I/O-Efficient Scaling Schemes for Distributed Storage Systems with CRS Codes

Si Wu, Yinlong Xu, Yongkun Li, and Zhijia Yang

Abstract—System scaling becomes essential and indispensable for distributed storage systems due to the explosive growth of data volume. Considering that fault-protection is a necessity in large-scale distributed storage systems, and Cauchy Reed-Solomon (CRS) codes are widely deployed to tolerate multiple simultaneous node failures, this paper studies the scaling problem of distributed storage systems with CRS codes. In particular, we formulate the scaling problem with an optimization model in which both the post-scaling encoding matrix and the data migration policy are assumed to be unknown in advance. To minimize the I/O overhead, we propose a three-phase optimization scaling scheme for CRS codes. Specifically, we first derive the optimal post-scaling encoding matrix under a given data migration policy, then optimize the data migration process using the selected post-scaling encoding matrix, and finally exploit the Maximum Distance Separable (MDS) property to further optimize the designed data migration process. Our scaling scheme requires minimal data movement while achieving uniform data distribution. Moreover, it requires to read fewer data blocks than conventional minimum data migration schemes, but still guarantees the minimum amount of migrated data. To validate the efficiency of our scheme, we implement it atop a networked file system. Extensive experiments show that our scaling scheme uses less scaling time than the basic scheme.

Index Terms—Scaling, cauchy reed-solomon codes, distributed storage systems, encoding matrix, data migration

1 Introduction

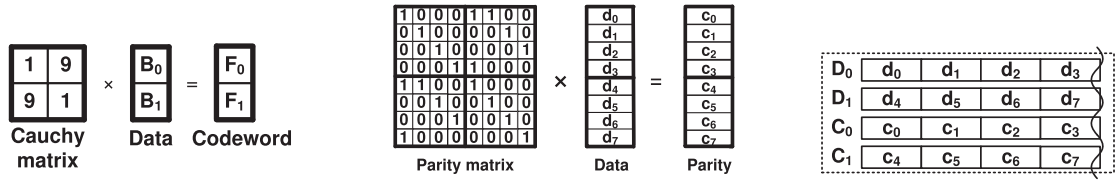
Storage systems have grown to the point where the system scale is tremendously huge and the components are built from inexpensive commodity storage devices, which leads to frequent component failures. Erasure codes are often deployed to provide fault tolerance in modern storage systems, such as Netapp [2], Cleversafe [10], Windows Azure [7] and Oceanstore [11]. The advantage of erasure codes over replication is that they can achieve higher reliability with the same amount of redundancy, or require less redundancy for satisfying the same reliability requirement. Since multiple simultaneous node failures occur from time to time in distributed systems, Cauchy Reed-Solomon (CRS) [1] codes are designed to tolerate any number of concurrent node failures, and they have been widely deployed in both academic projects and commercial systems such as Cleversafe [10] and Oceanstore [11]. On the other hand, the amount of digital data that needs to be stored continues to explode, storage systems have a continuously growing demand on storage capacity and I/O performance, so it is quite often to add new storage devices to a storage system so as to increase its storage capacity and improve its I/O performance. Therefore, fault tolerance and system scaling have become two necessities for modern storage systems.

This paper studies the scaling problem in CRS-coded storage systems. System scaling of a fault tolerated storage system can be regarded as an application atop the storage system. To accomplish the scaling process, original data must be rebalanced among the old devices and the newly-added devices. Besides, in storage systems with redundant mechanisms, the associated parities must also be updated accordingly. Hence, scaling of distributed storage systems inevitably incurs disk usage and network consumption. Modern storage systems are widely deployed in multi-cloud and data center services where the network resources are rather scarce. Thus, one important objective of performing system scaling is to minimize the amount of I/Os and network transfers so as to achieve fast scaling and allow more flexible scheduling of the foreground user tasks.

There have been numerous approaches to RAID scaling designed to minimize the amount of migrated data during the scaling processes, including FastScale [16], MDM [3], GSR [12], MiPiL [15], SDM [13], etc. These scaling schemes are called minimum data migration schemes in the following of this paper. For example, FastScale [16] aims at minimizing the migration I/Os for RAID-0 scaling. MiPiL [15] moves the minimum amount of data blocks from old devices to new devices for RAID-5 scaling. SDM [13] minimizes the data migration traffic according to the future parity layout for various RAID-6 codes. However, none of these approaches can be applied efficiently to CRS codes due to the special coding mechanisms. Please refer to Section 3 for more details about the difference between CRS scaling and conventional scaling methods.

- *S. Wu and Y. Li are with the School of Computer Science and Technology, University of Science and Technology of China and the AnHui Province Key Laboratory of High Performance Computing, University of Science and Technology of China. E-mail: {wusi, ykli}@mail.ustc.edu.cn.*
- *Y. Xu is with the School of Computer Science and Technology, University of Science and Technology of China and the Collaborative Innovation Center of High Performance Computing, National University of Defense Technology, China, Changsha 410073, China. E-mail: ylxu@ustc.edu.cn.*
- *Z. Yang is with the School of Computer Science and Technology, University of Science and Technology of China. E-mail: yangzj@mail.ustc.edu.cn.*

Manuscript received 17 Aug. 2015; accepted 24 Nov. 2015. Date of publication 4 Dec. 2015; date of current version 10 Aug. 2016.
Recommended for acceptance by R. Brightwell.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2015.2505722



(a) Cauchy matrix of a CRS code.

(b) Parity matrix representation of a CRS code ($k = 2, m = 2, \omega = 4$).

(c) Data layout of a CRS code ($k = 2, m = 2, \omega = 4$).

Fig. 1. Background on CRS code.

To address the scaling challenge due to CRS codes, we formulate CRS scaling with an optimization model in which both the post-scaling encoding matrix and the data migration policy are supposed to be unknown in advance. Thus, to optimize the CRS scaling process, we must design both the post-scaling encoding matrix and the data migration policy. Note that the Jerasure library [9] provides the encoding matrices for CRS codes whose encoding performance are optimized. Therefore, the encoding matrices of CRS codes for practical deployment are usually generated from Jerasure. For a CRS scaling scenario, the pre-scaling encoding matrix is given and can be generated from Jerasure. However, if the post-scaling encoding matrix is also derived from Jerasure, the I/O overhead and network bandwidth traffic may be very large. To address this challenge, we try to minimize the amount of I/Os and network transfers for CRS scaling in three steps: (i) we first design the post-scaling encoding matrix from the pre-scaling matrix with a naive data migration scheme, (ii) then optimize the data migration process using the selected post-scaling encoding matrix, (iii) and finally apply the Maximum Distance Separable (MDS) property to the designed data migration process to further optimize the migration I/Os. The main contributions of this paper are as follows.

- We formulate the scaling procedure of CRS codes as an optimization model that is composed of a post-scaling encoding matrix and a data migration policy. Based on this model, we propose a three-phase optimization scaling scheme for CRS codes.
- We propose an efficient method to design the post-scaling encoding matrix, which optimizes the I/O overhead and network traffic for CRS scaling.
- We propose a searching algorithm to design efficient online data migration schemes, which further reduces the I/O cost for CRS scaling.
- We develop a new scheme to further reduce the read of data blocks over existing minimum data migration schemes while still guarantee the minimum amount of migrated data by using parity blocks to decode the required data blocks. With this scheme, the I/O cost for CRS scaling gets further reduced.
- We implement our scaling scheme atop a real networked file system and demonstrate via extensive evaluations that our scaling scheme reduces the scaling time over transitional schemes under practical system settings.

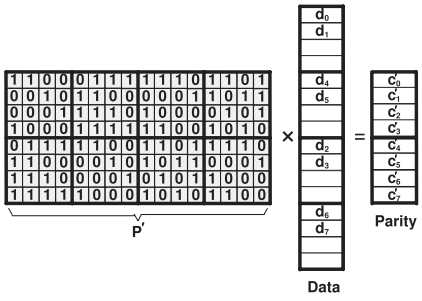
The remainder of the paper proceeds as follows. Section 2 reviews background on CRS codes. Section 3 motivates the demand on CRS scaling. Section 4 presents our method developed for designing efficient encoding matrices, and Section 5 illustrates the searching algorithm for finding efficient data migration approaches. Section 6 gives the algorithm which further reduces the read of data blocks in the data migration process. Simulation studies are presented in Section 7, and experimental evaluations are given in Section 8. Finally, Section 9 concludes this paper.

2 Background on CRS Codes

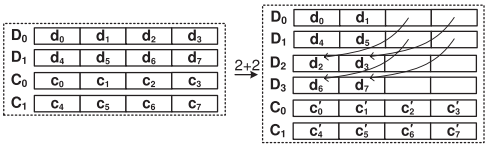
We first review the background on CRS codes. A storage system with a CRS code can be denoted by a triad (k, m, ω). It is composed of $n = k + m$ disks, of which k disks hold data blocks (labeled by $D_0, \ldots, D_{k-1}$) while the remaining m disks hold parity blocks (labeled by $C_0, \ldots, C_{m-1}$). The class of CRS codes have the desired property that the system can tolerate any m out of n disk failures, and this property is called *Maximum Distance Separable* property. Each disk is partitioned into *strips* of fixed size. Each strip stores ω data/parity blocks. A set of $n = k + m$ strips, each from a separate disk, form a *stripe*. The encoding is performed within a stripe. For practical deployment, the identities of the data and parity disks are rotated across different stripes to alleviate hot spots that may occur since parity disks usually require more accesses than data disks.

A CRS code can be represented by an $n\omega \times k\omega$ *encoding matrix* (G) over $GF(2)$. It first constructs an $n \times k$ distribution matrix from an $m \times k$ Cauchy matrix, both over a *Galois Field* $GF(2^\omega)$. Given $X = \{x_0, \ldots, x_{m-1}\}$, $Y = \{y_0, \ldots, y_{k-1}\}$ with $x_i, y_j \in GF(2^\omega)$ and $X \cap Y = \emptyset$, the Cauchy matrix defined by X and Y has an entry of $1/(x_i + y_j)$ at row i, column j. Note that the subscripts used throughout this paper are all zero-indexed. Fig. 1a shows an example of the cauchy matrix defined by $X = \{1, 2\}, Y = \{0, 3\}$ in $GF(2^4)$. The distribution matrix is composed of an identity matrix in the first k rows and a Cauchy matrix in the following m rows. The arithmetic is performed over $GF(2^\omega)$, where addition is simple bit-wise exclusive OR (XOR), while multiplication is more complex, which is typically implemented with a multiplication table or a discrete logarithm table.

A CRS code subsequently transfers an $n \times k$ distribution matrix over $GF(2^\omega)$ to an $n\omega \times k\omega$ encoding matrix over $GF(2)$ to eliminate the expensive Galois Field multiplications. Its main idea is as follows. Each element $e \in GF(2^\omega)$ can be expressed by a vector $V(e)$ of ω bits, i.e., the binary representation of e. Each element e may also be represented by an $\omega \times \omega$ matrix $\tau(e)$ over $GF(2)$ with the ith column of $\tau(e)$ being $V(e * 2^i)$ (i is zero-indexed). Thus, an $n \times k$ distribution matrix over $GF(2^\omega)$ can be transferred to an $n\omega \times k\omega$ encoding matrix over $GF(2)$ by substituting any e of its



(a) A randomly constructed post-scaling parity matrix.



