ESet: Placing Data towards Efficient Recovery for Large-scale Erasure-Coded Storage Systems

Chengjian Liu*, Xiaowen Chu*†, Hai Liu* and Yiu-Wing Leung*
*Department of Computer Science, Hong Kong Baptist University
Kowloon Tong, Hong Kong
† Institute of Research and Continuing Education, Hong Kong Baptist University
Shenzhen, China
Email:{cscjliu,chxw,hliu,ywleung}@comp.hkbu.edu.hk

***Abstract*—Erasure coding has been extensively deployed in distributed storage systems to ensure high reliability and low storage overhead. However, erasure coding requires much more disk I/O to recover a damaged data block than replication does, resulting in very long data recovery time. Data placement algorithm can be tailored to speed up data recovery process by exploiting I/O parallelism. However, existing algorithms that obtain good I/O parallelism for replication can not directly work with erasure-coded storage systems; and other algorithms for both replication based and erasure-coded storage systems overlook the importance of recovery I/O parallelism, which may jeopardize the service quality and reliability of these systems.**

In this paper, we present a data placement strategy named *ESet* which brings recovery efficiency for each host in a distributed storage system. We define a configurable parameter named *overlapping factor* for system administrator to easily achieve desirable recovery I/O parallelism. Our simulation results show that *ESet* can significantly improve the data recovery performance without violating the reliability requirement by distributing data and code blocks across different failure domains.

I. Introduction

In recent years, the amount of data stored in data center storage systems has already reached PB-level and still keeps increasing. E.g., in 2015, European Centre for Medium-Range Weather Forecast reported that its storage capacity reached 100 PB and experienced an annual growth rate of 45% [1]. For storage systems at such a large scale, data replication as a reliability mechanism brings very high storage cost. Many storage systems have begun to adopt erasure coding to prevent data loss [2][3] and meanwhile reduce storage overhead. E.g., HDFS[4][5] and Azure [6] have already supported erasure coding. An erasure code takes two integer parameters *n* and *k*, $n > k$, where k data blocks are coded into n blocks such that any k out of the n blocks can be used to recover the original k data blocks. So the storage overhead with (n, k) code is defined by *n/k*. For instance, Facebook transfers its blob storage from Haystack [7] to F4 [2] with $n = 14$ and $k = 10$, which reduces its storage overhead from 3.6x to 1.4x.

Many recent studies have revealed that data centers are frequently suffering disk-level [8][9] and host-level [10][11] failures[12][13][14][15]. This makes data recovery part of daily work for storage systems. For example, Facebook warehouse transfers around 100 TB of data each day to recover data from its failed disks and hosts [3].

Data placement scheme plays a critical role in erasure-coded storage systems. First of all, the data placement scheme should guarantee very high data reliability and availability that can withstand a certain level of disk failures, host failures, and rack failures. Secondly, it should achieve a desirable data recovery throughput. However, recovering a failed host or disk is very time consuming for erasure-coded storage systems[16]. Using the popular Reed-Solomon code as an example, to recover an individual data block requires to fetch a total of *k* blocks. Consider a scenario that a data center needs to recover a failed disk with 1 TB of data, and $k = 10$. Then it requires to access 10 TB of data to recover the failed disk. Assume the disk I/O throughput is 100 MB/s, then the recovery time will be at least 10^5 seconds if the 10 TB of data blocks can only be accessed sequentially. To reduce the recovery time, it is imperative to distribute those 10 TB of data blocks among many different disks at the very beginning, and hence we can exploit parallel disk I/O to speed up the data recovery process. Another approach to reducing data recovery time is to design new erasure code that requires less data to recovery a failed data block. In this paper, we focus on exploiting I/O parallelism.

A simple yet popular data placement scheme is random data placement, which distributes data and code blocks randomly. This makes storage systems able to aggregate I/O from more than *k* hosts or disks to recover a failed host or disk. However, the recovery performance may not be always guaranteed. E.g., Facebook's F4 took two days to recover the data on a host as the data and code blocks for the failed hosts are not well distributed [2]. Another side-effect is that, during the recovery period (a.k.a. degraded read), service latency can become 10x than normal mode and service quality decreases greatly.

To guarantee recovery performance for each host, a data placement algorithm must ensure that adequate hosts can participate in the recovery process. In this paper, we present a data placement scheme named *ESet* that can fulfil the fundamental data reliability requirement, and meanwhile achieve desirable data recovery performance. Our major contributions can be summarized as follows:

(1) We designed a data placement algorithm named *ESet* for bringing efficient data recovery in large-scale erasure-coded storage systems.

978-1-5090-2279-3/16/$31.00 ©2016 IEEE

(2) *ESet* conforms to reliability requirements of large-scale storage systems.
(3) We conducted extensive simulation studies and illustrated how *ESet* improves data recovery performance over existing random data placement algorithms.

The rest of this paper is organized as follows. We first introduce related works in section II. In section III, we present the formulation of the data placement problem. Section IV presents the data reliability analysis that leads to our design of *ESet*. Section V gives the detailed design of our placement algorithm. Section VI provides the performance evaluation. The last section draws the conclusion of our work.

II. Related Work

The essential goal of a storage system is to store data and provide quality of service for data access. When a storage system scales to more than thousands of hosts and tens of thousands of disks, failures are daily happened events. To prevent data loss caused by failures, data recovery is being performed everyday. During the data recovery process, the accesses to the lost data (called degraded reads) suffer long latency due to data decoding operations. This makes recovery performance play a crucial role for storage system to keep quality of service. A report [17] revealed that after a major failure event, some 60% needs more than 4 hours to restore its service due to necessary recovery. And around 20% needs more than a whole day to come back to normal service. This indicates that a storage system must take recovery performance into consideration when making data layout optimization, otherwise the system service quality would be heavily damaged.

