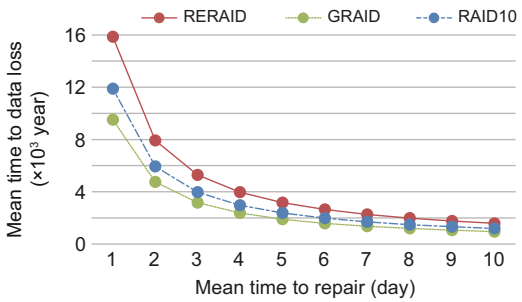

Fig. 8 Mean time to data loss achieved at different disk array levels

6 Evaluation

6.1 Experimental setup

In the experiments, we test the performance and energy consumption when $k = 2$ in erasure coding, comparing RERAID with GRAID, RAID10, and ERAID. These four configurations, which are evaluated and compared in detail, are:

1. RERAID: We implement RERAID with four primary disks and four mirrored disks, like RAID10. Each primary disk of RERAID reserves 1 GB (or more) to perform as a code cache, using erasure coding and log technologies.

2. GRAID: We implement GRAID with four primary disks, four mirrored disks, and one log disk.

3. ERAID and RAID10: We implement RAID10 and ERAID with four primary disks and four mirrored disks.

When receiving requests under the energy-efficient state, there is no operation regarding flushing the disk. There are four disks running in our system, five disks running in GRAID, eight disks running in RAID10, and four disks running in ERAID. When flushing the disk, all the disks will run. The performance evaluation is conducted on a platform of server-class hardware with an Intel® Xeon® 5110 1.60 GHz CPU and 2 GB DDR RAM. In the system, the disk module is a 500 GB hard disk with the Fedora Linux 8-i386 operating system and a 1000 Mb/s Ethernet card.

We use a portable wave analysis device (ZH-102) to test the energy consumption. The storage server, wave analysis device, and client framework are shown in Fig. 9. The AC to DC module converts alternating current to direct current, and from this module, we can obtain 5 and 12 V power supplies for the disks. Two lines connect the wave analysis device with the 5 V line and 12 V line, separately. When testing, through the Ethernet, the wave analysis device sends data acquired from the 5 and 12 V lines to a client. The client records the current value every second. Thus, we can calculate the change of power consumption.

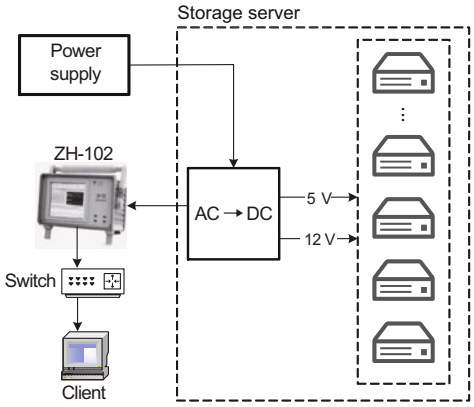

Fig. 9 The experimental pro of RERAID

We use four '.spc' traces as load to evaluate the performance and energy consumption. They are Rsrch, Web, Mds, and User, collected from enterprise servers at Microsoft Research Cambridge (Table 3). On the client, we use the trace tool btreplay to replay the trace. Through the replay, the request in the trace is sent to RERAID in the form of iSCSI request.

Table 3 Features of traces

Trace	Write ratio	IOPS	ARS (KB)	Source
User	59%	83.87	22.66	User home directories
Web	70%	50.32	14.99	Web SQL server
Mds	88%	18.41	9.19	Media server
Rsrch	91%	21.17	8.93	Research projects

IOPS: input/output operations per second; ARS: average required size

6.2 Energy conservation

For our RERAID, there are two modes: high-power mode and low-power mode. In low-power mode, half of all the disks are spun down to conserve energy and code caches are used for logging; we refer to this period as the logging period. In high-power mode, all disks are active and the data in the code cache is destaged to corresponding disks; we refer to this period as the destaging period. As shown in Fig. 10a, under our four traces, the logging period and destaging period are interleaved and the duration of the destaging period is far shorter than that of the logging period. We show the number of destagings and the destaging period ratio when the code cache is 1 GB in Table 4. We have run the entire period but the rest looks the same, so we give only a 60-h picture here.

