symmetry

MDPI

Article

Cache-Based Matrix Technology for Efficient Write and Recovery in Erasure Coding Distributed File Systems

Dong-Jin Shin [1] and Jeong-Joon Kim [2,*]

[1] Department of Computer Engineering, Anyang University, Anyang-si 14028, Republic of Korea; djshin@gs.anyang.ac.kr
[2] Department of Software, Anyang University, Anyang-si 14028, Republic of Korea
* Correspondence: jjkim@anyang.ac.kr

Abstract: With the development of various information and communication technologies, the amount of big data has increased, and distributed file systems have emerged to store them stably. The replication technique divides the original data into blocks and writes them on multiple servers for redundancy and fault tolerance. However, there is a symmetrical space efficiency problem that arises from the need to store blocks larger than the original data. When storing data, the Erasure Coding (EC) technique generates parity blocks through encoding calculations and writes them separately on each server for fault tolerance and data recovery purposes. Even if a specific server fails, original data can still be recovered through decoding calculations using the parity blocks stored on the remaining servers. However, matrices generated during encoding and decoding are redundantly generated during data writing and recovery, which leads to unnecessary overhead in distributed file systems. This paper proposes a cache-based matrix technique that uploads the matrices generated during encoding and decoding to cache memory and reuses them, rather than generating new matrices each time encoding or decoding occurs. The design of the cache memory applies the Weighting Size and Cost Replacement Policy (WSCRP) algorithm to efficiently upload and reuse matrices to cache memory using parameters known as weights and costs. Furthermore, the cache memory table can be managed efficiently because the weight–cost model sorts and updates matrices using specific parameters, which reduces replacement cost. The experiment utilized the Hadoop Distributed File System (HDFS) as the distributed file system, and the EC volume was composed of Reed–Solomon code with parameters (6, 3). As a result of the experiment, it was possible to reduce the write, read, and recovery times associated with encoding and decoding. In particular, for up to three node failures, systems using WSCRP were able to reduce recovery time by about 30 s compared to regular HDFS systems.

Keywords: erasure coding; replication; matrix; distributed file system; encoding; decoding

Citation: Shin, D.-J.; Kim, J.-J. Cache-Based Matrix Technology for Efficient Write and Recovery in Erasure Coding Distributed File Systems. *Symmetry* **2023**, *15*, 872. https://doi.org/10.3390/sym15040872

Academic Editor: Alexander Zaslavski

Received: 28 February 2023
Revised: 29 March 2023
Accepted: 4 April 2023
Published: 6 April 2023

1. Introduction

With the advancement of IT technology, various services utilizing big data, artificial intelligence, and the Internet of Things (IoT) have emerged in recent times [1]. In particular, big data, which refers to large and complex data sets, is being generated in various forms, from structured to unstructured data. Traditionally, relational databases have been used to store data. As a file system, Redundant Array of Inexpensive Disks (RAID) has often been used as a method to increase data stability by combining hard disks in an array. However, with the emergence of big data, distributed file systems have become popular as a more convenient and safe way to store data in case of server failure [2].

Distributed file systems typically distribute and store data using replication and EC techniques. In the data storage mode of the replication technique, the original data are divided into blocks, replicated, and distributed across multiple servers. When a command to read data is executed from the master server, the blocks distributed and stored across

the slave servers are combined into a single unit of data. In addition, even if a server fails and becomes unresponsive, the data stored on other servers can be used to recover the lost data [3]. However, due to the cost of distributing and storing data on each server, EC techniques have emerged as an alternative solution to increase the space efficiency of distributed storage systems. Unlike the replication technique, the EC technique uses a unique algorithm to encode the original data and create a parity block. The data block and the parity block are then distributed and stored on each server. Although the EC technique improves symmetrical space efficiency compared to the replication technique, there is an overhead when writing or reading data due to the involvement of many servers and disks [4]. Regarding the overhead, there have been studies on client overhead to solve disk Input/Output (I/O) problems as well as studies on solving the problem of data bottlenecks in terms of network traffic [5,6]. To address the issue of client overhead, this paper proposes a method to maximize the efficiency of the matrix by uploading the matrix generated during encoding and decoding through the EC algorithm to the cache memory, allowing for faster access.

The methodology of cache-based matrix technology applies the WSCRP algorithm for encoding and decoding [7]. The WSCRP algorithm operates based on the Weighting Replacement Policy (WRP) algorithm, which proposes a page replacement policy to be used in cache memory by introducing a new parameter called weights [8]. The WSCRP algorithm proposes a page replacement policy that extends the weights of the WRP algorithm by adding a new parameter called cost. The experiment applied a distributed file system, HDFS, which supports the EC algorithm. The EC algorithm selected for the experiment was Reed–Solomon (RS), and a (6, 3) volume was used, consisting of six data blocks and three parity blocks. The experimental evaluation targeted systems without cache memory, systems with Least Recently Used (LRU) and Least Frequently Used (LFU) basic cache memory algorithms, and systems with WRP and WSCRP algorithms, comparing a total of five writing times, reading times, and recovery times.

This paper begins with an introduction in Section 1 and examines the basic theory of distributed file systems in Section 2. Section 3 provides a summary of the improved methodologies in EC-based distributed file systems, along with related studies. Section 4 introduces the proposed cache-based matrix technology. In Section 5, the paper compares systems that applied the methodology to the distributed file system. Finally, Section 6 presents our conclusion.

2. Background

HDFS is a representative distributed file system, provided in an open-source format through the Apache Project, and many companies use it to build services. In a replication method of distributed file systems, such as HDFS, the write mode divides the original data into blocks, and the divided blocks are replicated and stored in each server. For instance, in replication techniques with 3x Factor elements, storing 300 GB (Gigabyte) of data requires 900 GB of storage space. Although data stability is increased in preparation for server failure, there is a disadvantage in that more physical capacity is required for symmetrical space efficiency. Figure 1 shows how data are written when nine nodes (servers) are configured through replication techniques.

When the storage command is given to the original data from the name node, the data are divided into blocks, and each block is replicated and written to the data node.

Distributed file systems that use EC instead of replication techniques are widely used to address symmetrical space efficiency aspects [9]. EC is a method of encoding and storing data using a unique algorithm. Representative algorithms exist in various forms, including XOR, RS, Liberation, and Weaver Code [10–13]. When data are written to the EC-based distributed file system, the original data are divided into data blocks and parity blocks through encoding. When data are read or recovered, data loss can be prevented in case of server failure through decoding. If the data are encoded using the RS (6, 3) volume, the original data are divided into six data blocks and encoded using the RS algorithm to

generate three parity blocks, which are then written to the storage. A total of nine servers are required to write the data, as indicated by the notation RS (6, 3). For instance, if you store 300 GB of data, only 300 GB of data blocks and 150 GB of parity blocks generated through encoding are needed, resulting in a total of 450 GB. This is more symmetrical and space-efficient than the replication technique. Figure 2 shows how blocks are distributed and written after encoding when the original data storage mode is initiated on the RS (6, 3) volume.

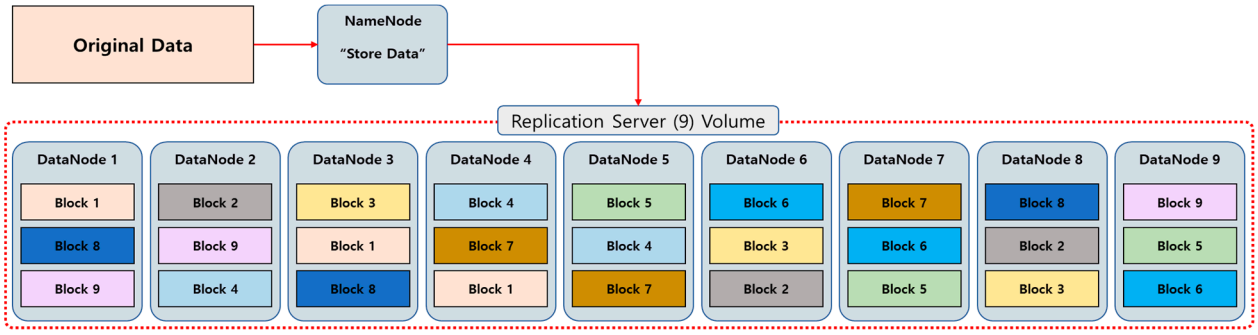

Figure 1. Distributed file system structure based on replication techniques.

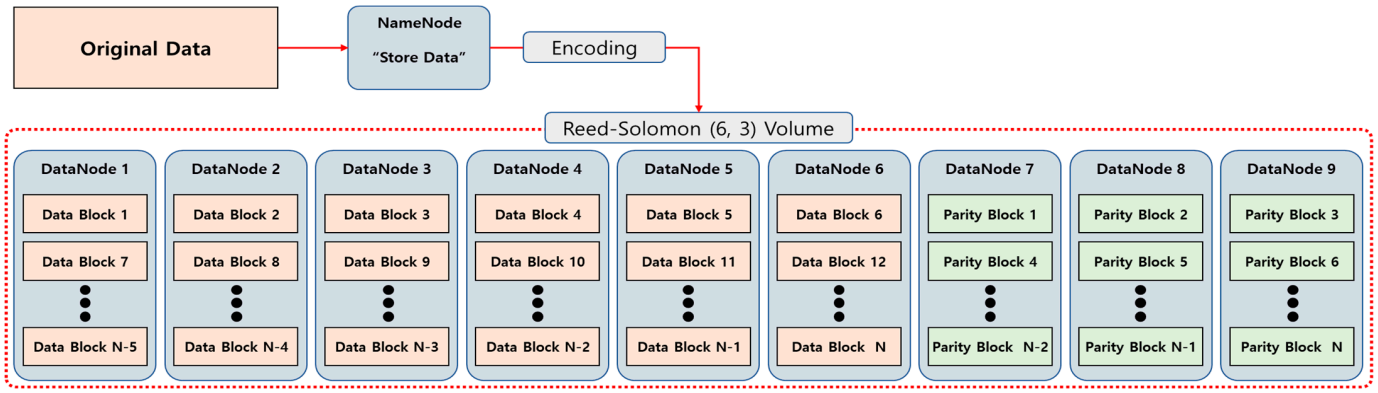

Figure 2. Distributed file system structure based on erasure coding techniques.