Recovery performance is an essential design goal for data placement algorithm[18]. Consider a storage system with *m* disks, when a disk fails, the recovery performance is optimal if *m*-1 disks can participate in its recovery and each disk contributes equal I/O for recovering the failed disk.

For replication based storage systems, stein system can be applied to achieve optimal recovery performance for a certain number of hosts. The optimal solution is achieved for recovery when some condition are satisfied [19]. However, it only works with some special numbers of hosts [20][21]. Thus, researchers are trying to design near-optimal stein system. A near optimal parallelism is proposed in [18] for storage systems with a few disks. In [22], it gives a data placement algorithm for replication to obtain optimal parallelism for replication for disks.

Copysets [23] addresses the issue of data loss for replication based storage systems by designing near-optimal stein system. It uses the scatter width to represent how many I/O parallelisms a host can obtain for its recovery. It selects hosts from different racks to form a group to avoid concurrent failures in the same group. Some permutations are made in the group and it makes each host distribute its replicates across hosts in the group so that each host can obtain near optimal recovery performance. When a host fails, all other hosts may contribute near equal I/O to recover the failed host. However, Copysets is not directly applicable to erasure-coded storage systems, because to restore a data block, only one host is needed for replication, but *k* hosts are required for erasure-coding.

Random data placement algorithms, such as RUSH [24][25], CRUSH [26], Random Slicing [27], can work with both replication based and erasure-coded storage system. These algorithms can guarantee reliability by randomly placing replicates across all failure domains, where each failure domain contains a set of disks or hosts that may become unavailable when a shared component failed. But they mainly focus on how to distribute data evenly on large-scale storage systems. Distributing data to obtain good recovery performance for each host is overlooked. Intuitively the number is large enough to achieve good recovery performance. However, due to randomness, the number of hosts participating in recovering a failed host can not be configured. And the recovery performance of each host may not satisfy the requirement of storage systems. Some storage systems map each host to many virtual storage nodes to accelerate host's recovery performance. When a host fails, all other hosts can participate in its recovery. However, it makes storage systems easy to suffer data loss when more than *n-k* disks fail concurrently.

In summary, existing data placement algorithms cannot make efficient recovery for erasure-coded storage system meanwhile guarantee system reliability, which would further jeopardize the service quality and reliability of a storage system. This motivates our work to design a placement algorithm for large-scale erasure-coded storage system to bring efficient recovery performance and ensure system reliability.

III. Problem Definition

In this section we present the system model of an erasure-coded distributed storage system, and then formulate the problem that we are trying to resolve.

A. System Model

Fig. 1 illustrates the physical layout of our targeted storage system architecture. It contains α racks denoted by R with subscripts indexed from 0 to $\alpha - 1$. The data center has β hosts in total. We represent them with H with subscripts indexed from 0 to $\beta - 1$. For simplicity, we consider a homogeneous system in this paper, i.e., all hosts are the same and they are evenly distributed in all racks. The whole system contains γ disks and each host carries γ/β disks.

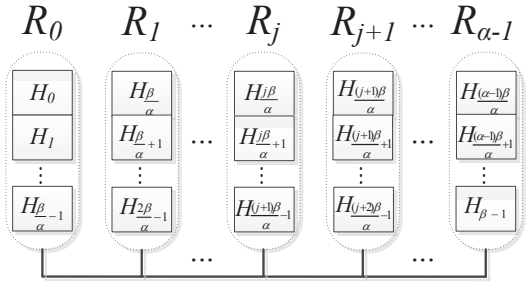


Fig. 1. Physical Layout of Storage System

Erasure Code: We use Reed-Solomon (RS) code as an example of erasure code in this paper. In an (n, k) RS code,

where $n > k$ and $k > 1$, k data blocks are put together to generate $n - k$ parity blocks (or code blocks). When no more than $n - k$ blocks fail in these blocks, any k remaining blocks can be used to restore the missing blocks. Although the encoding/decoding operations involve huge computation, many existing works have resolved this challenge with multi-core CPU and many-core GPU accelerations [28]. Currently, disk I/O overheads are the major challenge for such storage systems.

Stripe: Stripe is the basic unit for managing erasure-coded data. For the case of RS code, a stripe consists of *n* equal size blocks, where *k* blocks are data blocks and the rest *n-k* blocks are parity blocks. An erasure-coded storage system can guarantee its data reliability as long as each stripe has at most *n-k* missing blocks. The data reliability of a distributed system depends on the number of stripes and the data reliability of each stripe, which in turn depends on the values of n and k. Usually there is a trade-off between the reliability and storage overhead n/k.

B. Problem Formulation

Given the above system model, the data placement problem can be converted to the following stripe construction problem:

Stripe Construction Problem: Given a data center with α racks, β hosts, and γ disks, how to construct a total of S stripes such that the data reliability is higher than a predefined threshold and the data recovery performance is maximized. The construction of a stripe implies how to store its n blocks in the set of γ disks.

IV. Reliability Analysis

The essential purpose of erasure coding is to prevent data loss. Thus we first study major failure types in distributed storage systems and analyze how to design a placement algorithm without violating reliability requirement.

A. Revisiting Failures

Data are stored on non-volatile disks. Given a disk, a failure by itself and a failure that makes it unavailable will cause the data become unavailable[29]. Typically three types of failures will cause disks become unavailable: disk-level failure, host-level failure, and rack-level failure that may cause a number of hosts become unavailable.

Many studies have been carried out to investigate disk-level failures. The mean-time-to-failure of a disk is said to be around 1 million hours in [8]. Failure trends for disks are studied in [9]. Their results show that disks from different vintages fail randomly. No obvious clues can be used to predict when a given disk will fail. What we can get is that the annual disk failure rate is around 1%.