Fig. 10b shows the observed power consumption distributions of web workloads. First, RERAID consumes less energy than GRAID and RAID10. RAID10 will never spin down partial disks for energy conservation and GRAID uses an extra disk for

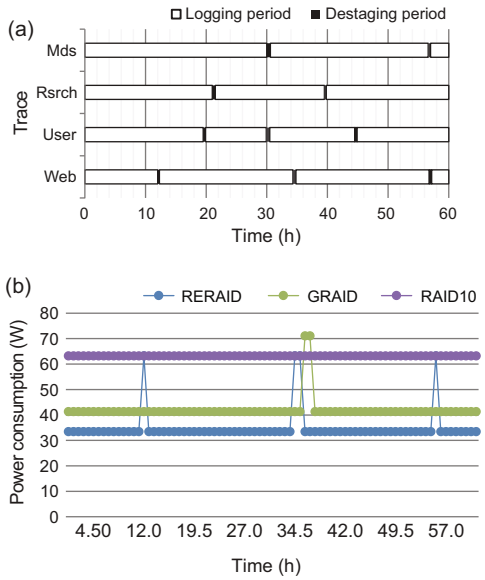

Fig. 10 Observed power consumption characteristics of RERAID: (a) alternated logging period and destaging period under different traces; (b) observed power consumption of web workload

Table 4 Destage of RERAID

Trace	Number of destagings	Ratio of time spent in destaging
Web	13	6.36%
User	11	4.65%
Mds	11	3.57%
Rsrch	7	2.19%

logging, so GRAID and RAID10 will consume more energy than RERAID in either the destaging period or the logging period. Second, we find that the time spent on destaging is relatively short, so it will not obviously impair the total performance.

We use the four workloads to compare RERAID, GRAID, and RAID10 (Fig. 11). First, we can see that RERAID has the best energy conservation. Compared with RAID10, RERAID saves 44.4% energy. This is because RERAID spins down partial disks for energy conservaiton in the logging period, but RAID10 keeps all disks active in both the desating period and the logging period. Compared with GRAID, RERAID saves 16.7%, as GRAID activates one more disk compared with RERAID. In addition, we find that the saved energy ratios are similar under various traces, and this is because most of time, partial disks are spun down. As shown in Table 4, the time spent in the destaging period is less than 6.36%.

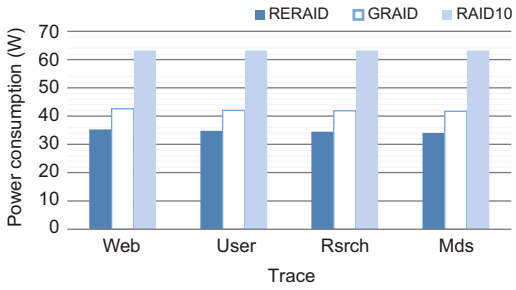

Fig. 11 Total energy consumption under different traces

6.3 Overall performance

Fig. 12 compares the average response time of RERAID with those of GRAID, ERAID, and RAID10 using the four traces. First, RERAID performs better than the others under Web, Mds, and Rsrch, because RERAID exploits the ECW algorithm and code cache area to improve write performance, and Web, Mds, and Rsrch are write-intensive workloads (Table 3). For the trace User, where the read ratio is the highest, the performance of RERAID is the worst. We know that there is a log disk in GRAID and ERAID and they write to the log disk and the primary disk group synchronously, but RERAID writes only to the code cache (log area). Second, we can see that RAID10 performs best, because it opens eight disks and provides good parallelizing reads and writes.