(b) A naive migration approach.

Fig. 2. A typical post-scaling parity matrix and a naive migration approach when scaling the CRS code in Fig. 1 with two new data disks.

entries with $\tau(e)$. The encoding and decoding operations can now be implemented over $GF(2)$ rather than $GF(2^{\omega})$. In each stripe of a (k, m, ω) CRS code, there are $k\omega$ data blocks denoted as $d_0, \ldots, d_{k\omega-1}$ and $m\omega$ parity blocks labeled by $c_0, \ldots, c_{m\omega-1}$. Note that in the following of this paper, data blocks and parity blocks strictly refer to the original data and the parity information generated, respectively.

The first $k\omega$ rows of the encoding matrix form an identity matrix, while the next $m\omega$ rows form a parity matrix, denoted as $P = (p_{i,j})_{m\omega \times k\omega}$. Fig. 1b presents the parity matrix of the CRS code with the Cauchy matrix shown in Fig. 1a. The data layout corresponding to Fig. 1b is shown in Fig. 1c. A column of P corresponds to a data block. If an entry of a column is one, the corresponding data block is encoded into a parity block. For example, in Fig. 1b, d_0 corresponds to the zeroth column. d_0 is encoded into c_0, c_4, c_7 as $p_{0,0} = p_{4,0} = p_{7,0} = 1$. A parity block corresponds to a row in P, which is the XOR sum of all data blocks with an entry one in the row. Again in Fig. 1b, c_0 corresponds to the zeroth row, $c_0 = d_0 \oplus d_4 \oplus d_5$ because $p_{0,0} = p_{0,4} = p_{0,5} = 1$.

3 Motivation

3.1 Problem Introduction

Suppose a (k, m, ω) CRS-coded storage system is scaled up to a $(k + t, m, \omega)$ one while keeping the same number of parity disks. To exploit the I/O parallelism in the scaled system, several data blocks in old data disks should be moved onto new ones. This process is called *data migration*. Besides, we should also scale up the encoding matrix since the pre-scaling encoding matrix has an order of $n\omega \times k\omega$ while the order of the post-scaling encoding matrix increases to $(n + t)\omega \times (k + t)\omega$. When the data migration policy and the post-scaling encoding matrix are both determined, the parity blocks should be modified so as to guarantee the MDS property in the scaled system. To modify the parity blocks, a system has to read several data blocks beyond the set of the migrated ones, read the specific parity blocks, then update the parity blocks, and finally write these parity blocks back. This process is termed *parity modification*. Since data layouts and encoding matrices are the same in different stripes, we only focus on the scaling process within one stripe.

A data migration scheme can be defined by a mapping

$$\sigma : ADDR \rightarrow ADDR^{'}, \tag{1}$$

where $ADDR = \{i | 0 \leq i \leq k\omega - 1\}$, $ADDR^{'} = \{i^{'} | 0 \leq i^{'} \leq (k + t)\omega - 1\}$. $\sigma(i) = i^{'}$ means that d_i, the ith data block of the pre-scaling stripe (i.e., the $(i \bmod \omega)$th data unit in $D_{\lfloor i/\omega \rfloor}$), is migrated to the $i^{'}$th data unit of the scaled stripe (i.e., the $(i^{'} \bmod \omega)$th data unit in $D_{\lfloor i^{'}/\omega \rfloor}$). We denote the scaling process of a CRS code as a triad $(\sigma, P, P^{'})$, where $P = (p_{i,j})_{m\omega \times k\omega}$ and $P^{'} = (p^{'}_{i,j})_{m\omega \times (k+t)\omega}$ are the pre-scaling and the post-scaling parity matrices, respectively.

We illustrate the scaling components (i.e., $(\sigma, P, P^{'})$) via an CRS scaling instance. Fig. 2 shows an example of adding two data disks to a storage system with the $(2, 2, 4)$ CRS code in Fig. 1. Fig. 2a presents a post-scaling parity matrix $P^{'} = (p^{'}_{i,j})_{m\omega \times (k+t)\omega}$, where $P^{'}$ is constructed from $X^{'} = \{0, 1\}$ and $Y^{'} = \{2, 3, 4, 5\}$. Fig. 2b presents a naive migration scheme defined by a mapping as follows:

$$\sigma(i) = \begin{cases} i, & \text{if } i = 0, 1, 4, 5 \\ i + 6 & \text{if } i = 2, 3, 6, 7. \end{cases}$$

Note that the space of data blocks that have already been moved (e.g., the second data unit in D_0) are treated as empty blocks, which can be reclaimed to gain contiguous space through *hole-punching* and *segment compaction*.

Our primary objective is to minimize the I/O overhead and hence the network consumption for the scaling process. Note that in applications with classic erasure codes, the amount of network transfers is equal to the amount of data read (written) in disks. The total amount of I/Os include the I/Os for both data migration and parity modification.

We optimize the amount of I/Os for CRS scaling in three steps: *(1) We optimize the post-scaling encoding matrix with a naive migration method*. Note that we only consider data migration approaches that achieve uniform data distribution after scaling and require minimal data movement, such as the one shown in Fig. 2b. The first step is to reduce the amount of data blocks being read for parity modification; *(2) we optimize the data migration approach with the selected post-scaling encoding matrix*. The second step is to reduce the amount of parity blocks being read (written) for parity modification; *(3) We explore into the selected data migration process, and use parity blocks to decode several data blocks.* The third step is to reduce the read of data blocks over the minimal data migration schemes while still guaranteeing the minimal amount of migrated data.

3.2 The Effect of Encoding Matrix

Comparing Fig. 1b with Fig. 2a, we see that the first parity block is modified as $c_0^{'} = c_0 \oplus d_1 \oplus d_2 \oplus d_3 \oplus d_4 \oplus d_6 \oplus d_7$. To do this, we should read $c_0, d_1, d_2, d_3, d_4, d_6, d_7$ to modify c_0. Furthermore, c_0, c_1, c_2, c_3, $d_0, d_1, d_2, d_3, d_4, d_5, d_6, d_7$ are required to be read to modify the parity blocks in disk C_0, and c_4, c_5, c_6, c_7, $d_0, d_1, d_2, d_3, d_4, d_5, d_6, d_7$ are required to be read to modify the parity blocks in C_1.

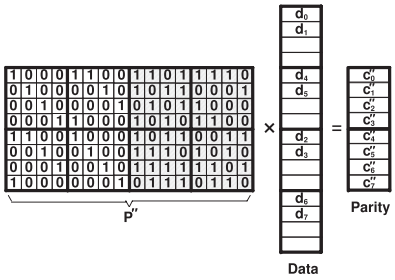


Fig. 3. Another post-scaling parity matrix when scaling the CRS coding in Fig. 1 with two new data disks.

In practical large-scale distributed storage systems, system scaling is typically operated through a coordinator node with computational capability. In particular, the coordinator node first issues the data migration process, and then performs the parity modification process. For data migration, the coordinator node first downloads the data blocks that are required to be migrated, and then writes these data blocks to new disks. The data blocks required for data migration are cached in the main memory of the coordinator node, and these data blocks will be later used for the parity modification process to save the overall I/O cost. For parity modification, the coordinator node first downloads certain additional data blocks beyond the set of the migrated data blocks, then reads the parity blocks that are required to be modified, and then updates the parity blocks, and finally writes them back to the corresponding parity disks. If a data block is required for the modification of multiple parity blocks, then it will be cached for later use so as to reduce the system I/O overhead.

For example, the amount of I/Os for data migration in Fig. 2a includes four reads of data blocks (i.e., d_2, d_3, d_6, d_7) and four writes of data blocks. The I/O overhead for parity modification includes four reads of data blocks (i.e., d_0, d_1, d_4, d_5), eight reads of parity blocks (i.e., $c_0 \sim c_7$), and eight writes of parity blocks.

Fig. 3 depicts another post-scaling parity matrix, $P'' = (p''_{i,j})_{m\omega\times(k+t)\omega}$, constructed from $X'' = \{1, 2\}$ and $Y'' = \{0, 3, 4, 5\}$. Comparing Fig. 1b with Fig. 3, we can conclude that only d_2, d_3, d_6, d_7 are needed to modify the eight parity blocks $c_0 \sim c_7$. Since the data blocks d_2, d_3, d_6, d_7 are available during the data migration process, they can be cached for parity modification. Thus, the I/O cost for parity modification is reduced to eight parity reads and eight parity writes.

We also emphasize that we can use the aggregate I/O technique [16] to reduce the number of I/O operations. For instance, in Fig. 3, we may use one read in D_0 (i.e., a read involving d_2, d_3), one read in D_1 (i.e., a read involving d_6, d_7), one read in C_0 (i.e., a read involving $c_0 \sim c_3$), one read in C_1 (i.e., a read involving $c_4 \sim c_7$) to fulfill the data (parity) being read. We may also use four writes to accomplish the data (parity) writes. The access aggregation technique can efficiently provide sequential disk accesses so as to reduce the disk access time. However, the amount of network transfers remains unchanged in the adoption of I/O aggregation.

3.3 The Effect of Data Migration

Given a pair of pre-scaling and post-scaling encoding matrices, the parity modification process will be determined by the data migration approach. Therefore, if we optimize the data migration process based on the given encoding matrices, the I/O overhead of the parity modification process may be further increased. For example, Fig. 4 depicts another data migration approach expressed by the following mapping:

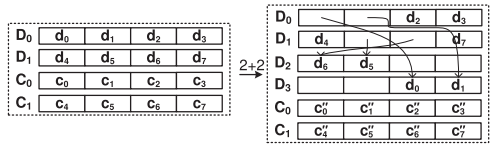


Fig. 4. Another data migration approach when scaling the CRS coding in Fig. 1 by adding two new data disks.

$$\sigma'(i) = \begin{cases} i, & \text{if } i = 2, 3, 4, 7 \\ i + 14; & \text{if } i = 0, 1 \\ 9(8), & \text{if } i = 5(6). \end{cases}$$

Note that this migration also achieves the minimal number of migration I/Os and uniform data distribution.

Given P in Fig. 1b and P'' in Fig. 3, for the data migration approach shown in Fig. 4, we only need to read d_0, d_1, d_5, d_6 to modify seven parity blocks $c_0, c_2, c_3, c_4, c_5, c_6, c_7$. The I/O overhead for parity modification is hence reduced to seven reads and seven writes of parity blocks.

3.4 MDS Property

We further observe that even the employed data migration approaches (e.g., the migration method in Fig. 4) are the ones with minimal data movement, we can still further optimize the amount of migration I/Os. In the following, we illustrate this observation via an example.

Note that the scaling process is initially implemented as a two-part sequential process: data migration and parity modification. Thus, as shown in Fig. 4, four reads of data blocks (i.e., d_0, d_1, d_5, d_6) and four writes of data blocks for minimal data migration schemes are indispensable. In fact, four reads of data blocks are the minimal amount of data blocks being read to achieve a minimal data migration scheme.