Host can fail for various reason. An HDFS based system with 40 PB data in 30 thousand hosts claims that around 24% host failures are caused by software issues such as garbage collection [30]. In [15], it claims that hosts may encounter failures due to restart and unknown reasons. And a failure burst will cause more than ten hosts in some racks become unavailable. It further gives 15 minutes as the time interval for determining if data recovery is necessary for an unavailable host. Facebook warehouse has on average around 50 hosts become unavailable for more than 15 minutes in each day in a month [3]. Thus, data recovery for host failure is the biggest challenge for data centers nowadays.

A whole rack failure is mainly caused by two reasons. First is network fault, which makes a rack unavailable for a while but does not cause data loss. Another one is power outage, where all hosts in the rack are forced to shutdown. A portion of hosts may be unable to restart as revealed in [15][23]. In this paper we will treat a rack failure as multiple host failure events.

B. Our Solution: ESet

***ESet*:** To minimize the probability of data loss, we propose to distribute the *n* blocks from a stripe across *n* distinct hosts. As the number of stripes in a distributed storage system is very large, we introduce the concept of *ESet* to simplify the process of stripe construction. An *ESet* is defined as a logical set of *n* virtual hosts who are carrying a set of stripes together. A virtual host here refers to a portion of the storage space of a physical host. Each stripe must belong to a single *ESet*, and each block of a stripe comes from a different virtual host in its *ESet*. By this design, an *ESet* can be regarded as a failure unit in the storage system. If no more than $n - k$ hosts fail in an *ESet*, there will be no data loss. But if more than $n - k$ hosts fail simultaneously in an *ESet*, then all data in the failed hosts could be lost. As illustrated in Fig. 2, the whole system contains ε *ESet*s, and they are indexed from 0 to $\varepsilon - 1$. Each *ESet* includes *n* distinct virtual hosts denoted as V_0 to V_{n-1}. Each virtual host is mapped from a concrete physical host, and each physical host can reside in different *ESet*s. E.g., in Fig. 2, physical host H_2 appears in three *ESet*s. Now the stripe construction problem is further converted to ESet construction problem, i.e., how to create a set of *ESet*s from the set of physical hosts.

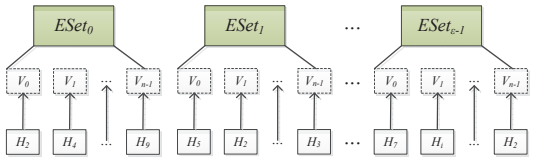


Fig. 2. *ESet*s

Erasure-Coded Storage System with ESet: The abstract view of our proposed erasure-coded storage system with ESet is demonstrated in Fig. 3. Users for the storage system interact with storage system with read service and write service provided by storage system. Logically, the storage system is consisted of ε *ESet*s generated from the physical hosts in the system. Each *ESet* carries the same number of stripes that can be used to store users' data blocks and parity blocks.

The users enjoy the quality of service for read and write services provided by the storage system. They do not need any detailed knowledge of the inside architecture of the storage

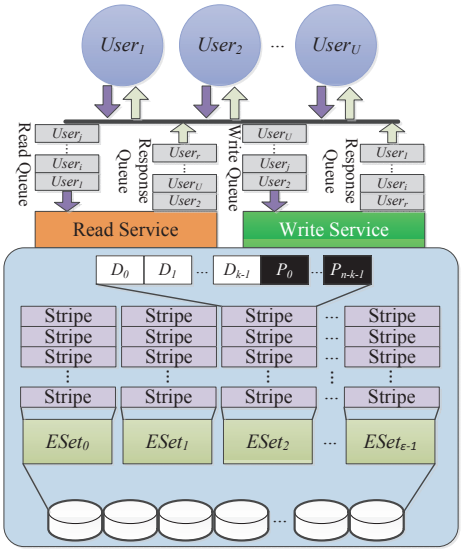


Fig. 3. Distributed Storage System Architecture with *ESet*

system. The read requests from users are queued at the read service and a response queue for responding the read request from users. The situation is similar to the write request and response from the storage system.

Overlapping Factor: Overlapping factor (denoted by symbol o) is a global system parameter that denotes how many *ESet*s a physical host sits it. When a host fails, there will be o sets of hosts, each contains k hosts from the corresponding *ESet*, who can participate in recovering all blocks in the failed host.

Recovery I/O Parallelism: Given a host, its recovery I/O parallelism represents how many sets of k hosts can work in parallel to help recover the host if it failed. We use the symbol O to represent each host's recovery I/O parallelism. O is equal to or small than o, as at most o sets can work in parallel to help recover a failed host.

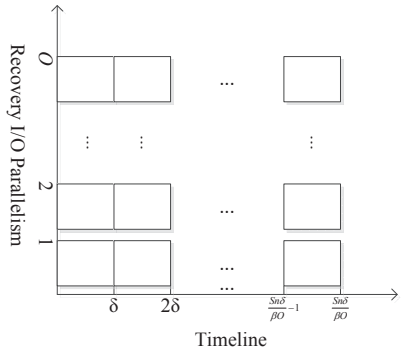


Fig. 4. Time For Recovering a Failed Host

Assume the whole storage system contains S stripes, and each host carries Sn/β blocks. A host resides in o *ESets* and each *ESet* carries $Sn/(\beta o)$ stripes. If a host fails, each of its *ESet*s can select k hosts to recover the $Sn/(\beta o)$ missing blocks. Assume the block size is b bytes and reading k blocks from k hosts requires time δ (assuming that the I/O bandwidth limitation for recovery of each host is b/δ). Fig. 4 illustrates the time for recovering all missing blocks for a failed host. There are O sets of k hosts working in parallel to recover the missing Sn/β blocks for a failed host in parallel.

