Even Data Placement for Load Balance in Reliable Distributed Deduplication Storage Systems

Min Xu[1], Yunfeng Zhu[2], Patrick P. C. Lee[1], Yinlong Xu[2]
[1]Department of Computer Science and Engineering, The Chinese University of Hong Kong
[2]School of Computer Science & Technology, University of Science & Technology of China
{*xum,pclee*}*@cse.cuhk.edu.hk, zyfl@mail.ustc.edu.cn, ylxu@ustc.edu.cn*

***Abstract*—Modern distributed storage systems often deploy deduplication to remove content-level redundancy and hence improve storage efficiency. However, deduplication inevitably leads to unbalanced data placement across storage nodes, thereby degrading read performance. This paper studies the load balance problem in the setting of a reliable distributed deduplication storage system, which deploys deduplication for storage efficiency and erasure coding for reliability. We argue that in such a setting, it is generally challenging to find a data placement that simultaneously achieves both read balance and storage balance objectives. To this end, we formulate a combinatorial optimization problem, and propose a greedy, polynomial-time *Even Data Placement (EDP)* algorithm, which identifies a data placement that effectively achieves read balance while maintaining storage balance. We further extend our EDP algorithm to heterogeneous environments. We demonstrate the effectiveness of our EDP algorithm under real-world workloads using both extensive simulations and prototype testbed experiments. In particular, our testbed experiments show that our EDP algorithm reduces the file read time by 37.41% compared to the baseline round-robin placement, and the reduction can further reach 52.11% in a heterogeneous setting.**

I. Introduction

Demands for scalable storage services have been surging in these years. Distributed storage systems aggregate multiple storage nodes (or servers) as a single storage pool, and provide scalable platforms for managing an ever-increasing scale of data. One major storage application is *backup storage*, in which users regularly generate data backups for their important data. To provide scalability, many commercial backup service providers [2, 3, 4] have deployed distributed storage systems for backup management.

Deduplication is a well-developed technology that improves storage efficiency of distributed storage systems. Backup storage products often use deduplication [9, 29] to remove content-level redundancy. Deduplication divides data into chunks, and stores only system-wide unique chunks and uses small references to refer duplicate chunks to already stored unique chunks. Deduplication can effectively reduce the storage space in practical storage workloads, e.g., by 20× [5].

However, deduplication causes duplicate data chunks to refer to existing identical data chunks that are scattered across different nodes in an unpredictable way, thereby making load balance difficult to achieve. Conventional data placement is unaware of deduplication, and attempts to write similar amounts of data to different nodes to maintain *storage balance*. Without deduplication, the similar amounts of data are read from different nodes, so *read balance* is also maintained. However, when deduplication is enabled, the chunks of a file may refer to existing duplicate chunks of another file in different nodes. Thus, file chunks may be clustered in a small number of nodes, thereby degrading read performance. In heterogeneous environments where nodes have varying I/O bandwidths [8], the degradation of read performance can be even more severe if the file chunks are clustered in low-bandwidth nodes. Poor read performance is undesirable for backup storage systems, as it prolongs the restore window. It also lengthens the system downtime during disaster recovery.

In this paper, we study the load balance problem in a reliable distributed deduplication storage system, which deploys deduplication for storage efficiency and erasure coding for reliability. Our storage system setting is similar to those in prior studies [9, 18, 19]. While the load balance problem described above is due to deduplication, we include erasure coding in our analysis to reflect a more practical storage system setting that addresses both storage efficiency and reliability.

We argue that in such a system setting, conventional data placement cannot simultaneously achieve both read balance and storage balance objectives. This motivates us to identify a deduplication-aware data placement policy that addresses the trade-off between read balance and storage balance. We make the following contributions.

- **Optimization problem:** We formulate a combinatorial optimization problem, whose objective is to find a data placement policy that maximizes read balance, while all nodes store similar amounts of data. Our problem formulation is based on parallel I/O access mode in a distributed storage system. It also addresses reliability based on erasure coding. We further extend the problem for heterogeneous environments.
- **Even data placement (EDP) algorithm:** The optimization problem is difficult to solve since it has a huge solution space and depends on the deduplication pattern. We thus propose a greedy *Even Data Placement (EDP)* algorithm, which determines an efficient data placement in polynomial time. We also propose an extended cost-based EDP (CEDP) algorithm to balance the read/write cost distributions.
- **Prototype implementation:** We implement a distributed storage system prototype that incorporates both deduplication and erasure coding. Our prototype is deployable in a networked environment.
- **Simulations and testbed experiments:** We conduct sim-

978-1-4673-7113-1/15/$31.00 ©2015 IEEE

ulations and testbed experiments under real-world workloads. We show that the baseline approach used in conventional data placement causes the data distributions of some files to deviate from the even distribution by over 50%, yet EDP effectively balances the data distribution. Testbed experiments show that our EDP algorithm reduces the file read time of the baseline by 37.41% and 52.11% in homogeneous and heterogeneous settings, respectively.

The rest of the paper is organized as follows: Section II presents the basics of reliable distributed deduplication systems. Section III explains the load imbalance issue via a motivating example. Section IV formulates the optimization problem that addresses load balance. Section V presents our proposed algorithms. Section VI describes the implementation details of our prototype. Section VII presents results from simulation and testbed experiments. Section VIII presents related work, and finally Section IX concludes the paper.

II. Basics

We present background details of a reliable distributed deduplication storage system considered in this paper.

A. Deduplication

Deduplication [23] improves storage efficiency by removing data with identical content. It divides input data into fixed-size or variable-size *chunks*, each of which is identified by a *fingerprint* computed by a cryptographic hash (e.g., MD5, SHA-1) of the chunk content. For variable-size chunking, the chunk boundaries are defined by content (e.g., Rabin fingerprinting [24]), and the chunk size distribution is configured by the average, minimum, and maximum chunk sizes. In both fixed-size chunking and variable-size chunking, two chunks are said to be identical if their fingerprints are identical, and we assume that the chance of fingerprint collisions of different chunks is negligible [23]. We also assume that a deduplication system maintains a fingerprint index to keep track of the chunks that are already stored.

This paper focuses on *inline* deduplication, which removes redundancy on the write path. Specifically, for a given set of chunks to be written, we first compare them, by querying the fingerprint index, with the currently stored chunks. If the fingerprint of a written chunk is new to the index, then the system regards the chunk as a *unique chunk*, meaning that the chunk will be stored and its fingerprint will be added to the index. Otherwise, if the fingerprint already exists, the system regards it as a *duplicate chunk*, meaning that the chunk is identical to another unique chunk of some previous files and will not be stored. The system will create a reference for the duplicate chunk to refer to an already stored chunk.

B. Erasure Coding

We consider a distributed storage system that achieves reliability via erasure coding, which provides higher fault tolerance than replication, but incurs significantly less storage overhead [28]. We consider an (n, k) erasure code configured with two parameters n and k, where $k < n$. It evenly divides data into k equal-size *data chunks*, and encodes them to form additional $n - k$ *parity chunks*, such that any k out of n data/parity chunks can reconstruct the original data. The collection of n data/parity chunks, which we call a *stripe*, will then be distributed to n distinct nodes.

If some data chunks (no more than $n-k$) are unavailable due to node failures, reads to unavailable data chunks are *degraded*, as they need to retrieve any k data/parity chunks of the same stripe from other non-failed nodes for decoding.