However, we may use a different implementation to further reduce the amount of I/Os for data migration. To achieve this, we first read all parity blocks (i.e., $c_0 \sim c_7$) into the coordinator node, then we use the parity blocks to decode the desired data blocks. The decoding mechanism can be demonstrated as follows:

$$d_0 = c_1 \oplus c_3 \oplus c_4 \oplus c_6$$
$$d_1 = c_0 \oplus c_1 \oplus c_2 \oplus c_3 \oplus c_5 \oplus c_6 \oplus c_7$$
$$d_5 = c_1 \oplus c_2 \oplus c_3 \oplus c_4 \oplus c_5 \oplus c_6 \oplus c_7$$
$$d_6 = c_0 \oplus c_2 \oplus c_3 \oplus c_5 \oplus c_6 \oplus c_7.$$

Finally, we write the data blocks into new disks to make them available as soon as possible, and also update the parity blocks and write them back. The decoding process works due to the MDS property that the parity blocks in two parity nodes suffice to decode the original data blocks. To obtain the decoding functions, we may use Gaussian Elimination or matrix inversion.

The data migration and parity modification processes are separated to exploit the MDS property in this implementation. The I/O overhead for data migration is reduced to four writes of data blocks. The I/O cost for parity modification increases to eight parity reads and seven parity writes

since we should read $c_0 \sim c_7$ but we only update $c_0, c_1, c_2, c_3, c_5, c_6, c_7$ and write them back. We can save four reads of data blocks in old data disks as well as the corresponding network consumption. Thus, we can further reduce the read of data blocks than the minimum data migration schemes but still guarantee the minimum amount of migrated data.

Motivated by the above examples, we aim at designing I/O-efficient scaling schemes for CRS codes. To achieve this, we first design the post-scaling encoding matrix to optimize the amount of data blocks being read for parity modification. We then design the data migration policy so as to optimize the amount of parity blocks being read (written) for parity modification. Finally, we exploit the MDS property to save the read of data blocks so that the I/O overhead for data migration can be saved.

4 The Design of Encoding Matrices

Fig. 2a shows that a randomly selected $P^{'}$ may induce high I/O overhead for parity modification. In this section, we give our method for designing the post-scaling parity matrix from a pre-scaling matrix with a naive data migration approach, so that the I/O overhead for the parity modification process can be minimized.

4.1 Jerasure Library

When constructing parity matrices for CRS codes, the primary concern is the encoding performance. The Jerasure library [9] uses matrix transformations to generate parity matrices with high encoding performance for CRS codes, and the Jerasure construction is as below.

Construction of the Jerasure parity matrix:

1. Construct an original Cauchy matrix $U = (u_{i,j})_{m \times k}$ from $X = \{0, \ldots, m-1\}$ and $Y = \{m, \ldots, m+k-1\}$, where $u_{i,j} = \frac{1}{i \oplus (m+j)}$. Note that the division arithmetic is performed over $GF(2^{\omega})$, $\oplus$ is XOR, and $+$ is regular integer addition.
2. For each column j $(0 \leq j \leq k-1)$, divide the elements by $u_{0,j}$ so that $u_{0,j}$ turns to element one.
3. For each row i $(1 \leq i \leq m-1)$, apply the following transformations.
   - Count the number of ones in the bit representation of row i.
   - For each j $(0 \leq j \leq k-1)$, count the number of ones in the bit representation of row i assuming all elements in row i are divided by $u_{i,j}$.
   - Select a value of j that produces the minimum number of ones, if it does reduce the number of ones in the bit representation of row i, then divide row i by $u_{i,j}$.
4. Transform $U = (u_{i,j})_{m \times k}$ into $P = (p_{i,j})_{m\omega \times k\omega}$.

Each of the above steps takes time of $O(mk)$, $O(mk)$, $O((m-1)k\omega^2)$, $O(mk\omega^2)$, respectively. Thus, the construction takes time of $O(2mk + (2m-1)k\omega^2)$.

The parity matrices of CRS codes for practical deployment are typically derived from Jerasure. However, we find that if we use both the pre-scaling and the post-scaling parity matrices from Jerasure, the I/O overhead for CRS scaling may be very high. For example, if we scale a $(3, 3, 4)$ CRS code to a $(6, 3, 4)$ one with the naive data migration whose mapping is as follows:

$$\sigma(i) = \begin{cases} i, & \text{if } i = 0, 1, 4, 5, 8, 9, \\ i + 10, & \text{if } i = 2, 3, 6, 7, 10, 11. \end{cases}$$

If we use both P and $P^{'}$ from Jerasure, we should read all data blocks (except for the data blocks being migrated) to modify all parity blocks.

We have examined many pairs of pre-scaling and post-scaling parity matrices from Jerasure, and found that most of them will induce high I/O overhead for CRS scaling. For example, suppose that we scale a (k, m, ω) CRS-coded storage system to a $(k+t, m, \omega)$ one for $m = 3, m \leq k, t \leq k$, $3 \leq \omega \leq 6$, there are 1,176 pairs of pre-scaling and post-scaling parity matrices from Jerasure, and 1,052 of them require to read all data blocks (except for the data blocks being migrated) to modify all parity blocks. When $m = 4, m \leq k$, $t \leq k, 3 \leq \omega \leq 6$, there are 1,114 pairs, and 1,015 pairs require to read all data blocks (except for the data blocks being migrated) to modify all parity blocks.

4.2 The Design of Post-Scaling Matrix

We first review the scaling up from $P = (p_{i,j})_{m\omega \times k\omega}$ in Fig. 1b to $P^{'} = (p^{'}_{i,j})_{m\omega \times (k+t)\omega}$ in Fig. 2a. Since $p_{0,1} = 0$, d_1 is not encoded into c_0. However, d_1 is encoded into $c^{'}_0$ because $p^{'}_{0,1} = 1$. Therefore, we should read d_1 to modify c_0 into $c^{'}_0$. Similarly, we should read d_4 to modify c_1 to $c^{'}_1$, as $p_{1,4} = 0$ but $p^{'}_{1,4} = 1$. Note that d_1 and d_4 are not migrated in Figs. 1 and 2, and so we have the following observation.

Observation: *If all entries in the first $k\omega$ columns of $P^{'}$ keep the same as that in P, then all un-migrated data blocks are not required for parity modification.*

Explanation: Suppose that an entry in the leftmost $k\omega$ columns of $P^{'}$, say $p^{'}_{i,j}$, equals to $p_{i,j}$ in P, and suppose that the data block d_j, which corresponds to the jth column of P (also the jth column of $P^{'}$), is not migrated. If $p^{'}_{i,j} = p_{i,j} = 1$, d_j will be encoded into both $c^{'}_i$ and c_i. Otherwise, d_j will be encoded into neither $c^{'}_i$ nor c_i.

The above observation indicates that if we keep the first $k\omega$ columns of P unchanged, *we only need the migrated blocks for parity modification, which can be cached when we read them for data migration.* Therefore, we can use the observation to greatly reduce the I/O overhead and network traffic for parity modification. It is highly desired since we can greatly release the load on old data disks which are continuously servicing the foreground user requests, and so we can improve the online user response time performance effectively. To achieve this objective, suppose that P is derived from sets $X = \{x_0, \ldots, x_{m-1}\}$ and $Y = \{y_0, \ldots, y_{k-1}\}$, we construct a post-scaling $P^{'}$ from X and $Y^{'} = Y \cup \{y^{'}_k, \ldots, y^{'}_{k+t-1}\}$, where $\{y^{'}_k, \ldots, y^{'}_{k+t-1}\} \subseteq GF(2^{\omega}) - X \cup Y$.

We emphasize that there also exists a trivial solution that satisfies the desired property of keeping the first $k\omega$ columns of $P^{'}$ identical to that in P. For example, if the size of the scaled stripe is determined in advance, we may first construct $P^{'}$ from Jerasure, and then set P identical to $P^{'}$ with t

dummy disks contributing zeros to the old parity blocks and being replaced with real disks upon scaling up. However, this solution is not applicable in practical storage systems, mainly because it requires the priori information of the scaled stripe size at the initial deployment. Precisely, the storage system must be initially deployed with P which is identical to $P^{'}$ for fault-tolerance. However, current storage systems have already been deployed with certain codes [2], [10], which makes the storage systems suffer from significant overhead when re-encoding. Besides, this solution should also take into consideration the optimization of the data migration process (See Sections 3.3 and 5). Thus, the way we design $P^{'}$ from a given P is more practical.

Given the naive data migration scheme, we construct a post-scaling parity matrix $P^{'}$ from P as follows.

Design of the post-Scaling parity matrix:

1. Select P from the Jerasure library. For each row i $(1 \leq i \leq m-1)$, record the value $u_{i,j}$ that improves the number of ones in the bit representation of row i, and we record $u_{i,j}$ as v_i.
2. Set $Y^{'} = \{m, \ldots, m+k-1, m+k, \ldots, m+k+t-1\}$ $(Y^{'} = Y \cup \{m+k, \ldots, m+k+t-1\})$. We then construct a Cauchy matrix $U^{'} = (u^{'}_{i,j})_{m \times (k+t)}$ from X and $Y^{'}$.
3. For each column j $(0 \leq j \leq k+t-1)$, divide the elements by $u^{'}_{0,j}$ to transform $u^{'}_{0,j}$ into element one.
4. For each row i $(1 \leq i \leq m-1)$, divide each element $u^{'}_{i,j}$ by v_i. This is to guarantee that the entries in the first k columns of $U^{'}$ are the same as those of U.
5. Finally, transform $U^{'} = (u^{'}_{i,j})_{m \times (k+t)}$ into $P^{'} = (p^{'}_{i,j})_{m\omega \times (k+t)\omega}$. The actions in step four also guarantee that the entries in the first $k\omega$ columns of $P^{'}$ are identical to those of P.

Note that the construction of the post-scaling encoding matrix with Jerasure takes time of $O(2m(k+t) + (2m-1)(k+t)\omega^2)$, while our method to construct $P^{'}$ takes time of $O(2m(k+t) + (m-1)(k+t) + m(k+t)\omega^2)$, mainly because we do not have to test and decide which element is to be divided for each row $1 \leq i \leq m-1$ in step 4. Thus, the time complexity of our method to construct $P^{'}$ is reduced over Jerasure.

The potential disadvantage of the designed $P^{'}$ is that it may affect the post-scaling encoding performance. In Section 7, we show via simulations that the encoding performance of our designed post-scaling parity matrices is almost the same as that of the post-scaling parity matrices derived from Jerasure.

5 The Design of Migration Process

In this section, we design the data migration policy to further reduce the I/O overhead for parity modification while achieving uniform data distribution and minimal data movement.

5.1 Problem Formulation

To evaluate the effect of the data migration process on the I/O overhead for parity modification, we define an *encoding set* as follows.