To summarize, the replication technique divides the original data into blocks and replicates and writes them. In contrast, the EC technique divides the original data into data blocks and generates parity blocks through encoding, and then distributes and writes them. In the case of RS (6, 3) volume, the original data are divided into six data blocks and written, and three parity blocks are generated through encoding and written. The replication technique requires more physical capacity as it stores multiple copies of data, while the EC technique offers a more symmetrical, space-efficient solution.

In a distributed file system, symmetrical space efficiency is calculated based on the number of failed servers and the number of servers that can participate in the recovery process. Table 1 shows the notation for the calculation of Equation (1), and Table 2 shows the comparison of symmetrical space efficiency of the distributed file systems shown in Figures 1 and 2, according to the calculations in Equation (1).

Table 1. Notation and symbols of Equation (1) and description.

Notation and Symbol	Definition
K	The number of data blocks
M	The number of blocks required for recovery

$$Space\ Efficiency = \frac{K}{K+M} \quad (1)$$

Table 2. Comparison of replication techniques and EC techniques.

Techniques	Fault Tolerance	Symmetrical Space Efficiency
Nine-way replication	8	33%
EC RS (6, 3)	3	67%

In the replication technique, K represents the number of original data blocks, while M represents the number of replicated blocks. In the EC technique, K represents the number of original data blocks, and M represents the number of parity blocks generated through encoding [14]. Since the replication technique consists of nine servers that copy blocks and write them to each server, even if only one server is available due to the failure of eight servers, data can be recovered if all the blocks necessary for recovery exist. However, the symmetrical space efficiency of the replication technique is low at 9/(9 + 18) = 33%. The EC-based RS (6, 3) volume uses the same nine servers as the replication technique, but three servers write parity blocks. If more than four servers fail beyond the number of parity blocks that can participate in the calculation, recovery cannot be performed. The replication technique recovers by combining divided blocks, while the EC technique is a recovery method that reconstructs data by calculating a specific algorithm. When comparing the symmetrical space efficiency, the EC technique is much better, with a difference of almost two times, 6/(6 + 3) = 67%. Figure 3 shows the encoding process in the EC-based distributed file system.

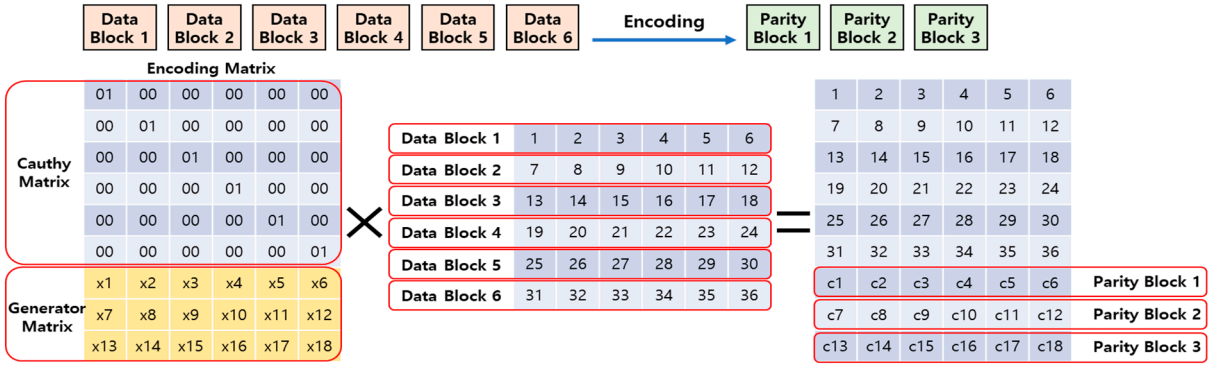

Figure 3. EC-based distributed file system encoding process.

When encoding is performed on the RS (6, 3) volume, the original data are divided into six data blocks. The six divided data blocks generate three parity blocks through an encoding process. The encoding process is expressed in Equation (2), and the related notation is shown in Table 3.

$$PB_i = \sum_{j=1}^{k} GM_{i,j} DB_{i,j}\ (j \leq 6) \quad (2)$$

Table 3. Notation and symbols of Equation (2) and description.

Notation and Symbol	Definition
PB_i	The parity blocks (i = rows)
$GM_{i,j}$	The generator matrix (i = rows, j = columns)
$DB_{i,j}$	The data blocks (i = rows, j = columns)

It is assumed that each piece of data of the six data blocks is composed of six items in a sequence. The generator matrix with three rows and six columns is created in a structure

consistent with the RS (6, 3) volume used, and a parity block is generated by summing the values of each data block and the values of the corresponding rows and columns of the generator matrix. The generator matrix is composed of a partial matrix with zero and a generator matrix with three rows and six columns, which is determined using the Cauchy matrix, depending on the volume of RS (6, 3) [15]. Figure 4 shows the decoding process of the EC-based distributed file system.

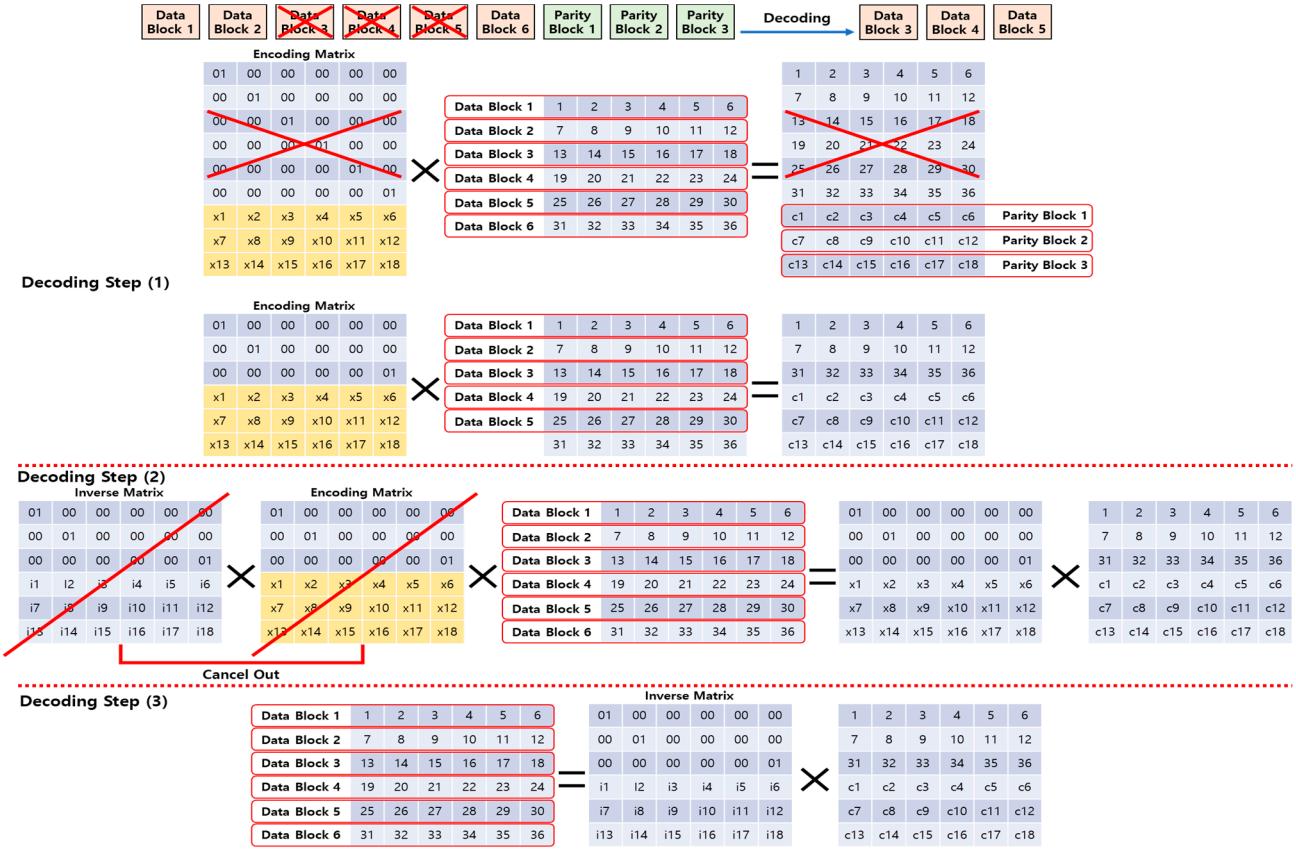

Figure 4. EC-based distributed file system decoding process.