C. Integration

To deploy both deduplication and erasure coding in a distributed storage system, we first apply deduplication to remove duplicate chunks, followed by applying erasure coding to the remaining unique chunks. Specifically, after deduplication, we divide data into non-overlapping groups of k unique chunks that are considered to be the data chunks of an erasure coding stripe. We then encode the k data chunks to form additional $n - k$ parity chunks.

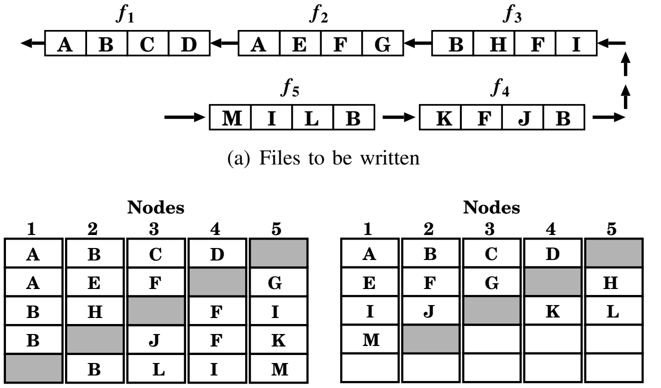
If fixed-size chunking is used in deduplication, the size of each encoded chunk will remain the same. On the other hand, if variable-size chunking is used, we first pad each k unique chunks with zeros to the maximum chunk size that has been configured before encoding. We store only the non-padded data chunks, and the parity chunks that have the maximum chunk size. If we need to decode unavailable data chunks due to failures, we first locally pad all chunks with zeros to the maximum chunk size before decoding.

III. Load Balance

Load balance is critical to the performance of a distributed storage system. We consider two aspects of load balance, namely *read balance* and *storage balance*, in which the system reads and stores the same (or similar) amount of data via each node, respectively. In storage systems with homogeneous nodes and without deduplication, storage balance evenly distributes data across nodes, and hence also implies read balance. Storage balance can be achieved by round-robin or random data placements. When erasure coding is used, *parity declustering* [13] can balance data and parity distribution by placing stripes across different subsets of nodes (assuming that the number of nodes is larger than the stripe size n).

In this work, we define the *baseline* policy for conventional data placement as follows. If the number of storage nodes is equal to the erasure coding stripe size, then we assign the data chunks to storage nodes in the round-robin fashion and keep the parity chunks rotated across stripes. Otherwise, if the number of storage nodes is larger than the erasure coding stripe size, then we first randomly select the same number of storage nodes as the stripe size as in parity declustering, followed by assigning the data chunks to the selected nodes in a round-robin fashion and keeping the parity chunks rotated across stripes.

However, with the baseline placement policy, deduplication inherently leads to uneven data distribution, thereby breaking the connection between read balance and storage balance. We



(a) Files to be written

(b) Without deduplication

(c) With deduplication

Fig. 1. Layouts of the baseline policy with/without deduplication (the data and parity chunks are represented in white and gray colors, respectively).

TABLE I
FILE READ TIMES OF THE BASELINE WITH/WITHOUT DEDUPLICATION.

	f_1	f_2	f_3	f_4	f_5
baseline (no dedup)	α/β	α/β	α/β	α/β	α/β
baseline (dedup)	α/β	$2\alpha/\beta$	$2\alpha/\beta$	$3\alpha/\beta$	$2\alpha/\beta$

motivate this issue via a toy example shown in Figure 1. Figure 1(a) shows a stream of five files with four chunks each, and we write them to a five-node storage system using $(5, 4)$ erasure coding (i.e., each stripe has four data chunks and one parity chunk). We assume parallel I/O accesses, in which read/write requests are issued to the storage system in parallel. Let α be the chunk size, and β be the I/O bandwidth of each storage node.

Figures 1(b) and 1(c) show the data layouts of the baseline policy without and with deduplication (the latter stores only unique chunks), respectively. With deduplication, although we write similar numbers of unique chunks to different nodes (the numbers differ by at most one), the chunks of a file may be clustered in a single node, thereby increasing the file read time. For example, three of four chunks of file f_4 are stored in node 2 (see Figure 1(c)). Table I shows the resulting read times of different files. We observe that the read times of files f_2, f_3, f_4, and f_5 all increase when deduplication is used, for example, by 2× or 3× when compared to without deduplication. Thus, maintaining storage balance cannot achieve read balance when deduplication is used.

The above example only considers homogeneous nodes. If a storage system comprises heterogeneous nodes, the file read time distribution can be more diverse since the chunks of a file may be aggregated in a bottlenecked node. Although the above example is contrived, we show via trace-driven analysis that read imbalance can occur when we apply deduplication to real-world workloads (see Section VII).

IV. PROBLEM

We formulate a combinatorial optimization problem that searches for a data placement policy that maximizes read balance, while preserving storage balance. We argue that the problem has a huge solution space, which motivates us to design an efficient but accurate data placement policy to solve the problem. We also address how our problem can be extended for heterogeneous settings.

We consider a reliable distributed storage system with N nodes. It deploys (n, k) erasure coding, where $k < n \leq N$, and places each stripe across n nodes. Suppose that we store t files, and check if the chunks of each file to be written can be deduplicated with the currently stored chunks (see Section II-A). Let $u_{i,j}$ and $d_{i,j}$ be the numbers of unique chunks and duplicate chunks, respectively, of file i in node j, where $1 \leq i \leq t$ and $1 \leq j \leq N$. Let $U_i = \sum_{j=1}^{N} u_{i,j}$ be the total number of unique chunks of file i.

A. Optimization Problem

We consider a storage system that is bottlenecked by the network I/O bandwidth, and its data in different nodes is accessed in parallel. We first assume that all nodes are homogeneous, while we later extend the model for a heterogeneous setting.

Suppose that the read time of a file is linear to the maximum number of data chunks (including unique and duplicate chunks) being read in a node. For file i, its read time (denoted by M_i) is given by:

$$M_i = \max_{1 \leq j \leq N} \{u_{i,j} + d_{i,j}\}, \text{ where } 1 \leq i \leq t. \quad (1)$$

Its lower bound (denoted by E_i) is the number of data chunks retrieved from a node when they are evenly placed across all nodes, i.e., when $E_i = u_{i,1} + d_{i,1} = u_{i,2} + d_{i,2} = \cdots = u_{i,N} + d_{i,N}$. We can show that

$$E_i = \frac{1}{N} \sum_{j=1}^{N} (u_{i,j} + d_{i,j}), \text{ where } 1 \leq i \leq t. \quad (2)$$

We maintain read balance by minimizing the difference between M_i and E_i for file i. While there are various ways to characterize read balance, in this paper, we consider one possible metric. We define a *read balance gap* G_i, which is a function of $u_{i,j}$'s and $d_{i,j}$'s, as follows:

$$G_i = 1 - \frac{E_i}{M_i}, \text{ where } 1 \leq i \leq t. \quad (3)$$

Read balance of file i is achieved by minimizing G_i, which attains minimum at zero (i.e., when the data chunks of file i are evenly placed across nodes). When G_i is close to one, the degree of read imbalance of file i becomes more severe.