Definition 1. *Given a parity matrix $P = (p_{i,j})_{m\omega \times k\omega}$, a data block d_j corresponds to the jth column in P. d_j will be encoded into a set of parity blocks which have an entry one in the jth column. We call such a set of parity blocks an* encoding set *corresponding to d_j, labeled by E_j.*

One example is Fig. 1b in which d_0 is encoded into three parity blocks c_0, c_4, c_7, thus, $E_0 = \{c_0, c_4, c_7\}$. Given a scaling process $(\sigma, P, P^{'})$, before scaling, a data block d_j corresponds to the jth column of P and is encoded into the parity blocks in E_j related to P. After the scaling process, d_j is migrated to the $\sigma(j)$th data unit of the scaled stripe. Therefore, d_j corresponds to the $\sigma(j)$th column of $P^{'}$ and is encoded into the parity blocks in $E^{'}_{\sigma(j)}$ related to $P^{'}$ after scaling. Take Figs. 1b and 2a as an example, before scaling, d_2 corresponds to the second column of P and is hence encoded into $E_2 = \{c_2, c_5\}$ related to P. While after the scaling process, d_2 is moved to the eighth data unit of the scaled stripe and corresponds to the eighth column of $P^{'}$. Thus, d_2 is encoded into $E^{'}_8 = \{c_0, c_2, c_3, c_4, c_5, c_7\}$ related to $P^{'}$. Therefore, we should read d_j to modify all the parity blocks in $E_j \oplus E^{'}_{\sigma(j)}$. The last example is to read d_2 to modify the parity blocks in $E_2 \oplus E^{'}_8 = \{c_0, c_3, c_4, c_7\}$.

Our target is to seek a data migration process σ, which minimizes

$$\sum_{j=0}^{k\omega-1} |E_j \oplus E^{'}_{\sigma(j)}|$$

so as to minimize the amount of parity blocks being modified. *Note that the entries in the first $k\omega$ columns of $P^{'}$ are the same as P according to the design of parity matrices in the last section. If d_j $(0 \leq j \leq k\omega - 1)$ is not migrated, $\sigma(j) = j$. Therefore, $E_j \oplus E^{'}_{\sigma(j)} = \emptyset$ for an un-migrated data block d_j.*

Our scaling method only moves data blocks from old data disks (i.e., $D_0 \sim D_{k-1}$) to new data disks (i.e., $D_k \sim D_{k+t-1}$) while no data is moved among old disks. Thus, the possible movement is from d_j in the pre-scaling stripe to the $j^{'}$th data unit of the scaled stripe, where $0 \leq j \leq k\omega - 1, k\omega \leq j^{'} \leq (k+t)\omega - 1$. If $\sigma(j) = j^{'}$, then the parity blocks in $E_j \oplus E^{'}_{j^{'}}$ need d_j for modification. Therefore, we may calculate $|E_j \oplus E^{'}_{j^{'}}|$ for each pair of $(j, j^{'})$ to record the I/O overhead for each possible data movement. The results are recorded with a matrix $A = (\alpha_{x,y})_{k\omega \times t\omega}$, where $\alpha_{x,y} = |E_x \oplus E^{'}_{k\omega+y}|$. $\alpha_{x,y}$ is the number of parity blocks which need d_x for their modifications when d_x is moved to $(k\omega + y)$th data unit in the scaled stripe.

To select a data migration approach with lower I/O cost, we first sort the entries in each row of $A = (\alpha_{x,y})_{k\omega \times t\omega}$ in the ascending order and store them in a matrix $B = (\beta_{x,y})_{k\omega \times t\omega}$. Besides, for each entry in each row after the sorting, we record the pre-sorting column index of it. The results are recorded with another matrix $R = (\gamma_{x,y})_{k\omega \times t\omega}$. Thus, $\beta_{x,y}$ is the number of parity blocks to be modified when migrating d_x to $(k\omega + \gamma_{x,y})$th data unit of the scaled stripe.

Take P in Fig. 1b and $P^{''}$ in Fig. 3 as an example to explain the generations of A, B, R. The entries in the first row of A are calculated from $|E_0 \oplus E^{''}_8|, \ldots, |E_0 \oplus E^{''}_{15}|$. The

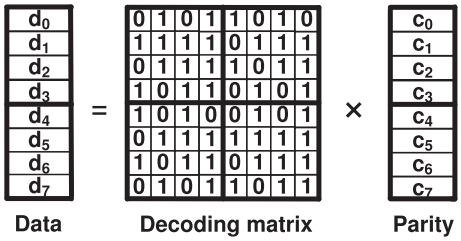


Fig. 5. The decoding matrix of the parity matrix in Fig. 1b.

entries are $5, 4, 5, 3, 6, 3, 1, 4$. After sorting the entries in the first row, the items become $1, 3, 3, 4, 4, 5, 5, 6$, stored in the first row of B. The pre-sorting column indices of the post-sorting items are $6, 3, 5, 1, 7, 0, 2, 4$, recorded as the first row of R. $\beta_{0,0} = 1$ is the minimum number of parity blocks to be modified if migrating d_0 to $(8 + \gamma_{0,0} = 14)$th data unit of the scaled stripe.

5.2 Algorithm Design

Using P and $P^{'}$ with orders of $m\omega \times k\omega$ and $m\omega \times (k+t)\omega$, respectively, we design a data migration scheme as follows.

Design of the data migration scheme:

1. We calculate the I/O overhead for each possible movement from d_j to the $j^{'}$th data unit of the scaled stripe, where $0 \leq j \leq k\omega - 1, k\omega \leq j^{'} \leq (k+t)\omega - 1$. The results are stored in A, B and R.
2. We search for an initial data migration method. We should migrate $\frac{t\omega}{(k+t)}$ data blocks from each old data disk. Thus, we select the smallest $\frac{t\omega}{(k+t)}$ elements from $\beta_{i\omega,0}$ to $\beta_{(i+1)\omega-1,0}$ for each i from 0 to $k-1$. Note that the selection of $\beta_{x,y}$ relates to a determined movement from d_x to the $(k\omega + \gamma_{x,y})$th data unit of the scaled stripe while $\beta_{x,y}$ is the induced I/O overhead.
3. We make some adjustments to the initial data migration method. We must make sure that the migration writes $\frac{k\omega}{(k+t)}$ data blocks to each new data disk to achieve the uniform data distribution. If migrating d_x to the $(k\omega + \gamma_{x,y})$th data unit in the scaled stripe by the selection of $\beta_{x,y}$ induces nonuniform data distribution, we select $\beta_{x,y+1}$ instead of $\beta_{x,y}$.
4. We obtain the final data migration approach through steps 2 and 3.

Note that in step 3, a movement from d_x to the $(k\omega + \gamma_{x,y})$th storage unit in the scaled stripe (induced by the selection of $\beta_{x,y}$) may induce nonuniform data distribution due to the following two reasons. One is that the $(k\omega + \gamma_{x,y})$th storage unit in the scaled stripe may cause more than $\frac{k\omega}{(k+t)}$ writes on a new data disk. The other is that the $(k\omega + \gamma_{x,y})$th storage unit may be overlapped with another storage unit indexed by $(k\omega + \gamma_{x',y'})$ of a previous selection of $\beta_{x',y'}$. If the above two cases happen, we repeatedly substitute the entry $\beta_{x,y}$ with $\beta_{x,y+1}$ until that the uniform distribution property is satisfied.

We still take P in Fig. 1b and $P^{''}$ in Fig. 3 as an example. In step 1, we calculate A, B, R as stated above. In step 2, we select $\beta_{0,0} = 1, \beta_{1,0} = 1, \beta_{5,0} = 2$ and $\beta_{6,0} = 1$ as candidates, which correspond to migrating d_0 to the 14th storage unit, d_1 to the 15th storage unit, d_5 to the 14th unit and d_6 to the 15th unit of the scaled stripe, respectively. In step 3, adding $\beta_{5,0} = 2$ and $\beta_{6,0} = 1$ will induce nonuniform data distribution, we thus replace them with $\beta_{5,1} = 3$ and $\beta_{6,1} = 4$, respectively. The movements are now migrating d_5 to the ninth unit and d_6 to the eighth unit of the scaled stripe, respectively. In step 4, we have a result data migration approach, which is also the one depicted in Fig. 4.

The time complexity of the searching algorithm is derived as follows. Step 1 takes time of $O(kt\omega^2)$, while step 2 and step 3 take times of $O(k\omega)$ and $O(kt^2\omega^2/(k+t))$, respectively. Therefore, the total time complexity is $O(kt^2\omega^2/(k+t) + kt\omega^2 + k\omega)$. We see that the time complexity of the searching algorithm is reasonable for practical deployment.

One side effect of the data migration approaches is that they may compromise the file access performance after scaling. In this paper, we are only concerned with designing data migration processes so as to save the amount of parity blocks being modified. For the design of data migration policies that achieve both good on-scaling performance and post-scaling read (write) performance, we pose it as a future work.

6 MDS Property

From Section 3.4, we observe that we can exploit the MDS property to save the amount of data blocks being read for data migration. In this section, we propose an algorithm to efficiently alleviate the read operations of several data blocks for the data migration process. Note that our design aims at reading fewer data blocks than the minimum data migration schemes but still guarantees the minimum amount of migrated data.

6.1 Problem Statement

Recall that a (k, m, w) CRS code can be expressed by an $n\omega \times k\omega$ encoding matrix of bits. Based on the encoding matrix, we give the following definitions.

Definition 2. *Given an $n\omega \times k\omega$ encoding matrix of a CRS code, an* encoding sub-matrix *is a $k\omega \times k\omega$ matrix extracted from the encoding matrix. It is formed by assuming data loss of any m disks and deleting the corresponding rows in the encoding matrix. It indicates how data (parity) blocks in the remaining k disks are encoded from the original $k\omega$ data blocks.*

For example, in Fig. 1b, the parity matrix P is also an encoding sub-matrix derived from disks C_0 and C_1. It shows how the parity blocks $c_0 \sim c_7$ are encoded from data blocks $d_0 \sim d_7$.

Definition 3. *A* decoding matrix *is the inverse matrix of an encoding sub-matrix, and it indicates how the original data blocks can be decoded by the data (parity) blocks in the surviving k disks.*

Continuing from the previous example, we can compute the decoding matrix for P in Fig. 1b. The decoding matrix is depicted in Fig. 5, and it shows how $d_0 \sim d_7$ are decoded by $c_0 \sim c_7$. Note that the construction of the CRS encoding

matrix guarantees that any encoding sub-matrix extracted is invertible, and so the decoding matrix does exist.

Here comes our problem. Given the post-scaling encoding matrix and the data migration scheme, how can we reduce the amount of data blocks being read for data migration as many as possible. We follow the data migration scheme designed in the last section (i.e., Section 5). We aim at eliminating the read of some migrated data blocks, and we use the parity blocks and other migrated data blocks to decode them. To save the amount of data blocks read, we may increase the amount of parity blocks read. However, we must guarantee that the total amount of read blocks can be reduced. We emphasize that we must only use other migrated data blocks along with the parity blocks to decode the desired blocks. For instance, in Section 3.4, we can use all parity blocks (i.e., $c_0 \sim c_7$) to decode the four migrated data blocks (i.e., d_0, d_1, d_5, d_6). we increase the read of parity blocks by one, but save the read of data blocks by four, and so the overall I/O cost and network traffic can be reduced.