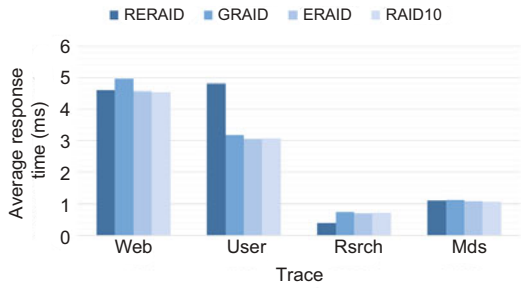

Fig. 12 Total average response time of different traces

6.4 Effect of erasure coding write

Fig. 13 shows the response time under different traces for read and write. As shown, RERAID has the worst read average response time and the best write average response time, compared with GRAID, ERAID, and RAID10. This is because writing to code cache sequentially is faster than writing to the data disk randomly, but reading from the code

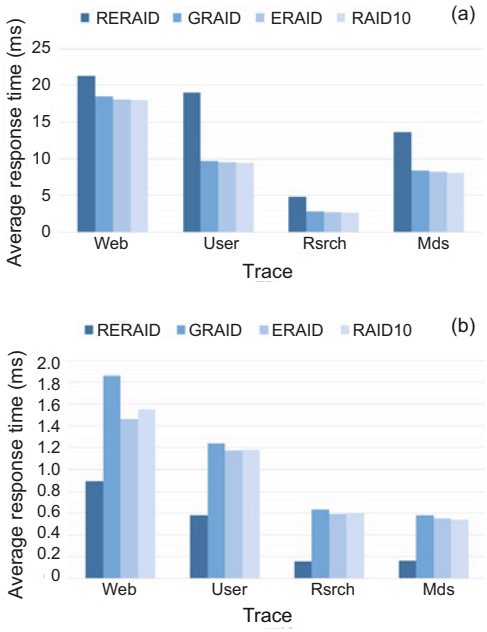

Fig. 13 Average response time under different traces for read (a) and write (b)

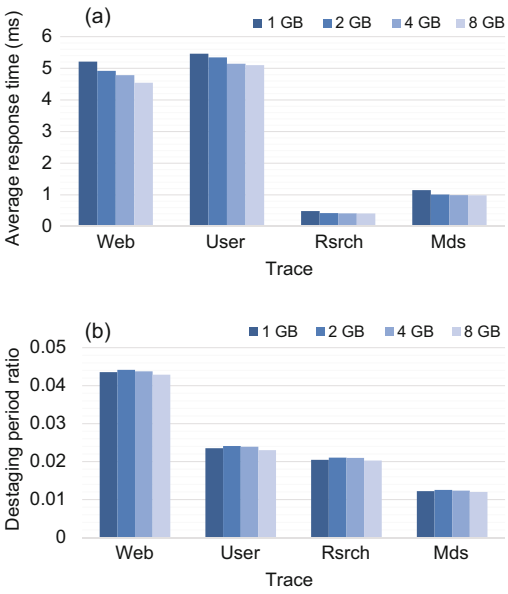

Fig. 14 The effects of code cache size on performance (a) and on the destaging period ratio (b)

cache (logging area) is slower than reading from the disk.

In Fig. 13a, for trace User, the read average response time of RERAID is about twice as long as that of any of the others. The reason is that trace User has the highest read request ratio, 41% (Table 3), and the speed of reading from the code cache is slower than that of reading from the data disk.

In Fig. 13b, RERAID outperforms other systems under various traces. Our erasure coding write combined with the ECW algorithm can optimize small-size writes into large-size writes, which can reduce the number of disk I/O operations to improve write performance.

In conclusion, our RERAID system deals with write requests well, but struggles with read requests.

6.5 Effect of code cache size

We test the effects of code cache size on performance in this section. We set the size of code cache to 1, 2, 4, and 8 GB (Fig. 14).

In Fig. 14a, as the code cache size increases, the overall performance increases slightly. This is because when flushing the disk, the performance decreases. So, when we increase the code cache size, we should decease the destage number for a better performance.

In Fig. 14b, the destaging period ratio is defined as the proportion of the destage time to the total time. Fig. 14b shows that changing the logging space alone will not influence the destaging interval ratio. The reason is that increasing cache size will prolong both the logging and destaging periods, and thus the ratio will not change.

Thus, we can conclude that destage has only a small effect on the performance.

7 Conclusions

In this paper, we developed a high-reliability and energy-efficient storage system named RERAID. It has multiple replicas to provide power proportionally for general reads and writes. It saves energy by spinning down the replica groups without migrating data or imposing extra requirements. Its code cache comprises part of the free space in the primary disk group and uses erasure coding to ensure the reliability of the data. The code cache is also used as a log to receive write requests and can merge many random small writes into a few large writes to improve the write performance when combined with the ECW algorithm. At the same time, it provides a flush disk mechanism to accelerate the flush and permit the operator to choose the right time to flush.