Missing blocks from a failed host are queued in O queues waiting to recovery. Blocks in the same queue are recovered in sequatial order. Each block take δ time to restore. As the time for decoding can be negligible when adopting PErasure and pipelined with I/O. The time for recovering all blocks from a failed host depends on the time for each queue to restore its all blocks. Thus the recovery time for a failed host can be represented by Eq. (1). Noting that O is equal to or smaller than o, as there are at most o sets of k hosts that can work in parallel for recovering a failed host.

$$t_r = \frac{Sn\delta}{\beta O} \tag{1}$$

Typically β is a fixed parameter for a data center storage system. S is determined by the amount of data stored in the system, n is determined by the erasure code and δ is limited by host I/O bandwidth. To reduce recovery time, we should increase O as much as possible, and recovery throughput will increase linearly with O increased, assuming the network bandwidth is enough.

We have mentioned that traditional placement algorithms mainly focus on how to distributes data evenly. They overlooked the importance of recovery I/O parallelism. O can be very small for a given host. It makes some hosts take days of time to recovery its whole data. Our design includes two parameters: o and O. Administrators can configure o to bring o sets participate in each failed host's recovery. And our Eset construction algorithm tries to make O as close to o as possible, therefore achieving near-optimal or optimal recovery performance.

C. Reliability Constraint

Assume some hosts from the same failure domain appear in the same *ESet*, the set will be prone to data loss if these hosts have high probability to encounter concurrent failures. Many hosts who overlap together for storing replicates or common stripes from different *ESet*s may fail concurrently. If the situation is common, the whole storage system will be prone to data loss. Assume the host failure probability is f. It is easy to observe that the data loss probability p of an *ESet* can be calculated by Eq. (2). If an *ESet* has less than n-k failed hosts, the stripes stored in the *ESet* will not suffer data loss. Here we assume the number of *ESet*s in a system is e, then the data loss probability of the whole storage system $P(S)$ can be calculated by Eq. (3).

$$p = \sum_{i=k}^{n} \binom{n}{i} (1-f)^i f \tag{2}$$

$$P(S) = p^e \tag{3}$$

The number of *ESet*s grow very fast as the number of hosts increases in a large scale storage system. Some works

have shown that the failure probability of hosts in data centers is around 1%. In Fig. 5, we compare three different configurations and demonstrate the reliability of the whole storage system with these configurations. When the size of *ESet*s reaches a million, the system with triplication and the one where *n=14* and *k= 10* can only provide data availability with less than 2-nines. This means data loss will happen in a large-scale storage system with thousands of hosts that stores PB-level data. Glacier provides data availability with 6-nines by decreasing *k*, with the side-effect of higher storage cost.

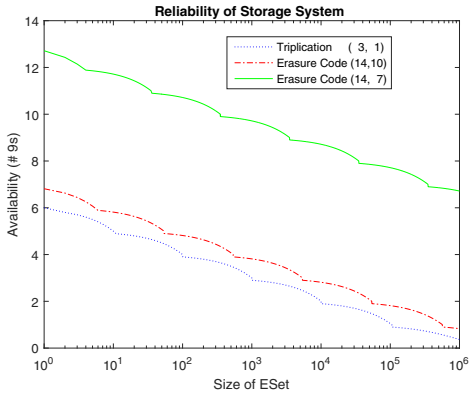


Fig. 5. Reliability of Storage System

As we can see from above, overlapping hosts from the same failure domain will increase the probability of data loss. Assume we have *n* hosts that have no common failure domain, that is *e* is 1, and each host's failure probability is 1%, Fig. 6 presents how many nines we can obtain with different *n* and storage efficiency. Thus to provide high availability, we need to partition hosts into different groups based on their failure domains. And the administrators of storage systems can configure *n* and *k* to fulfill their reliability requirements.

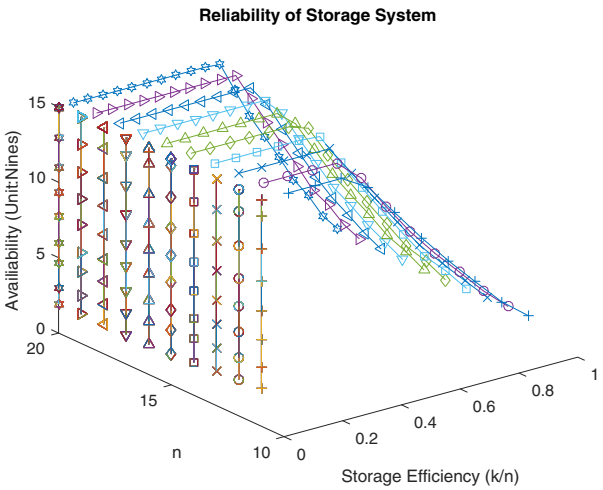


Fig. 6. Reliability From Independent Domain

TABLE I
MAIN NOTATIONS

Operation	Meaning
$F(x)$	Return failure domains of x, where x can be a rack and group
$\|F(x_1) \cap F(x_2)\|$	Return the number of common failure domains of x_1 and x_2
$R[i].W$	The weight of rack i, here used to indicate the number of hosts belongs to it
$Sort(R[])$	Merge racks that has less hosts than o and has most common failure domains. Sort racks in descending order based on their weight.

V. DESIGN OF ESET

In this section, we present the design of our data placement algorithm. First, we show how to select *n* hosts from different failure domains for each *ESet*. And then we present how each host resides in different *ESet* to obtain its desired recovery I/O parallelism.

A. Grouping for Reliability

To form *ESet* with hosts from independent failure domains, we first partition hosts into different groups. For each group, we select *o* hosts to form a row. There are *n* rows in total for each group. Hosts in different rows come from different failure domains. Thus, any *n* hosts come from *n* rows of a group share no common failure domain. The whole system contains *g* groups in total, as illustrated in Fig. 7. Each host inside a group only cooperates with hosts from other rows in its group to form an *ESet*. This can minimize data loss probability caused by each failure domain.