With deduplication, the distribution of duplicate chunks of a file depends on the previously stored files. Rearranging the placement of duplicate chunks may achieve a more balanced placement of the file, but this incurs expensive I/Os and unbalances the placements of previous files. Thus, we fix $d_{i,j}$'s in the gap G_i (see Equation 3), and we carefully place the unique chunks (i.e., search for a distribution $u_{i,j}$'s) to minimize G_i.

In addition to read balance, we also balance the storage of the unique chunks of t files. We write the same number of unique chunks (denoted by C) to each node, given by:

$$C = \frac{1}{N}\sum_{i=1}^{t} U_i, \tag{4}$$

To make memory management easier, we write chunks on a *per-batch* basis, in which we fix the number of unique chunks C to be written to each node and determine the number of files t accordingly. Specifically, we first buffer all C unique chunks to be written to each node (i.e., a total of $N \times C$ unique chunks in a batch), and then decide the appropriate data placement.

We now pose a combinatorial optimization problem, whose objective is to find a data placement policy (i.e., a set of $u_{i,j}$'s) that achieves read balance while preserving storage balance.

Problem 1:

$$\begin{aligned}
\text{Minimize} \quad & \sum_{i=1}^{t} G_i \\
\text{subject to} \quad & \sum_{j=1}^{N} u_{i,j} = U_i, \ \forall i \in [1,t] \\
& \sum_{i=1}^{t} u_{i,j} = C, \\
& 0 \le u_{i,j} \le U_i, \ \forall i \in [1,t], j \in [1,N].
\end{aligned}$$

Here, the objective is to minimize the sum of the read balance gaps G_i's for all file i. The first constraint ensures that all unique chunks of each file will be written to one of the nodes. The second constraint preserves storage balance by writing C unique chunks to each node. The last constraint bounds the range of each $u_{i,j}$.

However, Problem 1 has a huge solution space. Note that we store a total of $N \times C$ unique chunks to N nodes. There are $\binom{N\times C}{C}$ ways to assign C unique chunks to the first node, $\binom{(N-1)\times C}{C}$ ways to assign another set of C unique chunks to the second node, and so forth. Thus, the total number of ways to place all unique chunks is $\binom{N\times C}{C} \times \binom{(N-1)\times C}{C} \times \cdots \times \binom{C}{C} = \frac{(N\times C)!}{(C!)^N}$. The solution space is too large even for small N and C. For instance, when $N = 5$ and $C = 4$, there are more than 3×10^{11} possible solutions. Since the objective function depends on the current deduplication patterns (i.e., the number of duplicate chunks in each node), the optimal solution needs to be determined in real time. However, the huge solution space makes the extensive search for an optimal solution infeasible. In Section V, we propose a greedy algorithm to efficiently solve the problem.

B. Heterogeneity Awareness

Modern distributed storage systems are often composed of heterogeneous nodes, so the read latencies of data chunks differ across nodes. We modify Problem 1 by introducing a weight w_j for each node j, which is defined as the read cost per chunk for node j. Thus, the read cost of file i (denoted by M'_i) is $M'_i = \max_{1\le j\le N}\{w_j(u_{i,j} + d_{i,j})\}$.

First, we derive the lower bound of the read cost of file i (denoted by E'_i), which is achieved when the read costs are evenly distributed across nodes, i.e., $E'_i = w_1(u_{i,1} + d_{i,1}) = w_2(u_{i,2} + d_{i,2}) = \cdots = w_N(u_{i,N} + d_{i,N})$. Thus,

$$E'_i = \frac{\sum_{j=1}^{N}(u_{i,j} + d_{i,j})}{\sum_{j=1}^{N}\frac{1}{w_j}}. \tag{5}$$

The read balance gap (denoted by G'_i) between the maximum read cost of file i and its lower bound can be defined as:

$$G'_i = 1 - \frac{E'_i}{M'_i}. \tag{6}$$

We replace the objective function of Problem 1 with $\sum_{i=1}^{t} G'_i$ and solve the problem subject to the same constraints.

We can obtain the weights w_j's for all nodes by, for example, periodic probe measurements [8]. Then the weight w_j may represent the reciprocal of the measured link bandwidth of node j. We assume that the weights are fairly stable and do not vary significantly over time, so the resulting data placement reflects the current system conditions. We pose the issue of performing accurate weight measurements as future work.

V. Even Data Placement Algorithm

In this section, we propose the *Even Data Placement (EDP)* algorithm, a polynomial-time greedy algorithm that aims to efficiently identify a near-optimal data placement solution to Problem 1 in Section IV-A. We also extend the EDP algorithm for the heterogeneous setting.

A. Main Idea

The EDP algorithm builds on two procedures: DISTRIBUTE and SWAP. The DISTRIBUTE procedure attempts to identify a node to place each chunk of a batch such that the increase in the summation of the read balance gaps is minimum. If more than one node satisfies this criterion, then we place the chunk at the node where the previous chunk of the same file resides so as to keep the chunks of the same file together. The SWAP procedure inspects the placement decision of DISTRIBUTE and attempts to swap the chunk positions of different file pairs to see if the summation of the read balance gaps can be further reduced. There are two types of unique chunks in a batch that can be swapped: (i) *non-shared chunk*, which appears in exactly one file in the batch, and (ii) *shared chunk*, which appears in more than one file in the batch. Since swapping shared chunks may affect the chunk distributions of multiple files, for simplicity, we only consider the swapping of non-shared chunks. Note that the EDP algorithm only operates on the chunk positions rather than the chunks, and does not incur any actual I/O.

B. Algorithm Details

Algorithm 1 shows the pseudo-code of the EDP algorithm. It takes the following inputs: (i) N, the number of storage nodes; (ii) C, the number of unique chunks to be placed in each node ; (iii) t, the number of files in a batch; (iv) $\mathbf{U} = (U_i | 1 \le i \le t)$, a vector where each entry is the number of unique chunks of each file; (v) $\mathbf{d} = (d_{i,j} | 1 \le i \le t, 1 \le j \le N)$, a vector where each entry is the number of duplicate

Algorithm 1 Even Data Placement Algorithm

```
1: function EDP($t, N, C, \mathbf{U}, \mathbf{F}, \mathbf{d}$)
2:     $u_{i,j} \leftarrow 0, 1 \le i \le t, 1 \le j \le N$
3:     $C_j \leftarrow C, 1 \le j \le N$
4:     $T \leftarrow 0$
5:     for $i$ = 1 to $t$ do
6:         DISTRIBUTE($i, T, \mathbf{U}, \mathbf{C}, \mathbf{F}, \mathbf{u}, \mathbf{d}$)
7:         $S \leftarrow \sum_{k=1}^{i} G_k$
8:         for $i'$ = 1 to $i - 1$ do
9:             SWAP($i, i', \mathbf{u}, \mathbf{d}, S$)
10:        end for
11:        $T \leftarrow T + U_i$
12:    end for
13: end function
14: function DISTRIBUTE($i, T, \mathbf{U}, \mathbf{C}, \mathbf{F}, \mathbf{u}, \mathbf{d}$)
15:    for $l$ = 1 to $U_i$ do
16:        $j^* \leftarrow$ ID of node that minimizes change of $G_i$
17:        $u_{i,j^*} \leftarrow u_{i,j^*} + 1$
18:        for each $i' \in \mathbf{F}_{l+T}$ do
19:            $d_{i',j^*} \leftarrow d_{i',j^*} + 1$
20:        end for
21:        $C_{j^*} \leftarrow C_{j^*} - 1$
22:    end for
23: end function
24: function SWAP($i, i', \mathbf{u}, \mathbf{d}, S$)
25:    for each storage node pair $(j, j')$ do
26:        $m_{i,j} \leftarrow$ number of non-shared chunks of file $i$ on node $j$
27:        $m_{i',j'} \leftarrow$ number of non-shared chunks of file $i'$ on node $j'$
28:        for $z$ = 1 to min($m_{i,j}$, $m_{i',j'}$) do
29:            $S_z \leftarrow \sum_{k=1}^{i} G_k$, if $z$ non-shared chunks of
               files $i$ and $i'$ are swapped between nodes $j$ and $j'$
30:        end for
31:        $z^* \leftarrow$ argmin$_{1 \le z \le \min(m_{i,j}, m_{i',j'})}$ $S_z$
32:        if $S_{z^*} < S$ then
33:            Swap $z^*$ chunks of file $i$,$i'$ between nodes $j$,$j'$
34:            Update placement of the swapped chunks
35:            $S \leftarrow S_{z^*}$
36:        end if
37:    end for
38: end function
```