By implementing the system and experiments with real system traces, our results show that

RERAID is effective in improving performance and saving energy without reducing the reliability. The results show that the performance of RERAID is better than those of others presented in this paper, especially when the write ratio in request is more than 80%. When the request is all composed of write requests, RERAID could improve significantly the write performance by 69.1% compared to other energy-saving solutions (e.g., GRAID, ERAID). In addition, it reduces the energy consumption by 44.4% compared to RAID10, and 16.7% compared to GRAID.

There is still much work to do, mainly in two directions. First, we will study how to apply RERAID in distributed storage environments. Second, we can implement the erasure coding with a higher tolerance to failures. The purpose is to find out which combination is the best for performance, energy efficiency, and reliability.

In conclusion, we believe that RERAID is an attractive disk array design and RERAID offers high performance, while achieving significant energy-saving performance.

References

Amazon, 2007. Amazon S3: Object Storage Built to Store and Retrieve Any Amount of Data from Anywhere. http://aws.amazon.com/s3/

Amur, H., Cipar, J., Gupta, V., *et al.*, 2010. Robust and flexible power-proportional storage. Proc. 1st ACM Symp. on Cloud Computing, p.217-228. https://doi.org/10.1145/1807128.1807164

Bhadkamkar, M., Guerra, J., Useche, L., *et al.*, 2009. BORG: Block-reORGanization for self-optimizing storage systems. Proc. Usenix Conf. on File and Storage Technologies, p.183-196.

Blaum, M., Brady, J., Bruck, J., *et al.*, 1994. EVENODD: an optimal scheme for tolerating double disk failures in RAID architectures. Proc. 21st Int. Symp. on Computer Architecture, p.245-254. https://doi.org/10.1109/isca.1994.288145

Borthaku, D., 2010. What is Apache Hadoop? http://hadoop.apache.org/

Chen, Y., Hsu, W., Young, H., 2000. Logging RAID—an approach to fast, reliable, and low-cost disk arrays. Euro-Par, p.1302-1312. https://doi.org/10.1007/3-540-44520-x_182

Colarelli, D., Grunwald, D., 2002. Massive arrays of idle disks for storage archives. Proc. ACM/IEEE Conf. on Supercomputing, p.1-11. https://doi.org/10.1109/sc.2002.10058

Corbett, P., English, B., Goel, A., *et al.*, 2004. Row-diagonal parity for double disk failure correction. Proc. 3rd USENIX Conf. on File and Storage Technologies, p.1-14.

Department of Energy, 2012. NETL shares computing speed, efficiency to tackle barriers. *Fossil Energy Today*, **1**(6):1-3.

EMC, 2008. ATMOS: Big. Smart. Elastic. http://www.emc.com/storage/atmos/atmos.htm

Eom, H., Hollingsworth, J.K., 2000. Speed vs. accuracy in simulation for I/O-intensive applications. Proc. 14th Int. Parallel and Distributed Processing Symp., p.315-322. https://doi.org/10.1109/ipdps.2000.846001

Ghemawat, S., Gobioff, H., Leung, S., 2003. The Google File System. Proc. 19th ACM Symp. on Operating Systems Principles, p.29-43. https://doi.org/10.1145/945445.945450

Hu, Y., Yang, Q., 1996. DCD—disk caching disk: a new approach for boosting I/O performance. *ACM SIGARCH Comput. Archit. News*, **24**(2):169-178. https://doi.org/10.1145/232974.232991

Li, D., Wang, J., 2004. EERAID: energy-efficient redundant and inexpensive disk array. Proc. 11th ACM SIGOPS European Workshop, p.29. https://doi.org/10.1145/1133572.1133577

Li, D., Wang, J., 2006. eRAID: a queuing model based energy saving policy. Proc. IEEE Int. Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, p.77-86. https://doi.org/10.1109/mascots.2006.23

Lu, L., Varman, P.J., Wang, J., 2007. DiskGroup: energy efficient disk layout for RAID1 systems. Proc. Int. Conf. on Networking, Architecture, and Storage, p.233-242. https://doi.org/10.1109/nas.2007.21