We label the decoding matrix by $V = (v_{i,j})_{k\omega \times k\omega}$. Let b_j be the block corresponding to the jth column of V, b_j may be a data block $d_{i'}$ or a parity block $c_{j'}$. Data block d_i corresponds to the ith row of V, and can be decoded by a set of blocks with a entry one in the ith row, which we denote as O_i. For example, in Fig. 5, the zeroth column corresponds to $b_0 = c_0$. The zeroth row corresponds to d_0, which can be decoded by $O_0 = \{c_1, c_3, c_4, c_6\}$.

Let σ denote the data migration policy, $\sigma(i) = i$ means d_i is an un-migrated data block while $\sigma(i) \neq i$ means d_i is a migrated one. Let M be the set of migrated data blocks that are considered to be decoded. Let H be the set of migrated data blocks which help for decoding. Let T be the set of parity blocks being read, and we must use T and H to decode M. Again in Fig. 5, $M = \{d_0, d_1, d_5, d_6\}$, $H = \emptyset$, and $T = \{c_0 \sim c_7\}$. Based on the designed data migration process in Section 5, we can develop the algorithm to save the migration I/Os, which will be described in the following section.

6.2 Algorithm Design

The main idea of our algorithm is as follows. We traverse all possible decoding matrices of an encoding matrix. For each decoding matrix, there are a total of $k\omega$ rows. For each row i, it corresponds to a data block d_i, and indicates a set of blocks O_i that can decode d_i. If d_i is a migrated data block and O_i consists of other migrated data blocks as well as the parity blocks, we then include d_i into M and put the data blocks and the parity blocks of O_i into H and T, respectively. We check a decoding matrix row by row, and enumerate all decoding matrices, such that we can gradually maximize the amount of data blocks in the set M.

Using the MDS property to save I/Os:

1. Initialize $M = \emptyset, H = \emptyset, T = \emptyset$.
2. For any m out of n disks, we assume data loss happens in these m disks and construct the *decoding matrix* (i.e., $V = (v_{i,j})_{k\omega \times k\omega}$).
   - For each row i ($0 \leq i \leq k\omega$), if $\sigma(i) \neq i$ (i.e., d_i is a migrated data block):
     - Compute block set O_i that can decode d_i.
     - For each $b_j \in O_i$, if $b_j = d_{i'}$ is another migrated data block or $b_j = c_{j'}$ is a parity block, set f = **true**; otherwise, set f = **false**.
     - If f = **true** (This is to guarantee that we only utilize other migrated data blocks and the parity blocks to decode d_i):
       - If $d_i \notin M \cup H$, put d_i into M. For each $b_j \in O_i$, if $b_j = d_{i'}$ is another migrated data block and $d_{i'} \notin M \cup H$, put $d_{i'}$ into H; if $b_j = c_{j'}$ is a parity block and $c_{j'} \notin T$, put $c_{j'}$ into T.
       - If $d_i \in H$: for each $b_j \in O_i$, if $b_j = d_{i'}$ and $b_j \in M$, set g = **false**; otherwise, set g = **true**.
       - If g = **true**, put d_i into M. For each $b_j \in O_i$, if $b_j = d_{i'}$ and $d_{i'} \notin H$, put $d_{i'}$ into H; if $b_j = c_{j'}$ and $c_{j'} \notin T$, put $c_{j'}$ into T.
3. Return M, H, T.

Note that the steps in which g = **true** are to to handle the case where d_i is a candidate to be put into M but d_i has already been in H. If no block of O_i belongs to M, then we put d_i into M and put the blocks of O_i into H and T.

For the example in Fig. 1b, we construct the decoding matrix for disks C_0 and C_1, and then obtain the decoding functions for d_0, d_1, d_5, d_6. Thus, we have $M = \{d_0, d_1, d_5, d_6\}$, $H = \emptyset$, and $T = \{c_0 \sim c_7\}$. The saving by the algorithm is desired since we can further release the load on old disks, which benefits more on the on-scaling user response time performance. Note that our design trades extra computation overhead for the I/O savings, and we will demonstrate in our experiments presented in Section 8 that the penalization is negligible compared to the savings.

The time complexity of the above algorithm is derived as follows. The total number of encoding sub-matrices is $\binom{n}{m}$. Each matrix inversion applied to an encoding sub-matrix to get a decoding matrix takes time of $O(k^3\omega^3)$. For each decoding matrix V, we traverse all its elements so as to obtain its decoding functions, and this takes time of $O(k^2\omega^2)$. Thus, the overall time complexity is $O((k^3\omega^3 + k^2\omega^2) \times \binom{n}{m})$. Note that the computational complexities of the above three algorithms are reasonable so that the algorithms can be run off-line.

7 Numerical Study

In this section, we numerically analyze the amount of I/Os of the post-scaling parity matrix selection algorithm, the data migration process design algorithm, and the MDS optimization algorithm presented above. We adopt five schemes: (1) Use the post-scaling parity matrix from Jerasure and the naive data migration process (abbr. as JM-NM), which is also the basic scheme. (2) Use the post-scaling parity matrix generated by the method in Section 4.2 and the naive data migration process (abbr. as OM-NM). (3) Use the post-scaling parity matrix generated by the method in Section 4.2 and optimize the data migration process through the algorithm in Section 5.2 (abbr. as OM-OM). (4) OM-OM optimized with the algorithm in Section 6.2 (abbr. as OM-OM-MDS). (5) OM-NM optimized with the algorithm in Section 6.2 (abbr. as OM-NM-MDS).

TABLE 1
I/O Usage of the Five Scaling Schemes

(k, m, t, ω)	JM-NM	OM-NM	OM-OM	OM-OM-MDS	OM-NM-MDS
$(3,3,1,4)$	$3/3+0$,12/12	$3/3+0$,12/12	$3/3+0$,7/7	$3/3+0$,7/7	$0/3+0$,12/12
$(3,3,2,4)$	$4/4+8$,12/12	$4/4+0$,12/12	$4/4+0$,10/10	$0/4+0$,12/10	$0/4+0$,12/12
$(3,3,3,4)$	$6/6+6$,12/12	$6/6+0$,12/12	$6/6+0$,9/9	$0/6+0$,12/9	$0/6+0$,12/12
$(4,3,1,4)$	$3/3+13$,12/12	$3/3+0$,11/11	$3/3+0$,8/8	$3/3+0$,8/8	$3/3+0$,11/11
$(4,3,2,4)$	$4/4+12$,11/11	$4/4+0$,11/11	$4/4+0$,8/8	$3/4+0$,8/8	$3/4+0$,11/11
$(4,3,3,4)$	$6/6+10$,12/12	$6/6+0$,12/12	$6/6+0$,9/9	$4/6+0$,10/9	$4/6+0$,12/12
$(4,4,2,4)$	$4/4+12$,14/14	$4/4+0$,14/14	$4/4+0$,13/13	$0/4+0$,16/13	$0/4+0$,16/14
$(4,4,3,4)$	$6/6+10$,16/16	$6/6+0$,16/16	$6/6+0$,14/14	$0/6+0$,16/14	$0/6+0$,16/16
$(4,4,4,4)$	$8/8+8$,16/16	$8/8+0$,16/16	$8/8+0$,14/14	$0/8+0$,16/14	$0/8+0$,16/16
$(5,4,2,4)$	$5/5+15$,16/16	$5/5+0$,16/16	$5/5+0$,15/15	$4/5+0$,15/15	$4/5+0$,16/16
$(5,4,3,4)$	$6/6+14$,16/16	$6/6+0$,16/16	$6/6+0$,14/14	$3/6+0$,16/14	$4/6+0$,16/16
$(5,4,4,4)$	$8/8+12$,16/16	$8/8+0$,16/16	$8/8+0$,15/15	$5/8+0$,16/15	$4/8+0$,16/16
$(6,3,1,4)$	$3/3+21$,12/12	$3/3+0$,11/11	$3/3+0$,9/9	$3/3+0$,9/9	$3/3+0$,11/11
$(6,3,2,4)$	$6/6+0$,12/12	$6/6+0$,12/12	$6/6+0$,9/9	$6/6+0$,9/9	$5/6+0$,12/12
$(6,3,3,4)$	$6/6+0$,11/11	$6/6+0$,11/11	$6/6+0$,10/10	$6/6+0$,10/10	$5/6+0$,11/11
$(6,5,2,4)$	$6/6+18$,20/20	$6/6+0$,19/19	$6/6+0$,19/19	$5/6+0$,19/19	$3/6+0$,20/19
$(6,5,3,4)$	$6/6+18$,19/19	$6/6+0$,19/19	$6/6+0$,18/18	$6/6+0$,18/18	$3/6+0$,20/19
$(6,5,4,4)$	$8/8+16$,20/20	$8/8+0$,20/20	$8/8+0$,20/20	$4/8+0$,20/20	$3/8+0$,20/20

The field power ω is fixed as 4.

7.1 Comparison of The Five Scaling Schemes

Table 1 shows a comparison of the amount of data reads/writes for data migration plus the amount of data reads and parity reads/writes for parity modification in one stripe. Note that the parameters around $k = 6$ are commonly deployed in practical storage systems.

As shown in Table 1, OM-NM can efficiently reduce the amount of data blocks read for parity modification over JM-NM, and OM-OM can further reduce the amount of parity blocks modified for parity modification over OM-NM. Besides, OM-OM-MDS can further save the amount of data blocks being read for data migration over OM-OM. One example is the case where $(k = 3, m = 3, t = 2, \omega = 4)$. The amount of data blocks read for parity modification of JM-NM is 8, while that of OM-NM is 0. The amount of parity blocks being read (written) of OM-NM is 12 (12), while those of OM-OM are 10 and 10, respectively. OM-OM requires four data blocks to be read for data migration, while OM-OM-MDS only requires 0. However, OM-OM-MDS must read more parity blocks to decode the data blocks, but the overall system I/O overhead is indeed reduced.

In the cases where $(k = 6, m = 5, t = 2, \omega = 4)$ and $(k = 6, m = 5, t = 4, \omega = 4)$, OM-NM-MDS requires fewer I/Os than OM-OM-MDS, mainly because the naive data migration process is more ordered. We will show in Section 7.2 that with different ω, there are more scaling parameter settings under which OM-NM-MDS outperforms OM-OM-MDS. Nevertheless, whichever of OM-OM-MDS and OM-NM-MDS requires fewer I/Os, we can choose it as our scaling scheme.

We next conduct a series of tests for $m = 3, m \leq k$, $t \leq k, 3 \leq \omega \leq 6$, and there are a number of 1,176 parameters. Among them, there are 1,052 cases where OM-NM outperforms JM-NM in the amount of data blocks being read for parity modification. There are 307 cases where OM-OM further reduces the amount of parity blocks modified over OM-NM. Besides, there are 706 cases where OM-OM-MDS (or OM-NM-MDS if OM-NM-MDS is better) reduces the amount of data blocks being read for data migration over OM-OM while still saving the overall amount of scaling I/Os. In summary, these results prove the efficacy of our schemes.