chunks of each of the t files on each of the N nodes; and (vi) $\mathbf{F} = (F_l | 1 \le l \le N \times C)$, a vector where each entry is the list of files in the batch that share the unique chunk l. It outputs $\mathbf{u} = (u_{i,j} | 1 \le i \le t, 1 \le j \le N)$, a vector where each entry indicates the number of unique chunks of each of the t files to place in each of the N nodes.

For each file, the EDP algorithm first calls DISTRIBUTE (Line 6) to assign unique chunks of each file i to the storage nodes. It also records the current objective value (Line 7). It then calls SWAP to attempt to swap the chunk positions of the current file with those of the previous files to further reduce the objective value (Lines 8-10).

In DISTRIBUTE, for each unique chunk (Line 15), EDP tentatively assigns it to each of the N nodes, and finds the node that minimizes the change of G_i if the chunk is assigned to it (Line 16). If there are multiple nodes that have the same minimal change of G_i, EDP assigns the chunk to the same node as the previous chunk. Then EDP increments the number of unique chunks for the current file on that node by one (Line 17). For other files in the batch that share this unique chunk, EDP increments the number of duplicate chunks of each such file on the selected node by one (Lines 18-20). Finally, EDP decrements the allowable number of chunks on the selected node by one (Line 21).

In SWAP, EDP inspects possible swaps of placement decisions for a given pair of files denoted by indices i and i' between each possible pair of nodes denoted by indices j and j' (Line 25). EDP focuses on the non-shared chunks, and counts the number of non-shared chunks of files i and i' on nodes j and j' as $m_{i,j}$ and $m_{i',j'}$, respectively (Lines 26-27). Then EDP tries all possible numbers of unique chunks, denoted by z, from one up to $\min(m_{i,j}, m_{i',j'})$, to swap (Line 28). For each z, EDP records the revised objective value if z non-shared chunks of files i and i' are swapped between nodes j and j' (Line 29). EDP picks z that minimizes the objective value as z^* (Line 31). If the objective value after swapping is smaller than that without swapping, the swap decision will be made (Lines 32-36).

C. Example

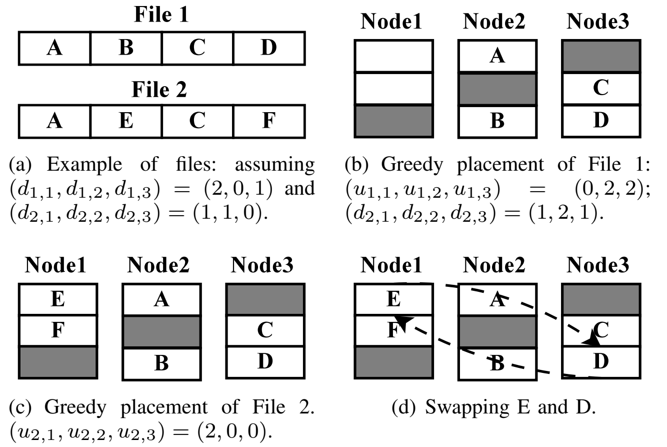
Figure 2 shows an illustrative example with two files in a batch to be stored on a system of three nodes using (3,2) erasure coding. We set $C = 2$. As shown in Figure 2(a), suppose that file 1 has four unique chunks and has $(d_{1,1}, d_{1,2}, d_{1,3}) = (2, 0, 1)$, and that file 2 has four unique chunks and has $(d_{2,1}, d_{2,2}, d_{2,3}) = (1, 1, 0)$.

To balance the placement of chunks of file 1, EDP places two unique chunks on nodes 2 and 3, as shown in Figure 2(b). As both chunks A and C are shared by file 2, EDP updates $(d_{2,1}, d_{2,2}, d_{2,3}) = (1, 2, 1)$. For file 2, DISTRIBUTE can only assign its two unique chunks to node 1. At the end of DISTRIBUTE, the distributions of unique and duplicate chunks of files 1 and 2 are: $(u_{1,1} + d_{1,1}, u_{1,2} + d_{1,2}, u_{1,3} + d_{1,3}) = (2, 2, 3)$ and $(u_{2,1} + d_{2,1}, u_{2,2} + d_{2,2}, u_{2,3} + d_{2,3}) = (3, 2, 1)$, respectively.

SWAP now tries swapping the positions of non-shared chunks of files 1 and file 2. Before SWAP, the objective value (i.e., the sum of read balance gaps of files 1 and 2) is $(1 - \frac{7/3}{3}) + (1 - \frac{2}{3}) = \frac{5}{9}$. Suppose now we try to swap chunks E and D, as shown in Figure 2(d). The revised distributions of unique and duplicate chunks of files 1 and 2 will become: $(u_{1,1} + d_{1,1}, u_{1,2} + d_{1,2}, u_{1,3} + d_{1,3}) = (3, 2, 2)$ and $(u_{2,1} + d_{2,1}, u_{2,2} + d_{2,2}, u_{2,3} + d_{2,3}) = (2, 2, 2)$, respectively. The objective value can reduce to $(1 - \frac{7/3}{3}) + (1 - \frac{2}{2}) = \frac{2}{9}$. This shows that swapping of chunks E and D can achieve better read balance.

D. Complexity Analysis

We now derive the worst-case complexity of the EDP algorithm. For each chunk of file i, DISTRIBUTE inspects all N storage nodes and a list of up to t files. Thus, the complexity of DISTRIBUTE on processing file i is $\mathcal{O}(U_i(N + t))$. On the other hand, the SWAP procedure scans possible swaps for file i and each of previous $i - 1$ files between every pair of nodes, and one swapping can involve up to C chunks. Thus, the complexity for SWAP to process file i is $\mathcal{O}((i - 1)CN^2)$. Hence,



(a) Example of files: assuming $(d_{1,1}, d_{1,2}, d_{1,3}) = (2,0,1)$ and $(d_{2,1}, d_{2,2}, d_{2,3}) = (1,1,0)$.

(b) Greedy placement of File 1: $(u_{1,1}, u_{1,2}, u_{1,3}) = (0,2,2)$; $(d_{2,1}, d_{2,2}, d_{2,3}) = (1,2,1)$.

(c) Greedy placement of File 2. $(u_{2,1}, u_{2,2}, u_{2,3}) = (2,0,0)$.