Errors have occurred in Data Block 3, Data Block 4, and Data Block 5, which need to be decoded. To do this, the corresponding row of the Cauchy matrix part used by Data Block 3, Data Block 4, and Data Block 5 must be deleted from the encoding matrix (Step 1). This generates an inverse matrix of the deleted encoding matrix, which is then multiplied on the left and right sides. The inverse matrix of the left side and the encoding matrix from which the error removed matrix are removed and are multiplied using the canceling out method (Step 2). Eventually, only the data composed of data blocks can be left, enabling the recovery of data blocks (Step 3). In other words, decoding involves calculating the inverse matrix through the encoding process of recovering the original data by using the previously written parity block. If expressed as a formula, the decoding process is shown in Equations (3)–(5), and the related notation is shown Table 4.

$$(EM_{i,j} - ER_{i,j})DB_{i,j} = PB_i, \;\; IM_{i,j} = (EM_{i,j} - ER_{i,j})^{-1} \tag{3}$$

$$IM_{i,j}(EM_{i,j} - ER_{i,j})DB_{i,j} = PB_i IM_{i,j} \tag{4}$$

$$DB_{i,j} = PB_i IM_{i,j} \tag{5}$$

During the decoding process, $(EM_{i,j} - ER_{i,j})$ is obtained by removing the row corresponding to the error from the encoding matrix using Equation (3), and then the inverse matrix of $(EM_{i,j} - ER_{i,j})$, $IM_{i,j}$, is calculated. Next, $IM_{i,j}$ is multiplied by both the left and

right sides of the original encoding matrix using Equation (4). Finally, multiplying $IM_{i,j}$ and $(EM_{i,j} - ER_{i,j})$ using Equation (5) results in only $DB_{i,j}$, the data block that needs to be restored, remaining. This step uses the canceling out method to remove any unnecessary elements. Through this process, the decoding matrix is created by deleting the erroneous row from the encoding matrix, finding its inverse, and performing matrix multiplication.

Table 4. Notation and symbols of Equations (3)–(5) and description.

Notation and Symbol	Definition
$EM_{i,j}$	The encoding matrix (i = rows, j = columns)
$ER_{i,j}$	The error is removed matrix (i = rows, j = columns)
$IM_{i,j}$	The inverse matrix of encoding matrix (i = rows, j = columns)

3. Related Works and Contributions

There exists a study that analyzes the problems occurring in the EC-based distributed file system, along with the cost analysis. Furthermore, there also exists an overview study that summarizes methodologies for improving the EC-based distributed file system [5,6]. In [5], the author divided the encoding/decoding process of the EC-based distributed file system into five steps—namely, client overhead, master process, parity calculation, data distribution, and slave process—and analyzed the cost of each step with increasing network speeds from 1 G to 100 G. The analysis revealed that the client overhead and data distribution processes were the costliest at 100 G network speeds. In [7], the author describes the overall flow of the study conducted on the EC distributed file system. Additionally, the EC algorithm is broken into four stages, providing a detailed introduction to the algorithm's characteristics and processes. Based on the aforementioned studies, this paper classified two methods for improving EC-based distributed file systems and organized related studies by directly investigating them. Section 3.1 includes studies that redesign EC-related algorithms or apply various recovery methods, whereas Section 3.2 focuses on studies that aim to reduce the bandwidth and traffic generated by encoding and decoding in terms of networks.

3.1. A Study of Erasure Coding Distributed File System Algorithm

In [16], the author proposes a new algorithm used in EC called Local Reconstruction Code (LRC), which reduces storage overhead by reducing the I/O bandwidth required for writing/reading compared to the previously used EC algorithm. In [17], the author applies the piggybacking methodology to the RS algorithm and proposes a new algorithm that reduces disk usage by approximately 30%. In [18], the author proposes an algorithm that presents an optimal recovery scenario for single-node failures using the Hadamard matrix. In [19], the author performed more optimization than the existing RS algorithm by redesigning the linear algebra used in RS. In [20], the author proposed Zebra, which uses various parameter values to encode the data hierarchically. Zebra automatically determines the number of layers based on data characteristics and operates dynamically, reducing storage and decoding overhead. In [21], the authors analyze the problems that arise when performing parallel recovery in EC and present three solutions: the disk contention avoidance method, the chunk allocation method, and the asynchronous recovery method. In [22], the author proposed clay code, a new algorithm for improving recovery bandwidth and disk I/O, to improve low storage overhead. In [23], the author proposed Founsure, a library that can solve the encoding and decoding costs arising from existing EC algorithms and increase the recovery bandwidth of data. In [24], the author presented a methodology that would improve the basic XOR operations of the EC algorithm to operate more efficiently, comparing it to the Intel-developed Jerasure Library. In [25], the author proposes a methodology in which system clients share files to increase the block access rate of distributed file systems. The smaller the client's cache size, the lower the performance improvement. However, as the client's cache size increases, the hit rate also increases, resulting in better

performance. In [26], the author combines low-latency persistent memory modules with distributed file systems. The cache between the client and the distributed file system maximizes file consistency and minimizes I/O overhead. In [27], the author proposes EC-Cache to address in-memory object caching dependencies arising from distributed file systems and to mitigate load balancing and I/O performance issues. The proposed system uses the same amount of memory but improves writing and reading times and enhances network load balancing and I/O performance. In [28], the author conducted a study by implementing a distributed layer cache system using the cache memory method on HDFS, a distributed file system. Files loaded from HDFS were cached in shared memory with direct access to the client library to improve file read and response times. In [29], the author proposes a full replica solution, a new cache policy, instead of a cache method consisting of fragments in an EC-based distributed file system. As a result of the simulation experiment, the cache hit rate increased, and the response time improved compared to the existing model.

3.2. A Study of Erasure Coding Distributed File System Network Traffic

In [30], the author proposed a delay recovery methodology that would allow the node to skip the failed node and perform the ensuing recovery operations to increase the overall recovery bandwidth. In [31], the author proposed a distributed reconstruction technique where data are not recovered by combining them in one node during decoding. Instead, partial parallel recovery can be performed in multiple nodes. In [32], the author proposed a T-Update methodology that transforms the structure of a complete node system into a tree structure to shorten the encoding update time that occurs during data updates. In [33], the author proposed a methodology for pipelining the node recovery process, which transfers the required blocks sequentially from the first node to the last node. In [34], the author proposed a XORInc framework, which encodes and decodes network flows in XOR form to effectively reduce network traffic and eliminate network bottlenecks. In [35], the author proposed an AggreTree method that allows for temporary aggregation and the partial decoding of blocks sent from servers and switches to effectively reduce traffic and eliminate bottlenecks in a network topology consisting of servers and switches. In [36], the author proposed NetEC, which improves the overall recovery performance by developing an Application-Specific Integrated Circuit (ASIC) switch that can apply the EC algorithm and aggregate the blocks received from the switch. In [37], the author proposed a methodology called File Aware Graph Recovery that enables recovery from a file perspective by mapping the information obtained during recovery, including file, block location, stripe shape, and access frequency. In [38], the author analyzes the costs associated with recovering data when multiple nodes fail and proposes an optimization methodology that uses a tree structure to minimize the costs incurred during recovery of both encoded blocks and original files.

3.3. Contribution

EC-based distributed file systems have been studied in two significant categories: improving algorithms and controlling network traffic. This paper proposes a cache-based matrix technique that can reduce client overhead introduced in [5]. Client overhead is the process of reducing the overhead that occurs during encoding and decoding. While some studies have focused on improving EC algorithms or suggesting new recovery methodologies, there are no studies that utilize the matrix used for encoding and decoding. Therefore, this paper analyzes the problem of matrices arising from encoding and decoding and proposes a methodology to upload the matrix to cache memory for quicker access, reducing unnecessary overhead, and increasing write, read, and recovery performance.

4. Problem Analysis and Methodology Proposal

This section analyzes the problems that arise from the matrix used in encoding and decoding in EC-based distributed file systems. To solve these problems, the paper proposes a cache-based matrix technology.

4.1. Matrix Analysis for Encoding and Decoding

In an EC-based distributed file system, data are encoded and written into data blocks and parity blocks based on the EC volume used during the encoding process. Data can be read through decoding when no server failure occurs. In addition, in the event of a failure of the server, an inverse calculation can be performed by the EC algorithm to recover the data. When encoding and decoding instructions are executed, matrices are generated according to the set volume, which are used to store and recover data. In particular, the matrix required for the decoding process varies depending on several EC properties, including the EC algorithm used, the EC volume size, the location of the failed server, and the number of failed servers. Figure 5 shows an example of a single server failure in a distributed file system configured with RS (6, 3) volumes.

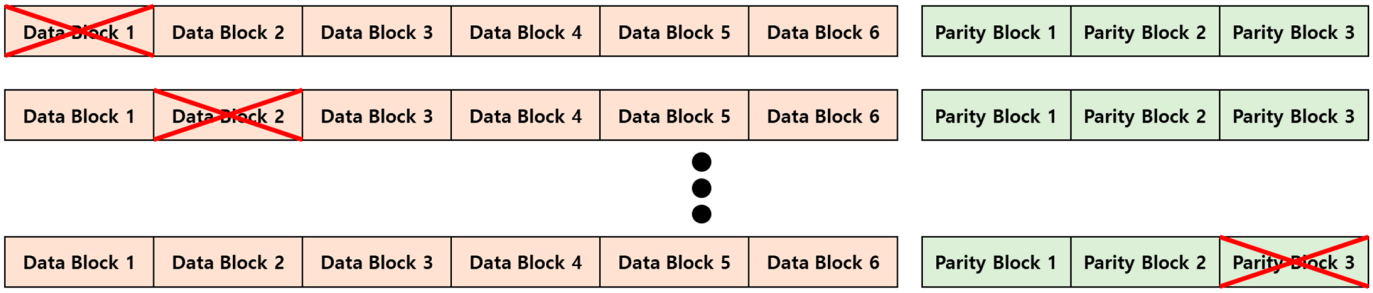

Figure 5. Per-server failure on RS (6, 3) volumes.