7.2 Impact of the Field Power ω

We now vary the field power ω to justify the applicability of our scaling scheme in adoption of more dynamic field sizes. Table 2 shows the amount of I/Os with different ω. For example, with $(k = 3, m = 3, t = 2, \omega = 5 \sim 8)$, OM-NM constantly outperforms JM-NM in terms of the amount of data blocks read for parity modification and OM-OM constantly outperforms OM-NM in terms of the amount of parity blocks modified. Moreover, OM-OM-MDS can further save the amount of data blocks read for data migration to reduce the overall system I/Os.

In the case where $(k = 5, m = 3, t = 2, \omega = 5)$, OM-NM-MDS is better than OM-OM-MDS in terms of the scaling I/Os. We further compare OM-NM-MDS with OM-OM-MDS through a series of tests. When $2 \leq k \leq 12, 2 \leq m \leq 6$, $1 \leq t \leq k, \omega = 5$, there are a total of 355 cases, among which there are 173 cases where OM-NM-MDS incurs fewer I/Os, and 137 cases where OM-OM-MDS incurs fewer I/Os, and 45 cases where OM-NM-MDS and OM-OM-MDS have the same amount of I/Os. When ω varies into 6, there are a number of 355 cases, and OM-NM-MDS performs better in 187 cases, while OM-OM-MDS performs better in 138 cases, and they have the same performance in the remaining 30 cases. When k grows even larger (e.g., $k = 28, 29$), OM-OM-MDS typically outperforms OM-NM-MDS. However, the situations where OM-NM-MDS is better still exist.

7.3 Post-Scaling Encoding Performance

We next compare the encoding performance between the post-scaling parity matrices with OM-NM and the post-scaling parity matrices from the Jerasure library. The parameters are $m = 3\ (4), m \leq k, t \leq k, 3 \leq \omega \leq 6$, and there

TABLE 2
I/O Usage of the Five Scaling Schemes With Different ω

(k, m, t, ω)	JM-NM	OM-NM	OM-OM	OM-OM-MDS	OM-NM-MDS
$(3, 3, 2, 5)$	$6/6 + 0{,}15/15$	$6/6 + 0{,}15/15$	$6/6 + 0{,}13/13$	$0/6 + 0{,}15/13$	$0/6 + 0{,}15/15$
$(3, 3, 2, 6)$	$6/6 + 12{,}17/17$	$6/6 + 0{,}17/17$	$6/6 + 0{,}17/17$	$0/6 + 0{,}18/17$	$0/6 + 0{,}18/17$
$(3, 3, 2, 7)$	$8/8 + 13{,}21/21$	$8/8 + 0{,}21/21$	$8/8 + 0{,}20/20$	$0/8 + 0{,}21/20$	$0/8 + 0{,}21/21$
$(3, 3, 2, 8)$	$9/9 + 15{,}24/24$	$9/9 + 0{,}24/24$	$9/9 + 0{,}21/21$	$0/9 + 0{,}24/21$	$0/9 + 0{,}24/24$
$(4, 3, 2, 5)$	$6/6 + 14{,}15/15$	$6/6 + 0{,}15/15$	$6/6 + 0{,}14/14$	$5/6 + 0{,}15/14$	$4/6 + 0{,}15/15$
$(4, 3, 2, 6)$	$8/8 + 16{,}18/18$	$8/8 + 0{,}18/18$	$8/8 + 0{,}16/16$	$5/8 + 0{,}17/16$	$5/8 + 0{,}18/18$
$(4, 3, 2, 7)$	$8/8 + 20{,}20/20$	$8/8 + 0{,}20/20$	$8/8 + 0{,}17/17$	$8/8 + 0{,}17/17$	$5/8 + 0{,}21/20$
$(4, 3, 2, 8)$	$10/10 + 22{,}24/24$	$10/10 + 0{,}24/24$	$10/10 + 0{,}20/20$	$10/10 + 0{,}20/20$	$6/10 + 0{,}24/24$
$(5, 3, 2, 5)$	$6/6 + 19{,}15/15$	$6/6 + 0{,}14/14$	$6/6 + 0{,}14/14$	$6/6 + 0{,}14/14$	$5/6 + 0{,}14/14$
$(5, 3, 2, 6)$	$8/8 + 22{,}18/18$	$8/8 + 0{,}18/18$	$8/8 + 0{,}17/17$	$8/8 + 0{,}17/17$	$7/8 + 0{,}18/18$
$(5, 3, 2, 7)$	$10/10 + 25{,}21/21$	$10/10 + 0{,}21/21$	$10/10 + 0{,}20/20$	$8/10 + 0{,}20/20$	$8/10 + 0{,}21/21$
$(5, 3, 2, 8)$	$10/10 + 30{,}23/23$	$10/10 + 0{,}23/23$	$10/10 + 0{,}21/21$	$10/10 + 0{,}21/21$	$8/10 + 0{,}23/23$

are a total of 2,290 cases. On average, the matrices with OM-NM have 1.94 percent more entries of one compared with the matrices from Jerasure. In some cases, the matrices with OM-NM have the same number of ones as the Jerasure matrices, e.g., when $(k = 6, m = 3, t = 3, \omega = 4)$, both the Jerasure matrix and the matrix with OM-NM have 172 ones. However, in most cases, the matrices with OM-NM have more entries of one. The worst case we find is that when $(k = 4, m = 4, t = 4, \omega = 5)$, there are 299 ones in the Jerasure matrix, while 342 ones in the matrix with OM-NM. Nevertheless, the performance penalty of our designed matrices is negligible compared to the savings in the scaling process.

8 Experimental Evaluations

In this section, we evaluate the performance of the proposed scaling schemes (i.e., JM-NM, OM-NM, OM-OM, OM-OM-MDS, and OM-NM-MDS) atop a networked file system and show that OM-NM, OM-OM, OM-OM-MDS (OM-NM-MDS) improves the scaling performance of JM-NM.

8.1 Methodology

We conduct our experiments atop an open-source networked file system called NCFS [6]. NCFS is a proxy-based distributed file system that interconnects multiple storage devices over a real network and transparently stripes data across the storage nodes. NCFS adopts a layered design that allows extensibility, such that various coding schemes can be readily integrated. In our experiments, we have implemented CRS codes based on Jerasure atop NCFS.

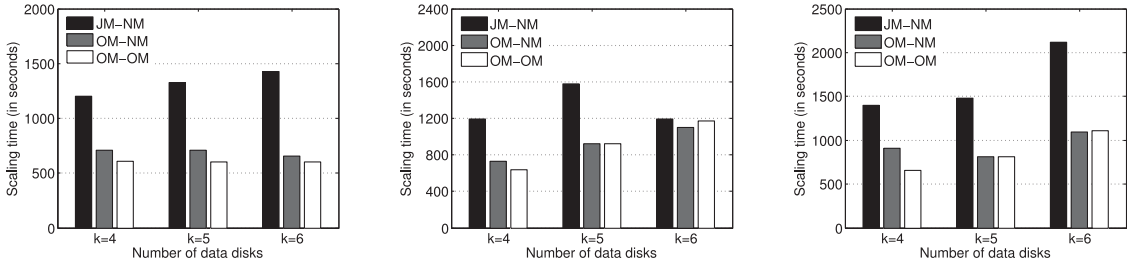
We further develop a scaling module atop NCFS. We have implemented the five scaling schemes (i.e., JM-NM, OM-NM, OM-OM, OM-OM-MDS, and OM-NM-MDS). Note that whichever of OM-OM-MDS and OM-NM-MDS incurs fewer amount of scaling I/Os, we select it as our scaling scheme. We implement the scaling operations with *a scaling thread*. For JM-NM, OM-NM, and OM-OM, the scaling thread runs in two steps: *i)* migrate the data blocks, *ii)* modify the parity blocks. Note that the data blocks for data migration in step *i)* are cached for parity modification in step *ii)* such that the total I/Os can be reduced. Also, a data block required for the modification of multiple parity blocks is cached to save the I/O cost in step *ii)*. For OM-OM-MDS and OM-NM-MDS, the scaling thread is implemented distinctly: *i)* read the data blocks in H and the parity blocks in T, *ii)* decode the data blocks of M, *iii)* write the data blocks of $M \cup H$ into new disks, *iv)* update the parities that are required to be modified, and *v)* write these parities back. Therefore, the data migration and parity modification processes are separated. In particular, the data migration part takes charge of reading the data blocks in H, decoding the data blocks of M, and writing the data blocks in $M \cup H$ to new disks. The parity modification part is composed of reading, updating, and writing the parity blocks (Note that the parity modification process requires only the migrated data blocks). A scaling thread is deployed at the proxy node of NCFS such that it is capable of coordinating the reads and writes among all disks.

We deploy the NCFS proxy on an Intel i3 Dual-Core 3.30 GHZ PC with 2 GB DRAM. The storage devices are connected via a Gigabit switch. The proxy node communicates with the storage nodes via the ATA over Ethernet protocol [5]. The cluster consists of 13 storage nodes with heterogeneous access performance and one proxy node, a total of 14 nodes. We emphasize that our main goal is to compare the relative scaling performance of the five scaling schemes, rather than their actual scaling performance in modern storage clusters.

We mainly focus on the scaling time (in seconds). The average scaling time is obtained as follows. We set the capacity of each disk as 1 GB, and upload 1 GB of data/parity blocks to each storage disk through the NCFS proxy using the CRS coding strategy. We then add a certain number of new data disks into the current NCFS configuration and perform the scaling operations. The scaling operations are performed three times for each case, and we compute the average scaling time.

In our experiments, we fix the strip size (i.e., ω) as 4 and compare the scaling time of various scaling schemes under different configurations of (k, m, t). In Table 2, we have validated the efficiency of our scaling scheme using different ω. Note that in CRS codes with fault-protection higher than two, ω should be as small as possible so as to achieve good encoding performance [8].

Our first set of experiments is to evaluate the scaling time performance of OM-OM and OM-NM over JM-NM. We compare the scaling time of OM-OM and OM-NM with that of JM-NM under the single-threaded and the multi-



(a) (m=3, t=1) (b) (m=3, t=2) (c) (m=4, t=1)

Fig. 6. Experiment 1.1 - Scaling performance with different number of data disks.

threaded implementations. As we see that the improvements of OM-OM and OM-NM over JM-NM are stable with different block sizes. For example in the case where $(k = 3, m = 3, t = 1)$, OM-NM outperforms JM-NM by 19.07 $\sim$ 19.86 perecent in scaling time with block size ranging from 512 to 4,096 KB, while OM-OM outperforms JM-NM by 19.10 $\sim$ 19.92 percent. Therefore, we fix the block size as 512 KB, and present the experimental results under different settings of (k, m, t) in Sections 8.2.1 to 8.2.3. We also study the impact of parallel scaling implemented with multi-threading in Section 8.2.4.