(d) Swapping E and D.

Fig. 2. Illustration of the EDP algorithm for two files, where $N = 3$, $C = 2$, $(n, k) = (3, 2)$.

the overall complexity of EDP when processing t files is $\mathcal{O}(\max\{\sum_{i=1}^{t} U_i(N+t), \sum_{i=1}^{t}(i-1)CN^2\}) = \mathcal{O}(CN^2t^2)$, which is in polynomial time.

E. Extensions

We consider two extensions, which account for heterogeneity and variable-size chunks.

Heterogeneity: Algorithm 1 can be modified slightly to adapt to the heterogeneous scenario, and we call it the *Cost-based Even Data Placement (CEDP)* algorithm. According to Section IV-B, the read balance gap is computed based on the node weights. For CEDP, the only change to EDP is to modify the calculation of the read balance gap in Lines 7, 16, and 29 of Algorithm 1. Other steps remain the same.

Variable-size chunking: We can extend Algorithm 1 to support variable-size chunking by calculating the read balance gap as a function of the number of bytes (rather than the number of chunks). We still write C unique chunks to each node to maintain storage balance, yet we modify $\mathbf{u}$ and $\mathbf{d}$ to indicate the numbers of unique and duplicate bytes, respectively. Specifically, let $\mathbf{L} = \{L_{i,j} | 1 \leq i \leq t, 1 \leq j \leq U_i\}$, where $L_{i,j}$ is the length of chunk j of file i. In DISTRIBUTE, we modify Lines 17 and 19 as $u_{i,j^*} \leftarrow u_{i,j^*} + L_{i,l}$ and $d_{i',j^*} \leftarrow d_{i',j^*} + L_{i,l}$, respectively. Also, in SWAP, we modify Lines 29 and 33 to update $u_{i,j}$ and $u_{i',j'}$ by the number of bytes of unique chunks that are swapped. We also modify the objective functions in Equations (3) and (6) in terms of numbers of bytes.

VI. IMPLEMENTATION

We implement a distributed storage system prototype that realizes deduplication and erasure coding and supports different data placement policies including the baseline (see Section III), EDP, and CEDP. Figure 3 shows the system architecture, which comprises three main components: one or multiple clients, a metadata server, and multiple storage nodes.

File writes operate on a per-batch basis (see Section IV-A). Specifically, a client divides files into chunks and computes the fingerprints. For each batch of chunks to be written, the client sends the fingerprints to the metadata server, which performs deduplication and runs the data placement policy (e.g., the baseline, EDP, or CEDP). The metadata server also maintains the file metadata to keep track of chunks associated with a file. It then responds to the client with the list of unique chunks and how they are placed across nodes. The client then applies erasure coding to the unique chunks, and writes the encoded chunks to different nodes in parallel.

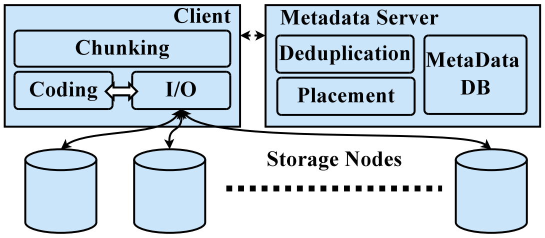


Fig. 3. Architectural overview of our prototype.

File reads operate in the reverse way. To read a file, the client queries the metadata server for all required chunks of a file. It then reads all chunks from the nodes in parallel. In the presence of node failures, the client issues degraded reads to retrieve k data and parity chunks from other surviving nodes to decode the unavailable chunks.

To integrate with erasure coding, we group every k unique chunks as the data chunks of a stripe. To balance the distribution of parity chunks, we enumerate all $\binom{N}{n}$ possible stripe permutations and n possible parity rotations. Thus, the resulting batch size is $N \times C = \binom{N}{n} \times n \times k$ data chunks. We pick C accordingly given N, n, and k.

To improve disk I/O efficiency, each storage node organizes data in *containers* [29], which serve as disk read/write units. We now configure each container to keep at most 64 chunks. When a node writes a chunk, it first appends the chunk to an in-memory container, which is flushed to disk when it is full. To read a chunk, the node reads the corresponding container as a whole into the local cache and retrieves the chunk.

We implement our prototype in C, and realize some operations using open-source libraries. We choose and implement 160-bit SHA-1 as fingerprints for deduplication using OpenSSL [20]. We also choose and implement Reed-Solomon coding [25] as our erasure coding scheme using Jerasure 2.0 [21] and GF-Complete [22]. In the metadata server, we maintain metadata in key-value databases and implement them using the Kyoto Cabinet library [10].

VII. EVALUATION

In this section, we compare the performance of our EDP and CEDP algorithms with that of the baseline placement policy defined in Section III. Our evaluation consists of two parts: simulations and testbed experiments, both of which are driven by real-world workloads.

A. Datasets

We drive our evaluation using three public datasets:

- *FSLHOME:* It is published by the File system and Storage Lab (FSL) at Stony Brook University [27]. It contains daily snapshots of the home directories of nine students on a shared network file system. Each snapshot comprises chunk fingerprints of multiple files obtained by variable-size chunking. Due to the large dataset size, we sample a subset of snapshots in year 2013. Our final FSLHOME dataset consists of a total of 104 snapshots, whose fingerprints are derived from the average chunk size of 4KB using variable-size chunking. Each snapshot has 197K to 210K files.

- *LINUX:* It contains 15 versions of *unpacked* Linux kernel source code [1], sampled from versions 2.6.35 to 3.16.3. Each version has size 393.77MB to 551.91MB of data with 30K to 50K files.
- *LINUXTAR:* It packs each version of the Linux kernel source code [1] into a single tarball file. It consists of 321 versions of uncompressed tarballs. Each tarball has size 4.93MB to 553.54MB of data.

Our evaluation studies the backup scenario. In each backup operation, we store a snapshot (for FSLHOME) or a version (for LINUX and LINUXTAR) and aim to achieve read balance for the files within each backup based on Problem 1. Note that the datasets are composed of different file size distributions: both FSLHOME and LINUX comprise many small files in each backup, while LINUXTAR contains one large tarball file in each backup. Thus, our evaluation addresses the impact of file size distributions on read balance.

For both FSLHOME and LINUX, we filter small files of size less than 16KB, so that we can distinguish more clearly the read balance gaps of the data placement schemes based on the large files with enough numbers of chunks. These small files only account for 2.53% and 5.84% of total sizes of FSLHOME and LINUX, respectively. Note that there is no small file in LINUXTAR.

The total logical sizes of FSLHOME, LINUX, and LINUXTAR are 6.46TB, 7.81GB, and 101GB, respectively (after we filter the small files of FSLHOME and LINUX). The physical storage sizes of FSLHOME, LINUX, and LINUXTAR after deduplication with 4KB variable-size chunking reduce to 0.34TB, 1.89GB, and 38.8GB, or equivalently, save 94.74%, 75.8%, and 61.50% of disk space, respectively. If we use deduplication with 4KB fixed-size chunking for LINUX and LINUXTAR, the savings are 70.04% and 53.01% disk space for LINUX and LINUXTAR, respectively.