Figure 5 shows a scenario where a single failure occurs among a total of nine servers, including servers that own Data Block 1 through Parity Block 3. When a decoding process is initiated for each server failure, a total of nine decoding matrices are generated, since a single failure occurs on nine servers. By calculating the number of cases when multiple failures occur, a total of 36 cases can be produced. The number of matrixes generated is equal to the number of failures allowed, which is the number of servers holding parity blocks, within the total number of servers. This can be expressed by Equation (6), which calculates the number of cases, and the related notation is shown in Table 5 [39].

$$Number\ of\ Decoding\ Matrix(nCr) = \frac{(n) \times (n-1) \times (n-2) \cdots (n-r+1)}{r \times (r-1) \times (r-2) \cdots 3 \times 2 \times 1} = \frac{n!}{(n-r)! \times r!} \quad (6)$$

Table 5. Notation and symbols of Equation (6) and description.

Notation and Symbol	Definition
n	The number of data node servers
r	The number of server fault
C	The combination calculates

For instance, if we calculate the triple failure of the RS (6, 3) volume, the result would be 84, which is equivalent to ${}_9C_3$. In an EC distributed file system that is configured with RS (6, 3) volumes, up to three server failures can be tolerated, and a maximum of 84 decoding matrices can be generated. The calculation of the number of cases from a single failure to multiple failures in various forms of EC volume is expressed in Table 6 by utilizing Equation (6).

Table 6. Number of decoding matrices based on the number of server failures.

Volume	Single Failure	Double Failure	Triple Failure	Quadruple Failure
RS (3, 2)	5	10	-	-
RS (6, 3)	9	36	84	-
RS (10, 4)	14	91	364	1001

Table 6 shows the maximum number of matrices that can be generated according to the number of server failures in the EC volume. When configured with RS (10, 4), a maximum of 1001 decoding matrices are generated from quadruple failure. However, generating all these matrices can result in unnecessary overhead. To address this issue, matrices that have the same structure as those used in previous encoding and decoding processes can be reused without having to create new ones. If these matrices are uploaded to cache memory, they can be accessed more quickly during encoding and decoding, which can help minimize unnecessary overhead.

4.2. Methodology of Cache-Based Matrix Technology

In order to upload a matrix to the cache memory and utilize it, the design of the cache memory is crucial. To implement the techniques proposed in this paper, the cache memory technique utilized is the WSCRP algorithm. The author has analyzed the problems associated with the replacement algorithm that is typically used in cache memory and proposed a new high-performance cache replacement algorithm called WSCRP, which is based on the WRP. WRP is an improved algorithm that builds on both LRU and LFU algorithms. The access time, which is an important performance indicator for cache memory, is faster than that of the main memory. To address this issue and efficiently improve cache memory access time, an adaptive replacement policy has been proposed. The adaptive replacement policy operates based on the LRU and LFU algorithms but assigns weights to each page based on specific parameters. These weights rank the pages based on their recentness, frequency, and reference rates. Additionally, since there is a reference rates parameter, a scheduling scheme in which a page with a low reference ratio is given higher priority is supported. In other words, the WRP algorithm prioritizes objects to be uploaded to the cache memory through weights and replaces low-ranking pages with new pages. Equation (7) shows the WRP algorithm, and the related notation is shown Table 7.

$$W_i = \frac{L_i}{F_i \times \Delta T_i} \tag{7}$$

Table 7. Notation and symbols of Equation (7) and description.

Notation and Symbol	Definition
W_i	The object i's weight value
L_i	The time of object i's last data access
F_i	The object i's frequency
$\Delta T_i\left(T_{ci} - T_{pi}\right)$	The average value of two data access times (T_{ci} is the last accessed time of object i, and T_{pi} is the penultimate accessed time)

If block j is in the buffer, a hit occurs in the cache memory, and the policy operates as follows if it is referenced:

1. L_i will be changed to $L_i + 1$ for every $i \neq j$.
2. For $i = j$, first we put $\Delta T_i = L_i$, $F_j = F_j + 1$ and then $L_j = 0$.

However, if the referenced block j is not in the buffer, a miss occurs, and the algorithm selects a block whose value of the weight function in the buffer is the highest among other values. When selecting, it searches for the object with the highest weight from the top to the bottom of the buffer, and if the weights of the objects are the same, the object that has been in the buffer for the longest time is selected for replacement. As a result, the weight values of the blocks in the buffer are updated on every access to the cache.

The WSCRP algorithm incorporates cost parameters into the adaptive replacement policy used in the WRP algorithm. Unlike the existing WRP algorithms, the WSCRP algorithm considers the size of the objects being replaced with cache memory when computing their weights. This prevents performance degradation that could occur based on the size of the objects being replaced. The weights of objects are calculated by adding cost parameters to the existing adaptive replacement policy. The WSCRP algorithm is shown in Equation (8), and the related notation is shown in Table 8.

$$WC_i = \frac{L_i}{F_i \times \Delta T_i} \times \frac{S_i}{CSV_i}, \ \ (CSV_i = C_i S_i H_i \div \sum_{i=1}^{n} C_i S_i R_i) \tag{8}$$

Table 8. Notation and symbols of Equation (8) and description.

Notation and Symbol	Definition
WC_i	The object *i*'s weight–cost value
S_i	The object *i*'s size
CSV_i	The object *i*'s cost-saving values
C_i	The object *i*'s network traffic consumption
H_i	How many valid copies of object *i* found in the cache
R_i	The total number of times data were requested

In the WSCRP algorithm, the S_i parameter represents the object's size, while the CSV_i parameter is a cost value added to the adaptive replacement policy. The primary goal of the WSCRP algorithm is to save resources between the cache and the main memory. The cost value determines when an object is removed from the cache. As the cost value determines when an object is removed from the cache. As the cost value approaches that of all objects, the overall consumption can be calculated, and the cost of consuming a single object can be determined. If the replacement cost of any one object is higher, caching it can result in cost savings. In the case of an object, if the value CSV_i is significant, it should be stored in the cache because it is more costly to perform a replacement and recache the cache. Therefore, this paper improves cache performance by adding the CSV_i parameter to the existing Equation (7) to determine which object should be removed from the cache when the cache is full. The S_i parameter takes up more cache space for extensive data, which may cause cache garbage to occur. This can result in unnecessary areas being freed up among the memory areas dynamically allocated by the program, thereby reducing the hit rate and increasing average access time. Therefore, it is best to prioritize caching small data. In addition, the CSV_i parameter can save the replacement cost of an object because it replaces the data with a more significant weight value if the cost of replacing one piece of data is higher than another.

The cache memory table structure is sorted in ascending order according to the *WC* value calculated in Equation (8). If an object existing in the cache memory is searched and hit, the *WC* value is recalculated and rearranged in ascending order. In other words, objects with low values move up because they are likely to be referenced, and objects with high values move down because they are less likely to be referenced again. Therefore, if the *WC* value is more significant than that of other objects, this object is considered unimportant, and it is removed first when the cache memory is full. If an object with the same *WC* value exists, the F_i parameter, which indicates the object's reference rates, is used as the second attribute to determine the object to be removed from the cache.

Cache-based matrix technology operates based on the WSCRP algorithm and aims to maximize the hit rate to effectively use the matrix uploaded to the cache memory. When a request for a matrix is received, the system first checks the cache memory to see if the matrix exists for storing and recovering data. If a matrix is present in the cache memory for encoding and decoding, it is returned to the name node where the command was executed without generating a new matrix. However, if a matrix does not exist in the cache memory, the system considers it as a new instruction method, searches for currently available data nodes, creates a new matrix, returns it to the name node, and adds it to the cache memory.

The *WC* value of the uploaded matrix is then calculated using the weight–cost model, and the *WC* value of the cache memory table structure is updated in ascending order. If there is insufficient space to store the matrix, the cache memory table alignment structure removes a matrix according to the update method until enough space is available to store the new data. The following Algorithm 1 shows the cache-based matrix technique.

Algorithm 1: Cache-based matrix technology.

```
1.  Parameter SCM: Calculates the size of the cache memory available
2.  Parameter SRM: Calculates the size of the requested matrix
3.  Parameter WC: Weight–cost values of matrix uploaded to cache memory
4.  If the requested matrix exists in the cache
5.      Find the matrix in the cache memory table
6.      Update cache memory table
7.  Else
8.      Check the live data nodes
9.      Available data nodes list transfer to name node
10.     Create matrix
11.     Add matrix to cache
12.     While SCM < SRM
13.       If there is only one maximum WC
14.         Remove the maximum WC in the cache memory table
15.       Else
16.         Use frequency value to calculate and remove WC
17. Update cache memory table
```

The algorithm for cache-based matrix technology first calculates the reference value *WC* through the weight–cost model when uploading to cache memory. It then calculates the free space of the cache memory, marks it as the size of cache memory (SCM), calculates the size of the matrix to be uploaded, and marks it as the size of the request matrix (SRM). If the name node performs encoding for data writing and decoding for reading and recovery, it checks whether the requested matrix exists in the cache memory. If it is present, the relevant matrix is found in the cache memory table, and encoding and decoding are performed through the matrix. The cache memory table is then updated by sorting the *WC* values of the used matrix in ascending order (Steps 3–5). If no matrix is found in the cache memory, a matrix must be generated. This is achieved by sending a list of available data nodes to the name node, which generates the matrix, uploads it to the cache, and performs encoding and decoding (Steps 6–10). If there is insufficient space in the cache memory to store the matrix, the algorithm removes the matrix with the maximum *WC* value through SCM and SRM. If there are multiple *WC* values, the algorithm removes the *WC* value, using frequency value as the second attribute (Steps 11–16).