We further evaluate the performance improvement by exploiting the MDS property. That is, we compare the five scaling schemes (i.e., JM-NM, OM-NM, OM-OM, OM-OM-MDS, and OM-NM-MDS) in terms of the total scaling time, and the results are shown in Section 8.2.5. As discussed in Section 7, we see that OM-OM-MDS can efficiently reduce the scaling time over OM-OM. Besides, in certain cases, OM-NM-MDS outperforms OM-OM-MDS in terms of the scaling time.

8.2 Results

8.2.1 Impact of Different Numbers of Data Disks

We now evaluate the scaling performance of JM-NM, OM-NM and OM-OM with different numbers of data disks. We select three typical sets of m and t, and then vary the number of data disks k from 4 to 6.

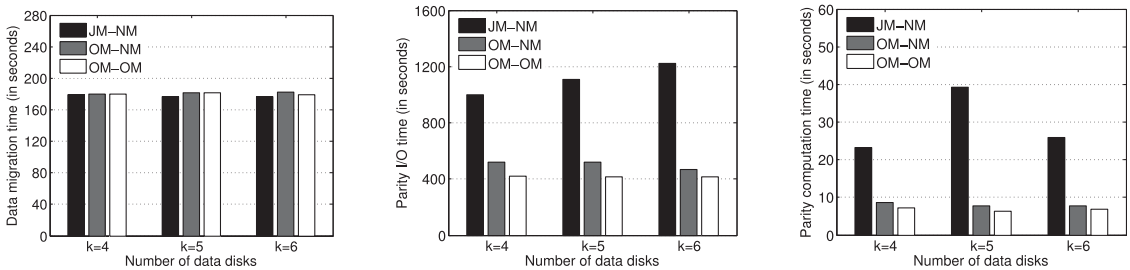
Figs. 6a, 6b and 6c provide the results under $(m = 3, t = 1), (m = 3, t = 2)$ and $(m = 4, t = 1)$, respectively. OM-NM constantly improves the scaling performance over JM-NM, while OM-OM further reduces the scaling time of OM-NM. Take the scaling case with $k = 4$ in Fig. 6a as an example, OM-NM reduces the scaling time of JM-NM by 41.25 percent, and OM-OM further reduces the scaling time of OM-NM by 14.42 percent.

Another important observation is that the improvements of both OM-NM and OM-OM over JM-NM become more significant as the number of data disks increases. For example, in Fig. 6a, OM-NM (OM-OM) saves the scaling time over JM-NM by 41.25 percent (49.72 percent) $\sim$ 53.95 percent (57.86 percent) with k ranging from 4 to 6. The main reason is that the amount of read data blocks of JM-NM goes up dramatically as k increases, while that of OM-NM (OM-OM) has a relatively stable increase. To be more detailed, the amount of read data blocks of JM-NM is typically $k\omega$, while that of OM-NM (OM-OM) is $\lfloor \frac{k\omega}{k+t} \rfloor$.

An exceptional case occurs when $(k = 6, m = 3, t = 2)$ in Fig. 6b, where the scaling time of JM-NM is almost identical to that of OM-NM. The reason is that the Jerasure matrix when $(k = 6, m = 3, t = 2, \omega = 4)$ has the same property as the matrix with OM-NM in the sense that the entries of the first $k\omega$ columns of P' are identical to that of P. Therefore, JM-NM in this case requires only the migrated data blocks for parity modification.

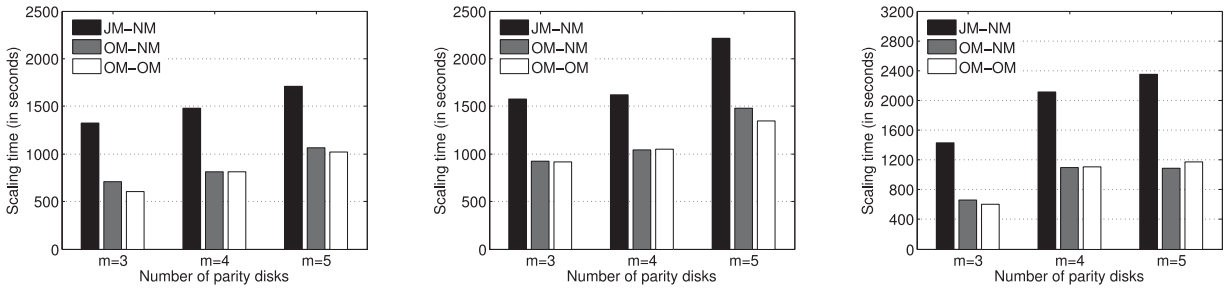
Recall that the scaling process can be divided into two main parts. In order to provide a detailed analysis of the scaling time reduction, we further divide the cost of parity modification into two parts, the I/O overhead and the computational cost for parity modification. Note that the I/O overhead includes reading additional data blocks and the related parity blocks for parity modification, as well as writing the modified parity blocks back. The computational cost takes charge of the computational process for the parity blocks.

Here we provide an analysis of time breakdown for the scaling operation. We select $(m = 3, t = 1)$ with k varying from 4 to 6, and show the results in Fig. 7. Take $(k = 4, m = 3, t = 1)$ as an example, the data migration part contributes to 14.90 percent of the overall scaling time, while the I/O overhead and computational cost for parity modification contribute to 83.17 and 1.93 percent, respectively. To summarize, the experimental results reveal that it is critical to optimize the I/O overhead for



(a) The data migration time (b) The parity I/O time (c) The parity computation time

Fig. 7. Experiment 1.2 - Time breakdown for the three scaling schemes with $(m = 3, t = 1)$.



(a) (k=5, t=1) (b) (k=5, t=2) (c) (k=6, t=1)

Fig. 8. Experiment 2 - Scaling performance with different number of parity disks.

parity modification in the scaling process, so as to reduce the overall scaling time.

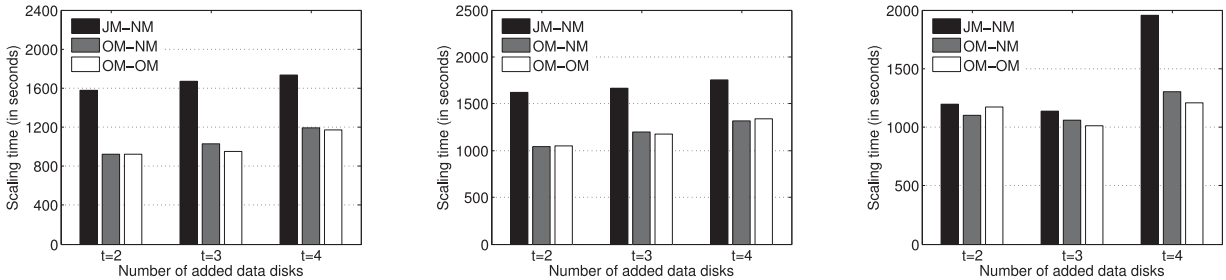
As shown in Fig. 7b, OM-NM reduces the parity I/O time of JM-NM by 48.17 $\sim$ 61.89 percent, while OM-OM has a reduction of 58.27 $\sim$ 66.11 percent. Recall that we compare the number of parity I/Os for the three scaling schemes with different scaling parameters in Section 7. Take $k = 4$ as an example, the number of parity I/Os for the three scaling schemes are 37, 22, 16, respectively. We observe from Fig. 7b that the ratio of the parity I/O time of the three scaling schemes is 1,001 s: 519 s:418 s, which is approximately proportional to 37:22:16. Therefore, these experimental results conform to our theoretical analysis.

8.2.2 Impact of Different Numbers of Parity Disks

Fig. 8 depicts the scaling performance with different number of parity disks. We vary m from 3 to 5, and also select three typical sets of k, t. As shown in Fig. 8, with the increased number of parity disks, the improvements of both OM-NM and OM-OM over JM-NM decrease. The reason is that as m increases, the amount of parity blocks read/written for all three scaling schemes increases roughly the same. Specifically, the amount of parity blocks read/written is approximately $m\omega$. For example, in Fig. 8a, OM-NM outperforms JM-NM in scaling time from 46.61 $\sim$ 37.69 percent, and OM-OM outperforms JM-NM in scaling time by 54.65 $\sim$ 40.45 percent.

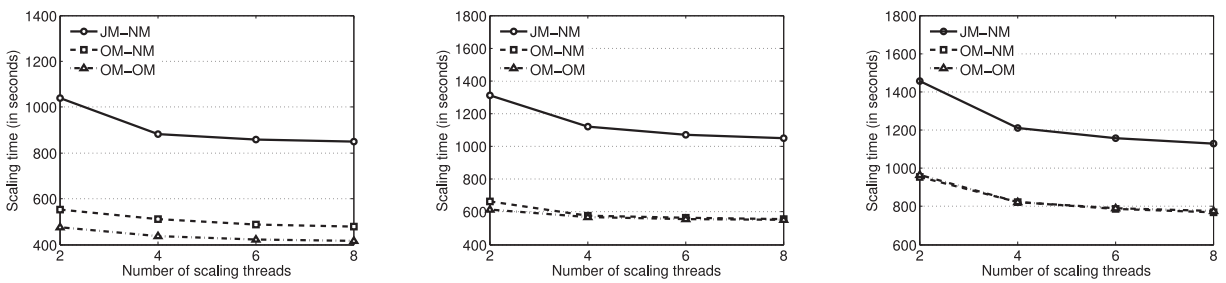
8.2.3 Impact of Different Numbers of New Disks

We now present the scaling performance versus the number of newly-added data disks. We vary t from 2 to 4 and select some typical combinations of (k, m). Figs. 9a, 9b, and 9c show the results for $(k = 5, m = 3), (k = 5, m = 4)$ and $(k = 6, m = 3)$, respectively. The improvements of OM-NM and OM-OM over JM-NM decrease as the number of new data disks increases. The reason is that the amount of data blocks being read of JM-NM keeps as a constant as t increases, while that of OM-NM (OM-OM) increases with the increased number of added data disks. For example in Fig. 9a, OM-NM (OM-OM) reduces the scaling time over JM-NM by 41.56 percent (43.08 percent) $\sim$ 31.11 percent (32.41 percent).



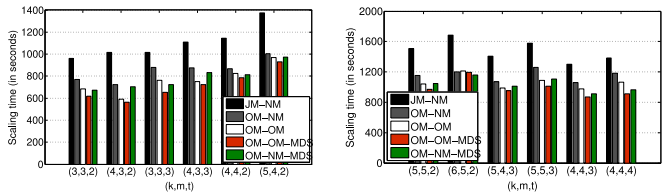
(a) (k=5, m=3) (b) (k=5, m=4) (c) (k=6, m=3)

Fig. 9. Experiment 3 - Scaling performance with different number of added data disks.



(a) (k=4, m=3, t=1) (b) (k=5, m=4, t=1) (c) (k=5, m=4, t=3)

Fig. 10. Experiment 4 - Multi-threaded scaling.



(a) Parameter set a (b) Parameter set b

Fig. 11. Experiment 5 - Comparison of the five scaling schemes.