B. Simulations

By default, our simulations consider a storage system with $N = 16$ nodes, and deploy $(n, k) = (14, 10)$ erasure coding as in [26]. We also set $C = \binom{16}{14} \times 14 \times 10/16 = 1,050$ (see Section VI) to balance the parity load. For the chunking scheme, we use 4KB variable-size chunking for FSLHOME, and 4KB fixed-size chunking for LINUX and LINUXTAR.

Effectiveness of EDP: We analyze the read balance problem and answer the following questions: (i) how severe read imbalance is in the baseline data placement policy; and (ii) how well the EDP/CEDP algorithm tackles the problem. We start with the homogeneous setting. We determine the chunk placements of the baseline and EDP algorithms based on chunk fingerprints and file metadata, and record the chunk distribution of each file. We then calculate the read balance gap of each file using Equation (3).

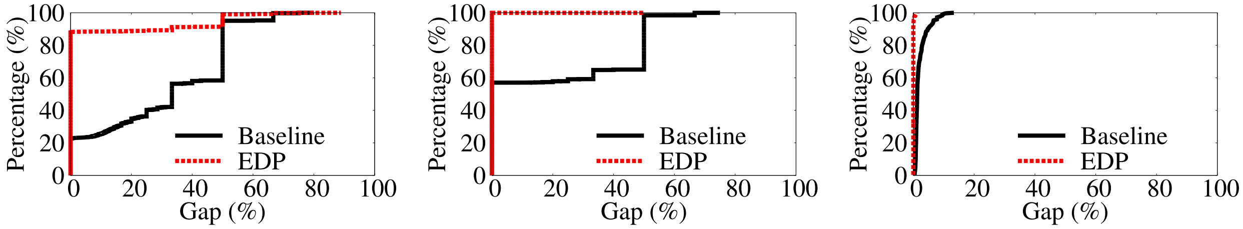
Figure 4 plots the cumulative distributions of files versus their gaps for FSLHOME, LINUX, and LINUXTAR. For FSLHOME, the baseline only keeps 22.59% of files evenly distributed; and around 75% of files have gaps between 10% and 50%. For LINUX, the baseline causes 42.95% of files to have gaps between 20% and 80%. For LINUXTAR, the baseline causes 13.3% of files to have non-zero gaps. On the other hand, EDP increases the percentage of evenly distributed files in FSLHOME, LINUX, LINUXTAR to 90%, 100%, and 100%, respectively.

Impact of erasure coding: We now study the read balance under different values of the system size N and erasure coding configurations (n, k). We consider two read balance metrics: (i) read balance and (ii) degraded read balance. For degraded reads, we consider the single node failure only (which is the common failure scenario [11, 26]), and simulate the failure by disabling the first node. We focus on the homogeneous setting, and compute both read balance and degraded read balance metrics of the baseline and EDP algorithms following Equation (3), in which we compute the gap as one minus the ratio of the even number of read chunks of a file to the maximum number of read chunks of the file over all nodes.

We focus on FSLHOME, while the results for LINUX and LINUXTAR are similar. Figures 5 compares the read balance and degraded read balance metrics for the baseline and EDP algorithms using FSLHOME. For read balance (see Figure 5(a)), the baseline leads to high gaps that range between 20% and 30%, while EDP reduces the gaps to 5%. EDP also keeps a smaller gap than the baseline in degraded read balance (see Figures 5(b)).

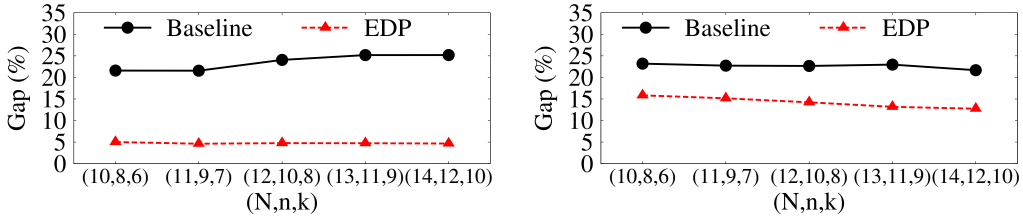
Impact of heterogeneity: We simulate heterogeneous environments with varying I/O bandwidths across nodes. We assume that the link bandwidths of storage nodes follow a uniform distribution so as to simulate a distributed storage environment [15]. Here, we randomly assign the bandwidth to each node using four uniform distributions: [1,120]Mbps, [10,120]Mbps, [30,120]Mbps, and [60,120]Mbps. We calculate the average improvement ratio of both EDP and CEDP algorithms over the baseline for each file, in terms of the reduction of read latency. For CEDP, the weight w_j $(1 \leq j \leq 16)$ associated with each storage node is set as the reciprocal of the I/O bandwidth of the node (see Section IV-B).

Figure 6 shows the results. When the I/O bandwidths are highly varying, CEDP improves the baseline by 50.89%, 48.90%, and 42.55% for FSLHOME, LINUX, and LINUXTAR, respectively, yet EDP shows very small improvements. For FSLHOME and LINUX, the improvements of EDP and CEDP become similar when the I/O bandwidths are less varying; for LINUXTAR, EDP is almost identical to the baseline (which conforms to the results in Figure 4(c)), and CEDP improves the latency of both the baseline and EDP. Overall, CEDP remains robust in heterogeneous environments.



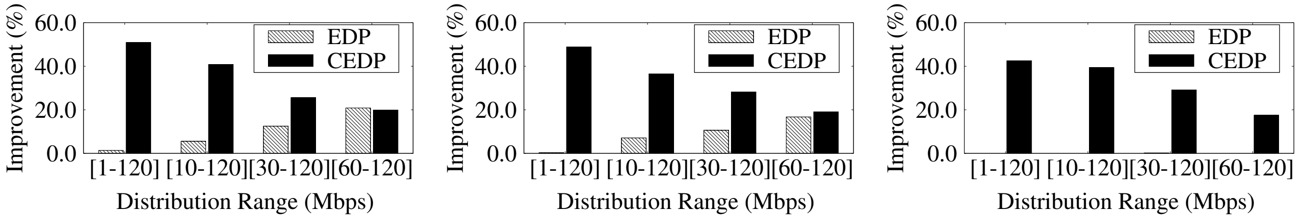
(a) FSLHOME (b) LINUX (c) LINUXTAR

Fig. 4. Cumulative distributions of files versus performance gaps for the baseline and EDP algorithms.



(a) Read balance (FSLHOME) (b) Degraded read balance (FSLHOME)

Fig. 5. Comparisons on read balance and degraded read balance between the baseline and EDP algorithms.



(a) FSLHOME (b) LINUX (c) LINUXTAR

Fig. 6. Read latency improvements of EDP and CEDP over baseline.