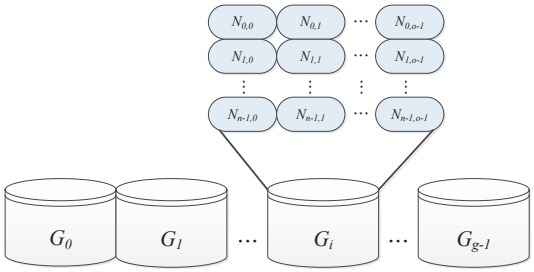


Fig. 7. Groups in System

Before presenting the algorithm of grouping, we give some notations in Table I. We define a function named *F(x)* to return the failure domain of *x*, where *x* can be a group or a rack. We use intersection operation of the failure domains of racks and each group to help us find common failure domains. It helps us select at most *o* hosts from a given rack to join a group as long as the rack has the least common failure domain with the group.

The generation of *g* groups is presented as pseudocode in Algorithm 1. The input contains an array with all racks inside it and the overlapping factor *o*. The first for loop initializes each group and finds *o* hosts from the first rack to form the first row of the group. The first inner loop finds the hosts to form the rest *n-1* rows of the group. The innermost loop

iterates all remaining racks to find a rack that has the least common failure domain with the given group.

Algorithm 1: Generate g *Groups* from α *Racks*

Input: $R[], \alpha, o$
Output: $G_0 ... G_{g-1}$

$RNum = \alpha$
for *i=0 to g-1* **do**
  $G_i = R[0]$
  $R[0].W = R[0].W - o$
  $F(G_i) = F(R[0])$
  for *j=1 to n-1* **do**
    $RPtr = NULL$
    for *l=0 to RNum-1* **do**
      if $R[l] \subseteq G_i$ **then**
        continue
      end
      if $RPtr = NULL$ **then**
        $RackPtr = R[l]$
        continue
      end
      if $|F(G_i) \cap F(RPtr)| > |F(G_i) \cap F(R[l])|$ **then**
        $RPtr = R[l]$
      end
    end
    $G_i = G_i \cup RPtr$
    $R.W = R.W - o$
    $F(G_i) = F(G_i) \cup F(RackPtr)$
    if $RPtr.W == 0$ **then**
      $RNum = RNum - 1$
    end
    $Sort(R[])$
  end
end

B. Generation of ESets

After generating groups, we can build *ESet*s that each has *n* hosts from different failure domains to store stripes. This is achieved by selecting *n* hosts from *n* distinct rows of a group. And it will make each *ESet* able to avoid data loss even when concurrent failures happen.

To put each host in *o ESet*s, we need to generate o^2 *ESet*s in total for each group. We have defined the number of *ESet*s as ε. Its relation to group number and overlapping factor can be depicted by Eq. (4).

$$\varepsilon = go^2 \tag{4}$$

Thus given *i*th group, we can map its hosts to *ESet*s from E_{io^2} to $E_{(i+1)o^2-1}$. Assume the index of an *ESet* is $io^2 + j$, where *j* is a variable with ranged from zero to $o^2 - 1$, we mapped its V_0 to a host from a group's first row and *j/o*th column. For its V_r, where *r* is bigger than zero and smaller than *n*-1, we mapped it to host from *r*th row and *((j/o(r-1)+j)mod o)*th column for *i*th group. The generation process of each *ESet* is showed in Algorithm 2. The algorithm can iterate *g* times to generate ε *ESet*s.

So far, we have illustrated how to generate all *ESet*s. Now we must assure each host appears in *o ESet*s. For hosts in the first row of a group, it is obvious that each host will appear in *o ESet*s. For hosts in other rows in a group, each host will appear in exactly an *ESet* with a host from the first row.

Algorithm 2: Generate Erasure Code Sets for a Group

Input: G_i, o, n
Output: $E_{io^2} ... E_{(i+1)o^2-1}$

$startIdx = io^2$
for *j=0 to* $o^2 - 1$ **do**
  $columnIdx = j/o$
  $E_{startIdx+j}.V_0 = G_i.N_{0,columnIdx}$
  for *r=1 to n-1* **do**
    $columnIdx = ((j/o)(r-1) + j) \ mod \ o$
    $E_{startIdx+j}.V_r = G_i.N_{r,columnIdx}$
  end
end

Because there are *o* hosts in the first row, hosts in other row will appear in *o ESet*s. Thus each host is mapped to *o ESet*s.

C. Recovery of a Failed Host

When a host fails, we need to find all *ESet*s that contain the failed host and use these *ESet*s to recover the failed host. The procedure of locating all *ESet*s contains two steps. First, given the group index of the failed host as *i*, we can calculate that the host is in *o ESet*s from E_{io^2} to $E_{(i+1)o^2-1}$. Then, use the row index *r* and column index *c* for the failed host to locate *o ESet*s that contain the failed host from E_{io^2} to $E_{(i+1)o^2-1}$.

The way of locating *o ESet*s from o^2 *ESet*s depends on the value of *r*. If *r* is equal to zero, *o ESet*s are from E_{io^2+co} to $E_{io^2+(c+1)o-1}$. Otherwise, we need to iterate the *ESet*s index from io^2 to $(i+1)o^2 - 1$ to find out all E_{io^2+j} that satisfy $c = (j/o(r-1) + j) mod \ o$.

After finding all *ESet*s, each set can select *k* hosts for recovering the failed host. In an optimal situation, *o ESets* can work in parallel to recover the failed host, which means *O* is equal to *o* for the failed host. But sometimes some *ESet*s have to work in sequential order for recovering the failed host since some hosts exist in these *ESet*s. Fig. 8 illustrates the situation with the *ESet*s generated by a group with *n*=4 and *o*=2. We set *k*=3. If $N_{0,0}$ fails, E_0 and E_1 can work in parallel for recovering it. But if the failed host is $N_{1,0}$, E_0 and E_2 must work in sequential order for the recovery, as $N_{3,0}$ exist in both sets.