8.2.4 Impact of Parallel Scaling

We now evaluate the scaling performance based on parallelized implementation. To boost the scaling performance, we implement a *parallel scaling architecture*. We use the *stripe-oriented* [4] technique to implement the parallel scaling. Specifically, multiple scaling threads are created, and each thread is associated with a group of stripes and performs the scaling process within its own group of stripes. Here we conduct experiments by increasing the number of scaling threads from 2 to 8. We show the results of the scaling time versus the number of scaling threads in Fig. 10, which reveals that OM-OM and OM-NM reduce the scaling time over JM-NM regardless of the number of scaling threads. In fact, as the number of scaling threads goes up from 2 to 4, the scaling time for the three scaling schemes drops dramatically. As the number of scaling threads further increases, the scaling time stays relatively stable. The reason is that the server used to deploy the scaling architecture is equipped with a dual-core CPU, which limits the expanding of the number of the scaling threads.

Besides, the scaling performance improvements are almost stable with different number of scaling threads. Take the scaling from a $(k = 4, m = 3)$ CRS code to a $(k = 5, m = 3)$ one in Fig. 10a as an example, OM-NM reduces the scaling time over JM-NM by 42.12 $\sim$ 46.78 percent with the number of scaling threads increasing from 2 to 8, while OM-OM outperforms JM-NM in scaling time by 50.31 $\sim$ 54.24 percent.

8.2.5 Impact of MDS Property

We now evaluate the performance improvement after exploiting the MDS property. That is, we compare the scaling time of the five scaling schemes (i.e., JM-NM, OM-NM, OM-OM, OM-OM-MDS, and OM-NM-MDS). Since we have evaluated the impact of different parameters (i.e., (k, m, t)) and also the influence of different number of scaling threads in the last section, we compare the five scaling schemes under certain scaling parameters (i.e., (k, m, t)) using a single thread. The scaling parameters include $(3, 3, 2), (4, 3, 2), (3, 3, 3), (4, 3, 3), (4, 4, 2), (5, 4, 2)$ and $(5, 5, 2), (6, 5, 2), (5, 4, 3), (5, 5, 3), (4, 4, 3), (4, 4, 4)$.

The results are depicted in Fig. 11. As shown in both Figs. 11a and 11b, OM-OM-MDS reduces the scaling time of OM-OM notably under all scaling parameters. Take Fig. 11a as an example, OM-OM-MDS outperforms OM-OM in terms of the scaling time by 9.34, 4.20, 13.99, 3.25, 4.85, and 4.17 percent. In Fig. 11b, OM-OM-MDS saves the scaling time of OM-OM by 6.48, 1.48, 3.60, 7.09, 11.07, and 14.43 percent. In $(k = 6, m = 5, t = 2)$ of Fig. 11b, OM-NM-MDS can be better than OM-OM-MDS, and OM-OM-MDS only outperforms OM-OM by 1.48 percent while OM-NM-MDS outperforms OM-OM by 4.79 percent.

We next provide a time breakdown to compare the effect of the five scaling schemes in more detail. To do that, we decompose both the data migration process and the parity modification process into multiple parts. The data migration cost is divided into three parts: the data read time, the data write time, and the data decode time. The parity modification cost is divided into four parts: the additional data read time, the parity read time, the update time, and the parity write time. We select $(k = 3, m = 3, t = 2)$, and show the results of the time breakdown in Table 3.

As shown in Table 3, the additional data read time for parity modification of JM-NM is 201.915 s, while that of OM-NM is reduced to 0.0 s. The parity read time and the parity write time of OM-NM is 293.265 and 275.284 s, while those of OM-OM are reduced to 240.740 and 228.619 s, respectively. A side advantage of OM-OM is to simplify the parity update process since OM-NM requires a time of 6.378 s for updating parities while OM-OM only requires a time of 3.989 s. However, the data read time for data migration of OM-OM is increased over OM-NM. This is because the data migration policies designed by OM-OM are sometimes not as ordered as OM-NM, which may increase the disk access time.

Compared to OM-OM, OM-OM-MDS can efficiently reduce the data read time for data migration. The overhead is some data decode time, which is 6.675 s for OM-OM-MDS. However, it is clear that the overhead is negligible compared to the savings. Besides, OM-OM-MDS increases the parity read time because it requires to read more parities to decode the data.

9 Conclusions

In this paper, we address the scaling problem for CRS codes from both the theoretical and the practical perspectives. Theoretically, we first present a method to design the

TABLE 3
Time Breakdown for the Five Scaling Schemes with $(k = 3, m = 3, t = 2)$

Scaling Scheme	Data migration time (s)			Parity modification time (s)			
	Data read	Data write	Decode	Data additional read	Parity read	Parity calculation	Parity write
JM-NM	105.119940	91.678932	0.0	201.915614	293.130786	11.837693	277.363755
OM-NM	104.810053	91.871019	0.0	0.0	293.265798	6.378754	275.284756
OM-OM	117.406128	92.236456	0.0	0.0	240.740590	3.989564	228.619976
OM-OM-MDS	0.0	94.806791	6.675051	0.0	284.942592	4.101241	228.820431
OM-NM-MDS	0.0	92.362616	6.432989	0.0	289.353425	6.332760	280.379103

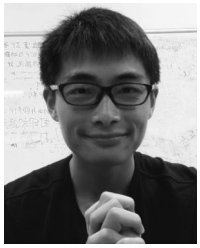
post-scaling parity matrices which can improve the parity modification efficiency, and then propose a searching algorithm to generate efficient data migration processes with very low computational complexity, and finally use the MDS property to improve the efficiency of data migration. For practical deployment, we implement our scaling scheme atop a networked file system. We conduct extensive experiments on the system testbed, and results demonstrate that our scaling scheme saves the scaling time over the basic scheme notably.

Acknowledgments

This work was supported by National Nature Science Foundation of China under Grant No. 61379038 and Huawei Innovation Research Program. The work of Yongkun Li was supported by National Nature Science Foundation of China under Grant No. 61303048, and Anhui Provincial Natural Science Foundation under Grant No. 1508085SQF214. An earlier conference version of the paper appeared in the 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS 2014) [14]. In this journal version, we add an MDS optimization algorithm on the designed data migration process to further optimize the migration I/Os, and more simulations and experiments.

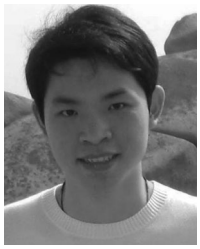
References

[1] J. Bloemer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman, "An XOR-based erasure-resilient coding scheme," *Int. Comput. Sci. Inst.*, Tech. Rep. TR-95-048, Aug. 1995.
[2] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar, "Row-diagonal parity for double disk failure correction," in *Proc. 3rd Conf. File Storage Technol.*, 2004, pp. 1–14.
[3] S. R. Hetzler, et al., "Data storage array scaling method and system with minimal data movement," US Patent 8,239,622, Aug. 7, 2012.
[4] M. Holland, G. A. Gibson, and D. P. Siewiorek, "Architectures and algorithms for on-line failure recovery in redundant disk arrays," *Distrib. Parallel Databases*, vol. 2, no. 3, pp. 295–335, Jul. 1994.
[5] S. Hopkins and B. Coile, AoE (ATA over Ethernet). The Brantley Coile Company, Inc., Tech. Rep. AoEr11, 2009.
[6] Y. Hu, C.-M. Yu, Y. K. Li, P. P. Lee, and J. C. Lui, "NCFS: On the practicality and extensibility of a network-coding-based distributed file system," in *Proc. Int. Symp. Netw. coding*, 2011, pp. 1–6.
[7] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin, et al., "Erasure coding in windows azure storage," in *Proc. USENIX ATC*, 2012, p. 2.
[8] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, Z. Wilcox-O'Hearn, et al., "A performance evaluation and examination of open-source erasure coding libraries for storage," in *Proc. 7th Conf. File Storage Technol.*, 2009, vol. 9, pp. 253–265.
[9] J. S. Plank, S. Simmerman, and C. D. Schuman, "Jerasure: A library in C/C++ facilitating erasure coding for storage applications-version 1.2," Univ. Tennessee, Knoxville, TN, USA, Tech. Rep. CS-08-627, 2008.
[10] J. K. Resch and J. S. Plank, "AONT-RS: Blending security and performance in dispersed storage systems," in *Proc. 9th USENIX Conf. File Stroage Technol.*, 2011, p. 14.
[11] S. Rhea, C. Wells, P. Eaton, D. Geels, B. Zhao, H. Weatherspoon, and J. Kubiatowicz, "Maintenance-free global data storage," *IEEE Internet Comput.*, vol. 5, no. 5, pp. 40–49, Sep. 2001.
[12] C. Wu and X. He, "GSR: A global stripe-based redistribution approach to accelerate RAID-5 scaling," in *Proc. 41st Int. Conf. Parallel Process.*, 2012, pp. 460–469.
[13] C. Wu, X. He, J. Han, H. Tan, and C. Xie, "SDM: A stripe-based data migration scheme to improve the scalability of RAID-6," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2012, pp. 284–292.
[14] S. Wu, Y. Xu, Y. Li, and Y. Zhu, "Enhancing scalability in distributed storage systems with cauchy reed-solomon codes," in *Proc. 20th IEEE Int. Conf. Parallel Distrib. Syst.*, 2014, pp. 5180–525.
[15] G. Zhang, W. Zheng, and K. Li, "Rethinking RAID-5 data layout for better scalability, *IEEE Trans. Comput.*, vol. 63, no. 11, pp. 2816–2828, Nov. 2012.
[16] W. Zheng and G. Zhang, "FastScale: Accelerate RAID scaling by minimizing data migration," in *Proc. 9th USENIX Conf. File Stroage Technol.*, 2011, p. 11.

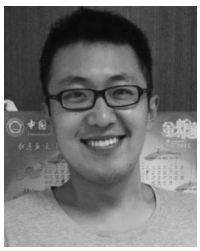


Si Wu received the bachelor's degree in computer science from the University of Science and Technology of China in 2011 and is currently working toward the PhD degree with the School of Computer Science and Technology, University of Science and Technology of China, Hefei, China. His research mainly focuses on erasure codes and distributed storage systems.

Yinlong Xu received the bachelor's degree in mathematics from Peking University in 1983, and the master and PhD degrees in computer science from the University of Science and Technology of China (USTC) in 1989 and 2004, respectively. He is currently a professor with the School of Computer Science and Technology at USTC. Currently, he is leading a group of research students in doing some networking and high-performance computing research. His research interests include network coding, wireless network, combinatorial optimization, design and analysis of parallel algorithm, parallel programming tools, etc. He received the Excellent PhD Advisor Award of Chinese Academy of Sciences in 2006.



Yongkun Li received the BEng degree in computer science from the University of Science and Technology of China in 2008 and the PhD degree in computer science and engineering from The Chinese University of Hong Kong in 2012. He is currently an associate researcher in the School of Computer Science and Technology, University of Science and Technology of China. His research mainly focuses on performance evaluation of networking and storage systems.



Zhijia Yang received the bachelor's degree in computer science from the University of Science and Technology of China in 2011 and is currently working toward the master degree with the School of Software Engineering, University of Science and Technology of China, Hefei, China. His research mainly focuses on distributed cache mechanisms.