C. Testbed Experiments

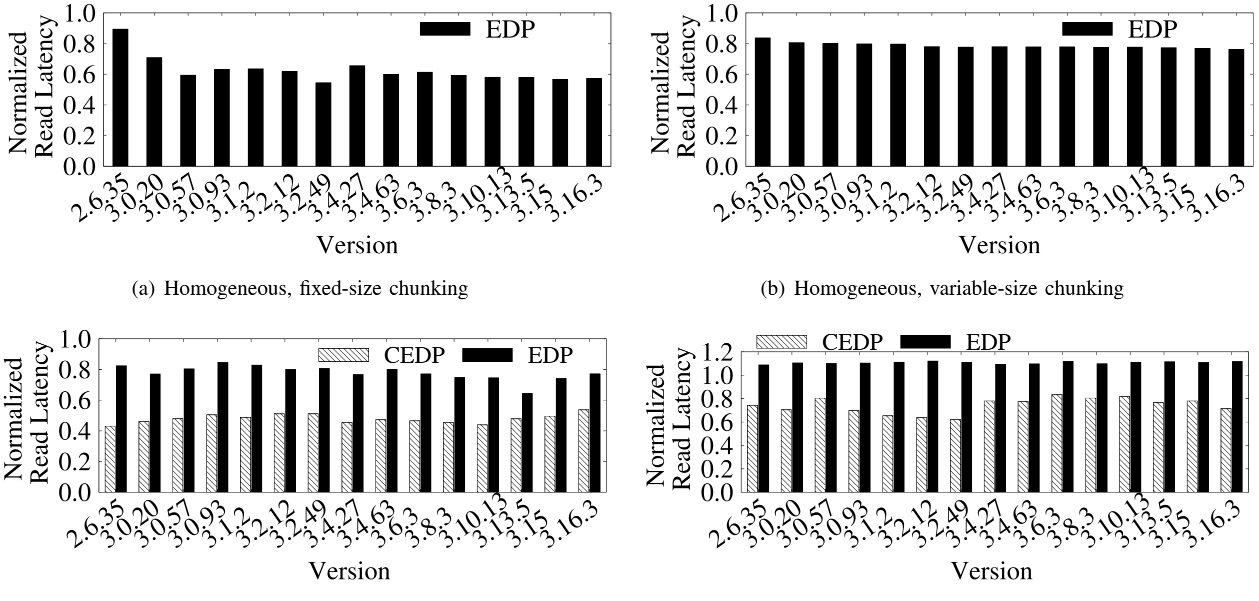
We evaluate via testbed experiments the read performance of various data placement algorithms using our prototype in Section VI. Our testbed consists of one client node, one metadata server node, and 16 storage nodes. We run each node on a multi-core machine, and interconnect all machines via a Gigabit Ethernet switch. The machines have varying CPU, RAM, and harddisk configurations. Here, we configure environments where the network transmission is the bottleneck. In a homogeneous setting, all nodes are configured with 100Mbps link bandwidth; in a heterogeneous setting, we configure five storage nodes with 10Mbps link bandwidth, six storage nodes with 100Mbps link bandwidth, and the five remaining storage nodes with 1Gbps link bandwidth. The client node and master node are configured with 1Gbps link bandwidth. All nodes run Ubuntu 12.04.2 with Linux kernel 3.5.

We mainly focus on LINUX and LINUXTAR so as to evaluate both fixed-size and variable-size chunking schemes. Both chunking schemes choose 4KB as the chunk size. The client writes each version, ordered by the version number, to the prototype. After that, the client reads all files of each version and records the read latency. We compute the normalized read latency of each file using EDP or CEDP with respect to that of the same file using the baseline. We then compute the average normalized latencies of all files. For CEDP, we set the weight w_j $(1 \leq j \leq 16)$ associated with each storage node as the reciprocal of the link bandwidth of the node. The testbed results are averaged over five runs.

Impact of chunking schemes: We compare the read latency results for both fixed-size and variable-size chunking schemes. In the interest of space, we only present the results of LINUX. Figures 7(a) and 7(b) show the results in the homogeneous testbed for both chunking schemes. On average, EDP reduces the read latency of the baseline by 37.41% and 21.38% for fixed-size and variable-size chunking schemes, respectively. We see that the normalized read latency of EDP is higher with variable-size chunking. The reason is that in variable-size chunking, the read latency may be determined by some chunks that have size larger than the average chunk size. This reduces the performance gap between EDP and the baseline.

Figures 7(c) and 7(d) present the results in the heterogeneous testbed. EDP is agnostic about heterogeneity. Its read latency reduction over the baseline drops to 22.12% for fixed-size chunking; even worse, it increases the read latency of the baseline by 10.79% for variable-size chunking. CEDP addresses the issue by taking into account the heterogeneous bandwidths. It reduces the averaged read latency over the baseline by 52.11% and 25.74% for fixed-size and variable-size chunking schemes, respectively.

Read latency distribution: We further examine the actual read latency distributions, as shown in Figure 8. Figures 8(a) and 8(b) show the read latency distributions for the files in



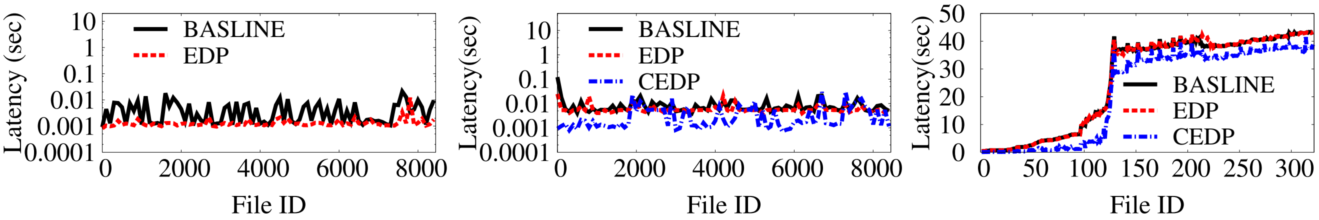
(a) Homogeneous, fixed-size chunking

(b) Homogeneous, variable-size chunking

(c) Heterogeneous, fixed-size chunking

(d) Heterogeneous, variable-size chunking

Fig. 7. Average normalized read latency for files of LINUX.



(a) Homogeneous, LINUX 3.16.3

(b) Heterogeneous, LINUX 3.16.3

(c) Heterogeneous, LINUXTAR

Fig. 8. Read latency distributions of Linux 3.16.3 and LINUXTAR (the files in the x-axis are sorted by the order that they are written to the prototype).

the last Linux version 3.16.3 using fixed-size chunking in both homogeneous and heterogeneous testbeds, respectively. In general, EDP improves the read latency for most of the files compared to the baseline in homogeneous prototype (see Figure 8(a)); in the heterogeneous prototype, CEDP reduces the read latency compared to both the baseline and EDP for most files (see Figure 8(b)). Figure 8(c) shows the read latency distributions for all 321 versions in LINUXTAR. CEDP outperforms the baseline by 35.74%, and can save up to 10 seconds for some backup versions.

D. Computational Overhead

EDP improves file read performance at the cost of extra computational overhead on the write path compared to the baseline. We evaluate the computational overhead of EDP by measuring the processing time of determining the placement of unique chunks in a batch. We conduct our evaluation on a machine with an Intel CPU at speed 3.4GHz. We evaluate processing time by generating unique chunks with `/dev/urandom` and feeding a batch of unique chunks to the EDP algorithm for processing. We vary the number of files inside the batch from 1 to $2,400$. We assume that each file is of equal size.

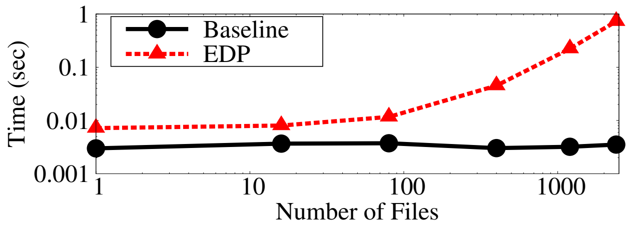


Fig. 9. Batch processing time of the baseline and EDP with various numbers of files in a batch.