4.3. Structure Design and CPU Performance of Cache-Based Matrix Technology

There are different architectural levels where caches can be located, but they are typically placed close to the front end to reduce the time and cost of accessing the backend service. Cache memory management methods can be classified into three types: centralized, global, and distributed cache structures. In a centralized structure, only one node can receive requests and access data. In a global structure, all nodes can access data using a single cache space. In a distributed cache structure, each node can access data using its own cache space. In this paper, the cache-based matrix technique employs a global structure where all data nodes use the cache space managed by the name node. The data node queries the name node for the matrix that encodes and decodes data, and the name node itself queries the storage space for data and delivers the matrices to the requested data node. If a distributed cache structure were used, each data node would manage its own matrix, which would lead to inefficient management due to the large number of resources, which would need to be managed by the name node. In addition, since the matrix is generated when

the name node performs data writing, reading, and recovery operations, the probability of cache misses would be high if each data node managed its own matrix. Therefore, a global cache structure is advantageous because it allows the name node to generate and store the matrix through responses from multiple data nodes, thereby reducing cache misses.

The encoding and decoding operations involved in erasure coding can be computationally intensive and require a significant amount of CPU resources. Therefore, a faster CPU can generally perform these operations more quickly, which can improve the overall performance of the distributed file system. Different models of CPUs have different architectures and features that can affect their performance in various ways. For example, some CPU models may have more cores or higher clock speeds, which can help improve the performance of encoding and decoding operations. Additionally, different CPU models may have distinct instruction sets, which can affect the performance of specific algorithms used in erasure coding. However, it is worth noting that other factors, such as memory bandwidth, storage speed, and network bandwidth, can also impact erasure coding performance. Therefore, while CPU performance is an essential factor to consider, it is not the only one. Consequently, it is important to consider the specific model and characteristics of the CPU when designing and optimizing erasure coding algorithms for distributed file systems.

5. Experiments

This section introduces the experimental environment and results used in the evaluation. In this paper, we compare five systems: basic HDFS, the basic cache memory method LRU, HDFS with LFU applied, and HDFS with WRP and WSCRP applied, using the methodology described in Section 4.

5.1. Experimental Environment

The algorithm described in Section 4 was implemented using the Java programming language and operated as a simulation environment. The simulation of the cache-based matrix technology was conducted on a workstation running the Ubuntu 20.04.4 operating system, with a XEON 4110 (8 core $\times$ 2) CPU, 128 GB of DDR4 memory, a 20 TB (Terabyte) HDD Disk, and four RTX 2080 graphics cards. The version of Java Development Kit (JDK) used for development was 11.0.9, and the distributed file system that the cache-based matrix technology was applied to was HDFS. The version of Hadoop that includes HDFS is 3.3.2. The EC policy supported by HDFS includes RS and XOR algorithms, and it determines the number of data and parity blocks. The available EC policies were RS-3-2-1024k, RS-6-3-1024k, RS-10-4-1024k, RS-LEGACY-6-3-1024k, and XOR-2-1-1024k. In this paper, the RS algorithm was applied using RS-6-3-1024k. The data block was divided into six servers, and the parity block was divided into three servers, requiring a total of ten servers because a name node was also needed. The striping cell size is 1024k; this determines the granularity of reading and writing the stripe, including buffer size and encoding works. In an EC-based distributed file system, the size of a cell refers to the unit used when blocks are calculated and stored through encoding. The block cells are composed of 1024 kilobytes. A stripe is a single unit stored as a data block, encoded, and stored when generating a parity block. For instance, if Data Blocks 1, 2, 3, 4, 5, and 6 are encoded to generate Parity Blocks 1, 2, and 3, one stripe is composed of nine blocks.

The experimental method involved generating dummy data of 10 GB and writing the data using encoding commands. To generate a particular size of data, the fallocate command supported by Ubuntu was used, and the "−l" option was added to create a dummy data file with a size of 10 GB using the "fallocate −l 10g dummy data" command, which was then encoded and written. For decoding, file reading times were measured when data nodes did not fail, and recovery time was measured by randomly failing data nodes. In case of a random failure, the number of failed servers was not allowed to exceed the number of servers that wrote the parity blocks described in Section 2. In other words,

since there were a total of nine servers (six servers storing data blocks and three servers storing parity blocks), only single, double, and triple failures had been set up.

The experimental metrics measured the time taken to encode (write) and decode (read and recover) the data, and the hit rate of the cache memory followed Equation (9):

$$Hit\ Rate(\%) = \frac{Number\ of\ cache\ hits}{(Number\ of\ cache\ hits + Number\ of\ cache\ misses)} \times 100 \quad (9)$$

Five items were used for comparative analysis: basic HDFS RS (6, 3) without cache memory technology, HDFS RS (6, 3) with basic cache memory structure LRU and LFU, WRP HDFS RS (6, 3) based on WRP, and WSCRP HDFS RS (6, 3) proposed in this paper.

5.2. Experiments Result and Discussion

The experimental evaluation first shows the result of measuring the overhead of the matrix.

In Figure 6, the time cost of uploading and downloading matrices generated during encoding and decoding to the cache memory is shown. As the number of encoding and decoding operations increases from 100 to 1000 times, the upload and download time of the matrix gradually increases. This is due to the increase in the number of matrices generated according to the number of encoding and decoding operations. However, the measurement time remains at around 4 s for the BASIC HDFS RS (6, 3) system, which does not utilize cache memory. Among the systems that utilize cache memory, the WSCRP HDFS RS (6, 3) system shows the lowest time cost, with a maximum encoding time of 6.4 s and a maximum decoding time of 28.4 s.

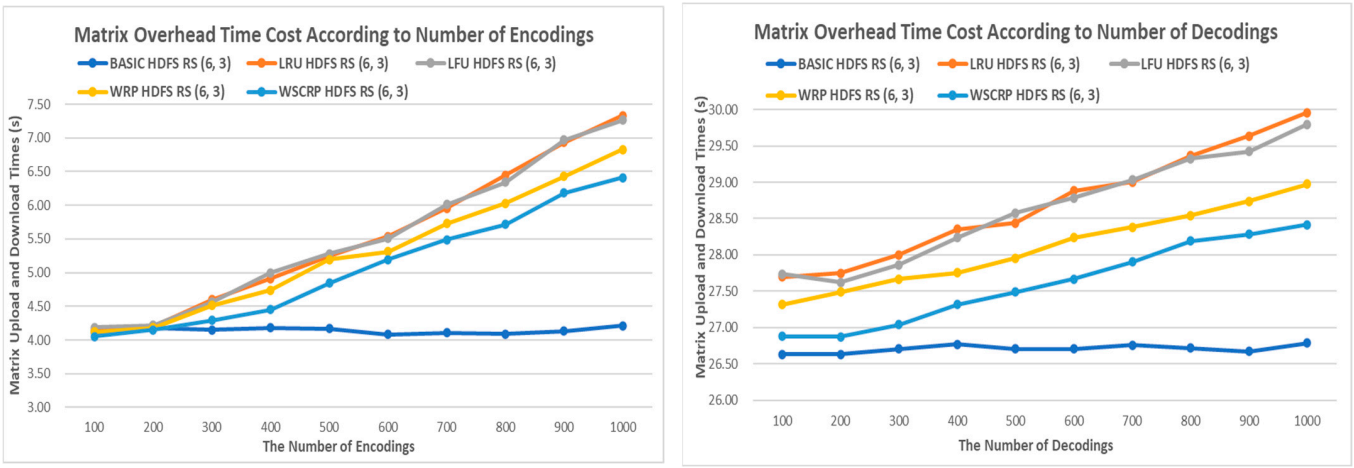

Figure 6. Matrix overhead time cost according to the number of encodings and decodings.

Figure 7 shows the space cost of the matrices stored in the cache memory. As the number of encodings and decodings increases, the matrices are cached, and the cache size increases gradually. The BASIC HDFS RS (6, 3) system does not utilize cache memory, and the initial measured size is maintained at approximately 14% during encoding and approximately 20% during decoding. However, among the systems that use cache memory, the WSCRP HDFS RS (6, 3) system can store more matrices as the number of encodings and decodings increases. The matrix size evaluated in terms of matrix overhead time and space cost is small at about 16 kilobytes. Although the time cost increases with an increasing number of encodings and decodings, it does not significantly affect the 1 Gbps network speed used in the experiment. Additionally, the space cost is sufficient to store the matrices, and systems utilizing cache memory can efficiently store the matrices through a replacement policy. Therefore, the costs of writing, reading, and recovering data are significantly affected, rather than the time and space cost of matrix overhead [40].

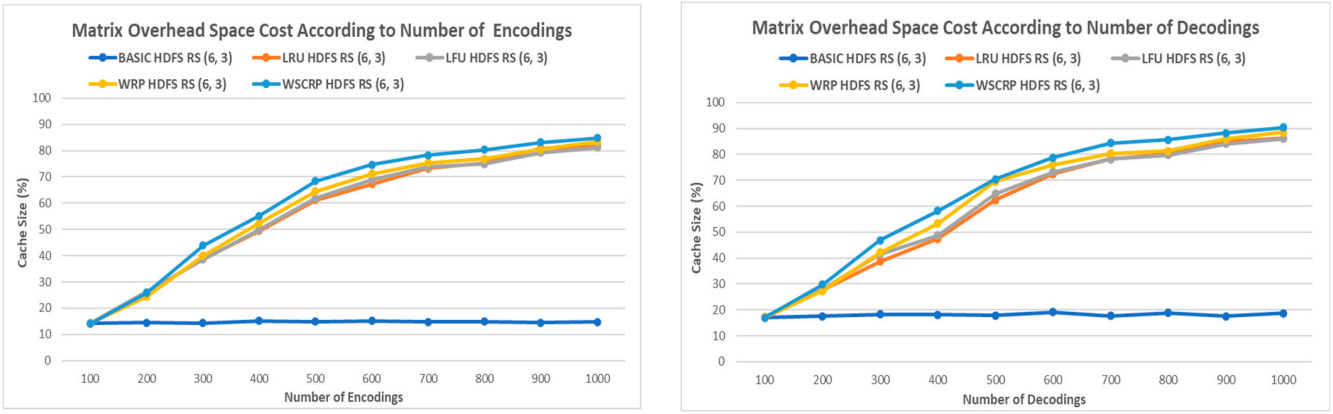

Figure 7. Matrix overhead space cost according to the number of encodings and decodings.