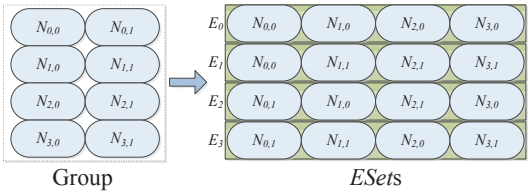


Fig. 8. An example of hosts mapped to *ESet*s

VI. Performance Evaluation

In this section, we evaluate the performance of *ESet* by simulations. Our evaluation comprises five parts: (i) we briefly introduce major parameters for our evaluation; (ii) we analyze the recovery I/O parallelism of *ESet*; (iii) we simulate and evaluate *ESet*'s recovery performance in data center storage

TABLE II
DATA CENTER CONFIGURATION OF SIMULATION

Name	Value	Meaning
α	200	Number of Racks
β	8400	Number of Hosts
γ	84000	Number of Disks
n	14	Number of Blocks Per Stripe
k	10	Number of Data Blocks Per Stripe
b	1000 MB	The Size of a Block
b/δ	100 MB/s	I/O Bandwidth

for a whole year; (iv) we simulate and evaluate *ESet*'s recovery performance under different cumulative failure distribution; (v) we simulate and evaluate *ESet*'s recovery performance with burst failures in an hour.

A. Evaluation Overview

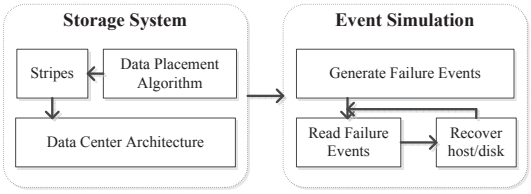


Fig. 9. Simulator Overview

To evaluate recovery performance of *ESet* under different circumstances, we build a simulator for our evaluation. The main components of our simulator are presented in Fig 9. First component is the storage system, which contains a data center architecture configured with parameters in Table II. We generate 40 PB data stored in stripes and evenly distributed on all disks according to a specified data placement algorithm. Besides *ESet*, we also implement RUSH [24][25] for comparison.

In our simulations, we assume that failure events conform to exponential distribution, which has been proved by many existing works. And we use λ to adjust failure rate.

B. Recovery I/O Parallelism Analysis

Ideally O is equal o to make each host obtains best recovery performance. But in reality O may be smaller than o with different parameters. Here we set n=14, k=10 and calculate the average recovery I/O parallelism for each host. To find average recovery I/O parallelism, first we calculate the O for each host in a group. Then we sum up their recovery I/O parallelism and divide by $o*n$ to get average recovery I/O parallelism for all hosts.

Fig. 10 illustrates the recovery I/O parallelism for different overlapping factor with n=14 and k=10. We can see O roughly increases with o increases. When o is a prime number, O reaches its maximum value. But when o is not a prime number, O may be only around half of o.

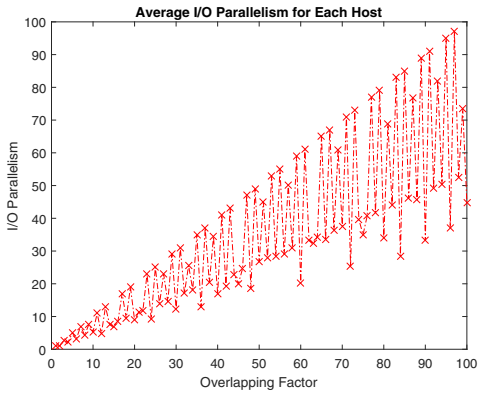


Fig. 10. Recovery I/O Parallelism for n=14 and k=10

TABLE III
PARAMETERS FOR A YEAR SIMULATION

Name	Value
λ	1
Disk Failure Percentage	4%
Host Failure Percentage	1%

C. Recovery Performance of Simulating A Year Failure

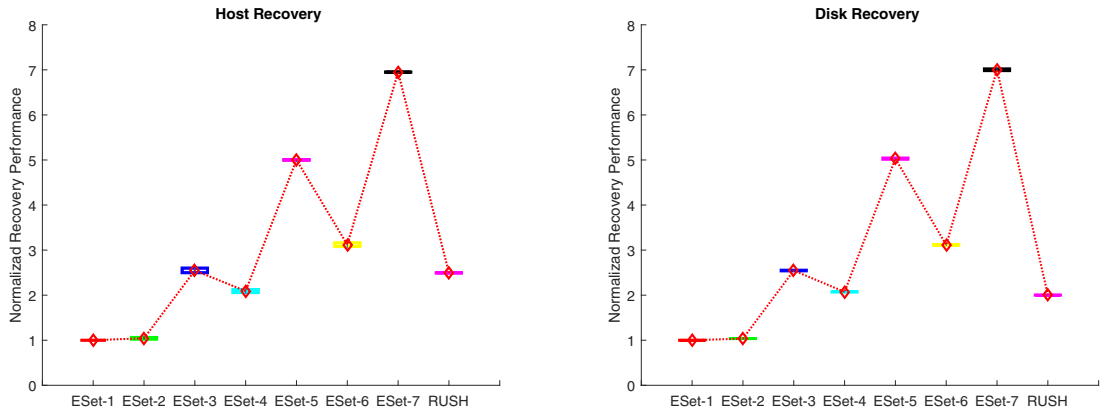
The major parameters used in this simulation are presented in Table. III. We use the recovery performance where the overlapping factor is 1 as the baseline for normalizing the recovery performance. Each simulation is conducted 30 times with overlapping factor from 1 to 7. We obtain the mean value of the recovery performance and calculate the standard deviation to see the stability of the recovery performance.

The simulation results are presented in Fig. 11. The number follows ESet- represents the overlapping factor. Here we can see that each *ESet* is very stable for recovering. Meanwhile the performance is similar to our I/O parallelism analysis in Fig.10. Besides, we also can see the recovery parallelism for RUSH with host recovery is around 2.5 and for disk recovery is around 2, which is a fixed value and not very efficient for recovering.