Figure 9 shows the processing times of the baseline and EDP algorithms with different numbers of files. As the number of files increases, the processing time of EDP increases at a rate faster than that of the baseline, since its complexity is quadratic to the number of files (see Section V-D). Nevertheless, the processing time remains small. For example, with 960 files, the average processing time of EDP is around 0.20s; if the chunk size is 4KB, the processing throughput is around 328.13MB/s. We can further increase the throughput, for example, by parallelizing the EDP algorithm.

VIII. Related Work

While deduplication effectively reduces storage space, it leads to degradation of read performance due to *fragmentation*, in which logically sequential data is scattered across physical address space. Recent studies propose techniques to improve read performance of deduplication systems, for example, by selective rewrites [12, 14, 17] or hybrid inline/offline deduplication [16]. Such read-enhancement techniques target a single node, while our work considers a distributed setting and improves read performance by maintaining load balance across nodes.

Reliability of chunks in distributed deduplication has also been studied. It is shown that chunks with different popularities have different degrees of reliability, so they should be replicated proportionally [6, 7]. Some distributed storage systems deploy deduplication and erasure coding to improve storage efficiency and data availability [9, 18, 19]. R-ADMAD [18] deploys deduplication with variable-size chunking over erasure-coded storage systems. Hydrastor [9] deploys deduplication and erasure coding for commercial backup storage. Reliability analysis in distributed deduplication is also studied [19]. Our work addresses load balance in a setting with deduplication and erasure coding, which to our knowledge is not addressed in prior studies.

IX. Conclusions

We study the load balance problem in a distributed storage system that employs two well-developed technologies namely deduplication and erasure coding. We point out that deduplication inherently cannot maintain read balance as the deduplicated chunks are unevenly distributed across nodes. We formulate a combinatorial optimization problem, and propose the *Even Data Placement (EDP) algorithm*, a polynomial-time greedy algorithm that minimizes the read balance gap subject to the storage balance constraint. We further extend the EDP algorithm for heterogeneous environments. Extensive trace-driven simulations and testbed experiments show that EDP improves read performance and preserves storage balance. In addition to storage efficiency and data availability respectively provided by deduplication and erasure coding, our work further enhances the parallel I/O performance of a distributed storage system through the design of a load-balance data placement scheme. The source code of our prototype is available for download at **http://ansrlab.cse.cuhk.edu.hk/software/edp**.

Acknowledgments

The work of Min Xu and Patrick Lee was supported in part by GRF CUHK413813 from the Research Grant Council of Hong Kong. The work of Yinlong Xu was supported by the National Natural Science Foundation of China under Grant No. 61379038.

References

[1] Linux kernel source tree. https://www.kernel.org/.

[2] Rackspace Cloud Backup Developer Guide. http://docs.rackspace.com/rcbu/api/v1.0/rcbu-devguide/rcbu-devguide-20140414.pdf.

[3] Symantec NetBackup 7.5. http://eval.symantec.com/mktginfo/enterprise/fact_sheets/b-symc_netbackup_7.5_DS_21219459-2.en-us.pdf.

[4] Whitewater Cloud Storage Gateways with HP Cloud Services. https://www.hpcloud.com/sites/default/files/Riverbed-Solution_brief.pdf.

[5] B. Andrews. Straight Talk about Disk Backup with Deduplication, 2013. ExaGrid.

[6] D. Bhagwat, K. Eshghi, D. D. E. Long, and M. Lillibridge. Extreme Binning: Scalable, Parallel Deduplication for Chunk-based File Backup. In *Proc. of IEEE MASCOTS*, 2009.

[7] D. Bhagwat, K. Pollack, D. Long, T. Schwarz, E. Miller, and J. Pâris. Providing High Reliability in A Minimum Redundancy Archival Storage System. In *Proc. of IEEE MASCOTS*, 2006.

[8] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging Endpoint Flexibility in Data-Intensive Clusters. In *Proc. of ACM SIGCOMM*, 2013.

[9] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki. Hydrastor: A Scalable Secondary Storage. In *Proc. of USENIX FAST*, 2009.

[10] FAL Labs. Kyoto cabinet. http://fallabs.com/kyotocabinet/.

[11] D. Ford, F. Labelle, F. Popovici, M. Stokely, V. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *Proc. of USENIX OSDI*, 2010.

[12] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, F. Huang, and Q. Liu. Accelerating Restore and Garbage Collection in Deduplication-based Backup Systems via Exploiting Historical Information. In *Proc. of USENIX ATC*, 2014.

[13] M. Holland, G. A. Gibson, and D. P. Siewiorek. Architectures and Algorithms for On-line Failure Recovery in Redundant Disk Arrays. *Distrib. Parallel Databases*, 2(3):295–335, 1994.

[14] M. Kaczmarczyk, M. Barczynski, W. Kilian, and C. Dubnicki. Reducing Impact of Data Fragmentation Caused by In-line Deduplication. In *Proc. of ACM SYSTOR*, 2012.

[15] J. Li, S. Yang, X. Wang, and B. Li. Tree-structured Data Regeneration in Distributed Storage Systems with Regenerating Codes. In *Proc. of IEEE INFOCOM*, 2010.

[16] Y.-K. Li, M. Xu, C.-H. Ng, and P. P. C. Lee. Efficient Hybrid Inline and Out-of-Line Deduplication for Backup Storage. *ACM Trans. on Storage*, 11(1):2, 2015.

[17] M. Lillibridge, K. Eshghi, and D. Bhagwat. Improving Restore Speed for Backup Systems that Use Inline Chunk-Based Deduplication. In *Proc. of USENIX FAST*, 2013.

[18] C. Liu, Y. Gu, L. Sun, B. Yan, and D. Wang. R-ADMAD: High Reliability Provision for Large-scale De-duplication Archival Storage Systems. In *Proc. of ACM ICS*, 2009.

[19] Y. Nam, G. Lu, and D. Du. Reliability-Aware Deduplication Storage: Assuring Chunk Reliability and Chunk Loss Severity. In *Proc. of IEEE IGCC*, 2011.

[20] OpenSSL. http://www.openssl.org.

[21] J. S. Plank and K. M. Greenan. Jerasure: A Library in C Facilitating Erasure Coding for Storage Applications – Version 2.0. Technical Report UT-EECS-14-721, University of Tennessee, January 2014.

[22] J. S. Plank, E. L. Miller, K. M. Greenan, B. A. Arnold, J. A. Burnum, A. W. Disney, and A. C. McBride. GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic. Version 1.0. Technical Report UT-CS-13-716, University of Tennessee, September 2013.

[23] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Storage. In *Proc. of USENIX FAST*, 2002.

[24] M. Rabin. *Fingerprinting by Random Polynomials*. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.

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

[26] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing Elephants: Novel Erasure Codes for Big Data. In *Proc. of VLDB Endowment*, 2013.

[27] V. Tarasov, A. Mudrankit, W. Buik, P. Shilane, G. Kuenning, and E. Zadok. Generating Realistic Datasets for Deduplication Analysis. In *Proc. of USENIX ATC*, 2012.

[28] H. Weatherspoon and J. D. Kubiatowicz. Erasure Coding Vs. Replication: A Quantitative Comparison. In *Proc. of IPTPS*, Mar 2002.

[29] B. Zhu, K. Li, and H. Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proc. of USENIX FAST*, 2008.