Figure 8 shows the writing time of encoding the generated data from 100 to 1000 times. As the number of encoding increases, the number of generated encoding matrices increases, so the overall writing time increases. Measurements from 100 to 1000 times showed that BASIC HDFS RS (6, 3) took 23 s to 45.6 s, LRU HDFS RS (6, 3) took 21.3 s to 42.1 s, and LFU HDFS RS (6, 3) took 21.1 s to 41.2 s. WRP HDFS RS (6, 3) took 19.9 s to 39.2 s, and WSCRP HDFS RS (6, 3) took 19.8 s to 36.8 s. When comparing BASIC HDFS RS (6, 3) with WSCRP HDFS RS (6, 3) based on 1000 encoding times, the writing time can be shortened by about 10 s. All the systems, except BASIC HDFS RS (6, 3), generate an encoding matrix, upload it to cache memory, and reuse the uploaded matrix for the next encoding, resulting in a small reduction in the overall writing time. In particular, WSCRP HDFS RS (6, 3) considering weights and parameters can save the most in writing time compared to LRU HDFS RS (6, 3) and LFU HDFS RS (6, 3) applying basic memory cache structures.

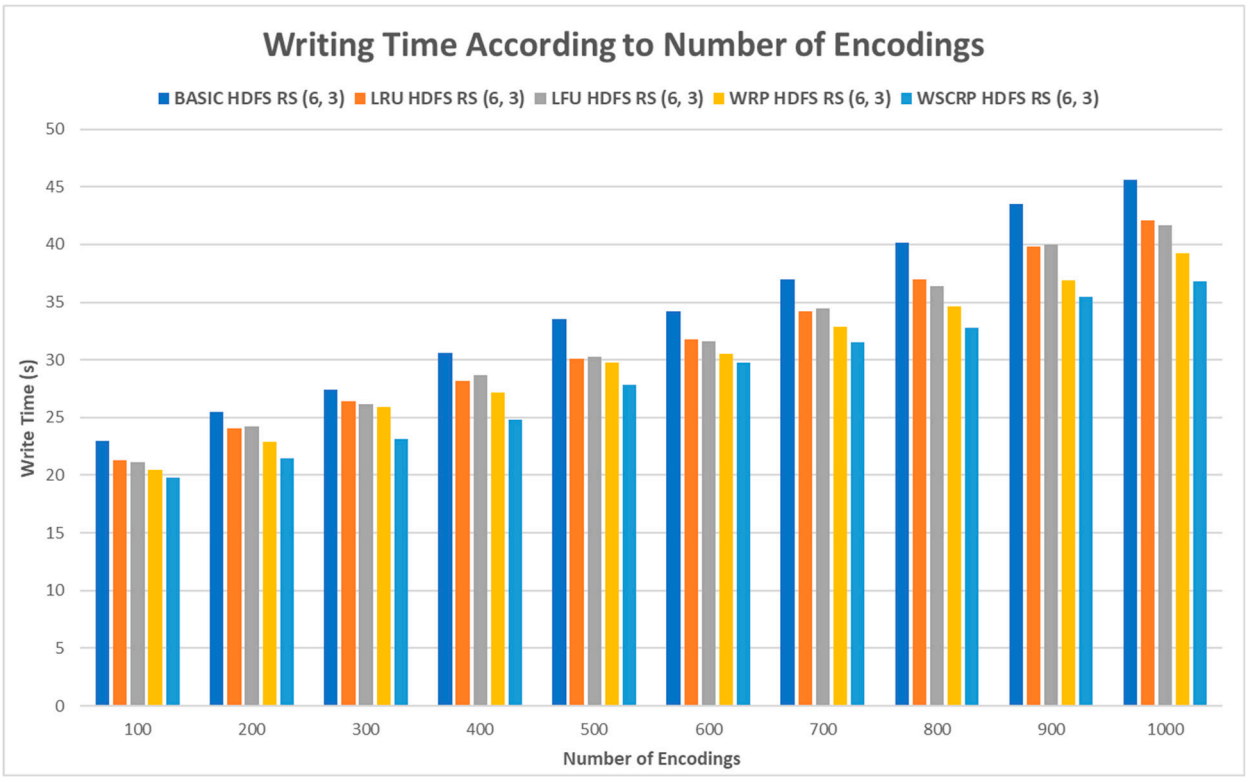

Figure 8. Writing time according to the number of encodings.

Figure 9 shows the reading time of decoding the encoded data from 100 to 1000 times. As the number of decodings increases, the number of generated decoding matrices increases, so the overall reading time increases. Measurements from 100 to 1000 times showed that BASIC HDFS RS (6, 3) took 285 s to 313.8 s, LRU HDFS RS (6, 3) took 282.8 s to 305.9 s, and LFU HDFS RS (6, 3) took 283.2 s to 304.2 s. WRP HDFS RS (6, 3) took 278.9 s to 295.8 s, and WSCRP HDFS RS (6, 3) took 270.5 s to 290.1 s. When comparing BASIC HDFS RS (6, 3) with WSCRP HDFS RS (6, 3) based on 1000 decoding times, the reading time can be shortened by about 23 s. All the systems, except BASIC HDFS RS (6, 3), generate a decoding matrix, upload it to cache memory, and reuse the uploaded matrix for the next decoding, resulting in a small reduction in overall reading time. In particular, WSCRP HDFS RS (6, 3) considering weights and parameters can save the most in reading time compared to LRU HDFS RS (6, 3) and LFU HDFS RS (6, 3) applying basic memory cache structures.

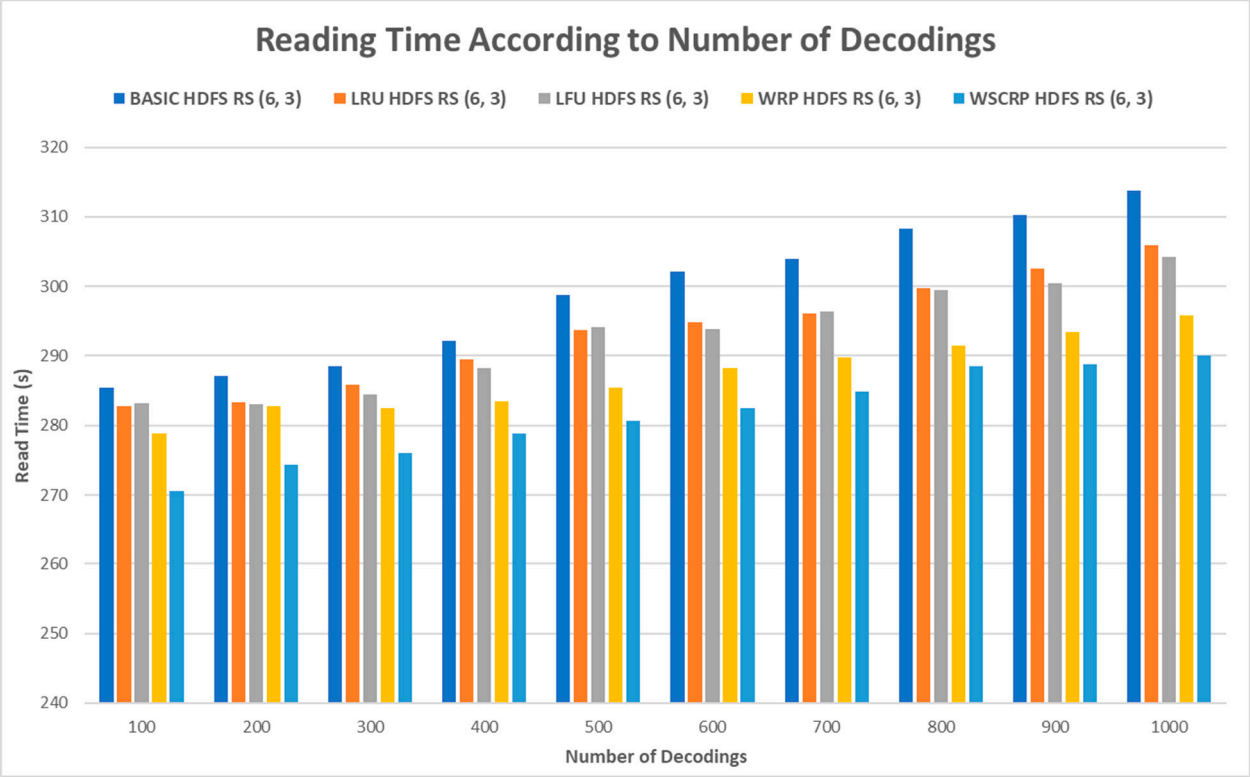

Figure 9. Reading time according to the number of decodings.

Figure 10 shows the recovery time measured when one to three nodes fail randomly, and decoding is performed 1000 times. When compared to the reading time in Figure 9, where decoding was performed 1000 times without failure, the recovery time measured by the five systems gradually increases as the number of random node failures in Figure 10 increases. This increase in decoding time is due to the selection of a data node that can respond, while excluding nodes with failures from the decoding matrix calculation. Among the five systems, WSCRP HDFS RS (6, 3) shows the shortest recovery time, measuring 296.1 s for a single failure of a random node, 305.3 s for double failures of a random node, and 336.8 s for triple failures of a random node, making it the most efficient system among the five.