Mao, B., Feng, D., Jiang, H., *et al.*, 2008. GRAID: a green RAID storage architecture with improved energy efficiency and reliability. Proc. IEEE Int. Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, p.113-120. https://doi.org/10.1109/mascot.2008.4770574

Menon, J., 1995. A performanee comparison of RAID-5 and log-struetured arrays. Proc. 4th IEEE In. Symp. on High Performance Distributed Computing, p.167-178. https://doi.org/10.1109/hpdc.1995.518707

Patterson, D.A., Gibson, G., Katz, R.H., 1988. A case for redundant arrays of inexpensive disks (RAID). Proc. ACM SIGMOD Int. Conf. on Management of Data, p.109-116. https://doi.org/10.1145/971701.50214

Plank, J.S., Xu, L.H., 2006. Optimizing Cauchy Reed-Solomon codes for fault-tolerant storage applications. Proc. 5th Int. Symp. on Network Computing and Applications, p.173-180. https://doi.org/10.1109/nca.2006.43

Pinheiro, E., Bianchini, R., 2004. Energy conservation techniques for disk array-based servers. Proc. 18th Annual Int. Conf. on Supercomputing, p.68-78. https://doi.org/10.1145/1006209.1006220

Pinheiro, E., Weber, W.D., Barroso, L.A., 2007. Failure trends in a large disk drive population. Proc. 5th USENIX Conf. on File and Storage Technologies, p.17-28.

Soundararajan, G., Prabhakaran, V., Balakrishnan, M., *et al.*, 2010. Extending SSD lifetimes with disk-based write caches. Proc. 8th USENIX Conf. on File and Storage Technologies, p.101-114.

Stodolsky, D., Gibson, G., Holland, M., 1993. Parity logging overcoming the small write problem in redundant disk arrays. *ACM SIGARCH Comput. Architect. News*, **21**(2):64-75. https://doi.org/10.1145/173682.165143

Thereska, E., Donnelly, A., Narayanan, D., 2011. Sierra: practical power-proportionality for data center storage. Proc. 6th Conf. on Computer Systems, p.169-182. https://doi.org/10.1145/1966445.1966461

Wang, J., Zhu, H.J., Li, D., 2008. eRAID: conserving energy in conventional disk-based RAID system. *IEEE Trans. Comput.*, **57**(3):359-374. https://doi.org/10.1109/tc.2007.70821

Weil, S., Brandt, S.A., Miller, E.L., *et al.*, 2006. Ceph: a scalable, high-performance distributed file system. Proc. 7th Conf. on Operating Systems Design and Implementation, p.307-320.

Wilkes, J., Golding, R., Staelin, R., *et al.*, 1996. The HP AutoRAID hierarchical storage system. *ACM Trans. Comput. Syst.*, **14**(1):108-136. https://doi.org/10.1145/225535.225539

Xin, Q., Miller, E.L., Schwarz, T., *et al.*, 2003. Reliability mechanisms for very large storage systems. Proc. 20th IEEE/11th NASA Goddard Conf. on Mass Storage Systems and Technologie, p.146-156. https://doi.org/10.1109/mass.2003.1194851

Xu, L.H., Bruck, J., 1999. X-code: MDS array codes with optimal encoding. *IEEE Trans. Inform. Theory*, **45**(1):272-276. https://doi.org/10.1109/18.746809

Yue, Y., Tian, L., Jiang, H., *et al.*, 2010. RoLo: a rotated logging storage architecture for enterprise data centers. Proc. IEEE 30th Int. Conf. on Distributed Computing Systems, p.293-304. https://doi.org/10.1109/ICDCS.2010.22

Yue, Y., He, B., Tian, L., *et al.*, 2016. Rotated logging storage architectures for data centers: models and optimizations. *IEEE Trans. Comput.*, **65**(1):203-215. https://doi.org/10.1109/tc.2015.2417539

Zhu, Q., Chen, Z., Tan, L., *et al.*, 2005. Hibernator: helping disk arrays sleep through the winter. Proc. 20th ACM Symp. on Operating Systems Principles, p.177-190. https://doi.org/10.1145/1095809.1095828