D. Recovery Performance of With Different λ Values

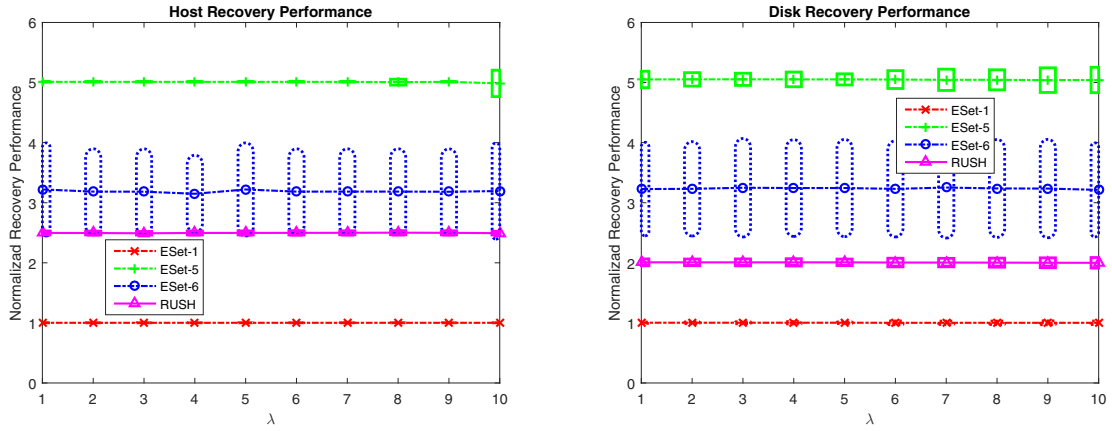
Here we set λ varied from 1 to 10, failure percentage for host as 1% and failure percentage for disk as 5% to see the performance of recovering when handling different failure density in a short period. When λ equals 1, the failures are nearly evenly distributed in a whole year. And when λ increases to 10, all failures happen in the first few months of the whole year.

The simulation results are showed in Fig. 12. We can see for *ESet*, where overlapping factor is 1, the recovery performance is very stable since its recovery I/O parallelism is always 1. There is no I/O contention for its recovery. For o=5, the recovery performance is also very stable for its host recovery. When λ reaches 10, it has higher standard deviation. We believe there are some hosts or disks failed in the same group and bringing I/O contention for the recovery. The contention



(a) Host Recovery Performance (b) Disk Recovery Performance

Fig. 11. Normalized Recovery Performance for A Year Simulation



(a) Host Recovery Performance (b) Disk Recovery Performance

Fig. 12. Normalized Recovery Performance for Different λ

is more obvious for its disk recovery with λ increasing. For o=6, it has very high standard deviation, that is because hosts in each group have different recovery I/O parallelism. Both o=5 and o=6 outperform RUSH for different λ.

E. Recovery Performance of Burst Failures in An Hour

After simulating recovery performance for a year, we also simulate how *ESet* performs during an hour of burst failures, which rarely happen but do exist in data center storage systems. We set host failure rate from 1‰to 10‰to see how *ESet* and *RUSH* perform.

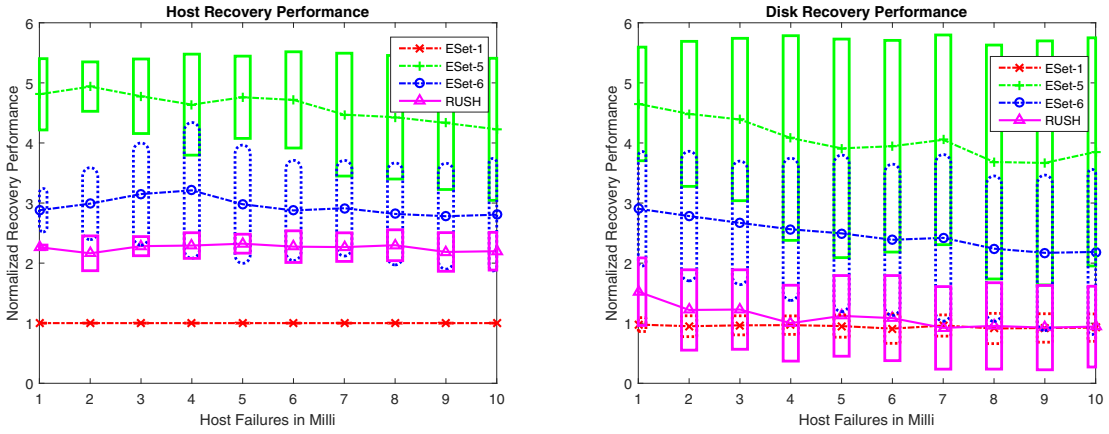
The recovery performance for host and disk are illustrated in Fig. 13. For host recovery performance, we can see when o=1 the performance is always very stable. And for o=5, the performance decreases as failure rate increases, which means more hosts and disks fail in the same group and introduce higher I/O contention with more failures. When o=6, the performance is not stable since failed hosts have different recovery I/O parallelism. And for disk recovery, except o=1, the performance decreases as host failure rate increases.

VII. Conclusion

In this paper, we showed the challenge of data placement algorithms to bring efficient recovery for erasure-coded storage systems. We proposed *ESet* to improve recovery performance by exploiting disk I/O parallelism. We demonstrated that *ESet* can achieve desirable recovery performance by adjusting overlapping factor. We believe that storage system administrator can easily achieve their desired recovery performance with *ESet*. Our simulation experiments showed that *ESet* can work well in data center storage systems with PB-level data. It can outperform *RUSH* by increasing overlapping factor. Currently our evaluation is limited by simulation. In the future work, we plan to implement our algorithm in a storage system and test its performance through real experiments.

Acknowledgement

We thank the anonymous reviewers for their valuable comments. This work is supported by Shenzhen Basic Research Grant SCI-2015-SZTIC-002.