The results of measuring writing and reading time by performing encoding and decoding without any failure show that the number of matrices used is reduced. However, as the maximum number of executions approaches 1000, both the writing and recovery times gradually increase. This is because although the number of matrices used is smaller than the number used in the presence of failures, the number of matrices increases with the

number of encodings and decodings, which in turn results in an increase in writing and reading time [41].

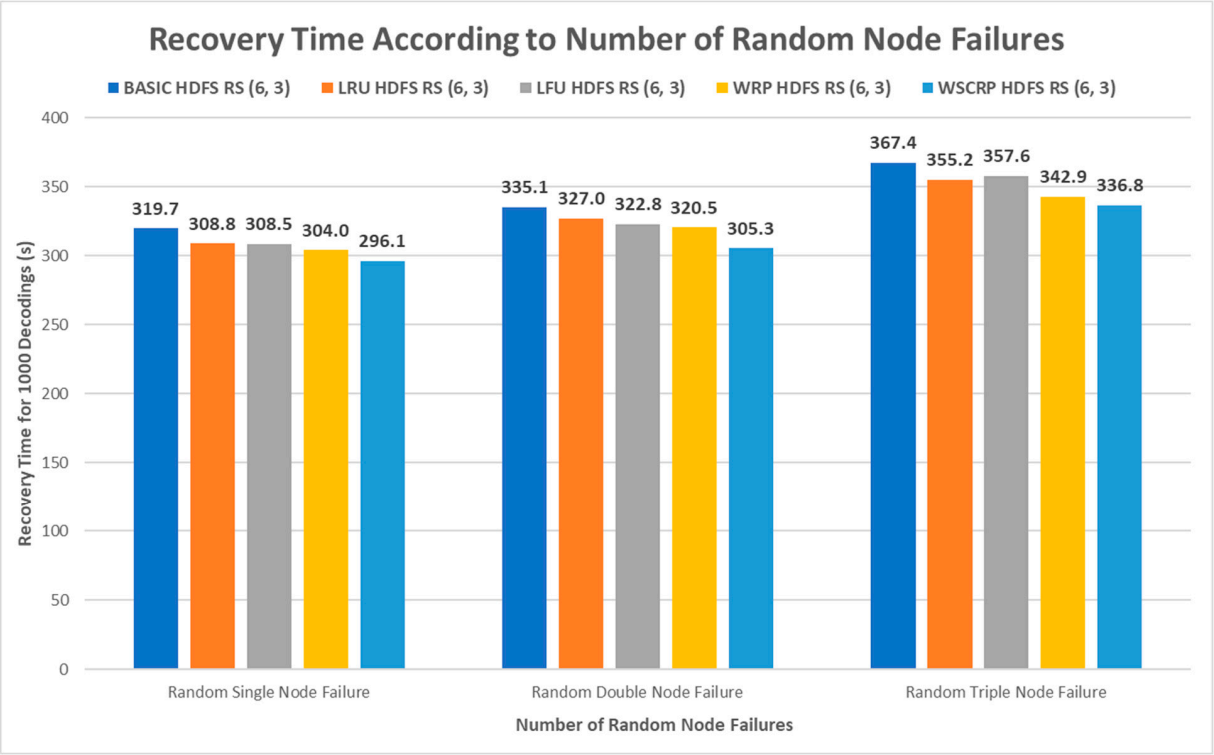

Figure 10. Recovery time according to the number of random node failures.

Figure 11 shows the hit rate of the matrix that was uploaded to and accessed from the cache memory. During the experiment shown in Figure 11, the hit rate was measured every 100 times while the recovery time of the random triple node failure (the third item in Figure 10) was being evaluated.

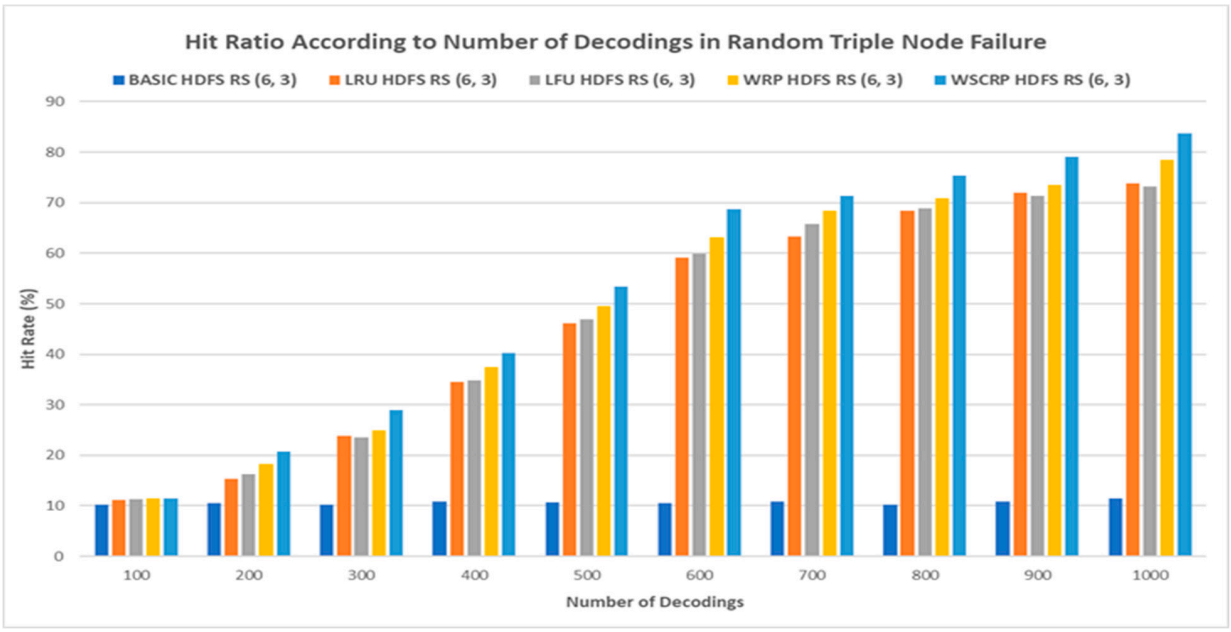

Figure 11. Hit rate according to the number of decodings.

BASIC HDFS RS (6, 3) does not upload the matrix generated by the decoding process to the cache memory. As a result, the hit rate remains around 10% even as the number of decodings increases. In contrast, the remaining four systems reuse the matrix uploaded to the cache memory, causing the hit rate to gradually increase with the number of decodings. The number of decodings increases rapidly up to 600. However, from 600 decodings onwards, the hit rate increases only slightly compared to the previous values. This is because, at 600 decodings, the cache memory is full and the page replacement policy is triggered. When the number of decodings reaches 1000, the hit rates of LRU HDFS RS (6, 3) and LFU HDFS RS (6, 3) show little difference, measuring at 73.8% and 73.2%, respectively. The hit rate of WRP HDFS RS (6, 3) is slightly superior to the LRU and LFU structures, measuring at 75.4%. As the number of decodings increases, the hit rate of WSCRP HDFS RS (6, 3) is measured to be higher than that of WRP HDFS RS (6, 3). It reaches its best value of 83.8% at 1000 decodings.

6. Conclusions and Future Directions

This paper proposes a cache-based matrix technique, which is a method for efficiently utilizing symmetrical space in EC-based distributed file systems. The technique uses the encoding used for file writing and the matrix generated when performing decoding used for file reading and recovery. Up to 1001 matrices based on the RS (10, 4) volume were generated, depending on the server that failed during file writing, reading, and recovery. Cache-based matrix technology can maintain a high cache memory hit rate, which can shorten file storage and recovery times during encoding and decoding. Additionally, since cache memory is utilized in terms of software, there is no cost to adding or changing hardware modules. The WSCRP algorithm underlying this technology can efficiently handle the page replacement policy of cache memory through weights and cost parameters. Therefore, this paper focuses on using matrices to reduce overhead in distributed file systems.

Future research directions are organized into three directions. First, the proposed technology will be applied not only to RS (6, 3), but also to various volumes of RS (3, 2), RS (8, 4), and RS (10, 4). Second, the proposed technology will be applied not only to HDFS but also to Ceph and GlusterFS, which are distributed file systems that support the EC algorithm. Finally, we will analyze the data blocks, parity blocks, and matrices that change when data are updated in the distributed file system. We plan to study how to reduce the overhead that occurs when updating data, as well as writing, reading, and recovering data.