(a) Host Recovery Performance

(b) Disk Recovery Performance

Fig. 13. Normalized Recovery Performance of Burst Failures in An Hour

REFERENCES

[1] M. Grawinkel, L. Nagel, M. Mäsker, F. Padua, A. Brinkmann, and L. Sorth, "Analysis of the ecmwf storage landscape," in *Proc. of the 13th USENIX Conference on File and Storage Technologies*, 2015.

[2] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang *et al.*, "F4: Facebooks warm blob storage system," in *11th USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[3] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster," *Proc. USENIX HotStorage*, 2013.

[4] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson, "Diskreduce: Raid for data-intensive scalable computing," in *Proceedings of the 4th Annual Workshop on Petascale Data Storage*. ACM, 2009, pp. 6–10.

[5] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson, "Diskreduce: Replication as a prelude to erasure coding in data-intensive scalable computing," *Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh*, 2011.

[6] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin *et al.*, "Erasure coding in windows azure storage," in *Proceedings of USENIX ATC*, 2012.

[7] D. Beaver, S. Kumar, H. C. Li, J. Sobel, P. Vajgel *et al.*, "Finding a needle in haystack: Facebook's photo storage." in *OSDI*, vol. 10, 2010, pp. 1–8.

[8] B. Schroeder and G. A. Gibson, "Understanding disk failure rates: What does an mttf of 1,000,000 hours mean to you?" *ACM Transactions on Storage (TOS)*, vol. 3, no. 3, p. 8, 2007.

[9] E. Pinheiro, W.-D. Weber, and L. A. Barroso, "Failure trends in a large disk drive population." in *FAST*, vol. 7, 2007, pp. 17–23.

[10] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, "Fast crash recovery in ramcloud," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 29–41.

[11] E. Heien, D. Kondo, A. Gainaru, D. LaPine, B. Kramer, and F. Cappello, "Modeling and tolerating heterogeneous failures in large parallel systems," in *High Performance Computing, Networking, Storage and Analysis, 2011 International Conference for*. IEEE, 2011, pp. 1–11.

[12] B. Schroeder, G. Gibson *et al.*, "A large-scale study of failures in high-performance computing systems," *Dependable and Secure Computing, IEEE Transactions on*, vol. 7, no. 4, pp. 337–350, 2010.

[13] S. Nath, H. Yu, P. B. Gibbons, and S. Seshan, "Subtleties in tolerating correlated failures in wide-area storage systems." in *NSDI*, vol. 6, 2006, pp. 225–238.

[14] K. M. Greenan, "Reliability and power-efficiency in erasure-coded storage systems," Ph.D. dissertation, Citeseer, 2009.

[15] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in globally distributed storage systems." in *OSDI*, 2010, pp. 61–74.

[16] H. Weatherspoon and J. D. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," in *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*, 2002.

[17] D. RESEARCH, "State of it recovery for smbs." [Online]. Available: https://axcient.com/state-of-it-recovery-for-smbs

[18] G. A. Alvarez, W. A. Burkhard, L. J. Stockmeyer, and F. Cristian, "Declustered disk array architectures with optimal and near-optimal parallelism," in *ACM SIGARCH Computer Architecture News*, vol. 26, no. 3. IEEE Computer Society, 1998, pp. 109–120.

[19] J. C. Koo and J. T. Gill, "Scalable constructions of fractional repetition codes in distributed storage systems," in *Communication, Control, and Computing, 2011 49th Annual Allerton Conference on*. IEEE, 2011, pp. 1366–1373.

[20] P. Kaski and P. R. Östergård, "There exists no (15, 5, 4) rbibd," *Journal of Combinatorial Designs*, vol. 9, no. 3, pp. 227–232, 2001.

[21] S. Houghten, L. Thiel, J. Janssen, and C. Lam, "There is no (46, 6, 1) block design*," *Journal of Combinatorial Designs*, vol. 9, no. 1, pp. 60–71, 2001.

[22] P. Shang, J. Wang, H. Zhu, and P. Gu, "A new placement-ideal layout for multiway replication storage system," *Computers, IEEE Transactions on*, vol. 60, no. 8, pp. 1142–1156, 2011.

[23] A. Cidon, S. M. Rumble, R. Stutsman, S. Katti, J. K. Ousterhout, and M. Rosenblum, "Copysets: Reducing the frequency of data loss in cloud storage." in *ATC*. Citeseer, 2013, pp. 37–48.

[24] R. Honicky and E. L. Miller, "A fast algorithm for online placement and reorganization of replicated data," in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*. IEEE, 2003, pp. 10–pp.

[25] R. Honicky and E. L. Miller, "Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE, 2004, p. 96.

[26] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "Crush: Controlled, scalable, decentralized placement of replicated data," in *ICS*. ACM, 2006, p. 122.

[27] A. Miranda, S. Effert, Y. Kang, E. L. Miller, I. Popov, A. Brinkmann, T. Friedetzky, and T. Cortes, "Random slicing: Efficient and scalable data placement for large-scale storage systems," *ACM Transactions on Storage (TOS)*, vol. 10, no. 3, p. 9, 2014.

[28] X. Chu, C. Liu, K. Ouyang, L. S. Yung, H. Liu, and Y.-W. Leung, "Perasure: a parallel cauchy reed-solomon coding library for gpus," in *IEEE International Conference on Communications*. IEEE, 2015.

[29] W. Jiang, C. Hu, Y. Zhou, and A. Kanevsky, "Are disks the dominant contributor for storage failures?: A comprehensive study of storage subsystem failure characteristics," *ACM Transactions on Storage (TOS)*, vol. 4, no. 3, p. 7, 2008.

[30] R. J. Chansler, "Data availability and durability with the hadoop distributed file system," *login: The USENIX Association Newsletter*, vol. 37, no. 1, 2013.