Author Contributions: Conceptualization, D.-J.S. and J.-J.K.; methodology, D.-J.S. and J.-J.K.; software, D.-J.S.; validation, J.-J.K.; formal analysis, D.-J.S. and J.-J.K.; investigation, D.-J.S.; resources, J.-J.K.; data curation, D.-J.S.; writing—original draft preparation, D.-J.S.; writing—review and editing, J.-J.K.; visualization, D.-J.S.; supervision, J.-J.K. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2022R1F1A1062953).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data are provided via commands that create dummy data in Linux.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

ASIC	Application-Specific Integrated Circuit
EC	Erasure Coding
GB	Gigabytes
HDFS	Hadoop Distributed File System

I/O	Input/Output
IoT	Internet of Things
JDK	Java Development Kit
LFU	Least Frequently Used
LRC	Local Reconstruction Code
LRU	Least Recently Used
RAID	Redundant Array of Inexpensive Disks
RS	Reed Solomon
SCM	Size of Cache Memory
SRM	Size of Request Matrix
TB	Terabytes
WRP	Weighting Replacement Policy
WSCRP	Weighting Size and Cost Replacement Policy

References

1. Sigov, A.; Ratkin, L.; Ivanov, L.A.; Xu, L.D. Emerging enabling technologies for industry 4.0 and beyond. *Inf. Syst. Front.* **2022**, 1–11. [CrossRef]
2. Macko, P.; Hennessey, J. Survey of Distributed File System Design Choices. *ACM Trans. Storage* **2022**, *18*, 1–34. [CrossRef]
3. Karun, A.K.; Chitharanjan, K. A review on hadoop—HDFS infrastructure extensions. In Proceedings of the 2013 IEEE Conference on Information & Communication Technologies, Thuckalay, India, 11–12 April 2013.
4. Shin, D.J.; Kim, J.J. Research on Improving disk throughput in EC-based distributed file system. *Psychology* **2021**, *58*, 9664–9671.
5. Kim, D.O.; Kim, H.Y.; Kim, Y.K.; Kim, J.J. Cost analysis of erasure coding for exa-scale storage. *J. Supercomput.* **2018**, *75*, 4638–4656. [CrossRef]
6. Balaji, S.B.; Krishnan, M.N.; Vajha, M.; Ramkumar, V.; Sasidharan, B.; Kumar, P.V. Erasure coding for distributed storage: An overview. *Sci. China Inf. Sci.* **2018**, *61*, 100301. [CrossRef]
7. Ma, T.; Hao, Y.; Shen, W.; Tian, Y.; Al-Rodhaan, M. An improved web cache replacement algorithm based on weighting and cost. *IEEE Access* **2018**, *6*, 27010–27017. [CrossRef]
8. Samiee, K. A replacement algorithm based on weighting and ranking cache objects. *Int. J. Hybrid Inf. Technol.* **2009**, *2*, 93–104.
9. Cook, J.D.; Primmer, R.; de Kwant, A. Compare cost and performance of replication and erasure coding. *Hitachi Rev.* **2014**, *63*, 304–310.
10. Luo, J.; Shrestha, M.; Xu, L.; Plank, J.S. Efficient encoding schedules for XOR-based erasure codes. *IEEE Trans. Comput.* **2013**, *63*, 2259–2272. [CrossRef]
11. Plank, J.S. A tutorial on Reed–Solomon coding for fault-tolerance in RAID-like systems. *Softw. Pract. Exp.* **1997**, *27*, 995–1012. [CrossRef]
12. Plank, J.S. The raid-6 liber8tion code. *Int. J. High Perform. Comput. Appl. Int. J. High Perform. C* **2009**, *23*, 242–251. [CrossRef]
13. Hafner, J.L. WEAVER Codes: Highly Fault Tolerant Erasure Codes for Storage Systems. In Proceedings of the FAST'05: 4th USENIX Conference on File and Storage Technologies, San Francisco, CA, USA, 13–16 December 2005.
14. Introduction to HDFS Erasure Coding in Apache Hadoop. Available online: https://blog.cloudera.com/introduction-to-hdfs-erasure-coding-in-apache-hadoop/ (accessed on 15 January 2023).
15. Plank, J.S. Erasure codes for storage systems: A brief primer. *Login* **2013**, *38*, 44–50.
16. Huang, C.; Simitci, H.; Xu, Y.; Ogus, A.; Calder, B.; Gopalan, P.; Yekhanin, S. Erasure coding in windows azure storage. In Proceedings of the USENIX ATC'12: The 2012 USENIX Conference on Annual Technical Conference, Boston, MA, USA, 13–15 June 2012.
17. Rashmi, K.V.; Shah, N.B.; Gu, D.; Kuang, H.; Borthakur, D.; Ramchandran, K. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster. In Proceedings of the 5th USENIX Workshop on Hot Topics in Storage and File Systems, San Jose, CA, USA, 27–28 June 2013.
18. Papailiopoulos, D.S.; Dimakis, A.G.; Cadambe, V.R. Repair optimal erasure codes through hadamard designs. *IEEE Trans. Inf.* **2013**, *59*, 3021–3037. [CrossRef]
19. Chen, B.; Ammula, A.K.; Curtmola, R. Towards server-side repair for erasure coding-based distributed storage systems. In Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, New York, NY, USA, 2–4 March 2015.
20. Li, J.; Li, B. Zebra: Demand-aware erasure coding for distributed storage systems. In Proceedings of the IEEE/ACM 24th International Symposium on Quality of Service (IWQoS), Beijing, China, 20–21 June 2016.
21. Kim, D.O.; Kim, H.Y.; Kim, Y.K.; Kim, J.J. Efficient techniques of parallel recovery for erasure-coding-based distributed file systems. *Comput. J.* **2019**, *101*, 1861–1884. [CrossRef]
22. Bashyam, K.R. Repair Pipelining for Clay-Coded Storage. In Proceedings of the 2021 International Conference on COMmunication Systems & NETworkS (COMSNETS), Bangalore, India, 5–9 January 2021.
23. Arslan, Ş.Ş. Founsure 1.0: An erasure code library with efficient repair and update features. *SoftwareX* **2021**, *13*, 100662. [CrossRef]
24. Uezato, Y. Accelerating XOR-based erasure coding using program optimization techniques. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, New York, NY, USA, 14–19 November 2021.

25. Muntz, D.; Honeyman, P. Multi-level Caching in Distributed File Systems. In Proceedings of the Winter USENIX Conference, San Francisco, CA, USA, 16 August 1991.
26. Zhang, J.; Wu, G.; Hu, X.; Wu, X. A Distributed Cache for Hadoop Distributed File System in Real-Time Cloud Services. In Proceedings of the ACM/IEEE 13th International Conference on Grid Computing, Beijing, China, 20–23 September 2012.
27. Rashmi, K.V.; Chowdhury, M.; Kosaian, J.; Stoica, I.; Ramchandran, K. EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, Savannah, GA, USA, 2–4 November 2016.
28. Anderson, T.E.; Canini, M.; Kim, J.; Kostic, D.; Kwon, Y.; Peter, S.; Reda, W.; Schuh, H.N.; Witchel, E. Assise: Performance and Availability via Client-local NVM in a Distributed File System. In Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation, Virtual Event, 4–6 November 2020.
29. Ruty, G.; Baccouch, H.; Nguyen, V.; Surcouf, A.; Rougier, J.L.; Boukhatem, N. Popularity-based full replica caching for erasure-coded distributed storage systems. *Clust. Comput.* **2021**, *24*, 3173–3186. [CrossRef]
30. Silberstein, M.; Ganesh, L.; Wang, Y.; Alvisi, L.; Dahlin, M. Lazy means smart: Reducing repair bandwidth costs in erasure-coded distributed storage. In Proceedings of the SYSTOR 2014 International Conference on Systems and Storage, New York, NY, USA, 30 June–2 July 2014.
31. Mitra, S.; Panta, R.; Ra, M.R.; Bagchi, S. Partial-parallel-repair (PPR) a distributed technique for repairing erasure coded storage. In Proceedings of the Eleventh European Conference on Computer Systems, New York, NY, USA, 18–21 April 2016.
32. Pei, X.; Wang, Y.; Ma, X.; Xu, F. T-update: A tree-structured update scheme with top-down transmission in erasure-coded systems. In Proceedings of the IEEE INFOCOM 2016—The 35th Annual IEEE International Conference on Computer Communications, San Francisco, CA, USA, 10–14 April 2016.
33. Li, R.; Li, X.; Lee, P.P.; Huang, Q. Repair Pipelining for Erasure-Coded Storage. In Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17), Santa Clara, CA, USA, 12–14 July 2017.
34. Wang, F.; Tang, Y.; Xie, Y.; Tang, X. XORInc: Optimizing data repair and update for erasure-coded systems with XOR-based in-network computation. In Proceedings of the 35th Symposium on Mass Storage Systems and Technologies (MSST), Santa Clara, CA, USA, 20–24 May 2019.
35. Xia, J.; Guo, D.; Xie, J. Efficient in-network aggregation mechanism for data block repairing in data centers. *Future Gener. Comput. Syst.* **2020**, *105*, 33–43. [CrossRef]
36. Qiao, Y.; Kong, X.; Zhang, M.; Zhou, Y.; Xu, M.; Bi, J. Towards in-network acceleration of erasure coding. In Proceedings of the Symposium on SDN Research, San Jose, CA, USA, 3 March 2020.
37. Zeng, H.; Zhang, C.; Wu, C.; Yang, G.; Li, J.; Xue, G.; Guo, M. FAGR: An efficient file-aware graph recovery scheme for erasure coded cloud storage systems. In Proceedings of the 2020 IEEE 38th International Conference on Computer Design (ICCD), Hartford, CT, USA, 18–21 October 2020.
38. Zhou, A.; Yi, B.; Liu, Y.; Luo, L. An Optimal Tree-Structured Repair Scheme of Multiple Failure Nodes for Distributed Storage Systems. *IEEE Access* **2021**, *9*, 21843–21858. [CrossRef]
39. Lee, K.H. Consideration of the Permutations and Combinations Taught in Secondary Schools. Master's Thesis, Yonsei University Graduate School of Education, Seoul, Republic of Korea, 1 June 2007.
40. Hafner, J.L.; Deenadhayalan, V.; Rao, K.K.; Tomlin, J.A. Matrix Methods for Lost Data Reconstruction in Erasure Codes. In Proceedings of the FAST'05: Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies, San Francisco, CA, USA, 13–16 December 2005.
41. Kim, J.J. Erasure-Coding-Based Storage and Recovery for Distributed Exascale Storage Systems. *Appl. Sci.* **2021**, *11*, 3298. [CrossRef]